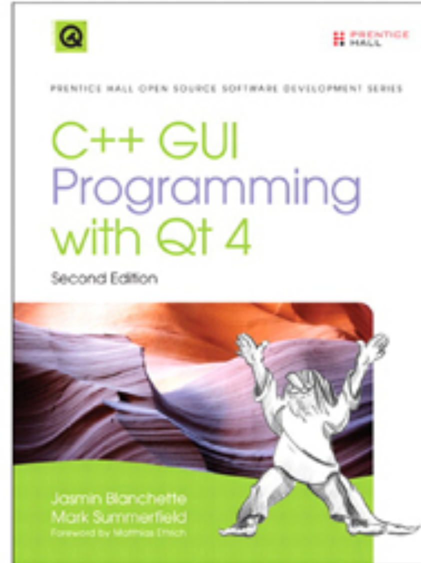
Qt 4 ile C++ GUI Programlama



Jasmin Blanchette; Mark Summerfield
Çeviren: Ufuk Uzun

YASAL AÇIKLAMALAR

Bu belgenin, *Qt 4 ile C++ GUI Programlama* çevirisinin **telif hakkı © 2010 *Ufuk Uzun'a***, özgün İngilizce sürümünün **telif hakkı © 2008 *Trolltech ASA'e*** aittir. Bu belgeyi, Open Publication Licence lisansının 1.0 ya da daha sonraki sürümünün koşullarına bağlı kalarak kopyalayabilir, dağıtabilir ve/veya değiştirebilirsiniz. Bu Lisansın özgün kopyasını http://www.opencontent.org/openpub/ adresinde bulabilirsiniz.

BU BELGE "ÜCRETSİZ" OLARAK RUHSATLANDIĞI İÇİN, İÇERDİĞİ BİLGİLER İÇİN İLGİLİ KANUNLARIN İZİN VERDİĞİ ÖLÇÜDE HERHANGİ BİR GARANTİ VERİLMEMEKTEDİR. AKSİ YAZILI OLARAK BELİRTİLMEDİĞİ MÜDDETÇE TELİF HAKKI SAHİPLERİ VE/VEYA BAŞKA ŞAHISLAR BELGEYİ "OLDUĞU GİBİ", AŞİKAR VEYA ZIMNEN, SATILABİLİRLİĞİ VEYA HERHANGİ BİR AMACA UYGUNLUĞU DA DAHİL OLMAK ÜZERE HİÇBİR GARANTİ VERMEKSİZİN DAĞITMAKTADIRLAR. BİLGİNİN KALİTESİ İLE İLGİLİ TÜM SORUNLAR SİZE AİTTİR. HERHANGİ BİR HATALI BİLGİDEN DOLAYI DOĞABİLECEK OLAN BÜTÜN SERVİS, TAMİR VEYA DÜZELTME MASRAFLARI SİZE AİTTİR.

İLGİLİ KANUNUN İCBAR ETTİĞİ DURUMLAR VEYA YAZILI ANLAŞMA HARİCİNDE HERHANGİ BİR ŞEKİLDE TELİF HAKKI SAHİBİ VEYA YUKARIDA İZİN VERİLDİĞİ ŞEKİLDE BELGEYİ DEĞİŞTİREN VEYA YENİDEN DAĞITAN HERHANGİ BİR KİŞİ, BİLGİNİN KULLANIMI VEYA KULLANILAMAMASI (VEYA VERİ KAYBI OLUŞMASI, VERİNİN YANLIŞ HALE GELMESİ, SİZİN VEYA ÜÇÜNCÜ ŞAHISLARIN ZARARA UĞRAMASI VEYA BİLGİLERİN BAŞKA BİLGİLERLE UYUMSUZ OLMASI) YÜZÜNDEN OLUŞAN GENEL, ÖZEL, DOĞRUDAN YA DA DOLAYLI HERHANGİ BİR ZARARDAN, BÖYLE BİR TAZMİNAT TALEBİ TELİF HAKKI SAHİBİ VEYA İLGİLİ KİŞİYE BİLDİRİLMİŞ OLSA DAHİ, SORUMLU DEĞİLDİR.

Tüm telif hakları aksi özellikle belirtilmediği sürece sahibine aittir. Belge içinde geçen herhangi bir terim, bir ticari isim ya da kuruma itibar kazandırma olarak algılanmamalıdır. Bir ürün ya da markanın kullanılmış olması ona onay verildiği anlamında görülmemelidir.

ÇEVİRMENİN NOTU

Merhaba,

Bu çeviri eserin içeriğinden bahsetmeden önce, onun var olma sebebini ve serüvenimi anlatmak istiyorum: Geçtiğimiz Mayıs ayında, Sakarya Üniversitesi Bilgisayar Mühendisliği bölümünde henüz daha 1. Sınıf öğrencisiyken ve hâlâ konsol ekranda çalışan programlar yazmakla meşgulken, gerek arkadaşlarım gerekse ben grafik arayüz programlamak için can atıyorduk. Dolayısıyla hemen araştırmaya koyuldum. Hâli hazırda C/C++ ile programlar yazdığımdan, doğal olarak C/C++ kullanarak grafik arayüz programlamanın yollarını arıyordum. Biraz araştırmadan sonra Qt ile tanıştım. Qt hakkındaki kısıtlı sayıda Türkçe kaynak bile, Qt ile grafik arayüz programlamanın en iyi tercihlerden biri olduğuna beni ikna etti. Ancak demin de bahsettiğim gibi Türkçe kaynakların kısıtlı olması nedeniyle Qt'u öğrenmemin zorlayıcı bir süreç olacağını kestirebiliyordum. İngilizce kaynaklara yönelmem gerekiyordu. Ancak benim gibi daha önce İngilizce bir kaynaktan yararlanarak bir şeyler öğrenmeye kalkışmamış biri için bu iş hiçte kolay olmayacaktı. Daha sonra "C++ GUI Programming With Qt 4; 2nd Edition" kitabının e-kitap versiyonunu edindim. Kitaba şöyle bir göz gezdirince, anlaşılır ve yararlı bir kaynak olduğuna karar verdim. Kitabı okumaya başlayınca, ilk izlenimlerimde haklı olduğumu anladım. Ancak, serüvenimin en başında yaşadığım Türkçe kaynak sıkıntısını düşününce, yapmam gereken şey apaçık ortadaydı: Bu kitabı Türkçeye çevirmek. Tabi bunda kitabın Open Publication Lisence ile yayınlamış olması ve İngilizcemin çok çok iyi olmamasına rağmen benim için bile anlaşılır olması(bazı kısımlar hariç) da etkili oldu. Böylece işe koyuldum. Fakat düşündüğümden yavaş ilerliyordum. Çeviri konusundaki tecrübesizliğimin de bunda etkiliydi elbette. Yine de, konu hakkında geniş bir Türkçe kaynağın olmayışını ve açık kaynak dünyasına bir katkıda bulunabilme ihtimalini düşünmek, başladığım işi bitirmemi sağladı.

Bu kitabı (C++ GUI Programming With Qt 4; 2nd Edition) çevirmemin nedeninden biraz bahsetmiş olsam da, en önemli nedenden bahsetmedim; o da, bu kitabın şuan için piyasadaki en güncel kaynak olması (2008 yılında yayınlanmasına rağmen). Diğer taraftan, Amazon.com'dan takip ettiğim kadarıyla biri Mayıs biride Ağustos ayında olmak üzere iki kitap yolda. Umarım bu kitapların Türkçe'ye kazandırılması bu kadar gecikmez.

Kitabın içeriği hakkında ise şunları söyleyebilirim: Bu çeviri eser, bahsi geçen kitabın tamamının değil, (ufak eksiklerle birlikte) 12 bölümünün çevirisini içermektedir. Çevirdiğim bölümleri belirlerken, temel ve orta seviyede Qt programlama öğrenmek isteyenlere hitap etme amacını güttüm. Çeviriyle alâkalı ise, eksikliklerin ve yanlışlıkların olabileceğini belirtmeliyim. Ancak, bunların olabildiğince az olması için elimden geleni yaptığımı da bilmelisiniz.

Son olarak, çeviri konusunda beni cesaretlendiren babama, Salt Okunur E-Dergi'de yazdığım yazıları aksattığımda bana aşırı tepki göstermeyen dergi ekibime ve özellikle attığım mesaja karşılık, kitabını çevirmemden memnuniyet duyacağını bildirerek beni mutlu eden, kitabın yazarlarından Jasmin Blanchette'a teşekkürlerimi sunmak isterim.

Okuyan herkese faydalı olması dileğiyle,

Ufuk Uzun
İstanbul, Türkiye
Mart 2010
ufukuzun.ce@gmail.com

EDİTÖRÜN NOTU

Sevgili Okuyucu,

Çalışan bir programcı olarak, her gün Qt'u kullanırım ve Qt'un C++ programcılarına getirdiği organizasyon, tasarım ve güçten gerçekten etkilenmiş bulunuyorum.

Qt hayata bir çapraz-platform GUI araç-takımı olarak başlamasına rağmen, gündelik programlamanın her cephesi için taşınabilirliği sağlayacak şekilde genişletildi: dosyalar, işlemler, ağ ve veritabanı erişimi bunlardan bazılarıdır. Qt'un bu geniş uygulanabilirliği sayesinde, sahiden kodunuzu bir kez yazarsınız ve sadece onu başka platformlar üzerinde yeniden derleyerek, uygulamanızı o platformlarda da çalıştırabilirsiniz. Bu, müşterilerinizin, sizin yazılımlarınızı başka platformlar üzerinde kullanması icap ettiğinde fevkalade değerlidir.

Elbette, eğer bir açık kaynak geliştiricisiyseniz, açık kaynak lisansı da mevcut olan Qt'un sunduğu her olanaktan siz de faydalanabilirsiniz.

Qt kapsamlı bir çevrimiçi yardım sağlar ve bu, yöneleceğiniz ilk başvuru kaynağınız olacaktır. Örnek programlar yararlıdır, fakat tersine mühendislik ve sadece örnekleri okumak Qt'u doğru kullanabilmek için yeterli olmayacaktır. İşte bu noktada, bu kitap resmin içine giriyor.

Bu gerçekten derli toplu ve düzgün bir kitaptır. Öncelikle, Qt üzerine yazılmış resmi ve en çok bahsi geçen kitap. Ve aynı zamanda harika bir kitap: iyi organize edilmiş, iyi yazılmış ve takip edilmesi ve öğrenilmesi kolay.

Bu kitabı okurken, tıpkı benim gibi çok eğleneceğinizi ve ondan çok şey öğreneceğinizi umuyorum.

Arnold Robbins
Nof Ayalon, İsrail
Kasım 2007

ÖNSÖZ

Neden Qt? Neden programcılar bizim gibi Qt'u seçerler? Elbette cevapları açıktır: Qt'un tek kaynak(single-source) uyumluluğu, özelliklerinin zenginliği, C++ performansı, kaynak kodlarının kullanılırlığı, dokümantasyonu, kaliteli teknik desteği ve Trolltech'in parlak pazarlama gereçleri arasında bahsedilen diğer öğeler. Bunların hepsi çok iyi, fakat en önemli noktayı kaçırıyor: Qt başarılı çünkü programcılar onu seviyor.

Nasıl olurda programcılar bir teknolojiyi sever, fakat diğerini sevmez? Kişisel olarak ben, yazılım mühendislerinin iyi hissettiren teknolojilerden hoşlandıklarını, fakat öyle olmayan diğer her şeyden nefret ettiklerini düşünüyorum. Böyle olmasa başka nasıl açıklayabilirdik; en parlak programcılardan bazılarının bir video kaydediciyi programlamak için yardıma ihtiyaç duymalarını, ya da birçok mühendisin şirketlerindeki telefon sisteminin işleyişinden dolayı üzgün görünmelerini? Trolltech'te telefon sistemimiz, diğer kişinin dâhili numarasını girmeye müsaade etmeden önce bizi iki saniye boyunca '*'a basmaya zorluyor. Eğer bunu yapmayı unutur ve hemen dâhili numarayı girmeye başlarsanız bütün numarayı yeniden girmek zorunda kalıyorsunuz. Peki neden '*'? Neden '#' ya da 1 ya da 5 ya da telefonun üstündeki diğer 20 tuştan herhangi biri değil? Ve neden iki saniye, neden bir ya da üç ya da bir buçuk değil? Neden bunlardan hiçbiri değil? Bu yüzden telefonu sinir bozucu bulurum ve mümkünse kullanmaktan kaçınırım. Ben de dâhil hiç kimse gelişigüzel şeyler yapmak zorunda olmaktan hoşlanmaz, özellikle bu gelişigüzel şeyler, aynı düzeyde gelişigüzel olaylara bağlı olduğunda.

Programlama da bizim telefon sistemimizi kullanmaya oldukça benzer, yalnız daha kötüsüdür. Ve Qt'un bizi kurtardığı şey tam olaraktan budur. Qt farklıdır. Bir kere, Qt mantıklıdır. Ve başka, Qt eğlencelidir. İşimize yoğunlaşmamıza izin verir. Qt'un orijinal mimarları bir problemle karşılaştıklarında, sadece iyi bir çözüm ya da hızlı bir çözüm veyahut basit bir çözüm aramazlar. Onlar doğru çözümü arar ve sonra bu çözümü belgelerler. Evet, hatalar da yaptılar, ve evet, bazı tasarım kararları testi geçemedi, fakat bir çok doğru şey de yaptılar.

Bizim için, Qt'da çalışmak bir sorumluluk ve bir ayrıcalıktır. Ayrıca sizin profesyonelliğinizin ve açık kaynak yaşamınızın daha kolay ve daha eğlenceli olmasına yardım etmekten de gurur duyuyoruz.

Qt'u kullanmaktan memnun olmamızın bir sebebi de onun çevrimiçi dokümantasyonudur. Fakat bu dokümantasyonların odağı öncelikli olarak, karmaşık gerçek dünya yazılımlarının nasıl inşa edildiğine dair bize çok az şey gösteren, kendi bireysel sınıflarıdır. Ve bu mükemmel kitap, bu boşluğu doldurur. Qt'un size sundukların, "Qt tarzı" programlamayı ve Qt'dan daha fazlasını almayı gösterir. Kitap iyi örnekler, tavsiyeler ve açıklamalar içerir.

Günümüzde çok sayıda ticari ve ücretsiz Qt uygulaması, satın alma veya indirme için hazır. Qt ile geliştirilmiş uygulamaları görmek bize gurur veriyor ve Qt'u daha iyi yapmak için ilham kaynağımız oluyor. Ve bu kitabın yardımıyla, her zamankinden daha yüksek kaliteli Qt uygulamaları geliştirileceğinden de eminim.

Matthias Ettrich
Berlin, Almanya
Kasım 2007

İÇİNDEKİLER

BÖLÜM 1: BAŞLARKEN 9

Merhaba Qt 9

Bağlantılar Kurma 11

Parçacıkları Yerleştirme 11

BÖLÜM 2: DİYALOGLAR OLUŞTURMA 15

QDialog Altsınıfı Türetme 15

Derinlemesine: Sinyaller ve Yuvalar 20

Hızlı Diyalog Tasarımı 22

Şekil Değiştiren Diyaloglar 28

Dinamik Diyaloglar 33

Yerleşik Parçacık ve Diyalog Sınıfları 34

BÖLÜM 3: ANA PENCERELER OLUŞTURMA 39

QMainWindow Altsınıfı Türetme 39

Menüler ve Araç Çubukları Oluşturma 43

Durum Çubuğunu Ayarlama 47

File Menüsünü Gerçekleştirme 49

Diyalogları Kullanma 54

Uygulama Ayarlarını Saklama 59

Çoklu Doküman İşleme 60

Açılış Ekranları 63

BÖLÜM 4: UYGULAMA İŞLEVLERİNİ GERÇEKLEŞTİRME 65

Merkez Parçacık 65

QTableWidget Altsınıfı Türetme 66

Yükleme ve Kaydetme 71

Edit Menüsünü Gerçekleştirme 73

Diğer Menüleri Gerçekleştirme 77

QTableWidgetItem Altsınıfı Türetme 80

BÖLÜM 5: ÖZEL PARÇACIKLAR OLUŞTURMA 88

Qt Parçacıklarını Özelleştirme 88

QWidget Altsınıfı Türetme 90

Özel Parçacıkları Qt Designer’a Entegre Etme 98

BÖLÜM 6: YERLEŞİM YÖNETİMİ 102

Parçacıkları Bir Form Üzerine Yerleştirme 102

Yığılı Yerleşimler ... 107
Bölücüler ... 108
Kaydırılabilir Alanlar ... 111
Yapışkan Pencereler ve Araç Çubukları ... 112
Çoklu Doküman Arayüzü ... 115
BÖLÜM 7: OLAY İŞLEME ... 123
Olay İşleyicilerini Uyarlama ... 123
Olay Filtreleri Kurma ... 128
Yoğun İşlem Sırasında Duyarlı Kalma ... 130
BÖLÜM 8: SÜRÜKLE-BIRAK ... 133
Sürükle-Bırakı Etkinleştirme ... 133
Özel Sürükleme Tiplerini Destekleme ... 137
Pano İşleme ... 142
BÖLÜM 9: ÖĞE GÖRÜNTÜLEME SINIFLARI ... 143
Öğe Görüntüleme Uygunluk Sınıflarını Kullanma ... 144
Önceden Tanımlanmış Modelleri Kullanma ... 150
Özel Modeller Gerçekleştirme ... 155
Özel Delegeler Gerçekleştirme ... 167
BÖLÜM 10: KONTEYNER SINIFLARI ... 172
Ardışık Konteynerler ... 172
Birleşik Konteynerler ... 179
Genel Algoritmalar ... 181
Karakter Katarları, Byte Dizileri ve Variantlar ... 183
BÖLÜM 11: GİRDİ/ÇIKTI ... 190
İkili Veri Okuma ve Yazma ... 190
Metin Okuma ve Yazma ... 195
Dizinler ... 200
Gömülü Kaynaklar ... 201
Süreçler Arası İletişim ... 201
BÖLÜM 12: VERİTABANLARI ... 207
Bağlanma ve Sorgulama ... 207
Tabloları Görüntüleme ... 213
Formları Kullanarak Kayıtları Düzenleme ... 215
Veriyi Tablo Biçiminde Formlar İçinde Sunma ... 218

BÖLÜM 1: BAŞLARKEN

Bu bölüm temel C++ ve Qt'un sağladığı işlevselliği birleştirerek birkaç küçük grafik kullanıcı arayüzü(GUI) uygulaması oluşturmayı gösterir. Bu bölüm ayrıca Qt'un en önemli iki fikri ile tanıştırır: sinyal/yuva mekanizması(signal/slot mechanism) ve yerleşimler(layouts).

Bölüm 2'ye geldiğimizde ise daha derinlere ineceğiz ve Bölüm 3'te gerçeğe daha uygun bir uygulama geliştirmeye başlayacağız.

Merhaba Qt

Çok basit bir Qt programı ile başlayalım. Önce onu satır satır inceleyeceğiz ve sonra nasıl derleyip çalıştıracağımızı öğreneceğiz.

```
1 #include <QApplication>
2 #include <QLabel>

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QLabel *label = new QLabel("Hello Qt!");
7     label->show();
8     return app.exec();
9 }
```

Satır 1 ve 2 `QApplication` ve `QLabel` sınıflarının tanımlarını programa dâhil eder. Her Qt sınıfı için, o sınıfın tanımını içeren ve onunla aynı isimde olan bir başlık dosyası vardır.

Satır 5 uygulama kaynaklarını yönetmek için bir `QApplication` nesnesi oluşturur. `QApplication`'un kurucusu `argc` ve `argv` argümanlarını ister, çünkü Qt kendine ait birkaç komut-satırı argümanını (command-line argument) destekler.

Satır 6 "Hello Qt!"u görüntüleyen bir `QLabel` parçacığı(widget) oluşturur. Qt ve Unix terminolojisinde, bir parçacık kullanıcı arayüzünde görsel bir öğedir. Bu terimin kaynağı "window gadget"tir ve bu terim Windows terminolojisindeki "control" ve "container" ile aynı anlamdadır. Butonlar(button), menüler(menu), kaydırma çubukları(scroll bar) ve çerçeveler(frame) parçacığa birer örnektirler. Parçacıklar başka parçacıklar içerebilirler; örneğin, bir uygulama penceresi genellikle bir `QMenuBar`, birkaç `QToolBar`, bir `QStatusBar`, ve bazı diğer parçacıkları içeren bir parçacıktır. Çoğu uygulama, uygulama penceresi olarak bir `QMainWindow` veya bir `QDialog` kullanır, fakat Qt oldukça esnektir, böylece herhangi bir parçacık pencere olarak kullanılabilir. Örneğimizde, `QLabel` parçacığı uygulama penceremizdir.

Satır 7 etiketi(label) görünür yapar. Parçacıklar her zaman gizlenmiş olarak oluşturulurlar, böylece onları göstermeden önce isteğimize göre uyarlayabiliriz.

Satır 8'de uygulamanın kontrolü Qt'a geçer. Bu noktada, program olay döngüsüne(event loop) girer. Bu, programın bir kullanıcı eylemi için beklediği (fare tıkladığında veya bir tuşa basmak gibi) bir tür

bekleme(stand-by) modudur. Kullanıcı eylemleri, programın cevaplayabildiği olayları("mesajlar" diye de geçer) üretir ve bunlar genellikle bir ya da daha fazla fonksiyonun yürütülmesidir. Örneğin, kullanıcı bir parçacığı tıkladığında, bir "fare butonuna basılma(mouse press)" ve bir "fare serbest(mouse release)" olayları üretilir. Bu açıdan GUI uygulamaları, genellikle girdi işleyen, sonuçlar üreten ve bir insan müdahalesi olmadan sonlandırılan geleneksel toplu iş programlarından(batch programs) büyük ölçüde farklıdırlar.

Sadelik olsun diye, `main()` fonksiyonunun sonunda `QLabel` nesnesi için `delete`'i çağırmadık. Bu bellek sızıntısı(memory leak) bunun gibi küçük bir program söz konusu olduğunda zararsızdır. Program sonlandırıldığında `QLabel` nesnesi için ayrılmış olan bellek işletim sistemi tarafından geri alınacaktır.

Artık programı kendi makinenizde deneyebilirsiniz. Şekil 1.1'deki gibi görünmelidir. Ondan önce, eğer makinenizde Qt yüklü değilse, Qt 4.3.2 veya daha yeni bir sürümünü bilgisayarınıza yüklemelisiniz. (İşletim sisteminize uygun bir Qt 4 kopyasını http://qt-project.org/ adresinden temin edebilirsiniz.) Bundan sonrası için, Qt 4'ün bir kopyasını doğru olarak makinenize yüklediğinizi varsayacağım. Ayrıca program kodlarının içinde olduğu `hello.cpp` adlı dosyanın `hello` adlı bir dizinin altında olması gerekir.

Şekil 1.1

Bir komut isteminden(command prompt) dizini `hello`'ya ayarlayın ve platformdan bağımsız proje dosyasını(`hello.pro`) oluşturmak için aşağıdakini yazın:

```
qmake -project
```

Proje dosyasından, platforma özgü bir makefile oluşturmak için aşağıdakini yazın:

```
qmake hello.pro
```

Şimdi, programı derlemek için `make`, ardından da programı çalıştırmak için `hello` yazın.

Bir sonraki örneğimize geçmeden önce biraz eğlenelim:

Şu satırı

```
QLabel *label = new QLabel("Merhaba Qt!");
```

şu kod parçası ile değiştirelim

```
QLabel *label = new QLabel("<h2><i>Hello </i> "
                           "<font color=red>Qt!</font></h2>");
```

ve yine aynı adımları izleyerek programı derleyip çalıştıralım. Uygulamamız çalıştığında Şekil 1.2'deki gibi görünecektir. Bu örneğin de gösterdiği gibi, bir Qt uygulamasının kullanıcı arayüzüne daha hoş bir hava katmak, basit HTML etiketleri(tag) kullanarak bile mümkün.

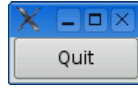
Şekil 1.2

Bağlantılar Kurma

İkinci örnek kullanıcı eylemlerine nasıl yanıt verildiğini gösterir. Uygulama kullanıcının tıklayıp programdan çıkabileceği bir butondan oluşuyor. Uygulamanın kaynak kodu, bir önceki örneğinki ile -ana parçacığımız `QLabel` yerine `QPushButton` kullanmamız ve bunu bir kullanıcı eylemine(butona tıklama) bağlamamız dışında- çok bezerdir.

Uygulamanın çalışan hali Şekil 1.3'de gösteriliyor. İşte kaynak kodu:

```
 1 #include <QApplication>
 2 #include <QPushButton>

 3 int main(int argc, char *argv[])
 4 {
 5     QApplication app(argc, argv);
 6     QPushButton *button = new QPushButton("Quit");
 7     QObject::connect(button, SIGNAL(clicked()),
 8                      &app, SLOT(quit()));
 9     button->show();
10     return app.exec();
11 }
```



Şekil 1.3

Qt parçacıkları bir kullanıcı eylemini ya da ortaya çıkan bir durum değişikliğini işaret etmek için sinyaller(signal) yayar.[*] Örneğin, `QPushButton` kullanıcı butona tıkladığında bir `clicked()` sinyali yayar. Bir sinyal bir yuvaya bağlanmış olabilir, böylece bir sinyal yayıldığında, yuva otomatik olarak yürütülür. Bizim örneğimizde, butonun `clicked()` sinyalini `QApplication` nesnesinin `quit()` yuvasına bağladık. Buradaki `SIGNAL()` ve `SLOT()` makroları sözdiziminin birer parçalarıdır.

[*] *Qt sinyalleri Unix sinyalleriyle bağlantısızdır. Bu kitapta, biz yalnızca Qt sinyalleriyle ilgileneceğiz.*

Bir önceki örnekte izlediğimiz adımları izleyerek kodu derleyip çalıştırdığınızda, eğer "Quit" butonuna tıklarsanız uygulamanın sona erdirileceğini göreceksiniz.

Parçacıkları Yerleştirme

Bölüm 1'in bu kısmında bir penceredeki parçacıkların geometrisini, yerleşimleri(layouts) kullanarak nasıl yöneteceğimizi ve sinyalleri ve yuvaları kullanarak iki parçacığı nasıl senkronize edebileceğimizi gösteren küçük bir uygulama oluşturacağız. Uygulama kullanıcının yaşını sorar. Kullanıcı, yaşını bir döndürme kutusu(spin box) ya da bir kaydırıcı(slider) yardımıyla girebilir.

Şekil 1.4

Uygulama 3 parçacıktan meydana gelir: bir `QSpinBox`, bir `QSlider`, ve bir `QWidget`. `QWidget` uygulamanın ana penceresidir. `QSpinBox` ve `QSlider` `QWidget` içinde yorumlanır; onlar `QWidget`'ın çocuklarıdırlar(children). Bir başka deyişle, `QWidget` `QSpinBox` ve `QSlider`'ın ebeveynidir(parent)

diyebiliriz. QWidget ise ebeveyne sahip değildir çünkü zaten kendisi bir üst-seviye pencere olarak kullanılmaktadır. QWidget ve onun tüm altsınıfları için kurucular ebeveyn parçacığı(parent widget) belirten bir QWidget * parametresi alır.

İşte kaynak kodu:

Kod Görünümü:

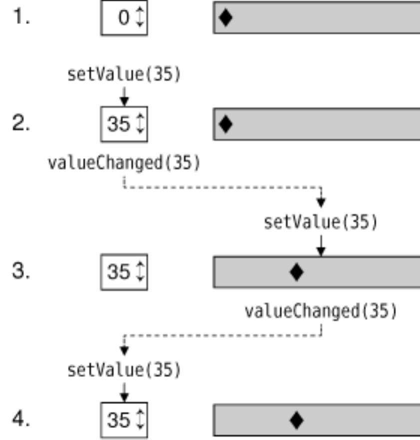
```
 1 #include <QApplication>
 2 #include <QHBoxLayout>
 3 #include <QSlider>
 4 #include <QSpinBox>

 5 int main(int argc, char *argv[])
 6 {
 7     QApplication app(argc, argv);

 8     QWidget *window = new QWidget;
 9     window->setWindowTitle("Enter Your Age");

10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12     spinBox->setRange(0, 130);
13     slider->setRange(0, 130);

14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                      slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                      spinBox, SLOT(setValue(int)));
18     spinBox->setValue(35);

19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22     window->setLayout(layout);

23     window->show();

24     return app.exec();
25 }
```

Satır 8 ve 9 QWidget'ı uygulamanın ana penceresi olarak hizmet verecek şekilde ayarlar. Pencerenin başlık çubuğunda görüntülenen metni ayarlamak için setWindowTitle() fonksiyonunu çağırırız.

Satır 10 ve 11 bir QSpinBox ve bir QSlider oluşturur, ve satır 12 ve 13 onların geçerli değişim aralıklarını ayarlar. Bir kullanıcının en fazla 130 yaşında olabileceğini rahatlıkla varsayabiliriz. Burada window'u QSpinBox ve QSlider'ın kurucularına, bu parçacığın(window) onların(QSpinBox ve QSlider) ebeveyni olduğunu belirtmek için aktarabilirdik, fakat burada gereksizdir çünkü yerleşim sistemi(layout system) bunu bizzat kendisi düşünür ve döndürme kutusunun ve kaydırıcının ebeveynini otomatik olarak ayarlar.

Satır 14 ve 16'da görünen iki QObject::connect() çağrısı döndürme kutusunu ve kaydırıcıyı senkronize ederek her zaman aynı değeri göstermelerini sağlar. Bir parçacığın değeri değiştiğinde, onun valueChanged(int) sinyali yayılır ve diğer parçacığın setValue(int) yuvası yeni değer ile çağrılır.

Satır 18 döndürme kutusunun değerini 35'e ayarlar. Bu olduğunda, QSpinBox 35 değerinde bir tamsayı argümanı ile valueChanged(int) sinyalini yayar. Bu argüman, kaydırıcının değerini 35'e ayarlayan QSlider'ın setValue(int) yuvasına aktarılır. Sonra kaydırıcı valueChanged(int) sinyalini yayar çünkü kendi değeri değişmiştir, bu da döndürme kutusunun setValue(int) yuvasını tetikler. Fakat bu noktada, setValue(int) hiçbir sinyal yaymaz çünkü zaten değeri 35'tir. Bu, sonsuz özyinelemeyi önler. Şekil 1.5 bu durumu özetliyor.

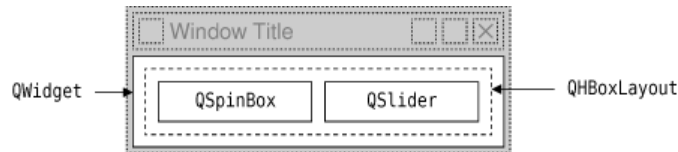


Şekil 1.5

Satır 19, 20, 21 ve 22'de, döndürme kutusu ve kaydırıcı parçacıklarını yerleşim yöneticisini(layout manager) kullanarak yerleştiririz. Bir yerleşim yöneticisi parçacıkların boyutunu ve konumunu ayarlayan bir nesnedir. Qt üç ana yerleşim yöneticisi sınıfına sahiptir:

- QHBoxLayout, parçacıkları dikey(horizontal) olarak soldan sağa doğru yerleştirir(bazı kültürlerde sağdan sola).
- QVBoxLayout, parçacıkları düşey(vertical) olarak en üstten aşağı doğru yerleştirir.
- QGridLayout, parçacıkları bir ızgaraya(grid) yerleştirir.

Satır 22'deki QWidget::setLayout() çağrısı yerleşim yöneticisini pencereye yükler. Perdenin arkasında, QSpinBox ve QSlider yerleşimin yüklendiği parçacığın çocukları olarak ayarlanırlar ve bu sebeple yerleşimin içine konacak bir parçacığı oluştururken açıkça bir ebeveyn belirtmemiz gerekmez.



Şekil 1.6

Hiçbir parçacığın boyutunu ya da konumunu açıkça ayarlamadığımız halde, QSpinBox ve QSlider yan yana hoş bir şekilde yerleşmiş görünüyor. Bunun nedeni, QHBoxLayout'un parçacıklara makul ölçüleri aşmayacak şekilde, otomatik olarak boyut ve konum tahsis etmesidir. Yerleşim yöneticileri bizi uygulamalarımızın ekran konumlarını kodlama angaryasından kurtarır ve pencerenin düzgünce yeniden boyutlandırılmasını sağlar.

Qt'un kullanıcı arayüzleri yaratma yaklaşımı anlaması basit ve çok esnektir. Qt programcılarının en çok kullandıkları ortak modele göre; o an ihtiyaç duyulan parçacıklar oluşturulur ve sonra eğer gerekirse özellikleri yine ihtiyaçlar doğrultusunda ayarlanır. Daha sonra parçacıklar programcılar tarafından, boyutlarının ve konumlarının otomatik olarak ayarlanacağı yerleşimlere eklenirler. Kullanıcı arayüzü davranışı ise Qt'un sinyal/yuva mekanizması kullanılarak bağlanan parçacıklar tarafından yönetilir.

BÖLÜM 2: DİYALOGLAR OLUŞTURMA

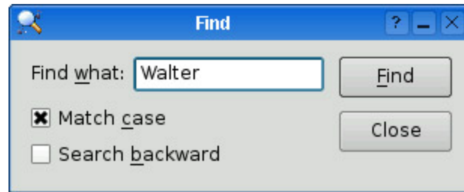
Bu bölüm, Qt'u kullanarak iletişim kutularının(dialog boxes) nasıl oluşturulduğunu öğretecek. İletişim kutuları kullanıcılara seçenekler sunar, kullanıcılara istedikleri değerleri verme ve kendi seçimlerini yapma izni verir. İletişim kutuları veya daha yaygın kullanılan ismiyle "diyaloglar", bir anlamda kullanıcılar ile uygulamaların birbirleriyle "konuşmalarını" sağlar.

Pek çok GUI uygulaması ana pencereyle birlikte bir menü çubuğu(menu bar), bir araç çubuğu(toolbar) ve düzinelerce diyalogdan meydana gelir. Kullanıcının seçimlerine direkt cevap veren diyalog uygulamaları oluşturmak da mümkündür(hesap makinesi uygulaması gibi).

İlk diyaloğumuzu, nasıl yapıldığını göstermek için, yalnızca kod yazarak oluşturacağız. Daha sonra, diyalogların Qt'un görsel tasarım aracı Qt Designer ile nasıl oluşturulduğunu göreceğiz. Qt Designer'ı kullanmak elle kod yazmaktan çok daha hızlıdır ve farklı tasarımları test etmeyi ve daha sonra tasarımları değiştirmeyi kolaylaştırır.

QDialog Altsınıfı Türetme

İlk örneğimiz "Find", tamamen C++ ile yazılmış bir diyalogdur. Şekil 2.1'de görebilirsiniz. Diyaloğu, uygun bir sınıf oluşturarak gerçekleştireceğiz. Böyle yaparak, onu kendi sinyal ve yuvalarıyla bağımsız bir bileşen yapacağız.



Şekil 2.1

Kaynak kodu iki dosyaya yayılır: `finddialog.h` ve `finddialog.cpp`. `finddialog.h` ile başlayacağız.

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H

3 #include <QDialog>

4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;
```

Satır 1 ve 2 (ve 27) başlık dosyasını birden fazla dâhil etmeye karşı korur.

Satır 3, Qt'da diyaloglar için temel sınıf olan `QDialog`'un tanımını dâhil eder. `Qdialog`, `QWidget`'tan türetilmiştir.

Satır 4, 5, 6 ve 7, diyaloğu gerçekleştirirken kullanacağımız Qt sınıflarının ön bildirimlerini yapar. Bir ön bildirim, C++ derleyicisini bir sınıfın varlığından -sınıf tanımının tüm detaylarını vermeden(sınıf tanımları genellikle kendi başlık dosyasında yer alır)- haberdar eder.

Sonra, `QDialog`'un bir altsınıfı olarak `FindDialog`'u tanımlarız:

```
 8 class FindDialog : public QDialog
 9 {
10     Q_OBJECT

11 public:
12     FindDialog(QWidget *parent = 0);
```

Sınıf tanımının başlangıcındaki `Q_OBJECT` makrosu, sinyaller veya yuvalar tanımlanan tüm sınıflarda gereklidir.

`FindDialog`'un kurucusu Qt parçacık sınıflarının tipik bir örneğidir. `parent` parametresi ebeveyn parçacığı belirtir. Varsayılan değeri, bu diyaloğun ebeveyne sahip olmadığı anlamına gelen boş(null) bir işaretçidir.

```
13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
15     void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

`signals` kısmında, kullanıcı "Find" butonuna tıkladığında diyaloğun yayacağı 2 sinyalin bildirimi yapılır. Eğer "Search backward" seçeneği seçilmişse, diyalog `findPrevious()` sinyalini, aksi halde `findNext()` sinyalini yayar.

`signals` aslında bir makrodur. C++ önişlemcisi, derleyici onu görmeden önce, onu standart C++'a dönüştürür. `Qt::CaseSensitivity` ise, `Qt::Sensitive` ve `Qt::Insensitive` değerlerini alabilen bir enum tipidir.

```
16 private slots:
17     void findClicked();
18     void enableFindButton(const QString &text);

19 private:
20     QLabel *label;
21     QLineEdit *lineEdit;
22     QCheckBox *caseCheckBox;
23     QCheckBox *backwardCheckBox;
24     QPushButton *findButton;
25     QPushButton *closeButton;
26 };

27 #endif
```

Sınıfın private(özel) kısmında 2 yuva bildirimi yaparız. Yuvaları gerçekleştirmek için, diyaloğun birçok çocuk parçacığına(child widget) erişmemiz gerekecek, bu nedenle de onları işaretçiler olarak tutarız. `slots` anahtar kelimesi de `signals` gibi bir makrodur.

private değişkenler için, sınıflarının ön bildirimlerini kullandık. Bu mümkündü çünkü hepsi işaretçi ve onlara başlık dosyası içinde erişmeyeceğiz, bu nedenle derleyicinin sınıf tanımlarının tamamına ihtiyacı yoktur.

Uygun başlık dosyalarını(<QCheckBox>, <QLabel>, vs.) dâhil edebilirdik, fakat ön bildirimleri kullanmak bir miktar daha hızlı derlenme sağlar.

Şimdi FindDialog sınıfının gerçekleştirimini içeren finddialog.cpp'ye bakacağız.

```
 1 #include <QtGui>

 2 #include "finddialog.h"
```

İlk olarak, Qt'un GUI sınıflarının tanımını içeren başlık dosyasını, yani <QtGui>'yi dâhil ederiz. Qt, her biri kendi kütüphanesinde bulunan değişik modüllerden meydana gelir. En önemli modüller QtCore, QtGui, QtNetwork, QtOpenGL, QtScript, QtSql, QtSv ve QtXml'dir. <QtGui> başlık dosyası, QtCore ve QtGui modüllerinin parçası olan tüm sınıfların tanımından meydana gelir. Bu başlık dosyasını dâhil etmek bizi her sınıfı tek tek dâhil etme zahmetinden kurtarır.

finddialog.h içine <QDialog>'u dâhil etme ve QCheckBox, QLabel, QLineEdit ve QPushButton'ın ön bildirimlerini kullanmak yerine, basitçe <QtGui>'yi dâhil edebilirdik. Ancak genellikle, böylesine büyük bir başlık dosyasını dâhil etmek kötü bir tarzdır, özelliklede daha büyük uygulamalarda.

```
 3 FindDialog::FindDialog(QWidget *parent)
 4     : QDialog(parent)
 5 {
 6     label = new QLabel(tr("Find &what:"));
 7     lineEdit = new QLineEdit;
 8     label->setBuddy(lineEdit);

 9     caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));

11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);

14     closeButton = new QPushButton(tr("Close"));
```

Satır 4'de temel sınıfın kurucusuna parent parametresini aktarırız. Sonra, çocuk parçacıkları oluştururuz. Karakter katarı deyimleriyle yapılan tr() fonksiyonu çağrıları, karakter katarı deyimlerini diğer dillere çevirmek için işaretler. Bu fonksiyon, QObject içinde ve Q_OBJECT makrosunu içeren her altsınıfta bildirilmiştir. Uygulamalarınızı diğer dillere çevirmek için acil bir planınız olmasa bile, kullanıcıya görünen karakter katarı deyimlerini tr() ile çevrelemek iyi bir alışkanlıktır.

Karakter katarı deyimlerinde, kısayol tuşları belirtmek için '&'ı kullanırız. Örneğin, Satır 11 kullanıcının -kısayolları destekleyen platformlarda- Alt+F tuş kombinasyonunu kullanarak aktif hale getirebileceği bir "Find" butonu oluşturur. '&', odağı kontrol etmek için de kullanılabilir: Satır 6'da kısayol tuşuyla(Alt+W) birlikte bir etiket oluştururuz ve Satır 8'de de etiketin arkadaşını(label's buddy) satır editörü(line editor) olarak ayarlarız. Arkadaş(buddy), etiketin kısayol tuşuna basıldığında odağı üstlenecek olan parçacıktır. Böylece, kullanıcı Alt+W tuş kombinasyonuna bastığında, odak satır editörüne (etiketin arkadaşı) kayar.

Satır 12'de setDefault(true)'yu çağırarak "Find" butonunu diyaloğun varsayılan butonu(default button) yaparız. Varsayılan buton, kullanıcı "Enter"a bastığında etkilenecek butondur. Satır 13'de "Find" butonunu devredışı bırakırız. Bir parçacık devredışı bırakıldığında, genelde gri gösterilir ve kullanıcı etkileşimine yanıt vermez.

```
15    connect(lineEdit, SIGNAL(textChanged(const QString &)),
16            this, SLOT(enableFindButton(const QString &)));
17    connect(findButton, SIGNAL(clicked()),
18            this, SLOT(findClicked()));
19    connect(closeButton, SIGNAL(clicked()),
20            this, SLOT(close()));
```

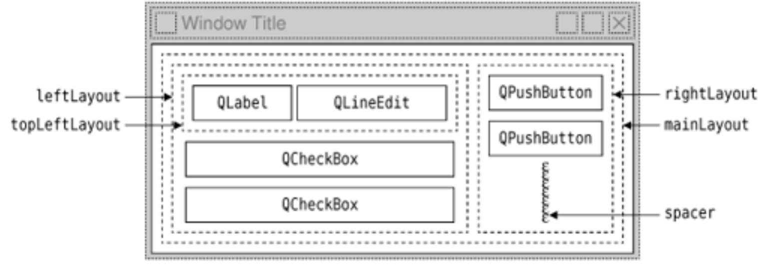
private yuva `enableFindButton(const QString &)`, satır editöründeki metin değiştiğinde, diğer private yuva `findClicked()`, kullanıcı "Find" butonuna tıkladığında çağrılır. Kullanıcı "Close"a tıkladığında da diyalog kendini kapatır. `close()` yuvası `QWidget`'ten miras alınır ve varsayılan davranışı parçacığı -onu silmeden- gizlemektir. `enableFindButton()` ve `findClicked()` yuvalarının kodlarına daha sonra bakacağız.

`QObject`, `FindDialog`'un atalarından biri olduğu için, `connect()` çağrılarının önüne `QObject::` önekini koymayı ihmal edebiliriz.

```
21     QHBoxLayout *topLeftLayout = new QHBoxLayout;
22     topLeftLayout->addWidget(label);
23     topLeftLayout->addWidget(lineEdit);

24     QVBoxLayout *leftLayout = new QVBoxLayout;
25     leftLayout->addLayout(topLeftLayout);
26     leftLayout->addWidget(caseCheckBox);
27     leftLayout->addWidget(backwardCheckBox);

28     QVBoxLayout *rightLayout = new QVBoxLayout;
29     rightLayout->addWidget(findButton);
30     rightLayout->addWidget(closeButton);
31     rightLayout->addStretch();

32     QHBoxLayout *mainLayout = new QHBoxLayout;
33     mainLayout->addLayout(leftLayout);
34     mainLayout->addLayout(rightLayout);
35     setLayout(mainLayout);
```

Sonra, yerleşim yöneticilerini(layout managers) kullanarak çocuk parçacıkları yerleştiririz. Yerleşimler (Layouts), parçacıkları ve diğer yerleşimleri içerebilirler. `QHBoxLayout`'ların, `QVBoxLayout`'ların ve `QGridLayout`'ların çeşitli kombinasyonlarını iç içe koyarak çok karmaşık diyaloglar oluşturmak mümkündür.

"Find" diyaloğu için, Şekil 2.2'de de görüldüğü gibi iki `QHBoxLayout` ve iki `QVBoxLayout` kullanırız. Dıştaki yerleşim, satır 35'te `FindDialog`'a yüklenen ve diyaloğun bütün alanından sorumlu olan ana yerleşimdir(mainLayout). Diğer üç yerleşim alt-yerleşimlerdir(leftLayout, rightLayout, topLeftLayout). Şekil 2.2'nin sağ altındaki küçük yay bir ara parçadır(spacer/stretch item). "Find" ve "Close" butonlarının altındaki anlamsız boşluğu doldurur ve butonların, yerleşimlerinin üstüne oturmasını sağlar.

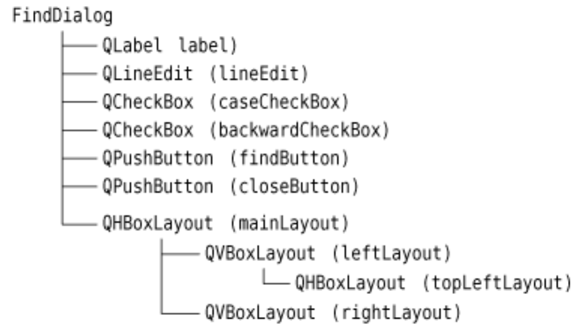


Şekil 2.2

Yerleşim yöneticisi sınıfları parçacık değildir. `QObject`'ten türetilmiş olan `QLayout`'tan türetilmişlerdir. Çalışan bir uygulamada, yerleşimler görünmezdir.

Alt-yerleşimler ebeveyn yerleşime eklendiklerinde(satır 25, 33 ve 34) otomatik olarak ebeveyne sahip olmuş olurlar. Sonra, ana-yerleşim diyaloğa yüklendiğinde(satır 35), ana-yerleşim de diyaloğun bir çocuğu olur ve böylece yerleşimlerdeki tüm parçacıklar ve yerleşimler diyaloğun çocukları olmuş olurlar. Bu ebeveyn-çocuk hiyerarşisi Şekil 2.3'te gösterilmiştir.

```
36     setWindowTitle(tr("Find"));
37     setFixedHeight(sizeHint().height());
38 }
```



Şekil 2.3

Son olarak, diyaloğun başlık çubuğunda görüntülenecek olan pencere başlığını ve değişmez(fixed) bir pencere yüksekliği ayarlarız. `QWidget::sizeHint()` fonksiyonu ise bir parçacığın ideal boyutunu döndürür.

Böylelikle `FindDialog`'un kurucusunun kritiğini bitirmiş olduk. Diyaloğun parçacıklarını ve yerleşimlerini `new` kullanarak oluşturduğumuz için, oluşturduğumuz her parçacık ve yerleşim için, `delete` diye adlandırılan bir yokedici(destructor) yazmamız gerekiyormuş gibi görülebilir. Fakat bu gereksizdir, çünkü ebeveyn yok edildiğinde, çocuk nesneler de Qt tarafından otomatik olarak silinirler.

Şimdi diyaloğun yuvalarına bakacağız:

```
39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43             caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                       : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
```

```
47      } else {
48          emit findNext(text, cs);
49      }
50 }

51 void FindDialog::enableFindButton(const QString &text)
52 {
53      findButton->setEnabled(!text.isEmpty());
54 }
```

findClicked() yuvası, kullanıcı "Find" butonuna tıkladığında çağrılır. "Search backward" seçeneğine bağlı olarak findPrevious() veya findNext() sinyalini yayar. emit anahtar kelimesi Qt'a özgüdür; diğer Qt genişletmeleri(extensions) gibi C++ önişlemcisi tarafından standart C++'a çevrilir.

enableFindButton() yuvası, satır editöründeki metin değiştiğinde çağrılır. Eğer satır editöründe metin varsa "Find" butonunu etkinleştirir, aksi halde devredışı bırakır.

Bu iki yuvayla diyaloğu tamamlamış olduk. Artık FindDialog parçacığını test etmek için main.cpp dosyasını oluşturabiliriz:

```
1 #include <QApplication>

2 #include "finddialog.h"

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }
```

Programı derlemek için her zamanki gibi qmake'i çalıştırın. Sonra programı çalıştırın. Eğer platformunuzda kısayol tuşları görünüyorsa, kısayolları kullanmayı deneyin. Parçacıklar arasında gezinmek için "Tab" tuşuna basın. Varsayılan tab sırası(tab order) parçacıklar oluşturulduğunda otomatik olarak düzenlenir. QWidget::setTabOrder()'i kullanarak tab sırasını değiştirebilirsiniz.

Derinlemesine: Sinyaller ve Yuvalar

Sinyal/yuva mekanizması Qt programlamanın temelidir. Uygulama programcılarına, birbiri hakkında hiçbir şey bilmeyen nesneleri birbirine bağlama olanağı verir. Biz şimdiden bazı sinyal ve yuvaları birbirine bağladık, kendi sinyal ve yuvalarımızın bildirimini yaptık, kendi yuvalarımızı gerçekleştirdik ve kendi sinyallerimizi yaydık. Şimdide bu mekanizmaya daha yakından bakalım.

Yuvalar, tipik C++ üye fonksiyonlarıyla hemen hemen özdeştirler. Sanal olabilirler, aşırı yüklenebilirler, public, protected ya da private olabilirler, diğer C++ üye fonksiyonları gibi direkt olarak çağrılabilirler ve parametreleri her tipten olabilir. Farklı olarak bir yuva bir sinyale bağlanabilir.

connect() ifadesi şu şekildedir:

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

Burada, `sender` ve `receiver` `QObject` işaretçileri, `signal` ve `slot` parametre ismi olmaksızın fonksiyon adlarıdır. `SIGNAL()` ve `SLOT()` makrolar ise esasen argümanlarını bir karakter katarına dönüştürürler.

Şimdiye kadar gördüğümüz örneklerde hep farklı sinyalleri farklı yuvalara bağladık. Başka olasılıklar da vardır:

- **Bir sinyal birden çok yuvaya bağlanmış olabilir:**

  ```
  connect(slider, SIGNAL(valueChanged(int)),
          spinBox, SLOT(setValue(int)));
  connect(slider, SIGNAL(valueChanged(int)),
          this, SLOT(updateStatusBarIndicator(int)));
  ```

  Sinyal yayıldığında, -özellikle belirtilmiş bir sıra yokken- yuvalar birbiri ardına çağrılırlar.

- **Birden çok sinyal aynı yuvaya bağlanmış olabilir:**

  ```
  connect(lcd, SIGNAL(overflow()),
          this, SLOT(handleMathError()));
  connect(calculator, SIGNAL(divisionByZero()),
          this, SLOT(handleMathError()));
  ```

  Sinyallerden herhangi biri yayıldığında yuva çağrılır.

- **Bir sinyal başka bir sinyale bağlanmış olabilir:**

  ```
  connect(lineEdit, SIGNAL(textChanged(const QString &)),
          this, SIGNAL(updateRecord(const QString &)));
  ```

  İlk sinyal yayıldığında ikinci sinyal de yayılır. Diğer taraftan, sinyal-sinyal bağlantıları sinyal-yuva bağlantılarından farksızdırlar.

- **Bağlantılar ortadan kaldırılabilir:**

  ```
  disconnect(lcd, SIGNAL(overflow()),
             this, SLOT(handleMathError()));
  ```

  Buna çok nadir ihtiyaç duyulur. Çünkü Qt, bir nesne silindiğinde, onunla ilişkili tüm bağlantıları otomatik olarak ortadan kaldırır.

Bir sinyali bir yuvaya(ya da başka bir sinyale) başarılı bir şekilde bağlamak istiyorsak, ikisi de aynı parametre tiplerine aynı sırada sahip olmalıdırlar.

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

İstisna olarak, bir sinyal bağlanacağı yuvadan daha fazla parametreye sahipse, fazladan parametreler önemsenmezler.

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

Eğer parametre tipleri uyuşmuyorsa, ya da sinyal veya yuva mevcut değilse, ve eğer uygulama hata ayıklama(debug) modunda inşa edilmişse, Qt çalışma sırasında uyarı verecektir. Benzer şekilde, eğer parametre isimleri yazılmışsa, Qt yine uyarı verecektir.

Şimdiye kadar, sinyal ve yuvaları parçacıklarla kullandık. Fakat mekanizmanın kendisi `QObject` içinde gerçekleştirilmiştir ve GUI programlamayla sınırlı değildir. Mekanizma her `QObject` altsınıfı tarafından kullanılabilir:

```
class Employee : public QObject
{
    Q_OBJECT

public:
    Employee() { mySalary = 0; }

    int salary() const { return mySalary; }

public slots:
    void setSalary(int newSalary);

signals:
    void salaryChanged(int newSalary);

private:
    int mySalary;
};

void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

`setSalary()` yuvasının nasıl gerçekleştirildiğine dikkat edin. `salaryChanged()` sinyalini sadece `newSalary != mySalary` olduğunda yaydık. Bu, çevrimsel bağlantıların sonsuz döngüler halini almamasını sağlar.

Hızlı Diyalog Tasarımı

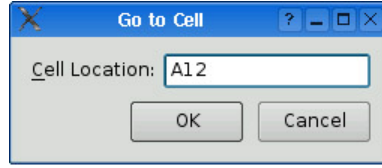
Qt elle kod yazmayı keyifli ve sezgisel yapacak şekilde tasarlanmıştır ve bunu, yalnızca C++ kodları yazarak Qt uygulamaları geliştiren programcılar iyi bilirler. Hâlâ, pek çok programcı formlar tasarlarken görsel bir yaklaşım kullanmayı tercih ederler, çünkü bunu elle kod yazmaktan daha doğal ve hızlı bulurlar. Denemeler yapabilmek ve tasarımlarını, elle kod yazarak oluşturulan formlara göre daha hızlı ve kolay değiştirebilmek isterler.

Qt Designer, programcılara görsel tasarım yeteneği sunarak onların seçeneklerini genişletir. Qt Designer, bir uygulamanın tüm formalarının ya da sadece birkaç formunun geliştirilmesinde kullanılabilir. Qt Designer ile oluşturulan formlar en nihayetinde yine C++ kodlarıdır, bu nedenlede Qt Designer geleneksel araç zinciri ile kullanılabilir ve derleyiciye hiçbir özel gereksinim yüklemez.

Bu kısımda, Qt Designer'ı Şekil 2.4'de görülen "Go to Cell" diyaloğunu oluşturmakta kullanacağız. İster kodla yapalım ister Qt Designer'la, bir diyalog oluşturmak her zaman aynı temel adımları atmamızı gerektirir:

1. Çocuk parçacıkları oluştur ve ilk kullanıma hazırla
2. Çocuk parçacıkları yerleşimler içine koy
3. Tab sırasını ayarla
4. Sinyal-yuva bağlantıları kur

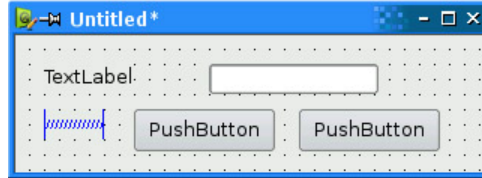
5. Diyaloğun özel yuvalarını gerçekleştir



Şekil 2.4

Qt Designer'ı çalıştırın. Qt Designer açıldığında karşınıza şablonların(templates) listesi gelecektir. "Widget"a, sonrada *Create*'e tıklayın. ("Dialog with Buttons Bottom" şablonu cazip gelebilir, fakat bu örnekte *OK* ve *Cancel* butonlarını -nasıl yapıldığını görmek için- elle kodlayarak oluşturacağız) Artık "Untitled" isimli bir pencereye sahip olmuş olmalısınız.

İlk adım, çocuk parçacıkları oluşturmak ve onları formun üzerine yerleştirmek. Bir etiket(label), bir satır editörü(line editor), bir yatay ara parça(horizontal spacer) ve iki butonu(push button) oluşturun. Her bir parça için, Qt Designer'ın parçacık kutusundaki(Widget Box) ismine ya da ikonuna tıklayarak formun üstüne sürükleyip, formda olmasını istediğimiz yere bırakalım. Şekil 2.5'deki gibi bir form elde etmeye çalışın. Fakat parçaların formdaki konumlarını ayarlamak için çokta fazla zaman harcamayın; Qt'un yerleşim yöneticileri onları daha sonra kusursuz olarak yerleştirecektirler.

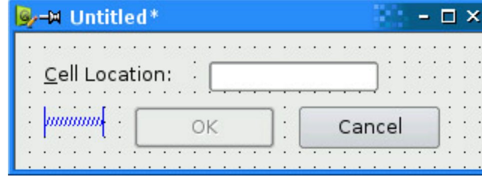


Şekil 2.5

Her bir parçacığın niteliklerini(properties) Qt Designer'ın nitelik editörünü(property editor) kullanarak ayarlayın:

1. TextLabel'a tıklayın. `objectName` niteliğinin "label" olduğundan emin olun ve `text` niteliğini "&Cell Location" olarak ayarlayın.
2. Satır editörüne tıklayın. `objectName` niteliğinin "lineEdit" olduğundan emin olun.
3. Birinci butona tıklayın. `objectName` niteliğini "okButton", `enabled` niteliğini "false", `text` niteliğini "OK" ve `default` niteliğini de "true" olarak ayarlayın.
4. İkinci butona tıklayın. `objectName` niteliğini "cancelButton", `text` niteliğini "Cancel" olarak ayarlayın.
5. Formun kendisini seçmek için formun arka planına tıklayın. `objectName` niteliğini "GoToCellDialog", `windowTitle` niteliğini "Go to Cell" olarak ayarlayın.

Şimdi &Cell Location şeklinde görünen metin etiketi(text label) dışındaki tüm parçacıklar güzel görünüyor. Arkadaşları(buddies) ayarlamaya izin veren özel bir moda girmek için Edit > Edit Buddies e tıklayın. Sonra, etiketi tıklayın ve kırmızı ok işaretini satır editörüne sürükleyip bırakın. Etiket artık Şekil 2.6'daki gibi, Cell Location olarak görünmeli. Arkadaş modundan çıkmak için Edit > Edit Widgets'ı tıklayın.

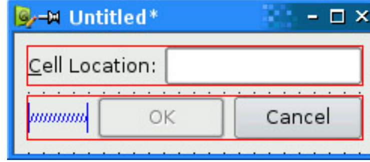


Şekil 2.6

Sıradaki adım parçacıkları form üzerine yerleştirmek:

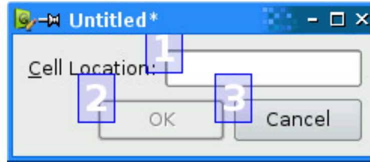
1. Cell Location etiketine tıklayın ve Ctrl'ye basılı tutarak satır editörüne de tıklayın, ikisini de seçmiş olacaksınız. Sonra, Form > Lay Out Horizontally'ye tıklayın.
2. Ara parçaya tıklayın ve Ctrl'ye basılı tutarak OK ve Cancel butonlarına da tıklayın. Sonra yine Form > Lay Out Horizontally'ye tıklayın.
3. Formun arka planını tıklayarak seçili olan parçaları seçimden çıkartın, sonrada Form > Lay Out Vertically'ye tıklayın.
4. Formu yeniden boyutlandırmak için Form > Adjust Size'a tıklayın.

Şekil 2.7'de görüldüğü gibi, yerleşimler oluşturulan formlar üzerinde kırmızı çizgiler görünür. Form çalışırken bu çizgiler görünmezler.



Şekil 2.7

Şimdi Edit > Edit Tab Order'a tıklayın. Burada, Şekil 2.8'de de görebileceğiniz mavi dikdörtgenlere tıklayarak, tab sırasını düzenleyebilirsiniz. Tab sırasını düzenledikten sonra, tab sırasını düzenleme modundan çıkmak için Edit > Edit Widgets'a tıklayın.



Şekil 2.8

Diyaloğu önizlemek için, Form > Prreview menü seçeneğini tıklayın. Tekrar tekrar Tab tuşuna basarak tab sırasını kontrol edin. Diyaloğu başlık çubuğundaki kapatma butonunu kullanarak kapatın.

Diyaloğu `gotocell` adlı bir klasöre `gotocelldialog.ui` olarak kaydedin ve aynı klasör içinde bir metin editörü kullanarak `main.cpp`'yi oluşturun:

```
#include <QApplication>
#include <QDialog>

#include "ui_gotocelldialog.h"

int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);

    Ui::GoToCellDialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();

    return app.exec();
}
```

Şimdi, `.pro` dosyası ve makefile oluşturmak için `qmake`'i çalıştırın(`qmake -project; qmake gotocell.pro`). `qmake` aracı, kullanıcı arayüzü dosyası `gotocelldialog.ui`'ı fark edecek ve Qt'un kullanıcı arayüzü derleyicisi(user interface compiler) `uic`'ı çağıracak derecede zekidir. `uic` aracı `gotocelldialog.ui`'ı C++'a çevirir ve sonucu `ui_gotocelldialog.h` içine yerleştirir.

Oluşturulan `ui_gotocelldialog.h` dosyası `Ui::GoToCellDialog` sınıfının tanımını içerir. Sınıf, formun çocuk parçacıkları ve yerleşimleri için tutulan üye değişkenlerin ve formu başlatan `setupUi()` fonksiyonunun bildirimlerini yapar. Oluşturulan sınıf şuna benzer:

```
class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...

    void setupUi(QWidget *widget) {
        ...
    }
};
```

Oluşturulan sınıf, herhangi bir ana sınıfa sahip değildir. Formu `main.cpp` içinde kullanırken bir `QDialog` oluşturur ve `setupUi()`'a aktarırız.

Eğer programı çalıştırırsanız, diyalog çalışacak, fakat kesinlikle istediğimiz gibi çalışmayacaktır:

- OK butonu her zaman devredışıdır.
- Cancel butonu hiçbir şey yapmaz.
- Satır editörü, yalnızca doğru hücre konumlarını(cell locations) kabul edeceğine, her metni kabul eder.

Diyaloğun doğru dürüst çalışmasını birkaç kod yazarak sağlayabiliriz. En doğru yaklaşım, `QDialog` ve `Ui::GoToCellDialog` sınıflarından türetilen yeni bir sınıf oluşturmak ve eksik fonksiyonelliği gerçekleştirmek olacaktır. Adlandırma kurallarımıza(naming convention) göre bu yeni sınıf `uic`'in oluşturduğu sınıf ile aynı isimde olmalıdır, fakat `Ui::` öneki olmaksızın.

Bir metin editörü kullanarak, `gotocelldialog.h` adında ve aşağıdaki kodları içeren bir dosya oluşturun:

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H

#include <QDialog>
```

```
#include "ui_gotocelldialog.h"

class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
    Q_OBJECT

public:
    GoToCellDialog(QWidget *parent = 0);

private slots:
    void on_lineEdit_textChanged();
};

#endif
```

Burada public kalıtım kullandık, çünkü diyaloğun parçacıklarına diyalog dışından da erişmek istiyoruz. Sınıfın gerçekleştirimi `gotocelldialog.cpp` dosyası içindedir:

```
#include <QtGui>

#include "gotocelldialog.h"

GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));

    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

Kurucuda, formu başlatmak için `setupUi()`'ı çağırdık. Burada çoklu kalıtıma teşekkür etmemiz gerekir, çünkü onun sayesinde `Ui::GoToCellDialog`'un üye fonksiyonlarına direkt olarak erişebiliyoruz. Kullanıcı arayüzünü oluşturduktan sonra, `setupUi()`, `on_objectName_signalName()` adlandırma kuralına uyan her yuvayı, uygun `objectName`'in `signalName()` sinyaline otomatik olarak bağlayacak. Bizim örneğimizde bu, `setupUi()`'ın şu bağlantıyı otomatik olarak kuracağı anlamına gelir:

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));
```

Ayrıca kurucuda, girdinin kapsamına sınırlama getiren bir geçerlilik denetleyicisi(validator) ayarladık. Qt bize yerleşik olarak üç adet geçerlilik denetleyicisi sınıfı sağlar: `QIntValidator`, `QDoubleValidator` ve `QRegExpValidator`. Burada, bir `QRegExpValidator`'ı "[A-Za-z][1-9][0-9]{0,2}" düzenli ifadesiyle kullandık ve bu şu anlama gelmektedir: bir büyük veya bir küçük harf, onu takip eden 1'le 9 arasında bir rakam ve onu da takip eden 0'la 9 arasında 0, 1 veya 2 rakama izin ver.

this 'i QRegExpValidator'ın kurucusuna aktararak, QRegExpValidator'ı GoToCellDialog nesnesinin çocuğu yaptık. Böyle yaparak, QRegExpValidator'ın silinmesi hakkındaki endişelerimizi ortadan kaldırdık; öyle ki ebeveyni silindiğinde o da otomatik olarak silinmiş olacak.

Qt'un ebeveyn-çocuk mekanizması QObject içinde gerçekleştirilir. Ebeveyni olan bir nesne oluşturduğumuzda (bir parçacık, geçerlilik denetleyicisi ya da herhangi başka türde bir şey), ebeveyn nesneyi çocuklarının listesine ekler. Ebeveyn silindiğinde, listedeki her bir çocuk da silinir. Çocuklar da kendi çocuklarını siler ve bu hiç çocuk kalmayıncaya dek devam eder. Ebeveyn-çocuk mekanizması bellek yönetimini(memory management) çok kolaylaştırır ve bellek sızıntıları riskini azaltır. delete'i çağırmamız gereken nesneler yalnızca new kullanarak oluşturduğumuz ebeveyni olmayan nesnelerdir. Ve ayrıca eğer bir çocuk nesneyi ebeveyninden önce silersek, Qt o nesneyi otomatik olarak ebeveyninin çocuklarının listesinden de silecektir.

Parçacıklar için ebeveynlerin ek bir anlamı daha vardır: Çocuk parçacıklar ebeveynin alanı içinde görünürler. Ebeveyn parçacığı sildiğimizde, çocuk parçacık yalnızca bellekten silinmez, aynı zamanda ekrandan da silinir.

Kurucunun sonunda, OK butonunu QDialog'un accept() yuvasına, Cancel butonunu da reject() yuvasına bağlarız. Her iki yuva da diyaloğu kapatır, fakat accept(), diyaloğun sonuç değerini 1'e eşit olan QDialog::Accepted'e, reject() ise diyaloğun sonuç değerini 0'a eşit olan QDialog::Rejected'e ayarlar. Diyaloğu kullanırken, bu sonucu kullanıcının OK'e tıkladığını görmek ve ona göre davranmak için kullanabiliriz.

on_lineEdit_textChanged() yuvası OK butonunu satır editörünün geçerli bir hücre konumu içermesine göre etkinleştirir ya da devredışı bırakır. OLineEdit::hasAcceptableInput() kurucuda ayarladığımız geçerlilik denetleyicisini kullanır.

Böylece diyaloğu tamamlamış olduk. Şimdi main.cpp'yi diyaloğu kullanmak için yeniden yazabiliriz:

```
#include <QApplication>

#include "gotocelldialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
    return app.exec();
}
```

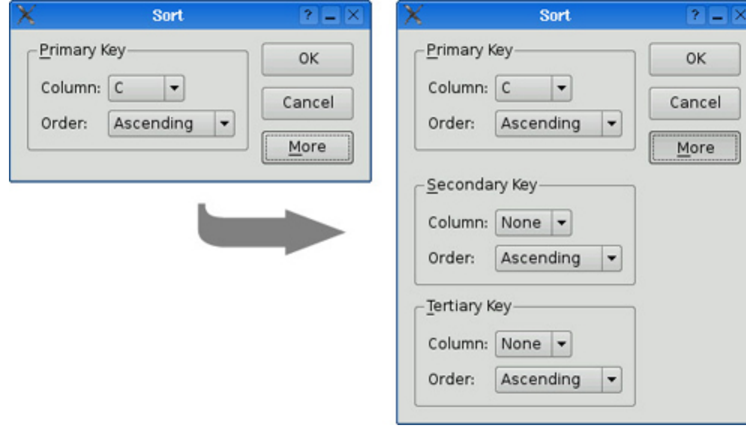
gotocell.pro'yu yeniden oluşturmak için qmake -project'i kullanın, makefile'ı güncellemek için qmake gotocell.pro'yu çalıştırın, make'i çalıştırarak uygulamayı yeniden derleyin ve çalıştırın.

Qt Designer'ı kullanmanın bir güzelliği de programcılara kaynak kodlarını değiştirme zorluğu olmadan formlarını değiştirme imkânı vermesidir. Saf C++ kodları yazarak bir form geliştirdiğinizde, tasarımı değiştirmeniz oldukça zaman kaybettirici olabilir. Eğer formunuzu Qt Designer'la oluşturduysanız, uic değiştirdiğiniz her form için yeniden kaynak kodu oluştururken hiç zaman kaybetmezsiniz.

Şekil Değiştiren Diyaloglar

Her zaman aynı parçacıkları gösteren diyaloglar oluşturmayı gördük. Bazı durumlarda, diyalogların şekil değiştirebilmesi arzu edilir. Şekil değiştiren diyalogların en çok bilinen iki türü genişleyen diyaloglar(extension dialogs) ve çok sayfalı diyaloglardır(multi-page dialogs). Her iki türden diyalog da Qt ile oluşturulabilir.

Genişleyen diyaloglar genellikle basit bir görünüm sunarlar fakat kullanıcıya basit ve genişletilmiş görünümler arasında geçiş yapabilme imkânını veren iki konumlu butona(toggle button) sahiptir. Genişleyen diyaloglar çoğunlukla hem sıradan kullanıcılara hem de ileri düzey kullanıcılara hitap etmesi istenen uygulamalarda kullanılır. Bu kısımda, Qt Designer'ı Şekil 2.9'da görünen genişleyen diyaloğu oluşturmakta kullanacağız.



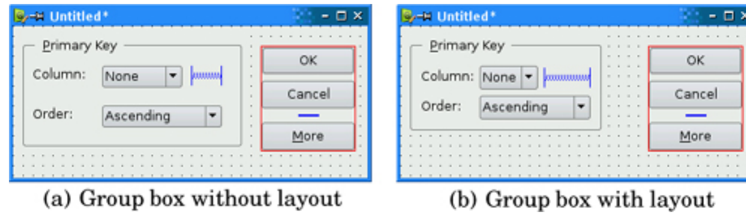
Şekil 2.9

Diyalog, bir hesap çizelgesi(spreadsheet) uygulaması içindeki bir ya da birkaç sütunu sıralamak için kullanabileceği bir Sort(sıralama) diyaloğudur. Diyaloğun basit görünümü kullanıcıya tek bir sıralama anahtarı girmesine imkân verir. Genişletilmiş görünüm ise iki ekstra sıralama anahtarı daha sağlar. Bir More butonu kullanıcının basit ve genişletilmiş görünümler arasında geçiş yapmasına izin verir.

Önce genişletilmiş görünüme sahip bir parçacığı Qt Designer'da oluşturacağız, ve sonra ikincil ve üçüncül anahtarları gizleyeceğiz. Parçacık karmaşık görünebilir, fakat Qt Designer'da bunu yapmak oldukça kolaydır. İşin sırrı önce birincil anahtar kısmını yapıp sonrada onu iki kere kopyalayarak ikincil ve üçüncül anahtarları elde etmekte:

1. File > New Form'u tıklayın ve "Dialog without Buttons" şablonunu seçin.
2. Bir OK butonu oluşturun ve formun sağ üst köşesine taşıyın. `objectName` niteliğini "okButton" olarak değiştirin ve `default` niteliğini "true" olarak ayarlayın.
3. Bir Cancel butonu oluşturun ve OK butonunun altına taşıyın. `objectName` niteliğini "cancelButton" olarak değiştirin.
4. Bir dikey ara parça oluşturun ve Cancel butonunun altına taşıyın, sonrada bir More butonu oluşturup dikey ara parçanın altına taşıyın. More butonunun `objectName` niteliğini "moreButton" olarak değiştirin, `text` niteliğini "&More" olarak ayarlayın ve `checkable` niteliğini "true" olarak ayarlayın.
5. OK butonuna tıklayın ve sonra Ctrl'ye basılı tutaraktan Cancel butonunu, dikey ara parçayı ve More butonunu da tıklayarak seçin, sonrada Form > Lay Out Vertically'ye tıklayın.
6. Bir Group Box, iki Label, iki Combo Box, ve bir Horizontal Spacer oluşturun ve form üzerinde herhangi bir yere koyun.

7. Group Box’ın sağ alt köşesinden tutun ve biraz genişletin. Sonra diğer parçacıkları Group Box içine taşıyın ve yaklaşık olarak Şekil 2.10(a)’daki gibi yerleştirin.
8. İkinci Combo Box’ın sağ kenarından tutarak, genişliğini birinci Combo Box’ın genişliğinin iki katı olacak kadar arttırın.
9. Group Box’ın `title` niteliğini “&Primary Key”, birinci Label’ın `text` niteliğini “Column:” ve ikinci Label’ın `text` nitaliğini “Order:” olarak ayarlayın.
10. Birinci Combo Box’a sağ tıklayın ve Qt Designer’ın Combo Box editörüne ulaşmak için Edit Items’i seçin. Bir “None” öğesi oluşturun.
11. İkinci Combo Box’a sağ tıklayın ve Edit Items’ı seçin. Bir “Ascendind”, bir de “Descending” parçası oluşturun.
12. Group Box’a tıklayın, sonra Form > Lay Out in a Grid’i tıklayın. Group Box’a tekrar tıklayın ve Form > Adjust Size’a tıklayın. Bu, yerleşimi Şekil 2.10(b)’deki gibi gösterecek.

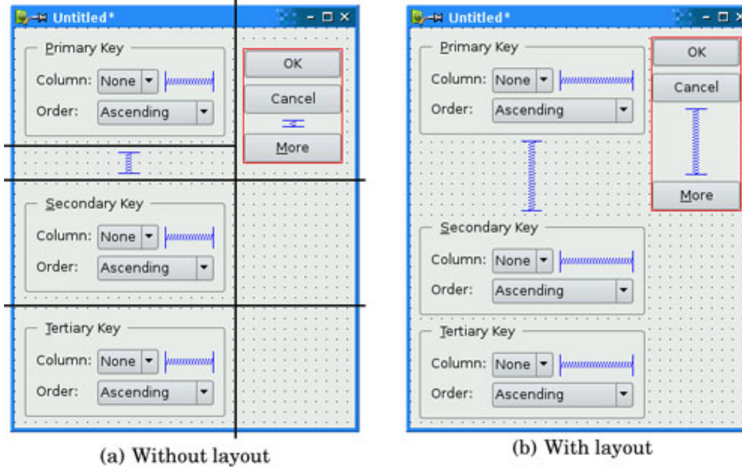


(a) Group box without layout (b) Group box with layout

Şekil 2.10

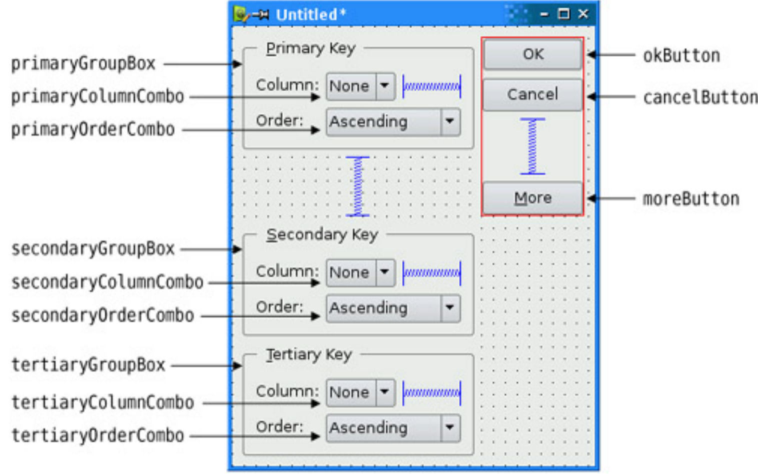
Şimdide Secondary Key ve Tertiary Key Group Box’larını ekleyeceğiz.

1. Diyalog penceresini ekstra bölümler için yeterince yüksek yapın.
2. Primary Key ve içindekilerin bir kopyasını oluşturmak için Ctrl tuşuna basılı tutaraktan Primary Key Group Box’ını tıklayın ve sürükleyip kendisinin altına bırakın. Bu işlemi bir kez de oluşturduğunuz kopyanın üstünde uygulayın ve üçüncü Group Box’ı da oluşturun.
3. `title` niteliklerini “&Secondary Key” ve “&Tertiary Key” olarak değiştirin.
4. Bir Vertical Spacer oluşturun ve Secondary Key Group Box’ı ile Primary Key Group Box’ı arasına yerleştirin.
5. Parçacıkları Şekil 2.11(a)’da göründüğü gibi, -hayali- bir ızgara kalıbı içinde düzenleyin.
6. Seçili parçacıkları seçimden çıkarmak için formu tıklayın, sonra Form > Lay Out in a Grid’i tıklayın. Şimdi, form Şekil 2.11(b) ile eşleşmeli.
7. İki Vertical Spacer’ın da `sizeHint` niteliğini [20, 0] olarak ayarlayın.



Şekil 2.11

Formu "SortDialog" olarak yeniden adlandırın ve `window title` niteliğini "Sort" olarak değiştirin. Çocuk parçacıkların isimlerini Şekil 2.12'de gösterildiği gibi değiştirin.

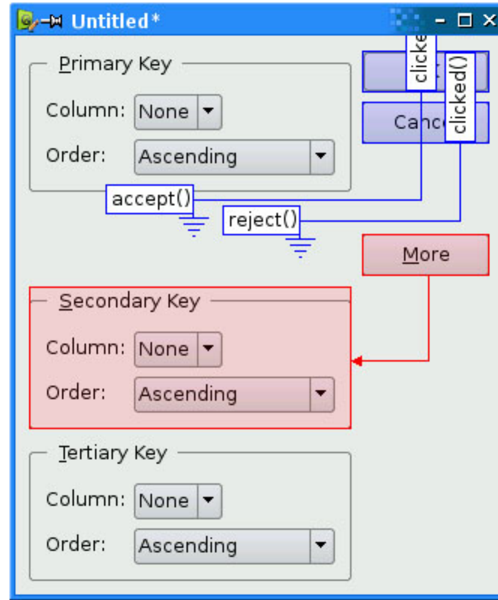


Şekil 2.12

Edit > Edit Tab Order'a tıklayın. Baştan aşağı sırayla tüm Combo Box'ları tıklayın, sonrada sağ taraftaki OK, Cancel ve More butonlarına sırayla tıklayın. Tab sıralama modundan çıkmak için Edit > Edit Widgets'a tıklayın.

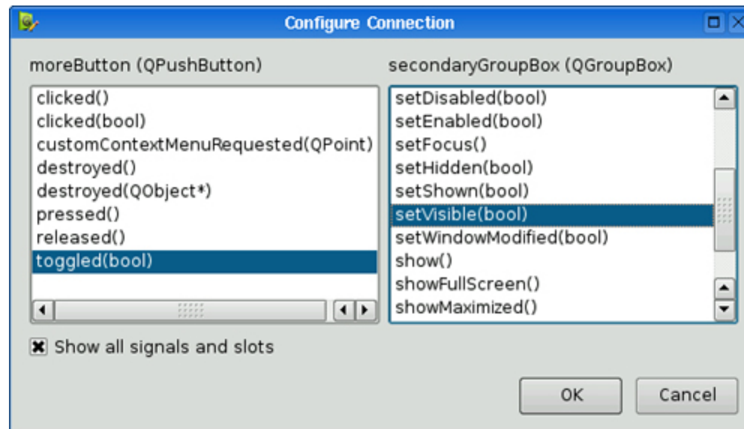
Böylece formumuz tasarlanmış oldu. Artık onu birkaç sinyal-yuva bağlantısı kurarak fonksiyonel yapmak için hazırız. Qt Designer parçacıklar arasında bağlantılar kurmamıza imkân verir. İki adet bağlantı kurmay ihtiyacımız var.

Qt Designer'ın bağlantı moduna girmek için Edit > Edit Signals/Slots'a tıklayın. Bağlantılar, Şekil 2.13'te gösterildiği gibi, formun parçacıkları arasındaki mavi oklarla temsil edilir ve Qt Designer'ın Signal/Slot Editor penceresinde listelenir. İki parçacık arasında bağlantı kurmak için gönderici(sender) parçacığı tıklayın ve kırmızı oku alıcı(receiver) parçacığa taşıyın ve bırakın. Sinyal ve yuva bağlantıları seçmenize imkân veren bir diyalog beliriverecek.



Şekil 2.13

İlk bağlantı okButton ile formun `accept()` yuvası arasında yapılacak. okButton'a tıklayın ve kırmızı oku formun boş bir yerine taşıyıp bırakın, sonra beliren Configure Connection diyaloğunda sinyal olarak `clicked()`'i yuva olarak `accept()`'i seçin ve sonrada OK'e tıklayın.



Şekil 2.14

İkinci bağlantı için, cancelButton'a tıklayın ve kırmızı oku formun boş bir yerine taşıyıp bırakın, ve Configure Connection diyaloğunda butonun `clicked()` sinyalini formun `reject()` yuvasına bağlayın.

Üçüncüsü moreButton ile secondaryGroupBox arasındaki bağlantıdır. moreButton'a tıklayın ve kırmızı oku secondaryGroupBox'a taşıyıp bırakın. Sinyal olarak `toggled(bool)`'u, yuva olarak `setVisible (bool)`'u seçin. Varsayılan olarak, Qt Designer `setVisible(bool)`'u listelemez, ancak Show all signals and slots seçeneğini seçili hale getirirseniz listelenecektir.

Dördüncü ve son bağlantı moreButton'ın `toggled(bool)` sinyali ile tertiaryGroupBox'ın `setVisible (bool)` yuvası arasındadır. Bu bağlantıyı da yapar yapmaz, Edit > Edit Widgets'a tıklayarak bağlantı modundan çıkın.

Diyaloğu `sortdialog.ui` adıyla, `sort` adlı bir klasöre kaydedin. Forma kod eklemek için, Go to Cell diyaloğunda kullandığımız gibi, burada da aynı çoklu kalıtım yaklaşımını kullanacağız.

Önce, aşağıdaki içerikle `sortdialog.h` dosyasını oluşturun:

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H

#include <QDialog>

#include "ui_sortdialog.h"

class SortDialog : public QDialog, public Ui::SortDialog
{
    Q_OBJECT

public:
    SortDialog(QWidget *parent = 0);

    void setColumnRange(QChar first, QChar last);
};

#endif
```

Şimdi, `sortdialog.cpp` dosyasını oluşturun:

Kod Görünümü:

```
 1 #include <QtGui>

 2 #include "sortdialog.h"

 3 SortDialog::SortDialog(QWidget *parent)
 4     : QDialog(parent)
 5 {
 6     setupUi(this);

 7     secondaryGroupBox->hide();
 8     tertiaryGroupBox->hide();
 9     layout()->setSizeConstraint(QLayout::SetFixedSize);

10     setColumnRange('A', 'Z');
11 }

12 void SortDialog::setColumnRange(QChar first, QChar last)
13 {
14     primaryColumnCombo->clear();
15     secondaryColumnCombo->clear();
16     tertiaryColumnCombo->clear();

17     secondaryColumnCombo->addItem(tr("None"));
18     tertiaryColumnCombo->addItem(tr("None"));
19     primaryColumnCombo->setMinimumSize(
20             secondaryColumnCombo->sizeHint());

21     QChar ch = first;
22     while (ch <= last) {
23         primaryColumnCombo->addItem(QString(ch));
24         secondaryColumnCombo->addItem(QString(ch));
25         tertiaryColumnCombo->addItem(QString(ch));
```

```
26          ch = ch.unicode() + 1;
27      }
28 }
```

Kurucu, diyaloğun ikincil ve üçüncül bölümlerini gizler. Aynı zamanda, diyaloğun kullanıcı tarafından yeniden boyutlandırılamaması için formun yerleşiminin `sizeConstraint` niteliğini `QLayout::SetFixedSize`’a ayarlar. Böylece yerleşim yeniden boyutlandırma sorumluluğunu üzerine alır ve çocuk parçacıklar göründüğünde ya da gizlendiğinde diyaloğu otomatik olarak yeniden boyutlandırır. Bu da diyaloğun her zaman ideal boyutta olmasını sağlar.

`setColumnRange()` yuvası, hesap çizelgesinde seçilen sütunlara göre, Combo Box’ların içeriğini yeniler. Bir de isteğe bağlı ikincil ve üçüncül anahtarlar için birer “None” öğesi ekleriz.

Satır 19 ve 20 hoş bir yerleşim sunar. `QWidget::sizeHint()` fonsiyonu bir parçacığın ideal boyutunu döndürür. Bu, farklı türde parçacıkların ya da farklı içerikli benzer parçacıkların yerleşim sistemi tarafından neden farklı boyutlandırıldığını açıklar.

İşte, ‘C’, ‘F’ ve aralarındaki sütunları Combo Box’ın seçeneklerine katan ve sonra diyaloğu gösteren test amaçlı bir `main()` fonksiyonu:

```
#include <QApplication>

#include "sortdialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

Böylece genişleyen diyaloğumuzu bitirdik. Bu örnekte de görüldüğü üzere, genişleyen bir diyalog tasarlamak, düz bir diyalog tasarlamaktan daha zor değildir: Tüm ihtiyacımız iki konumlu(toggle) bir buton, birkaç ekstra sinyal-yuva bağlantısı ve yeniden boyutlandırılamayan(non-resizable) bir yerleşim. Uygulamalarda, genişlemeyi kontrol eden butonun basit görünümdeyken “Advanced >>>” metni ile görüntülenmesi, genişletilmiş görünümdeyken “Advanced <<<” metni ile görüntülenmesi yaygındır. Qt içinde bunu gerçekleştirmek, `QPustButton` her tıklandığında `setText()`’i çağırmak suretiyle çok kolaydır.

Şekil değiştiren diyalogların bir diğer yaygın tipi, yine Qt ile oluşturulması oldukça kolay olan çok-sayfalı diyaloglardır. Bu tür diyaloglar çok farklı yollarla oluşturulabilir.

Dinamik Diyaloglar

Dinamik diyaloglar(dynamic dialogs) çalışma sırasında Qt Designer `.ui` dosyalarından oluşturulan diyaloglardır. `.ui` dosyalarını `uic` kullanarak C++ koduna dönüştürmek yerine, dosyayı çalışma sırasında `QUiLoader` sınıfını kullanarak yükleyebiliriz:

```
QUiLoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
    ...
```

```
}
```

Çocuk parçacıklara `QObject::findChild<T>()` kullanarak erişebiliriz:

```
QComboBox *primaryColumnCombo =
        sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
    ...
}
```

`findChild<T>()` fonksiyonu, verilen isim ve tiple eşleşen çocuk nesneleri döndüren bir şablon üye fonksiyondur(template member function).

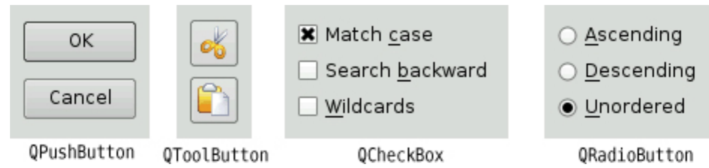
`QUiLoader` sınıfı ayrı bir kütüphanede yer alır. Bir Qt uygulamasında `QUiLoader`'ı kullanabilmek için, uygulamanın `.pro` dosyasına şu satırı eklememiz gerekir:

```
CONFIG += uitools
```

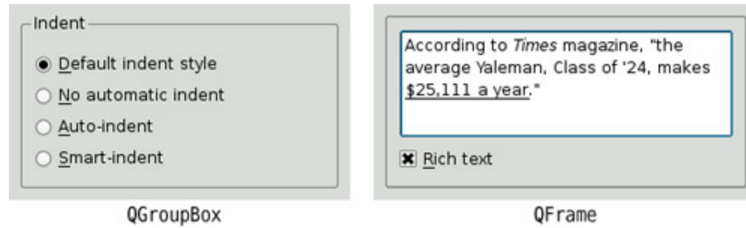
Dinamik diyaloglar, uygulamayı yeniden derlemeksizin, formun yerleşimini değiştirebilmemizi mümkün kılar.

Yerleşik Parçacık ve Diyalog Sınıfları

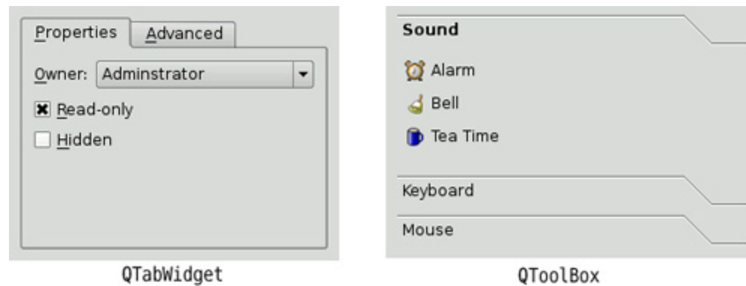
Qt, birçok durumda yaygın olarak ihtiyaç duyulan parçacıkları ve diyalogları bize yerleşik olarak sağlar. Bu kısımda size birçoğunun ekran görüntüsünü sunuyoruz:



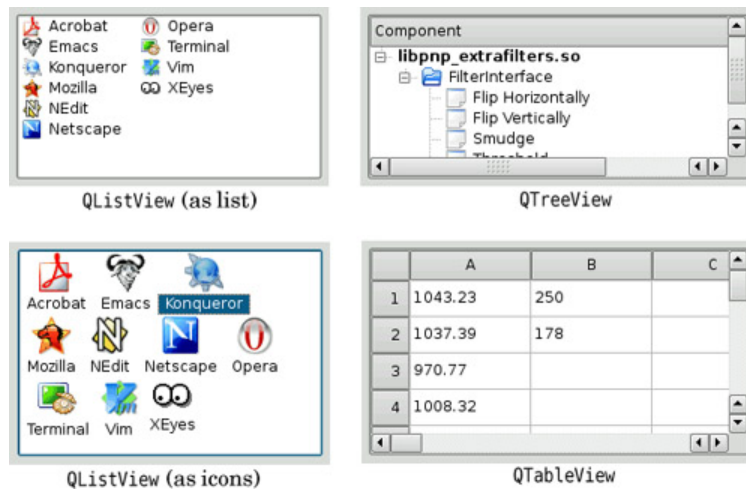
Şekil 2.15



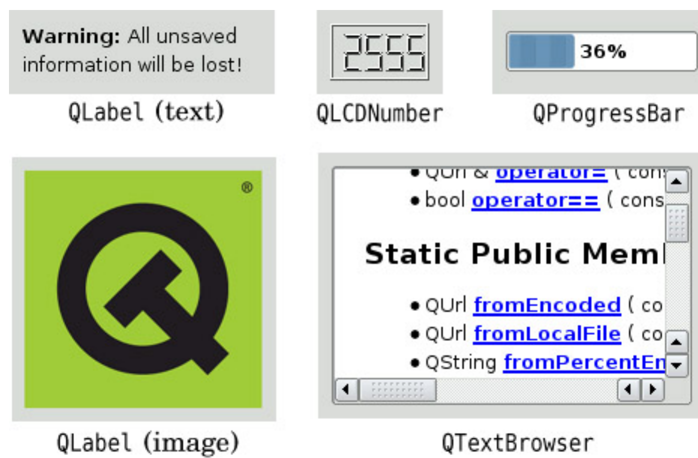
Şekil 2.16



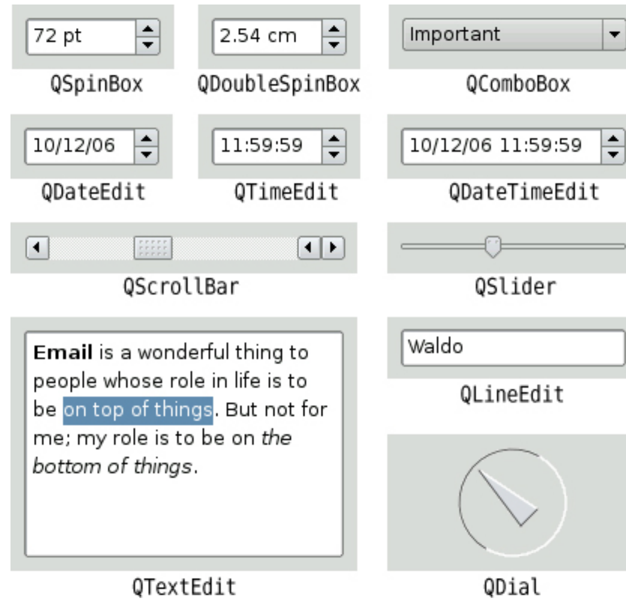
Şekil 2.17



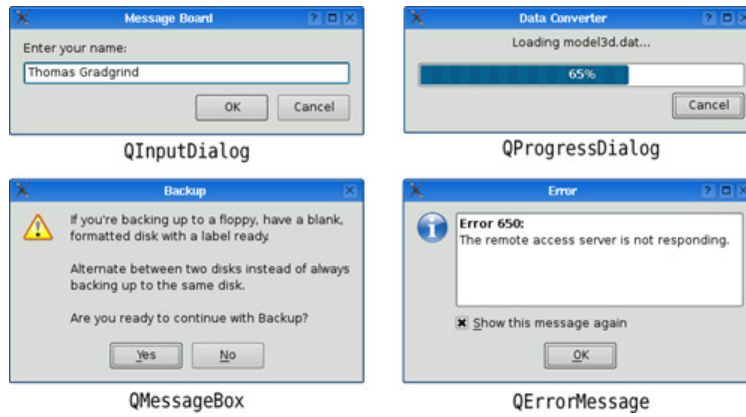
Şekil 2.18



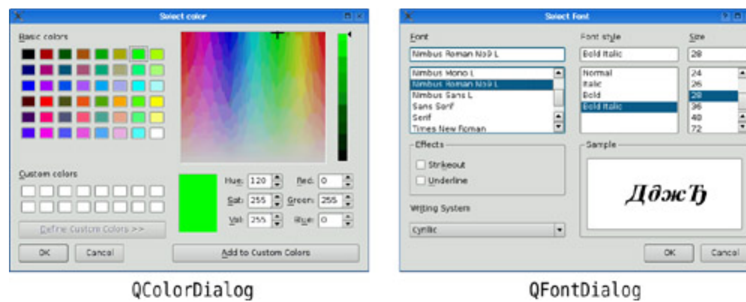
Şekil 2.19



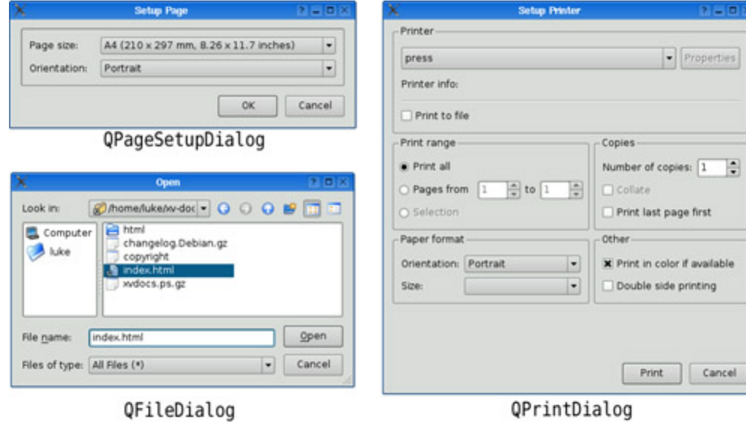
Şekil 2.20



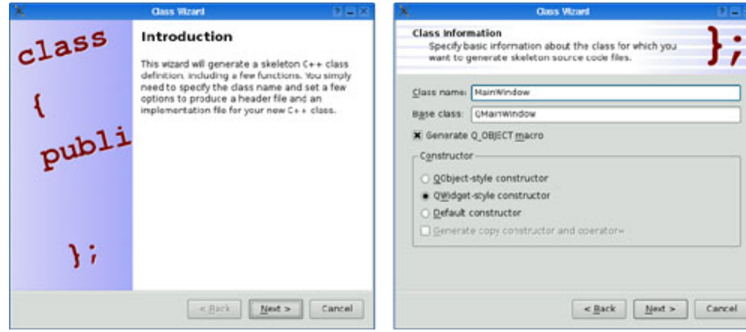
Şekil 2.21



Şekil 2.22



Şekil 2.23



Şekil 2.24

Qt bize dört çeşit buton sağlar: `QPushButton`, `QToolButton`, `QCheckBox` ve `QRadioButton` (Şekil 2.15). `QPushButton` ve `QToolButton`, çoğunlukla tıklandıkları zaman bir eylem başlatmak için kullanılırlar, fakat iki konumlu buton olarak da davranabilirler. `QCheckBox` bağımsız on/off seçenekleri için kullanılabilir. Hâlbuki `QRadioButton`'lar genellikle birbirini dışlayandırlar.

Qt'un konteyner parçacıkları(container widgets) diğer parçacıkları kapsayan(içine alan) parçacıklarıdır (Şekil 2.16 ve 2.17).

`QTabWidget` ve `QToolBox` çok-sayfalı parçacıklardır. Her bir sayfa bir çocuk parçacıktır ve sayfalar 0'dan başlayarak numaralandırılırlar. `QTabWidget`'ların şekilleri ve tablarının konumları ayarlanabilir.

Öğe görüntüleme sınıfları(item view classes), büyük miktarda verileri işlemede kullanılır ve sıklıkla kaydırma çubukları(scroll bars) kullanırlar (Şekil 2.18). Kaydırma çubuğu mekanizması, öğe görüntüleme sınıfları ve diğer kaydırılabilir(scrollable) parçacık çeşitleri için temel sınıf olan `QAbstractScrollArea` içinde gerçekleştirilir.

Qt kütüphanesi biçimlendirilmiş metni görüntülemede ve düzenlemede kullanabilecek zengin bir metin motoru(text engine) içerir. Bu motor, font özelliklerini, metin hizalanışlarını, listeleri, tabloları, resimleri ve köprüleri(hyperlinks) destekler.

Qt bize, yalnızca bilgi görüntülemede kullanılan birkaç parçacık sağlar (Şekil 19). Bunların en önemlisi `QLabel`'dır. `QLabel` düz metin, HTML ve resim görüntülemede kullanılabilir.

QTextBrowser, biçimlendirilmiş metin görüntüleyebilen, bir saltokunur(read-only) QTextEdit altsınıfıdır. Bu sınıf, büyük biçimlendirilmiş metin dokümanları için QLabel'dan daha çok tercih edilir, çünkü QLabel'ın aksine, gerektiğinde otomatik olarak kaydırma çubukları sağlayabilirken aynı zamanda klavye ve fare navigasyonu için kapsamlı bir destek de sağlar. Qt Assistant 4.3 kullanıcılara dokümantasyonu sunarken QTextBrowser'ı kullanır.

Qt, Şekil 2.20'de de görüldüğü üzere veri girişi için birkaç parçacık sağlar. QLineEdit bir girdi maskesi(input mask) veya bir geçerlilik denetleyicisi ya da her ikisini de kullanarak girdileri sınırlayabilir. QTextEdit QAbstractScrollArea'nın, büyük miktardaki metinleri düzenlemede yetenekli bir altsınıfıdır. Bir QTextEdit düz metin ya da zengin metin düzenlemeye ayarlanabilir. İkinci durumda, Qt'un zengin metin motorunun desteklediği bütün öğeleri görüntüleyebilir. Hem QLineEdit hem de QTextEdit pano(clipboard) ile tamamen entegredir.

Qt, çok yönlü bir mesaj kutusu(message box) ve bir hata(error) diyaloğu sağlar(Şekil 2.21). Zaman alan işlemlerin ilerlemesi QProgressDialog ya da QProgressBar kullanılarak gösterilebilir. QInputDialog, kullanıcının tek bir satır metin ya da tek bir sayı girmesi istendiğinde çok kullanışlıdır.

Qt, kullanıcının renk, font veya dosya seçmesini veya bir dokümanı yazdırmasını kolaylaştıran diyaloglardan standart bir set sağlar. Bunlar Şekil 2.22 ve 2.23'de gösteriliyor.

Son olarak, QWizard sihirbazlar(wizards) oluşturmak için bir çatı(framework) sağlar. Sihirbazlar, kullanıcının öğrenmeyi zor bulabileceği karmaşık ya da az karşılaşılan görevler için kullanışlıdırlar. Bir sihirbaz örneği Şekil 2.24'de gösteriliyor.

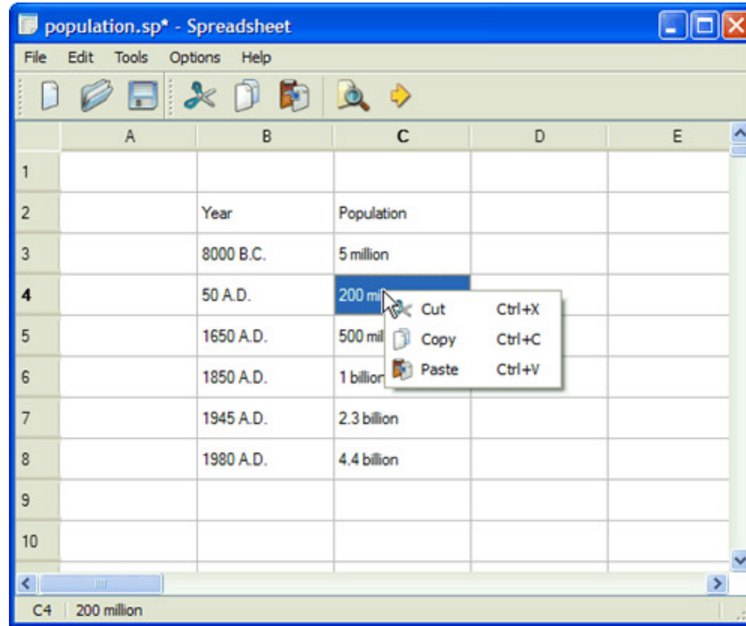
Birçok kullanıma hazır işlev yerleşik parçacıklar ve diyaloglar tarafından sağlanır. Daha özelleştirilmiş gereksinimler, parçacık niteliklerini ayarlayarak ya da sinyaller ve yuvalar bağlamak ve yuvalarda özel davranışlar gerçekleştirmek suretiyle giderilebilirler.

Bazı durumlarda, sıfırdan bir özel parçacık(custom widget) oluşturmak uygun olabilir. Qt bunu basitleştirir ve özel parçacıklar da tıpkı Qt'un yerleşik parçacıkları gibi platform bağımsızlığına sahip olurlar. Özel parçacıklar da Qt Designer'a entegre edilebilirler ve böylece Qt'un yerleşik parçacıklarıyla aynı şekilde kullanılabilirler.

BÖLÜM 3: ANA PENCERELER OLUŞTURMA

Bu bölüm, Qt'u kullanarak nasıl ana pencereler(main windows) oluşturulduğunu öğretecek. Sonunda, bir uygulamanın kullanıcı arayüzünü tam olarak oluşturabileceksiniz.

Bir uygulamanın ana penceresi, uygulamanın kullanıcı arayüzünün üstüne inşa edildiği bir çatı(framework) sağlar. Şekil 3.1'de görünen Spreadsheet(Hesap Çizelgesi) uygulamasının ana penceresi, bu bölümün belkemiğini oluşturacak. Spreadsheet uygulamasında, Bölüm 2'de oluşturduğumuz Find, Go to Cell ve Sort diyalogları da kullanılır.



Şekil 3.1

QMainWindow Altsınıfı Türetme

Bir uygulamanın ana penceresi `QMainWindow`'dan altsınıf türeterek oluşturulur. Mademki `QDialog` ve `QMainWindow` `QWidget`'tan türetilmiştir, o halde, Bölüm 2'de diyaloglar oluşturmakta kullandığımız teknikler, ana pencereler oluşturmak için de uygundurlar.

Ana pencereler Qt Designer kullanılarak da oluşturulabilirler, fakat bu bölümde, nasıl yapıldığını örnekle açıklamak amacıyla, her şeyi kodlayarak yapacağız. Eğer siz daha görsel bir yaklaşımı tercih ediyorsanız, Qt Designer'ın çevrimiçi kılavuzundaki "Creating Main Windows in Qt Designer" adlı bölüme bakabilirsiniz.

Spreadsheet uygulamasının ana penceresinin kaynak kodları, `mainwindow.h` ve `mainwindow.cpp` dosyalarına yayılır. Başlık dosyası ile başlayalım:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
```

```
#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void closeEvent(QCloseEvent *event);
```

MainWindow sınıfını, QMainWindow'un bir altsınıfı olarak tanımlarız. Kendi sinyal ve yuvalarını sağladığı için Q_OBJECT makrosunu içerir.

closeEvent() fonksiyonu, kullanıcı pencereyi kapattığında otomatik olarak çağrılan, QWidget içindeki sanal(virtual) bir fonksiyondur. MainWindow içinde yeniden gerçekleştirilmiştir ve bu sayede kullanıcıya, değişiklikleri diske kaydetmek isteyip istemediğini sorabiliriz.

```
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort();
    void about();
```

File > New ve Help > About gibi bazı menü seçenekleri MainWindow içinde private yuvalar olarak gerçekleştirilirler. Birçok yuva için dönüş değeri void'dir, fakat save() ve saveAs() yuvaları bir bool değer döndürürler. Bir yuva bir sinyale cevaben işletildiğinde, dönüş değeri görmezden gelinir, fakat bir yuvayı bir fonksiyon olarak çağırdığımızda, dönüş değerini, herhangi bir C++ fonksiyonunun dönüş değeri gibi kullanmamız da mümkündür.

```
    void openRecentFile();
    void updateStatusBar();
    void spreadsheetModified();

private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
```

Ana pencere, kullanıcı arayüzünü desteklemek için birkaç private yuvaya ve birkaç private fonksiyona daha ihtiyaç duyar.

```
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QStringList recentFiles;
    QString curFile;

    enum { MaxRecentFiles = 5 };
    QAction *recentFileActions[MaxRecentFiles];
    QAction *separatorAction;

    QMenu *fileMenu;
    QMenu *editMenu;
    ...
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QAction *newAction;
    QAction *openAction;
    ...
    QAction *aboutQtAction;
};

#endif
```

Private yuvalara ve fonksiyonlara ek olarak, `MainWindow` birçok private değişkene sahiptir. Onların hepsini, onları kullanırken açıklayacağız.

Şimdi gerçekleştirimi inceleyelim:

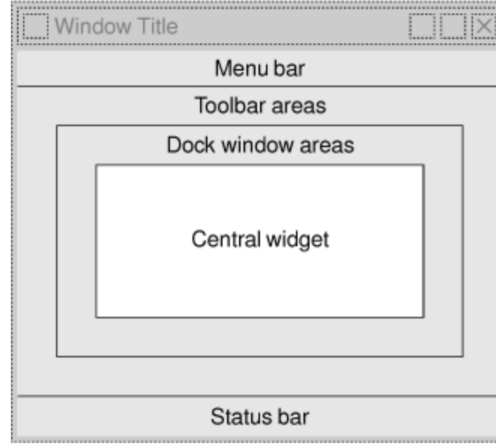
```
#include <QtGui>

#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

Altsınıflarımızda kullandığımız tüm Qt sınıflarının tanımları içeren `<QtGui>` başlık dosyasını dâhil ederek başlarız. Bazı özel başlık dosyalarını daha dâhil ederiz, bilhassa Bölüm 2'deki `finddialog.h`, `gotocelldialog.h` ve `sortdialog.h`'ı.

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);

    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();

    readSettings();

    findDialog = 0;
```

```
    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

Kurucuya, bir Spreadsheet parçacığı oluşturarak ve onu ana pencerenin merkez parçacığı(central widget) olacak şekilde ayarlayarak başlarız. Merkez parçacık ana pencerenin ortasına oturur(Şekil 3.2). Spreadsheet sınıfı, bazı hesap çizelgesi kabiliyetleri olan (hesap çizelgesi formülleri için destek gibi) bir QTableWidget altsınıfıdır.



Şekil 3.2

Ana pencerenin geri kalanını kurmak için createActions(), createMenus(), createContext-Menu(), createToolBars() ve createStatusBar() private fonksiyonlarını çağırırız. Uygulamanın depolanmış ayarlarını okumak için readSettings() private fonksiyonunu çağırırız.

findDialog işaretçisini boş(null) bir işaretçi olarak ilklendiririz. MainWindow::find() ilk kez çağrıldığında, FindDialog nesnesini oluşturacağız.

Kurucunun sonunda, pencerenin simgesi(icon) olarak icon.png'yi ayarlarız. Qt, BMP, GIF, JPEG, PNG, PNM, SVG, TIFF, XBM ve XPM de dâhil olmak üzere birçok resim formatını destekler. QWidget::set-WindowIcon() çağrısı, pencerenin sol-üst köşesinde görünen simgeyi ayarlar.

GUI uygulamaları genelde birçok resim kullanırlar. Uygulamaya resimler sağlamanın farklı metotları vardır. En yaygınları şunlardır:

- Resimleri dosyalarda depolamak ve çalışma sırasında yüklemek.
- Kaynak koduna XPM dosyaları dâhil etmek.(Bu işe yarar, çünkü XPM dosyaları geçerli C++ dosyalarıdır.)
- Qt'un kaynak mekanizmasını(resource mechanism) kullanmak.

Burada, Qt'un kaynak mekanizmasını kullanırız, çünkü bu dosyaları çalışma sırasında yüklemekten daha kullanışlıdır ve desteklenen her dosya formatıyla başarılı olur. Biz resimleri bir kaynak ağacı(source tree) içinde, images adlı bir altdizinde saklamayı tercih ettik.

Qt'un kaynak sistemini kullanmak için, bir kaynak dosyası oluşturmalı ve kaynak dosyasını belirten bir satırı `.pro` dosyasına eklemeliyiz. Bu örnekte kaynak dosyasına `spreadsheet.qrc` ismini verdik ve bu nedenle `.pro` dosyasına şu satırı ekleriz:

```
RESOURCES = spreadsheet.qrc
```

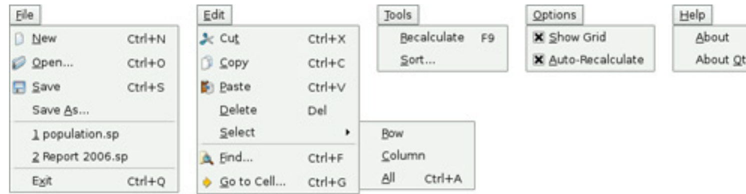
Kaynak dosyası basit bir XML formatı kullanır. İşte kullandığımız kaynak dosyasından bir alıntı:

```
<RCC>
<qresource>
    <file>images/icon.png</file>
    ...
    <file>images/gotocell.png</file>
</qresource>
</RCC>
```

Kaynak dosyaları uygulamanın yürütülebilir haline derlenir, böylece kaybolmazlar. Kaynaklardan yararlanırken, yol öneki(path prefix) `:/`'ı kullanırız. Kaynaklar herhangi bir dosya olabilirler (yani sadece resimler değil), ve kaynakları Qt'un bir dosya ismi beklediği birçok yerde kullanabiliriz.

Menüler ve Araç Çubukları Oluşturma

Birçok modern GUI uygulaması menüler, bağlam menüleri(context menus) ve araç çubukları sağlar. Menüler, kullanıcıların uygulamayı keşfetmesine ve yeni şeyler yapmayı öğrenmesine olanak verirken, bağlam menüleri ve araç çubukları sık kullanılan işlevlere hızlı erişmeyi sağlar. Şekil 3.3 Spreadsheet uygulamasının menülerini gösteriyor.



Şekil 3.3

Qt, menüler ve araç çubukları programlamayı eylem(action) kavramı sayesinde basitleştirir. Bir eylem, menüler ve araç çubukları eklenebilen bir öğedir. Qt'da menüler ve araç çubukları oluşturmak şu adımları gerektirir:

- Eylemler oluştur ve ayarla.
- Menüler oluştur ve eylemlere yerleştir.
- Araç çubukları oluştur ve eylemlere yerleştir.

Spreadsheet uygulamasında, eylemler `createActions()` içinde oluşturulur:

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(QKeySequence::New);
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

New eylemi bir hızlandırıcıya (New), bir ebeveyne(ana pencere), bir simgeye, bir kısayol tuşuna ve bir durum ipucuna(status tip) sahiptir. Birçok pencere sistemi, belirli eylemler için standartlaştırılmış klavye kısayollarına sahiptir. Örneğin, New eylemi Windows, KDE ve GNOME'da Ctrl+N, Mac OS X'te Command+N kısayoluna sahiptir. Uygun `QKeySequence::StandardKey` enum değerini kullanarak, Qt'un uygulamanın çalıştığı platforma göre doğru kısayolları sağlamasını garantiye alırız.

Eylemin `triggered()` sinyalini ana pencerenin private `newFile()` yuvasına bağlarız. Bu bağlantı, kulanıcı File > New menü öğesini seçtiğinde ya da New araç çubuğu butonunu tıkladığında ya da Ctrl+N kısayoluna bastığında `newFile()` yuvasının çağrılmasını sağlar.

Open, Save ve Save As eylemleri New eylemiyle çok benzerdirler, bu nedenle doğruca File menüsünün "recent opened files(son açılan dosyalar)" bölümüne geçeceğiz:

```
...
for (int i = 0; i < MaxRecentFiles; ++i) {
    recentFileActions[i] = new QAction(this);
    recentFileActions[i]->setVisible(false);
    connect(recentFileActions[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
}
```

`recentFileActions` dizisini eylemlere ekledik. Her eylem gizlidir ve `openRecentFile()` yuvasına bağlanmıştır. Daha sonra, son açılan dosyalar eylemlerinin nasıl görünür yapıldığını ve kullanıldığını göreceğiz.

```
exitAction = new QAction(tr("E&xit"), this);
exitAction->setShortcut(tr("Ctrl+Q"));
exitAction->setStatusTip(tr("Exit the application"));
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

Exit eylemi daha önce gördüklerimizden az da olsa farklıdır. Bir uygulamayı sonlandırmak için standartlaştırılmış bir tuş dizisi yoktur, bu nedenle tuş dizisini kendimiz açıkça belirtiriz. Bir diğer farklılıkta, pencerenin `close()` yuvasına bağlamış olmamızdır.

Şimdi, Select All eylemine geçebiliriz:

```
...
selectAllAction = new QAction(tr("&All"), this);
selectAllAction->setShortcut(QKeySequence::SelectAll);
selectAllAction->setStatusTip(tr("Select all the cells in the "
                                 "spreadsheet"));
connect(selectAllAction, SIGNAL(triggered()),
        spreadsheet, SLOT(selectAll()));
```

`selectAll()` yuvası, `QTableWidget`'ın bir atası `QAbstractItemView` tarafından sağlanır, bu nedenle onu kendimiz gerçekleştirmemiz gerekmez.

Hadi ileri -Options menüsündeki Show Grid eylemine- şıçrayalım:

```
...
showGridAction = new QAction(tr("&Show Grid"), this);
showGridAction->setCheckable(true);
showGridAction->setChecked(spreadsheet->showGrid());
showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
                                "grid"));
```

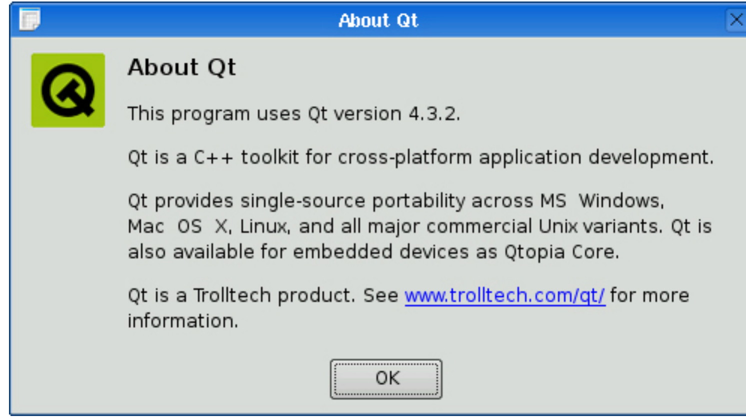
```
    connect(showGridAction, SIGNAL(toggled(bool)),
            spreadsheet, SLOT(setShowGrid(bool)));
```

Show Grid checkable bir eylemdir. Checkable *action*lar menüde bir kontrol-imi(check-mark) ile sunulur ve araç çubuklarında iki konumlu butonlar gibi gerçekleştirilirler. Eylem açık olduğunda, `Spreadsheet` bileşeni bir ızgara(grid) gösterir.

Show Grid ve Auto-Recalculate eylemleri bağımsız checkable eylemlerdir. Qt, `QActionGroup` sınıfı sayesinde birbirini dışlayan/ayrışık eylemleri de destekler.

```
    ...
    aboutQtAction = new QAction(tr("About &Qt"), this);
    aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
    connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

About Qt eylemi için(Şekil 3.4), `qApp` global değişkeni sayesinde erişilebilir olan, QApplication nesnesinin `aboutQt()` yuvasını kullanırız.



Şekil 3.4

Eylemleri oluşturduğumuza göre şimdi de onları içeren bir menü sistemi inşa etme aşamasına ilerleyebiliriz:

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
```

Qt'da, menüler `QMenu` örnekleridirler. `addMenu()` fonksiyonu, belirtilen metin ile bir `QMenu` parçacığı oluşturur ve menü çubuğuna ekler. `QMainWindow::menuBar()` fonksiyonu bir `QMenuBar` işaretçisi döndürür. Menü çubuğu ilk `menuBar()` çağrısında oluşturulur.

File menüsünü oluşturarak başlarız. Sonra ona New, Open, Save ve Save As eylemlerini ekleriz. Benzer öğeleri görsel olarak birlikte gruplandırmak için bir ayırıcı(separator) ekleriz. (Başlangıçta gizli olan)Eylemleri

recentfileActions dizisinden eklemek için bir for döngüsü kullanırız ve en sona exitAction eylemini ekleriz. Ayırıcılardan birini işaretçi olarak tutarız. Böylece iki ayırıcının arasında hiçbir şey yokken yani hiç "son açılan dosya" yokken, ayırıcılardan birini gizleyebiliriz.

```
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAction);
    editMenu->addAction(copyAction);
    editMenu->addAction(pasteAction);
    editMenu->addAction(deleteAction);

    selectSubMenu = editMenu->addMenu(tr("&Select"));
    selectSubMenu->addAction(selectRowAction);
    selectSubMenu->addAction(selectColumnAction);
    selectSubMenu->addAction(selectAllAction);

    editMenu->addSeparator();
    editMenu->addAction(findAction);
    editMenu->addAction(goToCellAction);
```

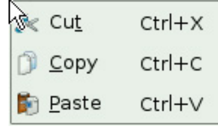
Sonra, Edit menüsünü oluşturur, File menüsü için yaptığımız gibi, istediğimiz konumlara QMenu::addAction() ile eylemler, QMenu::addMenu() ile altmenüler(submenu) ekleriz. Altmenüler de menüler gibi QMenu örnekleridirler.

```
    toolsMenu = menuBar()->addMenu(tr("&Tools"));
    toolsMenu->addAction(recalculateAction);
    toolsMenu->addAction(sortAction);

    optionsMenu = menuBar()->addMenu(tr("&Options"));
    optionsMenu->addAction(showGridAction);
    optionsMenu->addAction(autoRecalcAction);

    menuBar()->addSeparator();

    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutAction);
    helpMenu->addAction(aboutQtAction);
}
```

Benzer biçimde Tools, Options ve Help menülerini de oluştururuz.

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Her Qt parçacığı QAction'larla ilişkilendirilmiş bir listeye sahip olabilir. Uygulamaya bir bağlam menüsü sağlamak için Spreadsheet parçacığına istenilen eylemleri ekleriz ve parçacığın bağlam menü politikasını(context menu policy) bu eylemleri içeren bir bağlam menü göstermeye ayarlarız. Bağlam menüler, bir parçacığa sağ tıklayarak ya da platforma özgü bir tuşa basarak çağrılırlar. Spreadsheet uygulamasının bağlam menüsü Şekil 3.5'te gösteriliyor.

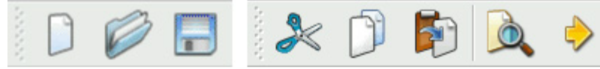


Şekil 3.5

Bağlam menüler sağlamanın daha sofistike bir yolu da `QWidget::contextMenuEvent()` fonksiyonunu uyarlamaktır: Bir `QMenu` parçacığı oluşturulur, istenilen eylemler eklenir ve `exec()` fonksiyonu çağrılır.

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);

    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

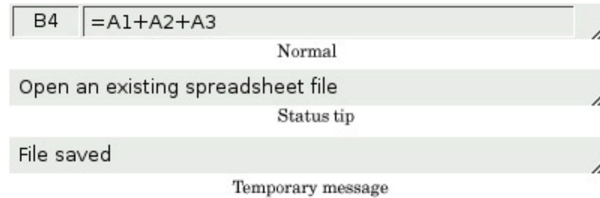
Araç çubukları oluşturmak menüler oluşturmakla çok benzerdir. Bir File araç çubuğu ve bir Edit araç çubuğu oluştururuz. Şekil 3.6'da görüldüğü üzere, tıpkı bir menü gibi, bir araç çubuğu da ayırıcılara sahip olabilir.



Şekil 3.6

Durum Çubuğunu Ayarlama

Menülerin ve araç çubuklarının tamamlanmasıyla birlikte, artık Spreadsheet uygulamasının durum çubuğunu(status bar) ele almaya hazırız. Normal durumda, durum çubuğu iki gösterge içerir: geçerli hücrenin konumu ve geçerli hücrenin formülü. Durum çubuğu, durum ipuçlarını ve diğer geçici mesajları göstermede de kullanılır. Şekil 3.7 durum çubuğunun her halini gösterir.



Şekil 3.7

Durum çubuğunu ayarlamak için `MainWindow` kurucusu `createStatusBar()` fonksiyonunu çağırır:

```
void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());
```

```
    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);

    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);

    connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
            this, SLOT(updateStatusBar()));
    connect(spreadsheet, SIGNAL(modified()),
            this, SLOT(spreadsheetModified()));
    updateStatusBar();
}
```

`QMainWindow::statusBar()` fonksiyonu durum çubuğuna bir işaretçi döndürür. (Durum çubuğu `statusBar()` ilk çağrıldığında oluşturulur.) Durum göstergeleri, gerektiğinde metnini değiştirdiğimiz basit `QLabel`'lardır. `QLabel`'lar durum çubuğuna eklendiğinde, durum çubuğunun çocukları yapılmak üzere otomatik olarak ebeveyn edindirilirler.

Şekil 3.7 bu iki etiketin(label) farklı boşluk gereksinimleri olduğunu gösterir. Hücre konum göstergesi çok küçük boşluğa ihtiyaç duyar ve pencere yeniden boyutlandırıldığında, her ekstra boşluk sağdaki hücre formül göstergesine gitmelidir. Bu, formül etiketinin `QStatusBar::addWidget()` çağrısı içinde genişleme katsayısı(stretch factor) 1 olarak belirtilerek başarılır. Konum göstergesi varsayılan olarak 0 genişleme katsayısına sahiptir. Bu, konum göstergesinin genişlemesinin tercih edilmediği anlamına gelir.

`QStatusBar` gösterge parçacıklarını yerleştirirken, her parçacık için `QWidget::sizeHint()` tarafından belirlenen ideal boyuta uymayı dener ve sonra genişleyebilen her parçacığı boşlukları doldurmak için genişletir. Bir parçacığın ideal boyutu parçacığın içeriğine bağlıdır ve biz içeriği değiştiğimizde değişir. Konum göstergesini sürekli yeniden boyutlandırılmasını önlemek için, minimum boyutunu içerebileceği en geniş metnin ("W999") genişliği ve bir ekstra boşluk olarak ayarlarız. Ayrıca hizalanışını(alignment) `Qt::AlingHCenter` ile yatay ortalanmış metin olarak ayarlarız.

Fonksiyonun bitimine yakın, `Spreadsheet`'in iki sinyalini `MainWindow`'un iki yuvasına bağlarız: `updateStatusBar()` ve `spreadsheetModified()`.

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

`updateStatusBar()` yuvası hücre konum ve hücre formül göstergelerini günceller. Kullanıcı, hücre imlecini yeni bir hücreye her taşıdığında çağrılır. Yuva, `createStatusBar()`'ın sonunda göstergeleri sıfırlamak için tipik bir fonksiyon olarak da kullanılır. Bu gereklidir, çünkü `Spreadsheet` başlangıçta `currentCellChanged()` sinyalini yaymaz.

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

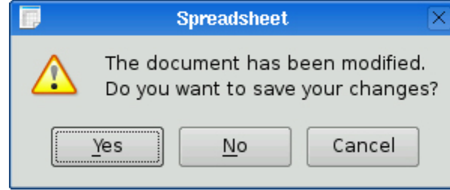
spreadsheetModified() yuvası windowModified niteliğini true olarak ayarlar ve başlık çubuğunu(title bar) günceller. Fonksiyon ayrıca konum ve formül göstergelerini günceller, böylece göstergeler geçerli durumu yansıtırlar.

File Menüsünü Gerçekleştirme

Bu kısımda, File menüsünün seçeneklerinin çalışması ve son açılan dosyalar listesinin düzenlenmesi için gerekli olan yuvaları ve private fonksiyonları gerçekleştireceğiz.

```
void MainWindow::newFile()
{
    if (okToContinue()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}
```

newFile() yuvası kullanıcı File > New menü seçeneğini ya da New araç çubuğu butonunu tıkladığında çağrılır. okToContinue() private fonksiyonu, eğer kaydedilmemiş bir değişiklik varsa "Do you want to save your changes?" diyaloğunun (Şekil 3.8) görüntülenmesini sağlar. Diyalog, eğer kullanıcı Yes veya No seçeneklerinden birini seçerse true, Cancel'ı seçerse false döndürür. Spreadsheet::clear() fonksiyonu hesap çizelgesinin tüm hücrelerini ve formüllerini temizler. setCurrentFile() private fonksiyonu pencere başlığını günceller, ek olarak curFile private değişkenini ayarlar ve son açılan dosyalar listesini günceller.



Şekil 3.8

```
bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
                        tr("The document has been modified.\n"
                           "Do you want to save your changes?"),
                        QMessageBox::Yes | QMessageBox::No
                        | QMessageBox::Cancel);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

okToContinue() içinde, windowModified niteliğinin durumunu kontrol ederiz. Eğer true ise, Şekil 3.8'de gösterilen mesaj kutusunu görüntüleriz. Mesaj kutusu Yes, No ve Cancel butonlarına sahiptir.

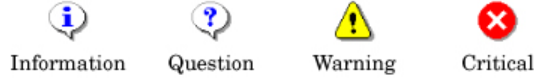
QMessageBox birçok standart buton sağlar ve otomatik olarak butonlardan birini varsayılan buton(default button; kullanıcı Enter'a bastığında aktif olan) yapmayı, birini de çıkış butonu(escape button; kullanıcı Esc'e

bastığında aktif olan) yapmayı dener. Farklı butonları varsayılan buton ve çıkış butonu olarak seçmek ve buton metinlerini değiştirmek de mümkündür.

`warning()` çağrısı ilk bakışta bir parça korkutucu görünebilir, fakat genel sözdizimi basittir:

```
QMessageBox::warning(parent, title, message, buttons);
```

`warning()`'e ek olarak, `QMessageBox` her biri farklı simgeye sahip, `information()`, `question()` ve `critical()` fonksiyonlarını sağlar. Bu simgeler Şekil 3.9'da gösteriliyor.



Şekil 3.9

```
void MainWindow::open()
{
    if (okToContinue()) {
        QString fileName = QFileDialog::getOpenFileName(this,
                                   tr("Open Spreadsheet"), ".",
                                   tr("Spreadsheet files (*.sp)"));
        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}
```

`open()` yuvası File > Open'a tekabül eder. `newFile()` gibi, önce kaydedilmemiş değişiklik varlığını denetlemek için `okToContinue()` fonksiyonunu çağırır. Kullanıcıdan yeni dosya ismini elde etmek için statik uygunluk fonsiyonu `QFileDialog::getOpenFileName()`'i kullanır. Fonksiyon bir dosya diyaloğunun görüntülenmesini sağlar, kullanıcıya bir dosya seçmesi imkânını verir ve bir dosya ismi ya da eğer kullanıcı Cancel'a tıkladıysa, boş bir karakter katarı döndürür.

`QFileDialog::getOpenFileName()`'in ilk argümanı ebeveyn parçacıktır. Diyaloglar için ebeveyn-çocuk ilişkisi, diğer parçacıklarınkiyle aynı anlamda değildir. Bir diyalog her zaman bir penceredir, fakat eğer bir ebeveyne sahipse, varsayılan olarak ebeveynin en üstünde ortalanır. Bir çocuk diyalog ebeveyniyle aynı görev çubuğu(taskbar) girdisini paylaşır.

İkinci argüman diyaloğun başlığıdır. Üçüncü argüman hangi dizinden başlanacağını söyler, bizim durumumuzda bu, geçerli dizindir.

Dördüncü argüman dosya filtreleri belirtir. Bir dosya filtresi, bir tanımlayıcı metin ve bir özel sembol kalıbından meydana gelir. Spreadsheet'in kendi dosya formatına ek olarak, Comma-separated values ve Lotus 1-2-3 dosyalarını da desteklemesini isteseydik, şöyle bir filtre kullanmamız gerekirdi:

```
tr("Spreadsheet files (*.sp)\n"
   "Comma-separated values files (*.csv)\n"
   "Lotus 1-2-3 files (*.wk1 *.wks)")
```

`open()` içinde, dosyaları yüklemek için `loadFile()` private fonksiyonu çağrıldı. Onu bağımsız bir fonksiyon yaptık, çünkü aynı işleve son açılan dosyaları yüklerken de ihtiyacımız olacak:

```
bool MainWindow::loadFile(const QString &fileName)
{
```

```
    if (!spreadsheet->readFile(fileName)) {
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
    return true;
}
```

Diskten dosya okumak için `Spreadsheet::readFile()`'ı kullanırız. Eğer yükleme başarılıysa, pencere başlığını güncellemek için `setCurrentFile()`'ı çağırırız; aksi halde, `Spreadsheet::readFile()` bir mesaj kutusu aracılığıyla kullanıcıya problemi bildirecektir.

Her iki durumda da, durum çubuğunda, uygulamanın ne yaptığı hakkında kullanıcıyı bilgilendirmek için 2 saniyeliğine(2000 milisaniye) bir mesaj görüntülenir.

```
bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}
bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

`save()` yuvası File > Save'e tekabül eder. Eğer dosya zaten bir isme sahipse yani daha önce açılmış ya da zaten kaydedilmişse, `save()` `saveFile()`'ı bu isimle çağırır; aksi halde, `saveAs()`'i çağırır.

```
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                               tr("Save Spreadsheet"), ".",
                               tr("Spreadsheet files (*.sp)"));
    if (fileName.isEmpty())
        return false;
    return saveFile(fileName);
}
```

`saveAs()` yuvası File > Save As'e tekabül eder. Kullanıcıdan bir dosya ismi elde etmek için `QFileDialog::getSaveFileName()`'i çağırırız. Eğer kullanıcı Cancel'a tıklarsa, `false` döndürülür.

Eğer dosya zaten varsa, `getSaveFileName()` fonksiyonu kullanıcıya üzerine yazmak isteyip istemediğini soracaktır. Bu davranış `getSaveFileName()`'e ek bir argüman olarak `QFileDialog::DontConfirmOverwrite`'ı aktararak değiştirilebilir.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

Kullanıcı File > Exit'a ya da pencerenin başlık çubuğundaki kapatma butonuna tıkladığında `QWidget::close()` yuvası çağrılır. Bu, parçacığa bir "kapat(close)" olayı gönderir. `QWidget::close-Event()`'i uyarlayarak, ana pencereyi kapatma girişimini önleyebiliriz ve pencereyi gerçekten kapatıp kapatmayacağımızı belirleyebiliriz.

Eğer kaydedilmemiş değişiklikler varsa ve kullanıcı Cancel'ı seçmişse, olayı "önemsemeyiz(ignore)" ve böylece pencere etkilenmez. Normal durumda ise olayı üstleniriz, sonuç olarak Qt pencereyi gizler. Ayrıca uygulamanın geçerli ayarlarını kaydetmek için `writeSettings()` private fonksiyonunu çağırırız.

Uygulama, son pencere kapatıldığında sonlandırılır. Gerekirse bu davranışı, `QApplication`'ın `quitOnLastWindowClosed` niteliğini `false` olarak ayarlayarak devre dışı bırakabiliriz. Bu durumda uygulama biz `QApplication::quit()`'i çağırana kadar çalışmaya devam eder.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);
    QString shownName = tr("Untitled");
    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*] - %2").arg(shownName)
                                   .arg(tr("Spreadsheet")));
}
QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}
```

`setCurrentFile()` içinde, `curFile` private değişkenini düzenlenen dosyanın adını saklaması için ayarlarız. Dosya ismini başlık çubuğunda göstermeden önce, dosyanın yolunu `strippedName()` ile sileriz.

Her `QWidget`, eğer pencerenin dokümanı kaydedilmemiş değişikliklere sahipse, `true` olarak ayarlanması gereken bir `windowModified` niteliğine sahiptir. Mac OS X'de, kaydedilmemiş dokümanlar pencerenin başlık çubuğundaki kapatma butonu içindeki bir nokta ile, diğer platformlarda, dosya ismini takip eden bir asteriks(*) ile belirtilirler. Qt, biz `windowModified` niteliğini güncel tuttuğumuz ve "[*]" işaretini asteriksin görünmesini istediğimiz yere yerleştirdiğimiz sürece, otomatik olarak bu davranışa bakar.

`setWindowTitle()` fonksiyonuna şu metni aktardık:

```
tr("%1[*] - %2").arg(shownName)
                .arg(tr("Spreadsheet"))
```

`arg()` iki "%n" parametresiyle kullanılır. `arg()`'ın ilk çağrısı "%1" yerine, ikinci çağrı "%2" yerine konur. Eğer dosya ismi "budget.sp" ise ve çeviri dosyası yüklenmemişse, sonuç karakter katarı "budget.sp[*] - Spreadsheet" olmalıdır. Şunu yazması daha kolay olacaktır:

```
setWindowTitle(shownName + tr("[*] - Spreadsheet"));
```

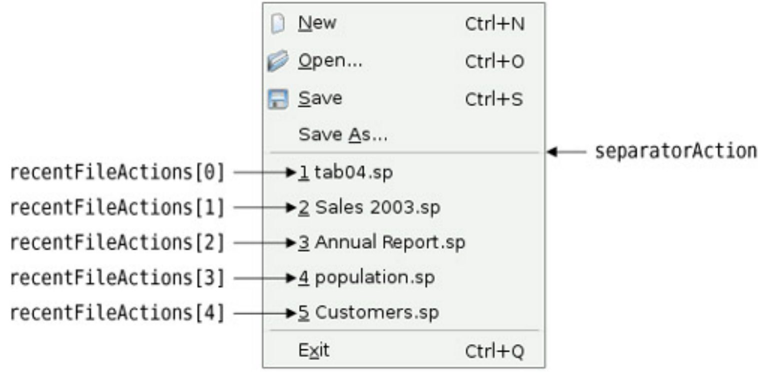
Fakat `arg()` kullanımı çevirmenler için daha fazla esneklik sağlar.

Eğer bir dosya ismi varsa, uygulamanın son açılan dosyalar listesi `recentFiles`'ı güncelleriz. Kopyaları önlemek amacıyla, listede bulunan dosya isimlerini silmek için `removeAll()` fonksiyonunu, sonrasında ilk öğe olarak dosya ismini eklemek için `prepend()`'i çağırırız. Listeyi güncelledikten sonra, File menüsündeki girdileri güncellemek için `updateRecentFileActions()` private fonksiyonunu çağırırız.

```
void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext()) {
        if (!QFile::exists(i.next()))
            i.remove();
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
        if (j < recentFiles.count()) {
            QString text = tr("&%1 %2")
                           .arg(j + 1)
                           .arg(strippedName(recentFiles[j]));
            recentFileActions[j]->setText(text);
            recentFileActions[j]->setData(recentFiles[j]);
            recentFileActions[j]->setVisible(true);
        } else {
            recentFileActions[j]->setVisible(false);
        }
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}
```

Daha fazla var olmayacak her dosyayı bir Java-tarzı yineleyici(Java-style iterator) kullanmak suretiyle silerek başlarız. Bazı dosyalar önceki bir oturumda kullanılmış fakat silinmiş olabilir. `recentFiles` değişkeni `QStringList`(`QString`'lerin listesi) tipindedir.

Sonra tekrar, bu sefer dizi-tarzı indeksleme(array-style indexing) kullanarak, dosyaların bir listesini yaparız. Her bir dosya için, '&' işaretinden oluşan bir karakter katarı oluştururuz; bir rakam (j + 1), bir boşluk ve dosya ismi (yolu olmadan). Uygun eylemleri bu metinleri kullanarak ayarlarız. Örneğin, eğer ilk dosya `C:\My Documents\tab04.sp` olsaydı, ilk eylemin metni "&1 tab04.sp" olacaktı. Şekil 3.10 sonuçta oluşan menüyü gösteriyor.



Şekil 3.10

Her eylem, `QVariant` tipinde bir birleşmiş "veri(data)" öğesine sahip olabilir. `QVariant` tipi birçok C++ ve Qt tipinin değerlerini tutabilir. Burada, eylemin "veri" öğesinde dosyanın tam ismini depolarız, böylece daha sonra kolaylıkla erişebiliriz. Aynı zamanda, eylemi görünür yaparız.

Eğer son açılan dosyalardan daha fazla dosya eylemi varsa, ekstra eylemleri açıkça gizleriz. Son olarak, eğer en az bir son açılan dosya varsa, ayırıcıyı görünür yaparız.

```
void MainWindow::openRecentFile()
{
    if (okToContinue()) {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

Kullanıcı bir son açılan dosyayı seçtiğinde, `openRecentFile()` yuvası çağrılır. `okToContinue()` fonksiyonu, herhangi bir kaydedilmemiş değişikliğin olduğu durumda kullanılır. `QObject::sender()`'ı kullanarak yuvanın hangi eylem tarafından çağrıldığını öğreniriz.

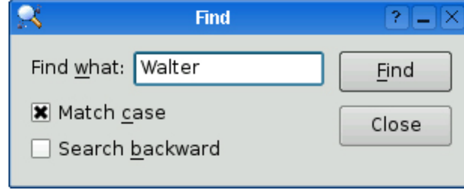
`qobject_cast<T>()` fonksiyonu, `moc` (meta object complier; Qt'un meta object derleyicisi) tarafından üretilmiş meta-information temelli bir dinamik dönüşümü(dynamic cast) yerine getirir. Standart C++ `dynamic_cast<T>()` fonksiyonuna benzemez. Qt'un `qobject_cast<T>()` fonksiyonu doğrudan dinamik kütüphane sınırları üstünde çalışır. Bizim örneğimizde, `qobject_cast<T>()`'ı bir `QObject` işaretçisini bir `QAction` işaretçisine dönüştürmekte kullanırız. Eğer dönüşüm başarılıysa (ki olmalı), eylemin verisinden sağladığımız tam dosya ismi ile `loadFile()`'ı çağırırız.

Bu arada, göndericinin(sender) bir `QAction` olduğunu bildiğimiz için, `static_cast<T>()` ya da geleneksel bir C-tarzı dönüşüm(C-style cast) kullansak da program çalışacaktır.

Diyalogları Kullanma

Bu kısımda, diyalogları Qt içinde nasıl kullanacağımızı (onları oluşturmayı, ilk kullanıma hazırlamayı, çalıştırmayı ve kullanıcının onlarla etkileşmesi sonucu ortaya çıkan seçimlere cevap vermeyi) açıklayacağız. Bölüm 2'de oluşturduğumuz Find, Go to Cell ve Sort diyaloglarını kullanacağız. Aynı zamanda bir tane de basit bir About box oluşturacağız.

Şekil 3.11'de gösterilen Find diyaloğu ile başlayacağız. Kullanıcının ana Spreadsheet penceresi ile Find diyaloğu arasında istediği gibi geçiş yapmasını istediğimiz için, Find diyaloğu *modeless* olmalıdır. *Modeless* pencere, uygulama içinde diğer pencerelerden bağımsız hareket eden bir penceredir.



Şekik 3.11

Modeless bir diyalog oluşturulduğunda, normalde, kullanıcı etkileşimine cevap veren, yuvalara bağlanmış sinyallere sahiptirler.

```
void MainWindow::find()
{
    if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL(findNext(const QString &,
                                            Qt::CaseSensitivity)),
                spreadsheet, SLOT(findNext(const QString &,
                                           Qt::CaseSensitivity)));
        connect(findDialog, SIGNAL(findPrevious(const QString &,
                                                Qt::CaseSensitivity)),
                spreadsheet, SLOT(findPrevious(const QString &,
                                               Qt::CaseSensitivity)));
    }

    findDialog->show();
    findDialog->raise();
    findDialog->activateWindow();
}
```

Find diyaloğu, kullanıcının hesap çizelgesi içinde metin arayabileceği bir penceredir. Kullanıcı Edit > Find'a tıkladığında `find()` yuvası çağrılır ve Find diyaloğu belirir. Bu noktada, birkaç senaryo mümkündür:

- Bu, kullanıcının Find diyaloğunu ilk çağırışıdır.
- Find diyaloğu daha önce çağrılmıştır, fakat kapatılmıştır.
- Find diyaloğu daha önce çağrılmıştır ve hâlâ açıktır.

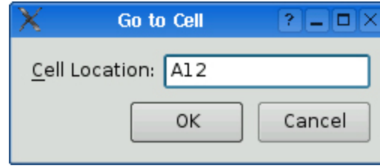
Eğer Find diyaloğu hâlihazırda bulunmuyorsa, onu oluştururuz ve `findNext()` ve `findPrevious()` sinyallerini uygun Spreadsheet yuvalarına bağlarız. Diyaloğu `MainWindow` kurucusu içerisinde oluşturabilirdik, fakat onun oluşumunu ertelemek uygulama başlangıcını daha hızlı yapar. Üstelik eğer diyalog hiç kullanılmazsa, hiç oluşturulmaz, böylece hem zaman ve hem de bellek tasarrufu sağlanır.

Sonra, pencerenin diğerlerinin üstünde ve aktif olarak açılmasını sağlamak için `show()`, `raise()` ve `ativeWindow()`'u çağırırız. Bir `show()` çağrısı tek başına gizlenmiş bir pencereyi görünür, en üstte ve aktif yapar, fakat Find diyaloğu zaten görünürken çağrılmış olabilir. Bu durumda, `show()` hiçbir şey yapmayacaktır ve bizde pencereyi en üstte ve aktif yapmak için `raise()` ve `activateWindow()`'u çağırırız. Bir alternatif şöyle yazılabilir:

```
if (findDialog->isHidden()) {
```

```
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->activateWindow();
}
```

Şimdi, Şekil 3.12'de gösterilen Go to Cell diyaloğuna bakacağız. Kullanıcının, uygulamadaki diğer pencerelerle onun arasında geçiş yapmasına izin vermeden, onu açmasını, kullanmasını ve kapatmasını istiyoruz. Bunun anlamı; Go to Cell *modal* olmalıdır. *Modal* pencere, çağrıldığında beliren ve uygulamayı bloke eden, kapanana kadar diğer işlemlere veya etkileşimlere engel olan bir penceredir. Daha evvel, dosya diyaloglarını ve mesaj kutularını *modal* olarak kullanmıştık.



Şekil 3.12

Bir diyalog, eğer `show()` kullanılarak çağrılıyorsa, *modeless*'tır (değilse *modal* yapmak için peşinen `setModal()`'ı çağırırız); eğer `exec()` kullanılarak çağrılıyorsa *modal*'dır.

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}
```

`QDialog::exec()` fonksiyonu, eğer diyalog kabul edilirse bir `true` değeri (`QDialog::Accepted`) döndürür, aksi halde bir `false` değeri (`QDialog::Rejected`) döndürür. Bölüm 2'de Qt Desginer kullanarak Go to Cell diyaloğunu oluşturduğumuzda, OK'i `accept()`'e, Cancel'ı da `reject()`'e bağladığımızı hatırlayın. Eğer kullanıcı OK'i seçerse, geçerli hücreyi, satır editörü içindeki değere ayarlarız.

`QTableWidget::setCurrentCell()` fonksiyonu iki argüman kabul eder: bir satır indeksi ve bir sütun indeksi. Spreadsheet uygulamasında, A1 (0, 0) hücresi, B27 ise (26, 1) hücresidir. `QLineEdit::text()` tarafından döndürülen `QString`'den satır indeksi elde etmek için, `QString::mid()`'i (başlangıç pozisyonundan karakter katarının sonuna kadar olan bir alt-karakter katarı döndürür) kullanarak satır numarasını çekeriz, `QString::toInt()`'i kullanarak satır numarasını (bir alt-karakter katarı) bir tamsayıya(integer) dönüştürürüz ve son olarak elde ettiğimiz tamsayı değerden 1 çıkartırız. Sütun numarası için, karakter katarının ilk karakterinin büyük harf formunun sayısal değerinden 'A' karakterinin sayısal değerini çıkartırız.

Şimdi, Sort diyaloğuna dönüyoruz. Sort diyaloğu *modal* bir diyalogdur. Şekil 3.13 bir sıralama örneği gösterir.

	A	B	C
1	George	Washington	1789-1797
2	John	Adams	1797-1801
3	Thomas	Jefferson	1801-1809
4	James	Madison	1809-1817
5	James	Monroe	1817-1825
6	John Quincy	Adams	1825-1829
7	Andrew	Jackson	1829-1837
8			

(a) Before sort

	A	B	C
1	John	Adams	1797-1801
2	John Quincy	Adams	1825-1829
3	Andrew	Jackson	1829-1837
4	Thomas	Jefferson	1801-1809
5	James	Madison	1809-1817
6	James	Monroe	1817-1825
7	George	Washington	1789-1797
8			

(b) After sort

Şekil 3.13

Kod Görünümü:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());
    if (dialog.exec()) {
        SpreadsheetCompare compare;
        compare.keys[0] =
              dialog.primaryColumnCombo->currentIndex();
        compare.keys[1] =
              dialog.secondaryColumnCombo->currentIndex() - 1;
        compare.keys[2] =
              dialog.tertiaryColumnCombo->currentIndex() - 1;
        compare.ascending[0] =
              (dialog.primaryOrderCombo->currentIndex() == 0);
        compare.ascending[1] =
              (dialog.secondaryOrderCombo->currentIndex() == 0);
        compare.ascending[2] =
              (dialog.tertiaryOrderCombo->currentIndex() == 0);
        spreadsheet->sort(compare);
    }
}
```

`sort()`'un kodları `gotoCell()` için kullandığımız aynı modeli izler:

- Yığın üstünde bir diyalog oluştururuz ve ilk kullanıma hazırlarız.
- Diyaloğu `exec()` kullanarak açarız.
- Eğer kullanıcı OK'e tıklarsa, kullanıcı tarafından girilen değerleri alırız.

`setColumnRange()` çağrısı, seçilen sütunları sıralama için hazır hale getirir. Örneğin, Şekil 3.13'te gösterilen seçimi kullanarak, `range.leftColumn()` 0 ('A' + 0 = 'A'), `range.rightColumn()` 2 ('A' + 2 = 'C') döndürecektir.

`compare` nesnesi birincil, ikincil ve üçüncül sıralama anahtarlarını ve onların sıralama düzenini depolar. (`SpreadsheetCompare` sınıfının tanımını bir sonraki bölümde göreceğiz.) Nesne `Spreadsheet::sort()` tarafından iki satırı karşılaştırmak için kullanılır. `keys` dizisi anahtarların sütun numaralarını depolar. Örneğin, seçim C2'den E5'e kadarsa, C sütunu 0 konumuna sahip olur. `ascending` dizisi düzenle ilişkili her anahtarı bir `bool` değer olarak tutar. `QComboBox::currentIndex()` o anda seçilmiş olan öğenin indeksini -0'dan başlayarak- döndürür. İkincil ve üçüncül anahtarlar için, geçerli öğeden ("None" öğesi nedeniyle) 1 çıkartırız.

sort() fonksiyonu işi yapar, fakat biraz kırılgandır. Sort diyaloğunun belirli bir yoldan (Combo Box'lar ve "None" öğeleriyle) gerçekleştirilişini üstlenir. Bunun anlamı; eğer Sort diyaloğunu yeniden tasarlarsanız, kodları yeniden yazmanız da gerekebilir. Bu yaklaşım, bir diyalog için eğer sadece bir yerden çağrılıyorsa elverişlidir.

Daha sağlıklı bir yaklaşım, bir SpreadsheetCompare nesnesi oluşturarak SortDialog sınıfını daha akıllı yapmaktır. Bu, MainWindow::sort()'u önemli derecede yalınlaştıracaktır:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());

    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

Bu yaklaşım gevşek bağlı bileşenlere yol açar ve hemen hemen her zaman birden fazla yerden çağırılacak diyaloglar için en doğru seçimdir.

Daha radikal bir yaklaşım ise, Spreadsheet nesnesine SortDialog nesnesi başlatıldığında bir işaretçi aktarmak ve böylece diyaloğun Spreadsheet üzerinde direkt olarak kullanılmasına imkân vermek olacaktır. Bu, sadece parçacığın belli bir örneği üzerinde çalışıldığı için SortDialog'u daha az genel yapar, hatta SortDialog::setColumnRange() fonksiyonunu ortadan kaldırarak kodu basitleştirir. Böylece MainWindow::sort() şu hali alır:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

Bu yaklaşım ilk olarak şunu sağlar: Çağıranın, diyaloğun özel bilgilerine ihtiyaç duyması yerine; diyalog, çağıran tarafından sağlanan veri yapılarının bilgisine ihtiyaç duyar. Bu yaklaşım, diyaloğun, değişiklikleri canlı canlı uygulaması gerektiğinde kullanışı olabilir. Fakat ilk yaklaşımın kullanılmasında karşılaştığımız kırılganlık gibi; bu üçüncü yaklaşım da, eğer veri yapıları değişirse kırılır.

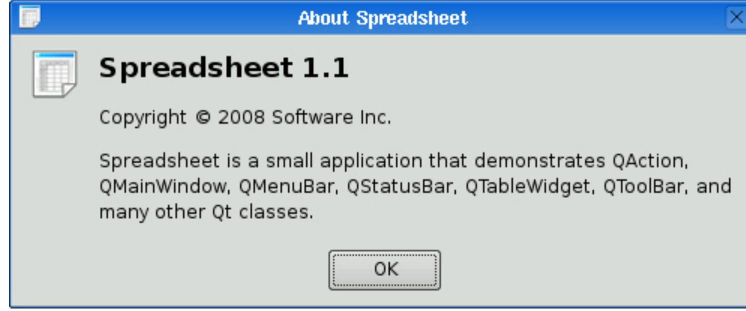
Bazı geliştiriciler, diyaloglar için sadece bir yaklaşımı seçerler ve ona sadık kalırlar. Bu, tüm diyaloglarında aynı modeli takip ettikleri için, yatkınlık ve kolaylık avantajlarına sahiptir, fakat kullanılmayan yaklaşımların avantajlarını kaçırır. İdeal olan ise, kullanılan yaklaşıma, diyaloğa göre karar verilmesidir.

Bu kısmı Hakkında kutusu(About box) ile toparlayacağız. Uygulama hakkındaki bilgileri sunmak için Find veya Go to Cell diyaloglarını yaptığımız gibi yine özel bir diyalog oluşturabilirdik, fakat Hakkında kutuları sıklıkla kullanıldıkları için, Qt daha basit bir çözüm sunar.

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
            tr("<h2>Spreadsheet 1.1</h2>"
               "<p>Copyright &copy; 2008 Software Inc."
```

```
                 "<p>Spreadsheet is a small application that "
                 "demonstrates QAction, QMainWindow, QMenuBar, "
                 "QStatusBar, QTableWidget, QToolBar, and many other "
                 "Qt classes."));
}
```

Hakkında kutusu, bir statik uygunluk fonksiyonu `QMessageBox::about()` çağrılarak sağlanır. Fonksiyon, `QMessageBox::warning()`'in ebeveyn pencerenin simgesi olarak standart "warning" simgesini kullanmasının dışında, `QMessageBox::warning()` ile çok benzerdir. Sonuçta elde ettiğimiz diyalog Şekil 3.14'te gösteriliyor.



Şekil 3.14

Şimdiye kadar, `QMessageBox` ve `QFileDialog`'tan birkaç statik uygunluk fonksiyonu kullandık. Bu fonksiyonlar bir diyalog oluşturur, onu ilk kullanıma hazırlar ve `exec()` ile çağırır. Daha az pratik olsa da, tıpkı diğer parçacıklar gibi bir `QMessageBox` ya da bir `QFileDialog` oluşturmak ve açıkça `exec()` ile ve hatta `show()` ile çağırmak da mümkündür.

Uygulama Ayarlarını Saklama

`MainWindow` kurucusu içinde, uygulamanın depolanmış ayarlarını yüklemek için `readSettings()`'i çağırdık. Benzer şekilde, `closeEvent()` içinde, ayarları kaydetmek için `writeSettings()`'i çağırdık. Bu iki fonksiyon, gerçekleştirilmesi gereken son `MainWindow` üye fonksiyonlarıdır.

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    settings.setValue("geometry", saveGeometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

`writeSettings()` fonksiyonu ana pencerenin geometrisini (konum ve boyut), son açılan dosyalar listesini ve Show-Grid ve Auto-Recalculate seçeneklerinin durumunu kaydeder.

Varsayılan olarak, `QSettings` uygulama ayarlarını platforma özgü yerlerde saklar. Windows'ta sistem kayıt defterini(system registry) kullanır; Unix'te veriyi metin dosyalarında saklar; Mac OS X'te Core Foundation Preferences API'yi kullanır.

Kurucu argümanları organizasyonun ismini ve uygulamanın ismini belirtir. Bu bilgi -platforma özgü bir yolla-ayarlar için bir yer bulmada kullanılır.

QSettings, ayarları anahtar-değer(key-value) çiftleri halinde saklar. Anahtar bir dosyanın sistem yoluna benzer. Altanahtarlar(subkeys), yol belirtmeye benzer bir sözdizimiyle (findDialog/matchCase gibi) ya da beginGroup() ve endGroup() kullanılarak belirtilebilirler.

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

Değer; bir int, bir bool, bir double, bir QString, bir QStringList ya da QVariant'ın desteklediği herhangi bir türde olabilir.

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    restoreGeometry(settings.value("geometry").toByteArray());

    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();

    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);

    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

readSettings() fonksiyonu writeSettings() tarafından kaydedilen ayarları yükler. value() fonksiyonunun ikinci argümanı, mevcut ayar olmadığı durumda varsayılan bir değer belirtir. Varsayılan değerler uygulama ilk çalıştığında kullanılırlar. Geometri ya da son açılan dosyalar listesi için verilen ikinci argüman olmadığı için pencere keyfi fakat mantıklı bir boyuta ve konuma sahip olacaktır, ve son açılan dosyalar listesi ilk çalışmada boş bir liste olacaktır.

MainWindow'da, readSettings() ve writeSettings() içindeki QSettings ilişkili tüm kodlarla tercih ettiğimiz bu düzenleme yaklaşımı, birçok olası yaklaşımdan yalnızca biridir. Bir QSettings nesnesi, uygulamanın çalışması sırasında, herhangi bir zamanda ve kodun herhangi bir yerinde, bazı ayarları sorgulamak ve değiştirmek için de oluşturulabilir.

Böylece MainWindow'un gerçekleştirimini tamamladık. İleriki kısımlarda, Spreadsheet uygulamasının çoklu doküman(multiple document) işleyebilmesini ve bir açılış ekranı(splash screen) gerçekleştirmeyi ele alacağız. İleriki bölümde de işlevlerini tamamlayacağız.

Çoklu Doküman İşleme

Artık, Spreadsheet uygulamasının main() fonksiyonunu kodlamak için hazırız:

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
```

```
    mainWin.show();
    return app.exec();
}
```

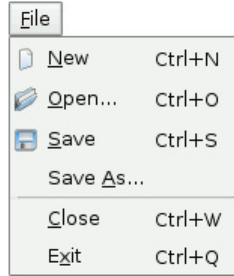
Bu `main()` fonksiyon şimdiye kadar yazdıklarımızdan biraz farklıdır: `MainWindow` örneğini `new` kullanmak yerine, yığındaki bir değişkenmiş gibi oluşturduk. `MainWindow` örneği fonksiyon sona erdirildiğinde otomatik olarak yok edilecektir.

`main()` fonksiyon ile gösterildiği gibi, Spreadsheet uygulaması bir tek ana pencere sağlar ve aynı zamanda yalnızca bir dokümanı işleyebilir. Eğer aynı anda birden fazla dokümanı düzenlemek istersek, Spreadsheet uygulamasının çoklu örneklerini başlatabiliriz. Fakat bu, kullanıcılar için -uygulamanın tek bir örneğiyle birçok ana pencere sağlayabiliyorken- uygun değildir.

Spreadsheet uygulamasını, çoklu dokümanları işleyebilmesi için değiştireceğiz. Öncelikle, biraz farklı bir File menüsüne ihtiyacımız var:

- File > New, var olan ana pencereyi kullanmak yerine, boş bir dokümanla yeni bir ana pencere oluşturur.
- File > Close geçerli ana pencereyi kapatır.
- File > Exit tüm pencereleri kapatır.

File menüsünün orijinal versiyonunda, Close seçeneği yoktu çünkü Exit ile aynı olurdu. Yeni File menüsü Şekil 3.15'te gösteriliyor.



Şekil 3.15

Bu da yeni `main()` fonksiyonudur:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    return app.exec();
}
```

Çoklu pencerelerle, `new` ile `MainWindow` oluşturmak mantıklıdır, çünkü o zaman bir ana pencereyi `delete` kullanarak yok ederek bellek kurtarabiliriz.

Bu da yeni `MainWindow::newFile()` yuvası:

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
```

```
}
```

Basitçe yeni bir MainWindow örneği oluştururuz. Yeni pencereyi herhangi bir işaretçide tutmamamız tuhaf görünebilir, fakat Qt tüm pencerelerin izini bizim için tuttuğundan, bir sorun oluşturmaz.

Bunlar da Close ve Exit için eylemlerdir:

```
void MainWindow::createActions()
{
    ...
    closeAction = new QAction(tr("&Close"), this);
    closeAction->setShortcut(QKeySequence::Close);
    closeAction->setStatusTip(tr("Close this window"));
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
            qApp, SLOT(closeAllWindows()));
    ...
}
```

QApplication::closeAllWindows() yuvası uygulamanın tüm pencerelerini kapatır, ancak bir tanesi kapat olayını(close event) reddeder. Burada bu davranışa kesinlikle ihtiyacımız vardır. Kaydedilmemiş değişiklikler için endişelenmemiz gerekmez, çünkü ne zaman bir pencere kapansa MainWindow::close-Event() işletilir.

Sanki uygulamanın çoklu pencereleri işleme yeteneğini sağlamışız gibi görünüyor. Ama ne yazık ki, gizli bir problem vardır: Eğer kullanıcı ana pencereleri oluşturup kapatmayı sürdürürse, sonunda makinenin belleği tükenebilir. Çünkü newFile()'da MainWindow parçacıkları oluşturmayı sürdürüyoruz fakat onları hiç silmiyoruz. Kullanıcı bir ana pencere kapattığında, varsayılan davranış onu gizlemektir, bu nedenle yine de bellekte durur. Birçok ana pencereyle bu bir problem olabilir.

Çözüm, kurucuda Qt::WA_DeleteOnClose niteliğini ayarlamaktır:

```
MainWindow::MainWindow()
{
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

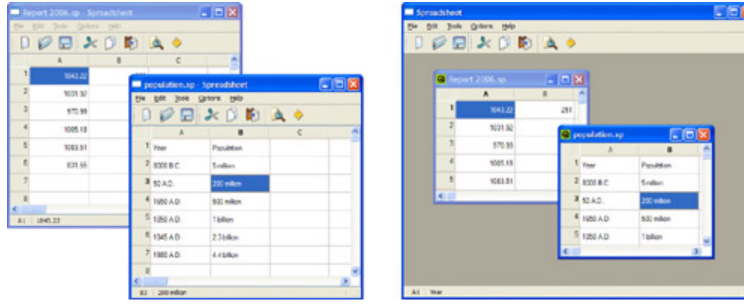
Bu, Qt'a, pencere kapandığında onu silmesini söyler. Qt::WA_DeleteOnClose niteliği, bir QWidget'ın davranışını etkilemek için ayarlanabilen birçok bayraktan(flag) biridir.

Bellek sızıntısı ilgilenmemiz gereken tek sorun değildir. Orijinal uygulamamızın tasarımında yalnızca bir ana pencereye sahip olacaktık. Çoklu pencerelerle birlikte, her bir pencere kendi son açılan dosyalar listesine ve kendi seçeneklerine sahip olur. Dolayısıyla, son açılan dosyalar listesi tüm uygulama için global olmalıdır. Bunu, recentFiles değişkenini statik olarak bildirerek başarabiliriz. Böylece, tüm uygulama için tek bir recentFiles örneği olmuş olur. Fakat sonra, File menüsünü güncellemek için çağırdığımız updateRecentFileActions()'ı tüm ana pencerelerden çağrılmasını sağlamalıyız. Bunu sağlayan kod şu şekildedir:

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

Kod, uygulamanın tüm pencerelerini yineleyerek `MainWindow` tipindeki tüm parçacıklar için `updateRecentFileActions()`'ı çağırmak için Qt'un `foreach` yapısını kullanır. Benzer bir kod, Show Grid ve Auto-Recalculate seçeneklerini eşzamanlı yapmak için ya da aynı dosyanın iki kere yüklenmediğinden emin olmak için de kullanılabilir.

Ana pencere başına bir doküman sağlayan uygulamalara SDI(single document interface) uygulamaları denir. Windows üzerindeki yaygın bir alternatifi olan MDI(multiple document interface), orta bölgesinde çoklu doküman pencerelerinin yönetildiği tek bir ana pencereye sahiptir. Qt, destekleyen tüm platformlarda, hem SDI hem de MDI uygulamalar oluşturmak için kullanılabilir.



Şekil 3.16

Açılış Ekranları

Uygulamaların birçoğu başlangıçta Şekil 3.17'deki gibi bir açılış ekranı(splash screen) sunar. Bazı geliştiriciler yavaş bir başlangıcı gizlemek için bir açılış ekranı kullanırken, bazıları da bunu 'pazarlama departmanlarını' memnun etmek için yaparlar. Qt uygulamalarına `QSplashScreen` sınıfını kullanarak bir açılış ekranı eklemek çok kolaydır.

Şekil 3.17

`QSplashScreen` sınıfı, ana pencere görünmeden önce bir resim gösterir. Uygulamanın başlatılma sürecinin ilerlemesi hakkında kullanıcıyı bilgilendirmek için resim üstüne mesajlar da yazabilir. Genelde, açılış ekranı kodu `main()` içinde yer alır(`QApplication::exec()` çağrısından önce).

QSplashScreen sınıfını kullanarak kullanıcıya, başlangıçta modülleri yükleyen ve ağ bağlantıları kuran bir açılış ekranı sağlayan örnek main() fonksiyonunun kodları şu şekildedir:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();

    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;
    splash->showMessage(QObject::tr("Setting up the main window..."),
                           topRight, Qt::white);
    MainWindow mainWin;

    splash->showMessage(QObject::tr("Loading modules..."),
                           topRight, Qt::white);
    loadModules();

    splash->showMessage(QObject::tr("Establishing connections..."),
                           topRight, Qt::white);
    establishConnections();

    mainWin.show();
    splash->finish(&mainWin);
    delete splash;

    return app.exec();
}
```

Böylelikle Spreadsheet uygulamasının kullanıcı arayüzünü tamamlamış olduk. Sonraki bölümde, çekirdek hesap çizelgesi işlevlerini gerçekleştirerek uygulamayı tamamlayacağız.

BÖLÜM 4: UYGULAMA İŞLEVLERİNİ GERÇEKLEŞTİRME

Önceki iki bölümde, Spreadsheet uygulamasının arayüzünün nasıl oluşturulduğunu açıkladık. Bu bölümde, temel işlevleri kodlayarak programı tamamlayacağız. Diğer şeylerin yanı sıra, dosyaları yüklemeyi ve kaydetmeyi, veriyi bellekte saklamayı, pano(clipboard) işlemlerini gerçekleştirmeyi ve QTableWigdet'ın hesap çizelgesi formülleri için destek vermesini sağlamayı göreceğiz.

Merkez Parçacık

Bir QMainWindow'un merkez alanı her türden parçacık tarafından tutulabilir. İşte tüm bu olasılıklara genel bir bakış:

1. **Standart bir Qt parçacığı kullanmak**
   QTableWidget veya QTextEdit gibi standart bir parçacık merkez parçacık olarak kullanılabilir. Bu durumda, dosyaları yükleme ve kaydetme gibi uygulama işlevleri başka bir yerde gerçekleştirilmelidir (örneğin bir QMainWindow altsınıfında).

2. **Özel bir parçacık kullanmak**
   Özel amaçlı uygulamalarda, genelde veriyi özel bir parçacıkta göstermek gerekir. Örneğin, bir simge editörü(icon editor) programında merkez parçacık bir IconEditor parçacığı olacaktır. Qt'da özel parçacıkların nasıl oluşturulduğu Bölüm 5'te açıklanmaktadır.

3. **Bir yerleşim yöneticisiyle sade bir QWidget kullanmak**
   Bazen, uygulamanın merkez alanı birçok parçacık tarafından tutulur. Bu, bir QWidget'ı diğer parçacıkların ebeveyni olarak kullanarak yapılabilir. Yerleşim yöneticisiyle de çocuk parçacıkların boyutları ve konumları ayarlanır.

4. **Bir bölücü(splitter) kullanmak**
   Çoklu parçacıkları birlikte kullanmanın bir diğer yolu da bir QSplitter kullanmaktır. QSplitter çocuk parçacıkları dikey ya da yatay olarak düzenler. Ayrıca, bölücü kullanmak kullanıcıya bir miktar boyutlandırma kontrolü verir. Bölücüler, diğer bölücülerde dâhil olmak üzere her türlü parçacığı içerebilirler.

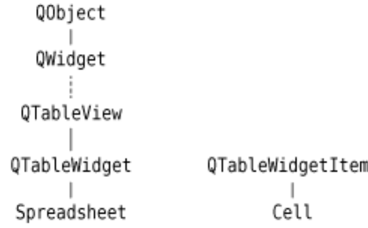
5. **MDI alanı kullanmak**
   Eğer uygulama MDI kullanırsa, merkez alan bir QMdiArea parçacığı tarafından tutulur ve her MDI penceresi bu parçacığın bir çocuğudur.

Yerleşimler, bölücüler ve MDI alanları standart Qt parçacıkları ya da özel parçacıklarla birlikte olabilirler.

Spreadsheet uygulaması için, bir QTableWidget altsınıfı merkez parçacık olarak kullanılabilir. QTableWidget sınıfı zaten hesap çizelgesi yeteneklerinin birçoğunu sağlar, fakat pano işlemlerini desteklemez ve "=A1+A2+A3" gibi hesap çizelgesi formüllerinden anlamaz. Bu eksik işlevleri Spreadsheet sınıfı içerisinde gerçekleştireceğiz.

QTableWidget Altsınıfı Türetme

Spreadsheet sınıfı, Şekil 4.1'de görüldüğü gibi QTableWidget'tan türetilmiştir. Kullanıcı boş bir hücreye metin girdiğinde, QTableWidget metni saklamak için otomatik olarak bir QTableWidgetItem oluşturur.



Şekil 4.1

QTablewWidget, model/görünüş(model/view) sınıflarından biri olan QTableView'den türetilmiştir.

Başlık dosyasından başlayarak Spreadsheet'i gerçekleştirmeye başlayalım:

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <QTableWidget>

class Cell;
class SpreadsheetCompare;
```

Başlık dosyası Cell ve SpreadsheetCompare sınıflarının ön bildirimleriyle başlar.

Bir QTableWidget hücresinin nitelikleri (metni, hizalanışı vb.) bir QTableWidgetItem içinde saklanır. QTableWidget'tan farklı olarak, QTableWidgetItem bir parçacık sınıfı değil, bir veri sınıfıdır. Cell sınıfı QTableWidgetItem'dan türetilir.

```
class Spreadsheet : public QTableWidget
{
    Q_OBJECT

public:
    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; }
    QString currentLocation() const;
    QString currentFormula() const;
    QTableWidgetSelectionRange selectedRange() const;
    void clear();
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    void sort(const SpreadsheetCompare &compare);
```

autoRecalculate() fonksiyonu yalnızca otomatik hesaplamanın(auto-recalculation) geçerliliğini döndürdüğü için satır içi(inline) gerçekleştirilmiştir.

Bölüm 3'te, MainWindow'u gerçekleştirirken Spreadsheet içindeki bazı public fonksiyonlara güvenmiştik. Örneğin, MainWindow::newFile()'dan, hesap çizelgesini sıfırlamak için clear()

fonksiyonunu çağırdık. QTableWidget'tan miras alınan bazı fonksiyonları da kullandık, bilhassa setCurrrentCell() ve setShowGrid()'i.

```
public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void selectCurrentRow();
    void selectCurrentColumn();
    void recalculate();
    void setAutoRecalculate(bool recalc);
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

signals:
    void modified();
```

Spreadsheet, Edit, Tools ve Options menülerinin eylemlerini gerçekleştiren bir çok yuva, ve bir de herhangi bir değişiklik meydana geldiğini bildiren modified() sinyalini sağlar.

```
private slots:
    void somethingChanged();
```

Spreadsheet sınıfı tarafından dâhilen kullanılan bir private yuva tanımlarız.

```
private:
    enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };

    Cell *cell(int row, int column) const;
    QString text(int row, int column) const;
    QString formula(int row, int column) const;
    void setFormula(int row, int column, const QString &formula);

    bool autoRecalc;
};
```

Sınıfın private kısmında, üç sabit, dört fonksiyon ve bir değişken bildiririz.

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};

#endif
```

Başlık dosyası SpreadsheetCompare sınıfının tanımıyla biter. Bunu Spreadsheet::sort()'un kritiğini yaparken açıklayacağız.

Şimdi gerçekleştirime bakacağız:

```
#include <QtGui>
```

```
#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidget(parent)
{
    autoRecalc = true;

    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);

    connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
            this, SLOT(somethingChanged()));

    clear();
}
```

Normal olarak, kullanıcı boş bir hücreye metin girdiğinde, `QTableWidget` metni tutmak için otomatik olarak bir `QTableWidgetItem` oluşturur. Bizim hesap çizelgemizde, onun yerine `Cell` öğesinin oluşturulmasını istiyoruz. Bu, kurucu içinde `setItemPrototype()` çağrılarak gerçekleştirilir. Dâhilen, `QTableWidget`, ilk örnek(prototype) olarak aktarılan öğeyi, yeni bir öğe gerektiğinde klonlar.

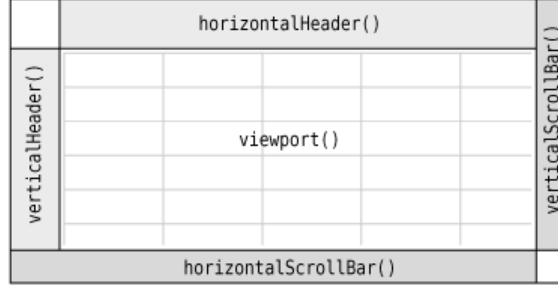
Kurucuda ayrıca, seçim modunu(selection mode) tek bir dikdörtgen seçmeye izin veren `QAbstractItemView::ContiguousSelection` olarak ayarlarız. Tablo parçacığının `item-Changed()` sinyalini private `somethingChanged()` yuvasına bağlarız; bu, kullanıcı bir hücreyi düzenlediğinde `somethinChanged()` yuvasının çağrılmasını sağlar. Son olarak, tabloyu yeniden boyutlandırmak ve sütun yüksekliklerini ayarlamak için `clear()`‘ı çağırırız.

```
void Spreadsheet::clear()
{
    setRowCount(0);
    setColumnCount(0);
    setRowCount(RowCount);
    setColumnCount(ColumnCount);

    for (int i = 0; i < ColumnCount; ++i) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString(QChar('A' + i)));
        setHorizontalHeaderItem(i, item);
    }

    setCurrentCell(0, 0);
}
```

`clear()` fonksiyonu, hesap çizelgesini sıfırlamak için `Spreadsheet` kurucusundan çağrılır. Ayrıca `MainWindow::newFile()`’dan da çağrılır.

Tüm seçimleri ve öğeleri temizlemek için `QTableWidget::clear()`’ı kullanabilirdik, fakat bu, başlıkları(header) geçerli boyutlarında bırakacaktı. Bunun yerine, tabloyu 0 x 0’a yeniden boyutlandırırız. Bu, başlıklar da dâhil tüm hesap çizelgesini temizler. Ondan sonra, tabloyu `ColumnCount` x `RowCount`(26 x 999)’a yeniden boyutlandırırız ve yatay başlıkları, `QTableWidgetItem`’ların içerdiği sütun isimleri “A”, “B”, …, “Z” ile doldururuz. Dikey başlık etiketlerini ayarlamamıza gerek yoktur, çünkü varsayılanları “1”, “2”, …, “999”dur. En sonunda hücre göstergesini A1 hücresine taşırız.

Bir `QTableWidget` birkaç çocuk parçacıktan ibarettir. En üstte yatay bir `QHeaderView`'e, solda dikey bir `QHeaderView`'e ve iki de `QScrollBar`'a sahiptir. Ortadaki alan, `QTableWidget`'ın üzerine hücreleri çizdiği görüntü kapısı(viewport) adında özel bir parçacık tarafından tutulur. Farklı çocuk parçacıklar `QTableWidget` ve `QAbstractScrollArea`'dan miras alınan fonksiyonlara direkt olarak erişebilirler (Şekil 4.2). `QAbstractScrollArea`, açılıp kapatılabilen, kaydırılabilir bir görüntü kapısı ve iki kaydırma çubuğu sağlar.



Şekil 4.2

```
Cell *Spreadsheet::cell(int row, int column) const
{
    return static_cast<Cell *>(item(row, column));
}
```

`cell()` private fonksiyonu, belli bir satır ve sütun için bir `Cell` nesnesi döndürür. Bir `QTableWidgetItem` işaretçisi yerine bir `Cell` işaretçisi döndürmesi dışında, hemen hemen `QTableWidget::item()` ile aynıdır.

```
QString Spreadsheet::text(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->text();
    } else {
        return "";
    }
}
```

`text()` private fonksiyonu, belli bir hücre için bir metin döndürür. Eğer `cell()` boş bir işaretçi döndürürse, hücre boştur, dolayısıyla boş bir karakter katarı döndürürüz.

```
QString Spreadsheet::formula(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->formula();
    } else {
        return "";
    }
}
```

`formula()` fonksiyonu hücrenin formülünü döndürür. Birçok durumda, formül ve metin aynıdır; örneğin, "Hello" formülü "Hello" karakter katarına eşittir, bu nedenle eğer kullanıcı hücre içine "Hello" girerse ve Enter'a basarsa, hücre "Hello" metinini gösterir. Fakat birkaç istisna vardır:

- Eğer formül bir sayı ise, öyle yorumlanır. Örneğin, "1.50" formülü 1.5 double değerine eşittir ve hesap çizelgesinde sağa yaslı duruma getirilir.
- Eğer formül tek tırnakla başlıyorsa, formülün geri kalanı metin olarak yorumlanır. Örneğin, "'12345" formülü "12345" karakter katarına eşittir.
- Eğer formül eşittir işaretiyle('=') başlıyorsa, formül aritmetik bir formül olarak yorumlanır. Örneğin, eğer A1 hücresinin içeriği "12" ve A2 hücresinin içeriği "6" ise, "=A1+A2" formülünün sonucu 18'e eşittir.

Bir formülü bir değer dönüştürme görevi Cell sınıfı tarafından yerine getirilir. Şuan için akılda kalması gereken şey, hücrede gösterilen metnin formülün kendisi değil, formülün sonucu olduğudur.

```
void Spreadsheet::setFormula(int row, int column,
                             const QString &formula)
{
    Cell *c = cell(row, column);
    if (!c) {
        c = new Cell;
        setItem(row, column, c);
    }
    c->setFormula(formula);
}
```

setFormula() private fonksiyonu belli bir hücre için formülü ayarlar. Eğer hücre zaten bir Cell nesnesine sahipse, onu yeniden kullanırız. Aksi halde, yeni bir Cell nesnesi oluştururuz ve tabloya eklemek için QTableWidget::setItem()'ı çağırırız. Sonda, eğer hücre ekranda gösterilirse, hücrenin yeniden çizilmesine neden olacak olan setFormula() fonksiyonunu çağırırız. Cell nesnesinin daha sonra silinmesi hakkında endişelenmemiz gereksizdir; QTableWidget hücrenin sahipliğini üstlenir ve doğru zamanda hücreyi otomatik olarak siler.

```
QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn())
           + QString::number(currentRow() + 1);
}
```

currentLocation() fonksiyonu, hesap çizelgesinin alışılmış formatına uygun bir şekilde (sütun harfini takiben satır numarası) geçerli hücre konumunu döndürür. Dönen değer, MainWindow::updateStatusBar() tarafından, konumu durum çubuğunda göstermede kullanılır.

```
QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}
```

currentFormula() fonksiyonu, geçerli hücre formülünü döndürür. MainWindow::updateStatusBar()'dan çağrılır.

```
void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}
```

somethingChanged() private yuvası, eğer "otomatik hesaplama" etkinse, tüm hesap çizelgesini yeniden hesaplar. Ayrıca modified() sinyalini yayar.

Yükleme ve Kaydetme

Artık, bir özel ikili(binary) format kullanarak, Spreadsheet dosyalarını yükleme ve kaydetmeyi gerçekleştireceğiz. Bunu, birlikte platformdan bağımsız ikili I/O sağlayan QFile ve QDataStream sınıflarını kullanarak yapacağız.

Bir Spreadsheet dosyası yazmakla başlayacağız:

Kod Görünümü:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("Cannot write file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }
    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_3);

    out << quint32(MagicNumber);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {

        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

writeFile() fonksiyonu, dosyayı diske yazmak için MainWindow::saveFile()'dan çağrılır. Başarılıysa true, hatalıysa false döndürür.

Verilen dosya ismi ile bir QFile nesnesi oluştururuz ve yazmak için dosyayı açarız. Ayrıca QFile üstünde işlemler yapan bir QDataStream nesnesi oluştururuz ve onu veri yazmada kullanırız.

Veri yazmadan hemen önce, uygulamanın imlecini(cursor) standart bekleme imleciyle (genellikle bir kum saatidir) değiştiririz ve tüm veri yazıldığında da normal imlece geri döneriz.

QDataStream, Qt tiplerinin birçoğunu desteklediği gibi temel C++ tiplerini de destekler. Sözdiziminde Standart C++ <iostream> sınıfı örnek alınmıştır. Örneğin,

```
out << x << y << z;
```

x, y ve z değişkenlerini bir akışa(stream) yazarken,

```
in >> x >> y >> z;
```

bir akıştan onları okur. Çünkü C++ ilkel tamsayı(integer) tipleri farklı platformlarda farklı boyutlarda olabilirler, en sağlamı bu değerleri, boyutlarının tanınabilir olmasını garanti eden `qint8`, `quint8`, `qint16`, `quint16`, `qint32`, `quint32`, `qint64` ve `quint64`'den birine dönüştürmektir.

Spreadsheet uygulamasının dosya formatı oldukça basittir. Bir Spreadsheet dosyası dosya formatını tanımlayan 32-bitlik bir sayı ile başlar(`MagicNumber`, `spreadsheet.h` içinde 0x7F51C883 olarak tanımlanan, isteğe bağlı rastgele bir sayıdır). Sonrasında her bir hücrenin satırını, sütununu ve formülünü içeren blokların bir serisi gelir. Boşlukları kaydetmek için boş hücreleri yazmayız. Format Şekil 4.3'te gösteriliyor.

Şekil 4.3

Veri tiplerinin ikili gösteriminin açık hali `QDataStream` tarafından belirlenir. Örneğin, bir `qint16` 2 byte olarak *big-endian* düzende, bir `QString` uzunluğunca Unicode karakteri olarak saklanır.

Qt tiplerinin ikili gösterimi, Qt 1.0'dan buyana oldukça geliştirildi. Gelecekteki Qt dağıtımlarında da geliştirilmeye devam edileceğe benziyor. Varsayılan olarak, `QDataStream` ikili formatın en son versiyonunu kullanır (Qt 4.3'te versiyon 9), fakat eski versiyonları okumak için de ayarlanabilir. Uygulamayı daha sonra daha yeni bir Qt dağıtımı kullanılarak yeniden derlenebileceğini düşünerek, herhangi bir uyumluluk problemini önlemek için, `QDataStream`'e açıkça, Qt'un versiyonuna aldırmadan versiyon 9'u kullanmasını sözleriz. (`QDataStream::Qt_4_3` 9'a eşit olan uygunluk sabitidir.)

`QDataStream` çok yönlüdür. Bir `QFile`'da, bir `QBuffer`'da, bir `QProcess`'te, bir `QTcpSocket`'te, bir `QUdpSocket`'te ya da bir `QSslSocket`'te kullanılabilir. Qt ayrıca, metin dosyalarını okumak ve yazmak için `QDataStream` yerine kullanılabilecek bir `QTextStream` sınıfı sunar.

Kod Görünümü:

```
bool Spreadsheet::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("Cannot read file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }

    QDataStream in(&file);
    in.setVersion(QDataStream::Qt_4_3);

    quint32 magic;
    in >> magic;
    if (magic != MagicNumber) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("The file is not a Spreadsheet file."));
        return false;
    }
```

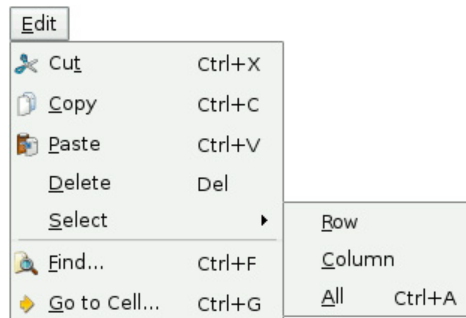
```
    clear();

    quint16 row;
    quint16 column;
    QString str;

    QApplication::setOverrideCursor(Qt::WaitCursor);
    while (!in.atEnd()) {
        in >> row >> column >> str;
        setFormula(row, column, str);
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

`readFile()` fonksiyonu `writeFile()` ile çok benzerdir. Dosyadan okumak için `QFile` kullanırız,fakat bu sefer `QIODevice::WriteOnly`'yi kullanmaktansa, `QIODevice::ReadOnly` bayrağını kullanırız. Sonra, `QDataStream` versiyonunu 9'a ayarlarız. Okuma formatı her zaman yazma ile aynı olmalıdır.

Eğer dosya, başlangıcında doğru sihirli sayıya(magic number) sahipse, hesap çizelgesindeki tüm hücreleri silmek için `clear()`'ı çağırırız ve sonrasında hücre verisini okuruz. Dosya sadece boş olmayan hücrelerin verisini içerdiği için, okumadan tüm hücrelerin temizlenmiş olmasını sağlamak zorundayız.

Edit Menüsünü Gerçekleştirme

Artık, uygulamanın Edit menüsüne ilişkin yuvaları gerçekleştirmeye hazırız. Menü Şekil 4.4'te gösteriliyor.



Şekil 4.4

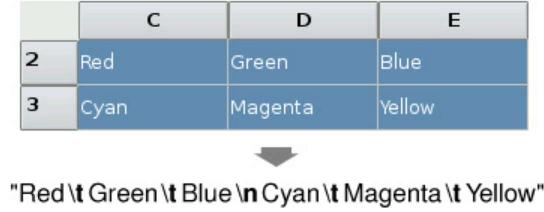
```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

`cut()` yuvası Edit > Cut'a tekabül eder. Cut, Copy ve onu takip eden bir Delete işlemi olduğu için gerçekleştirimi basittir.

```
void Spreadsheet::copy()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str;

    for (int i = 0; i < range.rowCount(); ++i) {
        if (i > 0)
            str += "\n";
        for (int j = 0; j < range.columnCount(); ++j) {
```

```
            if (j > 0)
                str += "\t";
            str += formula(range.topRow() + i, range.leftColumn() + j);
        }
    }
    QApplication::clipboard()->setText(str);
}
```

`copy()` yuvası Edit > Copy'ye tekabül eder. Geçerli seçimi yineler. Her bir seçili hücrenin formülü bir `QString`'e eklenir, satırlar newline('\n') karakteri kullanılarak, sütunlar tab('\t') karakteri kullanılarak ayrılırlar. Bu, Şekil 4.5'te bir örnekle açıklanmıştır.



Şekil 4.5

Sistem panosu Qt'da `QApplication::clipboard()` statik fonksiyonu sayesinde kolaylıkla kullanılabilinir. `QClipboard::setText()`'i çağırarak, bu uygulama ve diğer uygulamaların desteklediği düz metni pano üzerinde kullanılabilir yaparız. Formatımız, ayırıcı olarak kullanılan tab ve newline karakterleri sayesinde, Microsoft Excel'de dâhil olmak üzere çeşitli uygulamalar tarafından anlaşılırdır.

`QTableWidget::selectedRanges()` fonksiyonu seçim aralıklarının bir listesini döndürür. Birden fazla olamayacağını biliyoruz çünkü kurucuda seçim modunu `QAbstractItemView::Contiguous-Selection` olarak ayarlamıştık. Kolaylık sağlaması için, seçim aralığını döndüren bir `selectedRange()` fonksiyonu tanımlarız.

```
QTableWidgetSelectionRange Spreadsheet::selectedRange() const
{
    QList<QTableWidgetSelectionRange> ranges = selectedRanges();
    if (ranges.isEmpty())
        return QTableWidgetSelectionRange();
    return ranges.first();
}
```

Eğer sadece bir seçim varsa, açıkça ve sadece ilkini döndürürüz. `ContiguousSelection` modu geçerli hücreye seçilmiş muamelesi yaptığı için her zaman bir seçim vardır. Fakat hiçbir hücrenin geçerli olmaması gibi bir hata olması olasılığına karşı korunmak için, bu durumunda üstesinden geliriz.

Kod Görünümü:

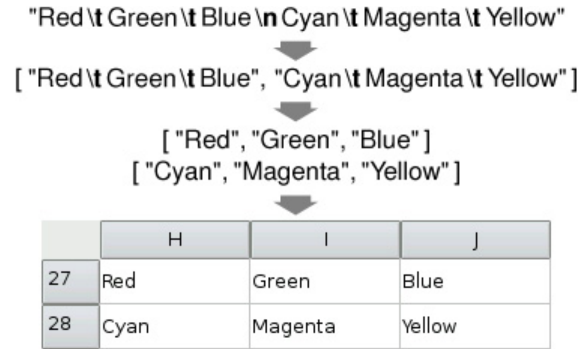
```
void Spreadsheet::paste()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str = QApplication::clipboard()->text();
    QStringList rows = str.split('\n');
    int numRows = rows.count();
    int numColumns = rows.first().count('\t') + 1;

    if (range.rowCount() * range.columnCount() != 1
            && (range.rowCount() != numRows
```

```
                || range.columnCount() != numColumns)) {
        QMessageBox::information(this, tr("Spreadsheet"),
                tr("The information cannot be pasted because the copy "
                   "and paste areas aren't the same size."));
        return;
    }

    for (int i = 0; i < numRows; ++i) {
        QStringList columns = rows[i].split('\t');
        for (int j = 0; j < numColumns; ++j) {
            int row = range.topRow() + i;
            int column = range.leftColumn() + j;
            if (row < RowCount && column < ColumnCount)
                setFormula(row, column, columns[j]);
        }
    }
    somethingChanged();
}
```

`paste()` yuvası Edit > Paste'e tekabül eder. Metni pano üstüne geçiririz ve karakter katarını `QStringList` haline getirmek için `QString::split()` statik fonksiyonunu çağırırız. Her bir satır listede bir karakter katarı halini alır.

Sonra, kopyalama alanının boyutunu belirleriz. Satırların sayısı `QStringList` içindeki karakter katarlarının sayısı, sütunların sayısı ilk satırdaki tab karakterlerinin sayısı artı 1'dir. Eğer sadece bir hücre seçilmişse, onu yapıştırma alanının sol üst köşesi olarak kullanırız; aksi halde, geçerli seçimi yapıştırma alanı olarak kullanırız.

Yapıştırma işlemini yerine getirmek için satırları yineler ve `QString::split()` kullanarak hücrelere ayırırız, fakat bu sefer tab karakterini ayırıcı olarak kullanırız. Şekil 4.6 bu adımları bir örnekle açıklıyor.



Şekil 4.6

```
void Spreadsheet::del()
{
    QList<QTableWidgetItem *> items = selectedItems();
    if (!items.isEmpty()) {
        foreach (QTableWidgetItem *item, items)
            delete item;
        somethingChanged();
    }
}
```

`del()` yuvası Edit > Delete'e tekabül eder. Eğer seçili öğeler varsa, fonksiyon onları siler ve `something-Changed()` fonksiyonunu çağırır. Seçimdeki `Cell` nesnelerinin her birini hücrelerden temizlemek için

delete kullanmak yeterlidir. QTableWidget, QTableWidgetItem'larının silindiğinin farkına varır ve eğer öğelerin hiçbiri görünür değilse otomatik olarak kendisini yeniden çizer. Eğer cell()'i silinmiş bir hücrenin konumuyla çağırırsak, boş bir işaretçi döndürecektir.

```
void Spreadsheet::selectCurrentRow()
{
    selectRow(currentRow());
}

void Spreadsheet::selectCurrentColumn()
{
    selectColumn(currentColumn());
}
```

selectCurrentRow() ve selectCurrentColumn() fonksiyonları Edit > Select > Row ve Edit > Select > Column menü seçeneklerine tekabül ederler. Gerçekleştirimleri QTableWidget'ın selectRow() ve selectColumn() fonksiyonlarına dayanır. Edit > Select > All ardındaki işlev, QTableWidget'tan miras alınan QAbstractItemView::selectAll() fonksiyonuyla sağlandığı için bizim gerçekleştirmemiz gerekmez.

```
void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() + 1;

    while (row < RowCount) {
        while (column < ColumnCount) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            ++column;
        }
        column = 0;
        ++row;
    }
    QApplication::beep();
}
```

findNext() yuvası şu şekilde işler: Eğer geçerli hücre C24 ise, D24, E24, …, Z24, sonra A25, B25, C25, …, Z25, ve Z999'a kadar arar.

Eğer bir eşleşme bulursak, geçerli seçimi temizler, hücre göstergesini eşleşen hücreye taşır ve Spreadsheet penceresini aktif ederiz. Eğer hiçbir eşleşme bulunmazsa, ses çıkararak (beep) aramanın başarısız sonuçlandığını bildiririz.

```
void Spreadsheet::findPrevious(const QString &str,
                               Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() - 1;

    while (row >= 0) {
        while (column >= 0) {
            if (text(row, column).contains(str, cs)) {
```

```
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            --column;
        }
        column = ColumnCount - 1;
        --row;
    }
    QApplication::beep();
}
```

`findPrevious()` yuvası, geriye doğru yinelemesi ve A1 hücresinde durması dışında `findNext()` ile benzerdir.

Diğer Menüleri Gerçekleştirme

Şimdi Tools ve Options menülerinin yuvalarını gerçekleştireceğiz. Bu menüler Şekil 4.7'de gösteriliyor.

Şekil 4.7

```
void Spreadsheet::recalculate()
{
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            if (cell(row, column))
                cell(row, column)->setDirty();
        }
    }
    viewport()->update();
}
```

`recalculate()` yuvası Tools > Recalculate'e tekabül eder. Gerektiğinde Spreadsheet tarafından otomatik olarak çağrılır.

Tüm hücreler üzerinde yineler ve her hücreyi "yeniden hesaplanması gerekli(requiring recalculation)" ile işaretlemek için `setDirty()`'yi çağırırız. `QTableWidget`, hesap çizelgesinde göstereceği değeri elde etmek için bir `Cell` üzerinde `text()`'i çağırdığında, değer yeniden hesaplanmış olacaktır.

Sonra, tüm hesap çizelgesini yeniden çizmesi için görüntü kapısı üzerinde `update()`'i çağırırız. `QTableWidget`'taki yeniden çiz kodu(repaint code) daha sonra, her görünür hücrenin görüntüleyeceği değeri elde etmek için `text()`'i çağırır. Çünkü her hücre için `setDirty()`'yi çağırdık; `text()` çağrıları henüz hesaplanmış bir değeri kullanacak. Hesaplama, görünmeyen hücrelerin de yeniden hesaplanmasını gerektirebilir ve `Cell` sınıfı tarafından yapılır.

```
void Spreadsheet::setAutoRecalculate(bool recalc)
{
    autoRecalc = recalc;
    if (autoRecalc)
        recalculate();
}
```

`setAutoRecalculate()` yuvası Options > Auto-Recalculate'e tekabül eder. Özellik açılmışsa, tüm hesap çizelgesini, güncel olduğundan emin olmak için derhal yeniden hesaplarız; daha sonra, `recalculate()` `somethingChanged()`'den otomatik olarak çağırılır.

Options > Show Grid için herhangi bir şey gerçekleştirmemiz gerekmez, çünkü `QTableWidget` zaten `QTableView`'den miras alınan bir `setShowGrid()` yuvasına sahiptir. Tek kalan, `MainWindow::sort()`'tan çağırılan `Spreadsheet::sort()`'tur:

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
    QList<QStringList> rows;
    QTableWidgetSelectionRange range = selectedRange();
    int i;

    for (i = 0; i < range.rowCount(); ++i) {
        QStringList row;
        for (int j = 0; j < range.columnCount(); ++j)
            row.append(formula(range.topRow() + i,
                               range.leftColumn() + j));
        rows.append(row);
    }

    qStableSort(rows.begin(), rows.end(), compare);

    for (i = 0; i < range.rowCount(); ++i) {
        for (int j = 0; j < range.columnCount(); ++j)
            setFormula(range.topRow() + i, range.leftColumn() + j,
                       rows[i][j]);
    }

    clearSelection();
    somethingChanged();
}
```

Sıralama geçerli seçim üzerinde işler ve `compare` nesnesinde saklanan sıralama anahtarı ve sıralama düzenine göre satırları yeniden düzenler. Verinin her bir satırını bir `QStringList` ile ifade ederiz ve seçimi satırların bir listesi şeklinde saklarız. Qt'un `qStableSort()` algoritmasını kullanırız ve basitlik olsun diye değere göre sıralamaktansa formüle göre sıralarız. İşlem, Şekil 4.8 ve 4.9'da örneklendirilerek gösteriliyor.

	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger", "Dijkstra", "1930-05-11"]
1	["Tony", "Hoare", "1934-01-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Donald", "Knuth", "1938-01-10"]

Şekil 4.8

index	value
0	["Donald", "Knuth", "1938-01-10"]
1	["Edsger", "Dijkstra", "1930-05-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Tony", "Hoare", "1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

Şekil 4.9

qSableSort() fonksiyonu bir başlama yineleyicisi(begin iterator), bir bitme yineleyicisi(end iterator) ve bir karşılaştırma fonksiyonu kabul eder. Karşılaştırma fonksiyonu, iki argüman alan(iki QStringList) alan ve eğer ilk argüman ikinci argümandan "daha küçük" ise true, aksi halde false döndüren bir fonksiyondur. Karşılaştırma fonksiyonu olarak aktardığımız compare nesnesi gerçek bir fonksiyon değildir, bir fonksiyonmuş gibi kullanılabilir.

qStableSort()'u uyguladıktan sonra, veriyi tablonun gerisine taşır, seçimi temizler ve something-Changed()'i çağırırız.

spreadsheet.h içinde, SpreadsheetCompare şöyle tanımlanmıştı:

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};
```

SpreadsheetCompare sınıfı özeldir, çünkü bir () operatörü gerçekleştirir. Bu bize sınıfı bir fonksiyonmuş gibi kullanma imkânı verir. Böyle sınıflara fonksiyon nesne (function object) ya da *functors* adı verilir. Bir fonksiyon nesnenin nasıl çalıştığını anlamak için basit bir örnekle başlayacağız:

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
}
```

Square sınıfı, parametresinin karesini döndüren bir operator()(int) fonksiyonu sağlar. Fonksiyonu compute(int) yerine operator()(int) olarak isimlendirerek, Square tipindeki bir nesneyi bir fonksiyonmuş gibi kullanabilme yeteneği kazandık.

```
Square square;
int y = square(5);
// y equals 25
```

Şimdi SpreadsheetCompare'i içeren bir örnek görelim:

```
QStringList row1, row2;
SpreadsheetCompare compare;
...
if (compare(row1, row2)) {
    // row1 is less than row2
}
```

compare nesnesi, yalın bir compare() fonksiyonuymuş gibi kullanılabilir. Ek olarak gerçekleştirimi üye değişkenlerinde saklanan tüm sıralama anahtarlarına ve sıralama düzenlerine erişebilir.

Bu tasarıya bir alternatif, sıralama anahtarlarını ve sıralama düzenlerini global değişkenlerde saklamak ve yalın bir `compare()` fonksiyonu kullanmak olacaktır. Ancak, global değişkenlerle hesaplama yapmak çirkindir ve "hoş" hatalara yol açabilir.

İşte, iki hesap çizelgesi satırını karşılaştırmakta kullanılan fonksiyonun gerçekleştirimi:

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                                    const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
                } else {
                    return row1[column] > row2[column];
                }
                }
        }
    }
    return false;
}
```

Operatör, eğer ilk satır ikinci satırdan daha küçük ise `true`; aksi halde `false` döndürür. `qStableSort()` fonksiyonu sıralamayı yapmak için bu fonksiyonun sonucunu kullanır.

`SpreadsheetCompare` nesnesinin `key` ve `ascending` dizileri `MainWindow::sort()` fonksiyonu içinde doldurulurlar. Her bir anahtar bir sütun indeksi ya da -1("None") tutar.

İki satırdaki ilişkili hücre girdilerini her bir anahtar için sırayla karşılaştırırız. Bir farklılık bulur bulmaz, `true` ve `false` değerlerinden uygun olan birini döndürürüz. Eğer tüm karşılaştırmalar eşit çıkıyorsa, `false` döndürürüz.

Böylece, Spreadsheet sınıfını tamamladık. Sıradaki kısımda `Cell` sınıfını irdeleyeceğiz.

QTableWidgetItem Altsınıfı Türetme

`Cell` sınıfı `QTableWidgetItem` sınıfından türetilir. Sınıf, `Spreadsheet` ile iyi çalışması için tasarlanır, fakat bu sınıfa özel bir bağımlılığı yoktur ve teoride her `QTableWidget` içinde kullanılabilir. İşte başlık dosyası:

Kod Görünümü:

```
#ifndef CELL_H
#define CELL_H

#include <QTableWidgetItem>

class Cell : public QTableWidgetItem
{
public:
    Cell();

    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
```

```
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();

private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;

    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};

#endif
```

Cell sınıfı QTableWidgetItem'ı iki private değişken ekleyerek genişletir:

- cachedValue hücre değerlerini bir QVariant olarak önbellekler.
- Eğer önbelleklenen değer güncel değilse, cacheIsDirty true'dur.

QVariant kullanırız, çünkü bazı hücreler QString bir değere sahipken, bazı hücreler double bir değere sahiptir.

cachedValue ve cacheIsDirty değişkenleri C++ mutable anahtar kelimesi kullanılarak bildirilirler. Bu bize bu değişkenleri const fonksiyonlarla değiştirebilme imkânı verir. Alternatif olarak, değeri text() her çağrıldığında yeniden hesaplayabilirdik, fakat bu gereksiz ve verimsiz olacaktır.

Sınıf tanımında hiç Q_OBJECT makrosu olmadığına dikkat edin. Cell, sinyalleri ya da yuvaları olmayan sade bir C++ sınıfıdır. Aslında, Cell sınıfında sinyallere ve yuvalara sahip olamayız, çünkü QTableWidget-Item, QObject'ten türetilmemiştir. Qt'un öğe(item) sınıfları QObject'ten türetilmez. Eğer sinyallere ve yuvalara ihtiyaç duyulursa, öğeleri içeren parçacıklar içinde ya da istisnai olarak QObject çoklu mirası ile gerçekleştirilirler.

İşte, cell.cpp'nin başlangıcı:

```
#include <QtGui>

#include "cell.h"

Cell::Cell()
{
    setDirty();
}
```

Kurucuda tek ihtiyacımız önbelleği bozuk(dirty) olarak ayarlamaktır. Bir ebeveyn aktarmamız gerekmez; hücre setItem() ile bir QTableWidget içine eklendiğinde, QTableWidget onun sahipliğini otomatik olarak üstlenecektir.

Her QTableWidgetItem, her bir veri "rolü(role)" için bir QVariant'a kadar veri tutabilir. En yaygın kullanılan roller Qt::EditRole ve Qt::DisplayRole'dür. Düzenleme rolü(edit role) düzenlenecek veri için, görüntüleme rolü(display role) görüntülenecek veri için kullanılır. Her ikisi için de veri genellikle

aynıdır, fakat Cell'de düzenleme rolü hücrenin formülüne, görüntüleme rolü hücrenin değerine (formül değerlendirmesinin sonucu) tekabül eder.

```
QTableWidgetItem *Cell::clone() const
{
    return new Cell(*this);
}
```

clone() fonksiyonu QTableWidget tarafından yeni bir hücre oluşturması gerektiğinde çağrılır; mesela, kullanıcı daha önce kullanmadığı boş bir hücreye bir şeyler yazmaya başladığında. QTableWidget-Item::setItemPrototype()'a aktarılan örnek, klonlanan öğedir.

```
void Cell::setFormula(const QString &formula)
{
    setData(Qt::EditRole, formula);
}
```

setFormula() fonksiyonu hücrenin formülünü ayarlar. Düzenleme rolü ile yapılan setData() çağrısından dolayı, bir uygunluk fonksiyonudur. Spreadsheet::setFormula()'dan çağrılır.

```
QString Cell::formula() const
{
    return data(Qt::EditRole).toString();
}
```

formula() fonksiyonu Spreadsheet::formula()'dan çağrılır. setFormula() gibi, fakat bu sefer öğenin EditRole verisine erişen bir uygunluk fonksiyonudur.

```
void Cell::setData(int role, const QVariant &value)
{
    QTableWidgetItem::setData(role, value);
    if (role == Qt::EditRole)
        setDirty();
}
```

Eğer yeni bir formüle sahipsek, cacheIsDirty'yi bir dahaki text() çağrısında hücrenin yeniden hesaplanmasını sağlamak için true olarak ayarlarız. text() fonksiyonu QTableWidgetItem tarafından sağlanan bir uygunluk fonksiyonudur ve data(Qt::DisplayRole).tostring() çağrısına denktir.

```
void Cell::setDirty()
{
    cacheIsDirty = true;
}
```

setDirty() fonksiyonu bir hücrenin değerini yeniden hesaplanmaya zorlamak için çağrılır. Sadece cacheIsDirty'yi true olarak ayarlar ve bu "cachedValue artık güncel değil" anlamını taşır. Yeniden hesaplama, gerekli oluncaya kadar icra edilmez.

```
QVariant Cell::data(int role) const
{
    if (role == Qt::DisplayRole) {
        if (value().isValid()) {
            return value().toString();
        } else {
            return "####";
        }
```

```
    } else if (role == Qt::TextAlignmentRole) {
        if (value().type() == QVariant::String) {
            return int(Qt::AlignLeft | Qt::AlignVCenter);
        } else {
            return int(Qt::AlignRight | Qt::AlignVCenter);
        }
    } else {
        return QTableWidgetItem::data(role);
    }
}
```

data() fonksiyonu QTableWidgetItem'dan uyarlanır. Eğer Qt::DisplayRole ile çağrılmışsa hesap çizelgesi içinde gösterilecek metni, eğer Qt:EditRole ile çağrılmışsa formülü döndürür. Eğer Qt::TextAlignmentRole ile çağrılmışsa, uygun bir hizalanış döndürür. DisplayRole durumunda, hücrenin değerini hesaplamada value()'ya güvenir. Eğer değer geçersizse (çünkü formül yanlıştır), "####" döndürürüz.

data() içinde kullanılan Cell::value() fonksiyonu bir QVariant döndürür. Bir QVariant, örneğin double ve QString gibi farklı tipteki değerleri saklayabilir ve fonksiyonların değişkeni diğer tiplere dönüştürmesini sağlar. Örneğin, double bir değişken için toString() çağrısı, double'ı temsil eden bir karakter katarı üretir. Varsayılan kurucu kullanılarak oluşturulmuş bir QVariant "boş (invalid)" bir değişkendir.

Kod Görünümü:

```
const QVariant Invalid;
QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;

        QString formulaStr = formula();
        if (formulaStr.startsWith('\'')) {
            cachedValue = formulaStr.mid(1);
        } else if (formulaStr.startsWith('=')) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            expr.append(QChar::Null);

            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (expr[pos] != QChar::Null)
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);
            if (ok) {
                cachedValue = d;
            } else {
                cachedValue = formulaStr;
            }
        }
    }
    return cachedValue;
}
```

value() private fonksiyonu hücrenin değerini döndürür. Eğer cacheIsDirty true ise, değeri yeniden hesaplamamız gerekir.

Eğer formül tek tırnakla başlarsa (örneğin "'12345"), tek tırnak 0 konumundadır ve değer, karakter katarının 1 konumundan sonuna kadar olan kısmıdır.

Eğer formül eşittir işareti("=") ile başlarsa, karakter katarını 1 konumundan alırız ve içerebileceği her boşluğu sileriz. Sonra deyimin değerini hesaplamak için evalExpression()'ı çağırırız. pos argümanı referans olarak aktarılır ve çözümlemeye başlanacak yerdeki karakterin konumu belirtir. evalExpression() çağrısından sonra, eğer çözümleme başarılmışsa, pos konumundaki karakter eklediğimiz QChar::Null karakteri olmalıdır. Eğer çözümleme başarısızsa, cachedValue'yu Invalid olarak ayarlarız.

Eğer formül tek tırnak ya da bir eşittir işareti ile başlamıyorsa, toDouble()'ı kullanarak formülü bir kayan noktalı(floating-point) sayıya dönüştürmeyi deneriz. Eğer dönüştürme işe yararsa, cachedValue'yu sonuçtaki sayı olarak ayarlarız; aksi halde, cahedValue'yu formül karakter katarı olarak ayarlarız. Örneğin, "1.50" formülü toDouble()'ın ok'i true olarak ayarlamasına ve 1.5 döndürmesine neden olurken, "Word Population" formülü toDouble()'ın ok'i false olarak ayarlamasına ve 0.0 döndürmesine neden olur.

toDouble()'a bir bool işaretçisi vererek, dönüşümün başarısını kontrol edebiliriz.

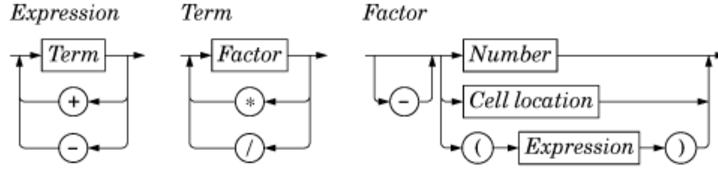
value() fonksiyonu const bildirilir. Derleyicinin bize cachedValue ve cacheIsValid'i const fonksiyonlarda değiştirebilmemize izin vermesi için, onları mutable değişkenler olarak bildirdik. value()'yu non-const yapmak ve mutable anahtar kelimelerini silmek cazip gelebilir, fakat bu derlenmeyecektir çünkü value()'yu const bir fonksiyon olan data()'dan çağırırız.

Formüllerin ayrıştırılması hariç, artık Spreadsheet uygulamasını tamamladık. Bu kısmın geri kalanı evalExpression() fonksiyonunu ve ona yardımcı iki fonksiyon olan evalTerm() ve evalFactor() fonksiyonlarını kapsar. Kod biraz karmaşıktır, fakat uygulamayı tamamlanması açısından gereklidir. Kod GUI programlamayla ilgili olmadığı için, gönül rahatlığıyla bu kısmı geçip Bölüm 5'ten okumaya devam edebilirsiniz.

evalExpression() fonksiyonu bir hesap çizelgesi deyiminin(expression) değerini döndürür. Bir deyim, '+' veya '-' operatörlerince ayrılmış bir ya da daha fazla terim(term) olarak tanımlanır. Terimler ise '*' veya '/' operatörleriyle ayrılmış bir ya da birden fazla faktör(factor) olarak tanımlanırlar. Deyimleri terimlere, terimleri faktörlere ayırarak, operatörlere doğru öncelik sırası uygulanmasını sağlarız.

Örneğin "2*C5+D6", birinci terimi "2*C5", ikinci terimi "D6" olan bir deyimdir. "2*C5" terimi birinci faktör olarak "2"ye ikinci faktör olarak da "C5"e sahiptir. "D6" terimi ise tek bir faktörden oluşur: "D6". Bir faktör; bir sayı ("2"), bir hücre adresi ("C5") ya da parantez içinde bir deyim olabilir.

Hesap çizelgesi deyimlerinin sözdizimi Şekil 4.10'da tanımlanmıştır. Gramerdeki her bir sembol için, onu ayrıştıran bir uygunluk üye fonksiyonu vardır. Ayrıştırıcılar, *recursive-descent parsers* olarak adlandırılan tarzda yazılırlar.



Şekil 4.10

Bir Deyimi ayrıştıran `evalExpression()` fonksiyonuyla başlayalım:

Kod Görünümü:

```
QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '+' && op != '-')
            return result;
        ++pos;

        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double
                && term.type() == QVariant::Double) {
            if (op == '+') {
                result = result.toDouble() + term.toDouble();
            } else {
                result = result.toDouble() - term.toDouble();
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

Önce, ilk terimin değerini almak için evalTerm()'i çağırırız. Sonraki karakter '+' ya da '-' ise, ikinci defa `evalTerm()`'i çağırarak devam ederiz; aksi halde, deyim tek bir terimden oluşuyor demektir ve bu durumda onun değerini tüm deyimin değeri olarak döndürürüz. İlk iki terimin değerini elde ettikten sonra operatöre göre işlemin sonucunu hesaplarız. Eğer iki terim de bir `double` ise, sonucu bir `double` olarak hesaplarız; aksi halde, sonucu `Invalid` olarak ayarlarız.

Hiç terim kalmayıncaya kadar böyle devam ederiz. Bu doğru çalışır, çünkü toplama ve çıkarma sol birleşimli(left-associative) aritmetik işlemlerdir. Yani "1-2-3", "1-(2-3)"e değil, "(1-2)-3"e eşittir.

Kod Görünümü:

```
QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '*' && op != '/')
            return result;
        ++pos;

        QVariant factor = evalFactor(str, pos);
```

```
        if (result.type() == QVariant::Double
                && factor.type() == QVariant::Double) {
            if (op == '*') {
                result = result.toDouble() * factor.toDouble();
            } else {
                if (factor.toDouble() == 0.0) {
                    result = Invalid;
                } else {
                    result = result.toDouble() / factor.toDouble();
                }
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

evalTerm(), çarpma ve bölme ile uğraşması dışında, evalExpression()'a çok benzer. evalTerm()'deki tek incelik, -bazı işlemcilerde hataya neden olduğu için- sıfıra bölmeden kaçınmaktır.

Kod Görünümü:

```
QVariant Cell::evalFactor(const QString &str, int &pos) const
{
    QVariant result;
    bool negative = false;

    if (str[pos] == '-') {
        negative = true;
        ++pos;
    }
    if (str[pos] == '(') {
        ++pos;
        result = evalExpression(str, pos);
        if (str[pos] != ')')
            result = Invalid;
        ++pos;
    } else {
        QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
        QString token;

        while (str[pos].isLetterOrNumber() || str[pos] == '.') {
            token += str[pos];
            ++pos;
        }
        if (regExp.exactMatch(token)) {
            int column = token[0].toUpper().unicode() - 'A';
            int row = token.mid(1).toInt() - 1;

            Cell *c = static_cast<Cell *>(
                              tableWidget()->item(row, column));
            if (c) {
                result = c->value();
            } else {
                result = 0.0;
            }
        } else {
            bool ok;
            result = token.toDouble(&ok);
```

```
            if (!ok)
                result = Invalid;
        }
    }

    if (negative) {
        if (result.type() == QVariant::Double) {
            result = -result.toDouble();
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

evalFactor() fonksiyonu evalExpression() ve evalTerm()'e göre biraz daha karmaşıktır. Faktörün negatif olup olmadığını not ederek başlarız. Sonra, bir açma paranteziyle başlayıp başlamadığına bakarız. Eğer öyleyse, evalExpression()'ı çağırarak, parantezin içeriğinin bir deyim olup olmadığına bakarız. Parantez içine alınmış deyimi ayrıştırırken, evalExpression() evalTerm()'i, evalTerm() evalFactor()'ı, o da tekrar evalExpression()'ı çağırır. Bu, ayrıştırıcı içinde bir özyineleme ortaya çıkarır.

Eğer faktör iç içe bir deyim değilse, sıradaki dizgeciği(token) ayıklarız (bir hücre adresi ya da bir sayı olmalıdır). Eğer dizgecik QRegExp ile eşleşirse, onu bir hücre referansı olarak alırız ve verilen adresteki hücre için value()'yu çağırırız. Hücre hesap çizelgesi içinde herhangi bir yerde olabilir ve diğer hücrelerle bağımlılığı olabilir. Bağımlılık bir problem değildir; sadece daha fazla value() çağrısını ve ("bozuk(dirty)" hücreler için) tüm bağlı hücreler hesaplanmış olana dek daha fazla ayrıştırmayı tetikleyecektir. Eğer dizgecik bir hücre adresi değilse, onu bir sayı olarak kabul ederiz.

BÖLÜM 5: ÖZEL PARÇACIKLAR OLUŞTURMA

Bu bölüm Qt'u kullanarak özel parçacıklar(custom widgets) oluşturmayı açıklar. Özel parçacıklar, var olan bir Qt parçacığından altsınıf türeterek ya da direkt olarak `QWidget`'tan altsınıf türeterek oluşturulabilirler. Biz her iki yaklaşımı da örnekle açıklayacağız. Ayrıca, özel parçacıkları Qt Designer'a entegre ederek yerleşik Qt parçacıkları gibi kullanabilmeyi de göreceğiz.

Qt Parçacıklarını Özelleştirme

Bazen, bir Qt parçacığının niteliklerinin, Qt Designer'da ayarlayarak ya da fonksiyonunu çağırarak, daha fazla özelleştirilmeye ihtiyaç duyulduğu durumlarla karşılaşırız. Basit ve direkt bir çözüm, ilgili parçacık sınıfından altsınıf türetmek ve ihtiyaçlara uygun olarak uyarlamaktır.

Bu kısımda, Şekil 5.1'de görülen onaltılı(hexadecimal) döndürme kutusunu(spin box) geliştireceğiz. `QSpinBox` sadece onlu(decimal) tamsayıları destekler, fakat ondan bir altsınıf türeterek, onaltılı değerleri kabul etmesini ve göstermesini sağlamak oldukça kolaydır.

Şekil 5.1

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H

#include <QSpinBox>

class QRegExpValidator;

class HexSpinBox : public QSpinBox
{
    Q_OBJECT
public:
    HexSpinBox(QWidget *parent = 0);

protected:
    QValidator::State validate(QString &text, int &pos) const;
    int valueFromText(const QString &text) const;
    QString textFromValue(int value) const;

private:
    QRegExpValidator *validator;
};

#endif
```

`HexSpinBox`, işlevlerinin birçoğunu `QSpinBox`'tan miras alır. Tipik bir kurucu sağlar ve `QSpinBox`'tan aldığı üç sanal fonksiyonu uyarlar.

```
#include <QtGui>
```

```
#include "hexspinbox.h"

HexSpinBox::HexSpinBox(QWidget *parent)
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
```

Varsayılan aralığı, QSpinBox'ın varsayılan aralığı(0'dan 99'a) yerine bir onaltılı döndürme kutusu için daha uygun olan "0'dan 255'e (0x00'dan 0xFF'ye)" olarak ayarlarız.

Kullanıcı, bir döndürme kutusunun değerini onun yukarı/aşağı oklarını tıklayarak ya da satır editörüne bir değer girerek değiştirebilir. Son olarak, kullanıcı girdilerini onaltılı sayılarla sınırlandırmak istiyoruz. Bunu başarmak için; bir ile sekiz arasında, her biri {'0', …, '9', 'A', …, 'F', 'a', …, 'f'} kümesinden olan karakterler kabul eden bir QRegExpValidator kullanırız.

```
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}
```

Bu fonksiyon, metnin doğru girildiğini görmek için QSpinBox tarafından çağrılır. Üç olası sonuç vardır: Invalid (metin düzenli deyimle eşleşmiyordur), Intermediate (metin geçerli bir değerin olası bir parçasıdır), ve Acceptable (metin geçerlidir). QRegExpValidator uygun bir validate() fonksiyonuna sahiptir, böylece basitçe ona yapılan çağrının sonucunu döndürürüz. Teoride, döndürme kutusunun aralığının dışına taşan değerler için Invalid ya da Intermediate döndürmeliyiz, fakat QSpinBox bu durumu yardım olmaksızın fark edecek derecede zekidir.

```
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
```

textFromValue() fonksiyonu bir tamsayı değeri bir karakter katarına dönüştürür. QSpinbox bu fonksiyonu, kullanıcı döndürme kutusunun yukarı/aşağı oklarına bastığında, satır editörünü güncellemek için çağırır. Değeri, küçük harfli onaltılıya dönüştürmek için ikinci bir 16 argümanı ile QString::number() statik fonksiyonunu çağırırız ve sonucu büyük harfli yapmak için QString::toUpper()'ı çağırırız.

```
int HexSpinBox::valueFromText(const QString &text) const
{
    bool ok;
    return text.toInt(&ok, 16);
}
```

valueFromText() fonksiyonu, ters dönüşümü (bir karakter katarından bir tamsayı değere) yerine getirir. QSpinBox bu fonksiyonu, kullanıcı döndürme kutusunun editör bölümüne bir değer girdiğinde ve Enter'a bastığında çağırır. QString::toInt() fonksiyonunu, yine 16 tabanını kullanarak, bir metni bir tamsayı değere dönüştürmede kullanırız. Eğer karakter katarı geçerli bir onaltılı sayı değilse, ok false olarak ayarlanır ve toInt() 0 döndürür. Burada, bu olasılığı düşünmek zorunda değiliz, çünkü geçerlilik denetleyicisi sadece geçerli onaltılı karakter katarları girilmesine izin verir. Aptal bir değişkenin(ok) adresini ayrıştırmak yerine, toInt()'e ilk argüman olarak boş bir işaretçi de aktarabilirdik.

Böylece, onaltılı döndürme kutusunu bitirdik. Diğer Qt parçacıklarını özelleştirme de aynı modeli izler: Uygun bir Qt parçacığı seç, ondan altsınıf türet ve bazı sanal fonksiyonların davranışlarını değiştirerek uyarla.

QWidget Altsınıfı Türetme

Birçok özel parçacık, var olan parçacıkların bir birleşimidir. Var olan parçacıkları birleştirerek inşa edilen özel parçacıklar genelde Qt Designer'da geliştirilirler:

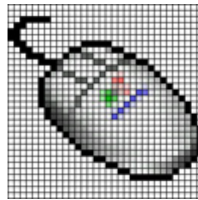
- "Widget" şablonunu kullanarak yeni bir form oluştur.
- Gerekli parçacıkları forma ekle ve onları yerleştir.
- Sinyal ve yuva bağlantılarını ayarla.
- Eğer sinyal ve yuvaların başarabildiğinin ötesinde bir davranış lazımsa, gerekli kodu `QWidget` ve `uic`-üretilmiş sınıflarından türetilen bir sınıf içine yaz.

Elbette var olan parçacıkların birleştirilmesi tamamıyla kod içinde de yapılabilir. Hangi yaklaşım seçilirse seçilsin, sonuçta ortaya çıkacak sınıf bir `QWidget` altsınıfıdır.

Eğer parçacık kendi sinyal ve yuvalarına sahip değilse ve herhangi bir sanal fonksiyon da uyarlanmamışsa, parçacığı, var olan parçacıkları bir altsınıf olmaksızın birleştirerek toparlamak da mümkündür. Bu yaklaşımı Bölüm 1'de, bir `QWidget`, bir `QSpinBox` ve bir `QSlider` ile Age uygulamasını oluştururken kullanmıştık. Yine de, kolaylıkla `QWidget`'tan bir altsınıf türetebilir ve altsınıfın kurucusu içinde `QSpinBox` ve `QSlider`'ı oluşturabilirdik.

Qt parçacıklarının hiçbiri eldeki işe uygun olmadığında ve arzu edilen sonucu elde etmek için var olan parçacıkları birleştirmenin ya da uyarlamanın hiçbir yolu yoksa dahi, istediğimiz parçacığı oluşturabiliriz. Bu, `QWidget`'tan altsınıf türeterek ve birkaç olay işleyiciyi(event handler) parçacığı çizmeye ve fare tıklamalarını yanıtlamaya uyarlayarak başarılabilir. Bu yaklaşım bize, parçacığımızın görünümünü ve davranışını tanımlamada ve kontrol etmede tam bir bağımsızlık sağlar. Qt'un yerleşik parçacıkları (örneğin `QLabel`, `QPushButton` ve `QTableWidget`) bu yolla gerçekleştirilmişlerdir. Eğer Qt'un içinde var olmasalardı, kendimiz, `QWidget` tarafından sağlanan, tamamıyla platformdan bağımsız public fonksiyonları kullanarak, oluşturmamız da mümkündü.

Bir özel parçacığın bu yaklaşımı kullanarak nasıl oluşturduğunu açıklamak için, Şekil 5.2'de görülen `IconEditor` parçacığını oluşturacağız. `IconEditor`, simge(icon) düzenleme programı olarak kullanılabilecek bir parçacıktır.



Şekil 5.2

Başlık dosyasının kritiğini yaparak başlayalım.

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H

#include <QColor>
```

```
#include <QImage>
#include <QWidget>

class IconEditor : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
    IconEditor(QWidget *parent = 0);

    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;
```

`IconEditor` sınıfı üç adet özel nitelik(property) bildirmek için `Q_PROPERTY()` makrosunu kullanır: `penColor`, `iconImage` ve `zoomFactor`. Her bir nitelik bir veri tipine, bir "oku(read)" fonksiyonuna ve isteğe bağlı bir "yaz(write)" fonksiyonuna sahiptir. Örneğin, `penColor` niteliğinin veri tipi `QColor`'dır ve `penColor()` ve `setPenColor()` fonksiyonları kullanılarak okunabilir ve yazılabilir.

Parçacığı Qt Designer kullanarak oluşturduğumuzda, özel nitelikler Qt Designer'ın nitelik editöründe `QWidget`'tan miras alınan niteliklerin altında görünür. Nitelikler `QVariant` tarafından desteklenen her tipte olabilir. `Q_OBJECT` makrosu, nitelikler tanımlayan sınıflar için gereklidir.

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private:
    void setImagePixel(const QPoint &pos, bool opaque);
    QRect pixelRect(int i, int j) const;

    QColor curColor;
    QImage image;
    int zoom;
};

#endif
```

`IconEditor` `QWidget`'tan aldığı üç protected fonksiyonu yeniden uyarlar ve aynı zamanda birkaç private fonksiyon ve değişkene sahiptir. Üç private değişken üç niteliğin değerlerini tutar.

Gerçekleştirim dosyası `IconEditor`'ın kurucusuyla başlar:

```
#include <QtGui>

#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent)
    : QWidget(parent)
{
```

```
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);

    curColor = Qt::black;
    zoom = 8;

    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgba(0, 0, 0, 0));
}
```

Kurucu, Qt::WA_StaticContents niteliği ve setSizePolicy() çağrısı gibi bazı incelikli yönlere sahiptir. Onları kısaca ele alacağız.

Kalem rengi siyah(black) olarak ayarlanır. Yakınlaştırma katsayısı(zoom factor) 8 olarak ayarlanır; bu, simgedeki her bir pikselin 8 x 8'lik bir kare olarak yorumlanacağı anlamına gelir.

Simge verisi image üye değişkeninde saklanır ve setIconImage() ve iconImage() fonksiyonlarından doğrudan erişilebilirdir. Bir simge düzenleme programı, genellikle kullanıcı bir simge dosyasını açtığında setIconImage()'ı ve kullanıcı onu kaydetmek istediğinde simgeye erişmek için iconImage()'ı çağıracaktır. image değişkeni QImage tipindedir. Onu, 16 x 16 piksele ve yarı saydamlığı(semi-transparency) destekleyen 32-bit ARGB formatına ilklendiririz. Resim verisini, resmi saydam bir renkle doldurarak temizleriz.

QImage sınıfı, bir resmi donanımdan bağımsız bir tarzda saklar. 1-bit, 8-bit ve 32-bit derinliklerini kullanamaya ayarlanabilir. 32-bit derinliğinde bir resim bir pikselin kırmızı, yeşil ve mavi bileşenlerinden her biri için 8 bit kullanır. Kalan 8 bit pikselin alfa bileşenini (opacity) saklar. Örneğin, katıksız bir kırmızı renginin kırmızı, yeşil, mavi ve alfa bileşenleri 255, 0, 0 ve 255 değerlerine sahiptir. Qt'da bu renk şu şekilde belirtilebilir:

```
QRgb red = qRgba(255, 0, 0, 255);
```

Ya da, madem renk saydam değil, dördüncü argüman olmadan şu şekilde de gösterilebilir:

```
QRgb red = qRgb(255, 0, 0);
```

QRgb sadece unsigned int değerler tutabilen bir tip tanımıdır ve qRgb() ve qRgba fonksiyonları, argümanlarını tek bir 32-bit ARGB tamsayı değer halinde birleştiren yerel fonksiyonlardır. Dolayısıyla şöyle yazmak da mümkündür:

```
QRgb red = 0xFFFF0000;
```

Birinci FF alfa bileşenine ikinci FF kırmızı bileşenine tekabül eder. IconEditor'ın kurucusunda alfa bileşeni olarak 0 olarak kullanarak QImage'ı saydam bir renkle doldururuz.

Qt, renkleri saklamak için iki tip sağlar: QRgb ve QColor. QColor birçok kullanışlı fonksiyonuyla Qt içinde renkleri saklamada yaygın olarak kullanılan bir sınıfken, QRgb sadece QImage içinde 32-bit piksel veriyi saklamada kullanılmak üzere tanımlanmış bir tiptir. IconEditor parçacığında, QRgb'yi sadece QImage ile ilgilendiğimizde kullanırız. penColor niteliği de dâhil olmak üzere diğer her şey için QColor'ı kullanırız.

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
```

```
        size += QSize(1, 1);
    return size;
}
```

sizeHint() fonksiyonu QWidget'tan uyarlanmıştır ve bir parçacığın ideal boyutunu döndürür. Burada ideal boyutu, resim boyutunun yakınlaştırma katsayısıyla çarpımı ile yakınlaştırma katsayısının 3 veya daha fazla olduğu durumlarda ızgaraya yerleştirebilmek için her bir yönden ekstra bir piksel olarak alırız.

Bir parçacığın boyut ipucu(size hint) genelde yerleşimlerle birleşmede kullanışlıdır. Qt'un yerleşim yöneticileri, bir formun çocuk parçacıklarını yerleştirdiklerinde, parçacığın boyut ipucuna mümkün olduğunca uymaya çalışırlar. IconEditor iyi bir yerleşime sahip olmak için güvenilir bir boyut ipucu bildirmelidir.

Boyut ipucuna ek olarak, parçacıklar, yerleşim sistemine nasıl genişleyip küçüleceklerini söyleyen bir boyut politikasına(size policy) sahiptirler. Kurucuda, yatay ve dikey boyut politikalarını ayarlamak amacıyla setSizePolicy()'yi QSizePolicy::Minimum argümanları ile çağırarak, her yerleşim yöneticisine, boyut ipucunun bu parçacığın minimum boyutu olduğunu söylemiş oluruz. Başka bir deyişle, parçacık eğer gerekirse genişletilebilir, fakat boyut ipucundan daha aza küçültülmemelidir. Bu, Qt Designer'da parçacığın sizePolicy niteliğini ayarlayarak da yapılabilir.

```
void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}
```

setPenColor() fonksiyonu geçerli kalem rengini ayarlar. Renk, yeni çizilmiş pikseller için kullanılabilir.

```
void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertToFormat(QImage::Format_ARGB32);
        update();
        updateGeometry();
    }
}
```

setIconImage() fonksiyonu, resmi düzenlemeye hazırlar. Eğer resim bir alfa tampon ile 32-bit yapılmamışsa, convertToFormat()'ı çağırırız. Kodun başka bir yerinde resim verisinin 32-bit ARGB değerler olarak saklandığını varsayacağız.

image değişkenini ayarladıktan sonra, parçacığın yeni resim kullanılarak yeniden çizilmesi için QWidget::update()'i çağırırız. Sonra, parçacığın içerdiği her yerleşime parçacığın boyut ipucunun değiştiğini söylemek için QWidget::updateGeometry()'yi çağırırız. Ondan sonra, yerleşim otomatik olarak yeni boyut ipucuna uydurulacaktır.

```
void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;

    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
```

```
}
```

setZoomFactor() yakınlaştırma katsayısını ayarlar. Sıfıra bölmeyi önlemek için, 1'in altındaki her değeri 1'e eşitleriz. Sonra, parçacığı yeniden çizmek ve yerleşim yöneticilerini boyut ipucu değişikliği hakkında bilgilendirmek için update() ve updateGeometry()'yi çağırırız.

penColor(), iconImage() ve zoomFactor() fonksiyonları başlık dosyasında yerel fonksiyonlar olarak gerçekleştirilir.

Şimdi, paintEvent() fonksiyonu için olan kodların kritiğini yapacağız. Bu fonksiyon IconEditor'ın en önemli fonksiyonudur. Parçacığın yeniden çizilmesi gerektiği her zaman çağrılır. QWidget'taki varsayılan gerçekleştirim hiçbir şey yapmaz, parçacığı boş bırakır.

Tıpkı Bölüm 3'te tanıştığımız closeEvent() gibi, paintEvent() de bir olay işleyicidir. Qt, her biri farklı tipte bir olayla ilişkili olan birçok olay işleyiciye sahiptir.

Bir çiz olayı(paint event) üretildiğinde ve paintEvent() çağrıldığında birçok durum vardır. Örneğin:

- Bir parçacık ilk kez gösterildiğinde, sistem parçacığı kendini çizmeye zorlamak için otomatik olarak bir çiz olayı üretir.
- Parçacık yeniden boyutlandırıldığında sistem bir çiz olayı üretir.
- Bir parçacık başka bir pencere tarafından gizlendiğinde ve sonra tekrar açığa çıktığında, gizlenen alan için bir çiz olayı üretilir (eğer pencere sistemi alanı saklamamışsa).

QWidget::update() ya da QWidget::repaint()'i çağırarak da bir çiz olayını zorlayabiliriz. Bu iki fonksiyon arasındaki farlılık; repaint() derhal yeniden çizmeye zorlarken, update()'in sadece bir çiz olayını Qt'un olayları işleyeceği zamana zamanlamasıdır. (Her iki fonksiyonda eğer parçacık ekranda görünür değilse hiçbir şey yapmaz.) Eğer update() birçok kez çağrılırsa, Qt titreşmeyi önlemek için ardışık çiz olaylarını tek bir çiz olayı içine sıkıştırır. IconEditor'da, her zaman update()'i kullanırız.

İşte kod:

Kod Görünümü:

```
void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if (zoom >= 3) {
        painter.setPen(palette().foreground().color());
        for (int i = 0; i <= image.width(); ++i)
            painter.drawLine(zoom * i, 0,
                             zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j,
                             zoom * image.width(), zoom * j);
    }
    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j) {
            QRect rect = pixelRect(i, j);
            if (!event->region().intersect(rect).isEmpty()) {
                QColor color = QColor::fromRgba(image.pixel(i, j));
                if (color.alpha() < 255)
                    painter.fillRect(rect, Qt::white);
```

```
                    painter.fillRect(rect, color);
                }
            }
        }
    }
```

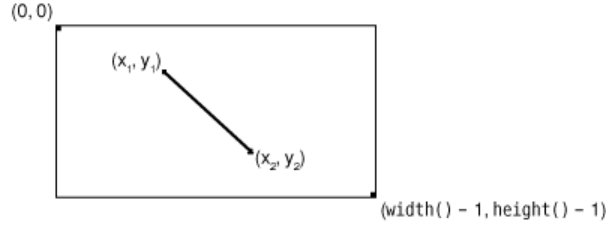
Parçacık üzerinde bir `QPainter` nesnesi oluşturarak başlarız. Eğer yakınlaştırma katsayısı 3 veya daha büyükse, `QPainter::drawLine()` fonksiyonunu kullanarak ızgara şeklinde yatay ve dikey satırlar çizeriz.

Bir `QPainter::drawLine()` çağrısı aşağıdaki sözdizimine sahiptir:

```
painter.drawLine(x1, y1, x2, y2);
```

Buradaki (`x1`, `y1`) bir ucun konumu, (`x2`, `y2`) diğer ucun konumudur. Ayrıca, fonksiyonun, dört `int` değer yerine iki `QPoint` alan aşırı yüklenmiş bir versiyonu da mevcuttur.

Bir Qt parçacığının sol üst pikseli (0, 0) konumuna, sağ alt pikseli (width() - 1, height() - 1) konumuna yerleşmiştir. Bu, geleneksel Kartezyen koordinat sistemi ile aynıdır, fakat Şekil 5.3'te de görüldüğü üzere, terstir.



Şekil 5.3

`QPainter` üstünde `drawLine()`'ı çağırmadan önce, `setPen()`'i kullanarak satırın rengini ayarlarız. Rengi kodlayabilirdik, siyah veya gri gibi, fakat daha iyi bir yaklaşım parçacığın renk paletini kullanmaktır.

Her parçacık, ne için hangi rengin kullanılması gerektiğini belirten bir palet ile donatılmıştır. Örneğin, parçacıkların arkaplan rengi (genellikle açık gri) için bir palet girdisi ve bir tane de arkaplanın üstündeki metnin rengi (genellikle siyah) için bir palet girdisi mevcuttur. Varsayılan olarak, bir parçacığın paleti, pencere sisteminin renk düzenini benimser. Paletten renkler kullanarak, `IconEditor`'ın kullanıcı tercihlerine uymasını sağlarız.

Bir parçacığın paleti üç renk grubundan meydana gelir: aktif(active), inaktif(inactive) ve devre dışı(disabled). Kullanılacak renk, parçacığın geçerli durumuna bağlıdır:

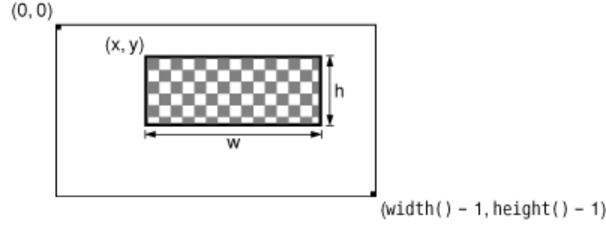
- `Active` grubu mevcut durumda aktif olan penceredeki parçacıklar için kullanılır.
- `Inactive` grubu diğer pencerelerdeki parçacıklar için kullanılır.
- `Disabled` grubu herhangi bir penceredeki devre dışı kalmış/bırakılmış parçacıklar için kullanılır.

`QWidget::palette()` fonksiyonu, parçacığın paletini bir `QPalette` nesnesi olarak döndürür. Renk grupları, `QPalette::ColorGroup` tipinin enumları olarak belirtilirler.

Çizmek için uygun bir fırça(brush) veya renk seçmek istediğimizde, doğru yaklaşım `QWidget::palette()`'den elde ettiğimiz geçerli paleti ve gerekli rolü kullanmaktır, örneğin, `QPalette::foreground()`. Her bir rol fonksiyonu bir fırça döndürür, fakat eğer sadece renk gerekiyorsa, `paintEvent()`'te yaptığımız

gibi, rengi fırçadan elde edebiliriz. Fırçalar varsayılan olarak, parçacığın durumuna uygun fırçayı döndürürler, bu nedenle bir renk grubu belirtmemiz gerekmez.

`paintEvent()` fonksiyonu, resmi çizerek biter. `IconEditor::pixelRect()`, yeniden çizilecek alanı tanımlayan bir `QRect` döndürür. (Şekil 5.4 bir dikdörtgenin nasıl çizildiğini gösteriyor.) Basit bir optimizasyon olarak, bu alanın dışında kalan pikselleri yeniden çizmeyiz.



Şekil 5.4

Yakınlaştırılmış bir pikseli çizmek için `QPainter::fillRect()`'i çağırırız. `QPainter::fillRect()` bir `QRect` ve bir `QBrush` alır. Fırça olarak bir `QColor` aktararak, bir katı dolgu modeli elde ederiz. Eğer renk tamamıyla ışık geçirmez(opaque) değilse, yani alfa kanalı 255'ten az ise, önce beyaz bir arka plan çizeriz.

```
QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}
```

`pixelRect()` fonksiyonu `QPainter::fillRect()` için uygun bir `QRect` döndürür. `i` ve `j` parametreleri `QImage` içindeki piksel koordinatlarıdır (parçacık içindeki değil). Eğer yakınlaştırma katsayısı 1 ise, iki koordinat sistemi tam olarak uyar.

`QRect` kurucusu `QRect(x , y, width, height)` şeklinde bir sözdizimine sahiptir; burada (`x`, `y`) dikdörtgenin sol üst köşesinin konumu, `width` x `height` dikdörtgenin boyutudur. Eğer yakınlaştırma katsayısı 3 veya daha fazla ise, boyutu yatay ve dikeyde birer piksel azaltırız, bu nedenle dolgu(fill), ızgara satırları üzerine çizilmez.

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->button() == Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

Kullanıcı bir fare butonuna bastığında, sistem bir "fare butonuna basma(mouse press)" olayı üretir. `QWidget::mousePressEvent()`'i uyarlayarak bu olayı cevaplandırabiliriz ve böylece, resmin fare imlecinin altındaki piksellerini düzenleyebilir ya da silebiliriz.

Eğer kullanıcı farenin sol butonuna basarsa, `setImagePixel()` private fonksiyonunu, ikinci argüman olarak `true`'yu göndererek çağırırız. (Böylece pikselin, geçerli kalem rengi ile düzenlenmesi gerektiğini

bildiririz.) Eğer kullanıcı farenin sağ butonuna basarsa, yine setImagePixel()'i çağırırız, fakat bu sefer ikinci argüman olarak, pikselleri temizlemeyi emreden false'u göndererek.

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->buttons() & Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

mouseMoveEvent() "fare taşınma" olaylarını işler. Bu olaylar varsayılan olarak, kullanıcı bir fare butonuna basılı tuttuğunda üretilirler. Bu davranışı, QWidget::setMouseTracking()'ı çağırarak değiştirmek mümkündür, fakat bu örnek için böyle yapmamız gerekmez.

Farenin sağ ya da sol butonuna basarak bir pikselin düzenlenebilmesi veya temizlenebilmesi gibi, bir pikselin üzerinde butona basılı tutmak ve duraksamak da bir pikseli düzenlemek veya temizlemek için yeterlidir. Bir seferde birden çok butona basılı tutmak mümkün olduğu için, QMouseEvent::buttons() tarafından döndürülen değer fare butonlarının bitsel bir OR'udur. Belli bir butona basılıp basılmadığını & operatörünü kullanarak test ederiz ve eğer basılmışsa setImagePixel()'i çağırırız.

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;

    if (image.rect().contains(i, j)) {
        if (opaque) {
            image.setPixel(i, j, penColor().rgba());
        } else {
            image.setPixel(i, j, qRgba(0, 0, 0, 0));
        }
        update(pixelRect(i, j));
    }
}
```

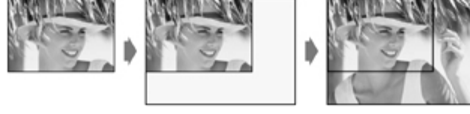
setImagePixel() fonksiyonu, bir pikseli düzenlemek ya da temizlemek için mousePressEvent() ve mouseMoveEvent()'ten çağrılır. pos parametresi, farenin parçacık üzerindeki konumudur.

İlk adım fare konumunu parçacık koordinatlarından resim koordinatlarına çevirmektir. Bu, fare konumunun x() ve y() bileşenlerini yakınlaştırma katsayısına bölerek yapılır. Sonra, noktanın doğru alan içinde olup olmadığını kontrol ederiz. Kontrol, QImage::rect() ve QRect::contains() kullanarak kolaylıkla yapılır; bu, i'nin 0 ve image.width() – 1 arasında olup olmadığını ve j'nin 0 ve image.height() – 1 arasında olup olmadığını etkin bir şekilde kontrol eder.

opaque parametresine bağlı olarak, resimdeki pikseli düzenler ya da temizleriz. Bir pikseli temizlemek aslında onu saydam olarak düzenlemektir. QImage::setPixel() çağrısı için, kalemin QColor'ını bir 32-bit ARGB değere dönüştürmeliyiz. Sonunda, yeniden çizilmesi gereken alanın bir QRect'i ile update()'i çağırırız.

Üye fonksiyonları bitirdik, şimdi kurucuda kullandığımız Qt::WA_StaticContents niteliğine döneceğiz. Bu nitelik Qt'a, parçacık yeniden boyutlandırıldığında içeriğinin değişmemesini ve parçacığın sol üst

köşesinde konumlanmış olarak kalmasını söyler. Qt bu bilgiyi, parçacık yeniden boyutlandırıldığında zaten görünen alanların gereksizce yeniden çizilmesinin önüne geçmede kullanır. Bu durum Şekil 5.5'te bir örnekle açıklanıyor.



Şekil 5.5

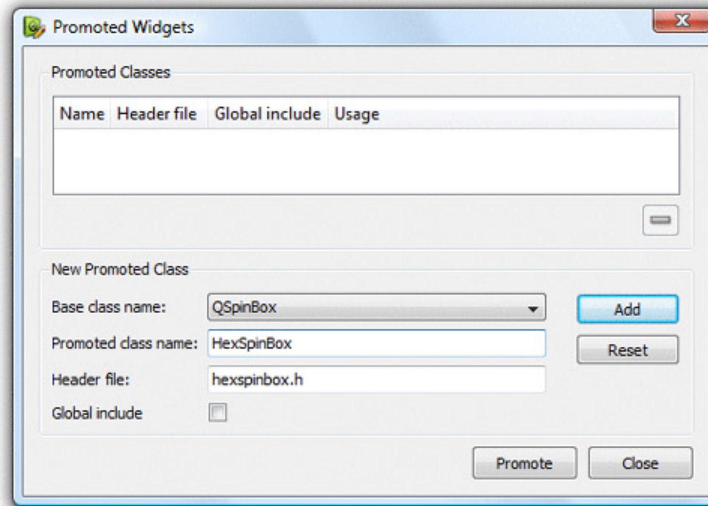
Bir parçacık yeniden boyutlandırıldığında, Qt, varsayılan olarak, parçacığın tüm görünür alanları için bir çiz olayı üretir. Fakat eğer parçacık `Qt::WA_StaticContents` niteliğiyle oluşturulmuşsa, çiz olayının alanı, daha önce görünmeyen piksellerle sınırlandırılır. Bu, parçacık daha küçük bir boyuta yeniden boyutlandırıldığında hiçbir çiz olayı üretilmeyeceği anlamına gelir.

Artık `IconEditor` parçacığı tamamlandı. Daha önceki bölümlerdeki bilgileri ve örnekleri kullanarak, `IconEditor`'ı bir pencere olarak, bir `QMainWindow`'da merkez parçacık olarak ya da bir yerleşim veya bir QScrollArea içinde çocuk parçacık olarak kullanan bir kod yazabilirdik. Sıradaki kısımda, parçacığı Qt Designer'a entegre etmeyi göreceğiz.

Özel Parçacıkları Qt Designer'a Entegre Etme

Özel parçacıkları Qt Designer içinde kullanmadan önce, Qt Designer'ı onlardan haberdar etmeliyiz. Bunu yapmak için iki teknik vardır: "Promotion(Terfi)" yaklaşımı ve "Plugin(Eklenti)" yaklaşımı.

Terfi yaklaşımı en hızlı ve kolay olanıdır.



Şekil 5.6

İşte, bu yaklaşımı kullanarak bir `HexSpinBox`, bir form içine adım adım şu şekilde eklenir:

1. Qt Designer'ın parçacık kutusundan formun üstüne sürükleyerek bir `QSpinBox` oluşturun.
2. Döndürme kutusuna(spin box) sağ tıklayın ve bağlam menüden Prote to Custom Widget'ı seçin.
3. Beliren diyaloğun sınıf ismi kısmına "HexSpinBox", başlık dosyası kısmına "hexspinbox.h" yazın.

uic tarafından üretilen kod, <QSpinBox> yerine hexspinbox.h'ı içerecek ve böylelikle bir HexSpinBox'ı somutlaştırmış olacak. HexSpinBox parçacığı Qt Designer'da, her özelliğini ayarlamamıza imkân veren bir QSpinBox tarafından temsil edilecek.

Terfi yaklaşımının güçlükleri, özel parçacığın spesifik özelliklerine Qt Designer içinde erişilememesi ve parçacığın kendi olarak yorumlanmamasıdır. Her iki problem de Eklenti yaklaşımı kullanılarak aşılabilir.

Eklenti yaklaşımı, Qt Designer'ın çalışma sırasında yükleyebildiği ve parçacığın örneğini oluşturmakta kullanabildiği bir eklenti kütüphanesi kreasyonu gerektirir. Qt Designer formu düzenleme ve önizlemede gerçek parçacığı kullanır, Qt meta-object sistemine şükürler olsun ki, Qt Designer, özelliklerinin listesini dinamik olarak oluşturabilir. Nasıl çalıştığını görmek için, önceki kısımdaki IconEditor'ı bir eklenti olarak Qt Designer'a entegre edeceğiz.

Önce, bir QDesignerCustomWidgetInterface altsınıfı türetmeliyiz ve bazı sanal fonksiyonlar uyarlamalıyız. Eklentinin kaynak kodunun iconeditorplugin adında bir dizinde, IconEditor'ın kaynak kodunun ise iconeditor adında paralel bir dizinde yer aldığını varsayacağız.

İşte sınıf tanımı:

```
#include <QDesignerCustomWidgetInterface>

class IconEditorPlugin : public QObject,
                         public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    IconEditorPlugin(QObject *parent = 0);

    QString name() const;
    QString includeFile() const;
    QString group() const;
    QIcon icon() const;
    QString toolTip() const;
    QString whatsThis() const;
    bool isContainer() const;
    QWidget *createWidget(QWidget *parent);
};
```

IconEditorPlugin altsınıfı, IconEditor parçacığını içeren bir factory sınıftır. QObject ve QDesignerCustonWidgetInterface'ten türetilmiştir ve moc'a ikinci ana sınıfın bir eklenti arayüzü olduğunu bildirmek için Q_INTERFACES makrosunu kullanır. Qt Designer, sınıfın örneklerini oluşturmak ve sınıf hakkında bilgi edinmek için, şu fonksiyonları kullanır:

```
IconEditorPlugin::IconEditorPlugin(QObject *parent)
    : QObject(parent)
{
}
```

Kurucu önemsizdir.

```
QString IconEditorPlugin::name() const
{
    return "IconEditor";
```

```
}
```

`name()` fonksiyonu eklenti tarafından sağlanan parçacığın adını döndürür.

```
QString IconEditorPlugin::includeFile() const
{
    return "iconeditor.h";
}
```

`includeFile()` fonksiyonu, eklentinin içerdiği parçacığın başlık dosyasının adını döndürür.

```
QString IconEditorPlugin::group() const
{
    return tr("Image Manipulation Widgets");
}
```

`group()` fonksiyonu, özel parçacığın üyesi olduğu parçacık kutusu grubunun ismini döndürür. Eğer isim henüz kullanımda değilse, Qt Designer parçacık için yeni bir grup oluşturacaktır.

```
QIcon IconEditorPlugin::icon() const
{
    return QIcon(":/images/iconeditor.png");
}
```

`icon()` fonksiyonu, Qt Designer'ın parçacık kutusunda özel parçacığı temsil etmekte kullanılacak olan simgeyi döndürür.

```
QString IconEditorPlugin::toolTip() const
{
    return tr("An icon editor widget");
}
```

`toolTip()` fonksiyonu, Qt Designer'ın parçacık kutusunda özel parçacık üzerine fare ile gelinip beklendiğinde görünecek olan araç ipucunu(tooltip) döndürür.

```
QString IconEditorPlugin::whatsThis() const
{
    return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
              "Programming with Qt 4</i> as an example of a custom Qt "
              "widget.");
}
```

`whatsThis()` fonksiyonu, Qt Designer'ın göstermesi için "What's This?(Bu Nedir?)" metnini döndürür.

```
bool IconEditorPlugin::isContainer() const
{
    return false;
}
```

`isContainer()` fonksiyonu eğer parçacık diğer parçacıkları içerebiliyorsa `true`; aksi halde, `false` döndürür. Örneğin, `QFrame` diğer parçacıkları içerebilen bir parçacıktır. Genelde, her Qt parçacığı diğer parçacıkları içerebilir, fakat Qt Designer, `isContainer()` `false` döndürdüğünde buna izin vermez.

```
QWidget *IconEditorPlugin::createWidget(QWidget *parent)
{
    return new IconEditor(parent);
}
```

Qt Designer, bir parçacık sınıfının bir örneğini oluşturmak için, `createWidget()` fonksiyonunu belirtilen ebeveynle çağırır.

```
Q_EXPORT_PLUGIN2(iconeditorplugin, IconEditorPlugin)
```

Eklenti sınıfını gerçekleştiren kaynak kodu dosyasının sonunda, eklentiyi Qt Designerca kullanılabilir yapmak için `Q_EXPORT_PLUGIN2()` makrosunu kullanmalıyız. İlk argüman, eklentiye vermek istediğimiz isimdir; ikinci argüman ise onu gerçekleştiren sınıfın ismidir.

Eklenti için oluşturulan `.pro` dosyası şuna benzer:

```
TEMPLATE      = lib
CONFIG       += designer plugin release
HEADERS       = ../iconeditor/iconeditor.h \
                iconeditorplugin.h
SOURCES       = ../iconeditor/iconeditor.cpp \
                iconeditorplugin.cpp
RESOURCES     = iconeditorplugin.qrc
DESTDIR       = $$[QT_INSTALL_PLUGINS]/designer
```

`qmake` aracı, bazı ön tanımlı değişkenlere sahiptir. Onlardan biri, Qt'un yüklendiği dizindeki `plugins` dizininin yolunu tutan `$$[QT_INSTALL_PLUGINS]`'tir. Siz eklentiyi inşa etmek için `make` veya `nmake` girdiğinizde, eklenti kendini otomatik olarak Qt'un `plugins/Designer` dizinine yükleyecektir. Eklenti bir kere inşa edildiğinde, `IconEditor` parçacığı Qt Designer içinde artık Qt'un yerleşik bir parçacığıymış gibi kullanılabilir.

Eğer birkaç özel parçacığı Qt Designer ile birleştirmek isterseniz, ya her biri için bir eklenti oluşturursunuz, ya da tümünü `QDesignerCustomWidgetCollectionInterface`'ten türetilen tek bir eklentide birleştirirsiniz.

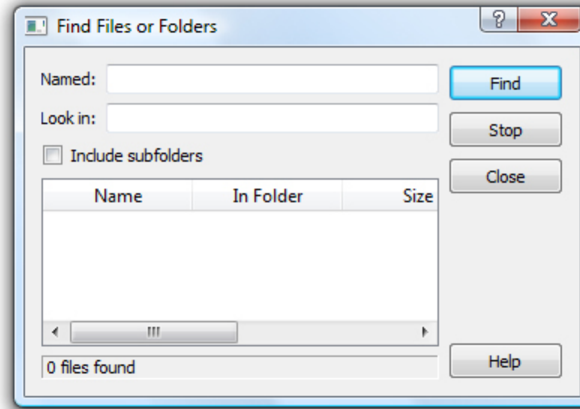
BÖLÜM 6: YERLEŞİM YÖNETİMİ

Bir forma yerleştirilmiş her parçacık, verilen uygun bir boyutta ve konumda olmalıdır. Qt, parçacıkları bir forma yerleştiren birkaç sınıf sağlar: `QHBoxLayout`, `QVBoxLayout`, `QGridLayout` ve `QStackedLayout`. Bu sınıflar çok kullanışlıdır, öyle ki hemen hemen her Qt geliştiricisi onları ya direkt kaynak kodu içinde ya da Qt Designer içinden kullanır.

Qt'un yerleşim sınıflarını(layout classes) kullanmanın bir diğer nedeni de, formların otomatik olarak farklı yazı tiplerine, dillere ve platformlara uymalarını sağlamalarıdır. Tüm bu sınıflar, kullanıcının değiştirebileceği esnek bir yerleşim sağlarlar. Bu bölümde onlardan bahsedeceğiz.

Parçacıkları Bir Form Üzerine Yerleştirme

Bir form üzerindeki çocuk parçacıkların yerleşimini yönetmenin üç basit yolu vardır: mutlak konumlama(absolute positioning), manüel yerleşim(manual layout) ve yerleşim yöneticileri(layout managers). Bu yaklaşımların her birini sırasıyla, Şekil 6.1'de gösterilen Find File diyaloğunda kullanırken inceleyeceğiz.



Şekil 6.1

Mutlak konumlama, parçacıkları yerleştirmenin en ilkel yoludur. Tüm konum ve boyut bilgileri elle kodlanarak girilir. İşte, mutlak konumlama kullanılarak yazılan bir `FindFileDialog` kurucusu:

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 200, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 200, 25);
    subfoldersCheckBox->setGeometry(9, 71, 256, 23);
    tableWidget->setGeometry(9, 100, 256, 100);
    messageLabel->setGeometry(9, 206, 256, 25);
    findButton->setGeometry(271, 9, 85, 32);
    stopButton->setGeometry(271, 47, 85, 32);
```

```
    closeButton->setGeometry(271, 84, 85, 32);
    helpButton->setGeometry(271, 199, 85, 32);

    setWindowTitle(tr("Find Files or Folders"));
    setFixedSize(365, 240);
}
```

Mutlak konumlamanın birçok dezavantajı vardır:

- Kullanıcı pencereyi yeniden boyutlandıramaz.
- Eğer kullanıcı olağandışı geniş bir yazı tipi seçerse ya da uygulama başka bir dile çevrilirse, bazı metinler kesik görünebilir.
- Parçacıklar bazı stiller için uygun olmayan boyutlara sahip olabilir.
- Konumlar ve boyutlar elle hesaplanmalıdır. Bu sıkıcı ve hataya eğilimlidir ve bakımı zahmetli hale getirir.

Mutlak konumlamaya bir alternatif manüel yerleşimdir. Manüel yerleşim ile parçacıkların konumları hâlâ mutlaktır, fakat boyutları pencerenin boyutuna orantılı yapılır. Bu, formun `resizeEvent()` fonksiyonunu çocuk parçacıklarının geometrisini ayarlaması için uyarlayarak gerçekleştirilir:

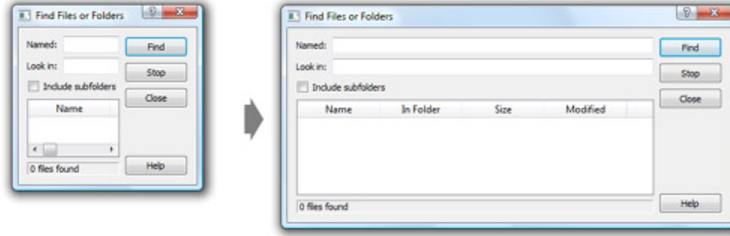
Kod Görünümü:

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    setMinimumSize(265, 190);
    resize(365, 240);
}

void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();

    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
    subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);

    tableWidget->setGeometry(9, 100, 156 + extraWidth,
                             50 + extraHeight);
    messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth,
                              25);
    findButton->setGeometry(171 + extraWidth, 9, 85, 32);
    stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
    closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
    helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85,
                            32);
}
```

`FindFileDialog` kurucusunda, formun minimum boyutunu 265 x 190, ilk boyutunu 365 x 240 olarak ayarlarız. `resizeEvent()` işleyicisinde, büyümesini istediğimiz parçacığa ekstra boşluk veririz. Bu, kullanıcı formu yeniden boyutlandırdığında, formun düzgünce ölçeklenmesini sağlar.

Tıpkı mutlak yerleştirme gibi, manüel yerleşim de, çokça elle kod yazmayı ve birçok sabitin programcı tarafından hesaplanmasını gerektirir. Böyle kod yazmak yorucudur, özelliklede tasarım değiştiğinde. Ve hâlâ metnin kesik görünme riski vardır. Bu riskin önünü almanın bir yolu vardır, fakat o da kodu daha fazla karmaşıklaştıracaktır.

Parçacıkları bir form üzerine yerleştirmenin en pratik yolu, Qt'un yerleşim yöneticilerini kullanmaktır. Yerleşim yöneticileri, her tipten parçacık için mantıklı varsayılanlar sağlar ve her bir parçacığın yazıtipine, stiline ve içeriğine bağlı olan boyut ipucunu hesaba katar. Yerleşim yöneticileri minimum ve maksimum boyutlara da saygı gösterir ve yazıtipi ve içerik değişikliklerine ve pencerenin yeniden boyutlandırılmasına karşılık yerleşimi otomatik olarak ayarlar. Find File diyaloğunun yeniden boyutlandırılabilir bir versiyonu Şekil 6.2'de gösteriliyor.



Şekil 6.2

En önemli üç yerleşim yöneticisi `QHBoxLayout`, `QVBoxLayout` ve `QGridLayout`'tur. Bu sınıflar, yerleşimler için sağlanan temel çatı(framework) olan QLayout'tan türetilmiştirler. Tüm bu sınıflar, Qt Designer tarafından tam olarak desteklenir ve ayrıca kod içinde direkt olarak kullanılabilir.

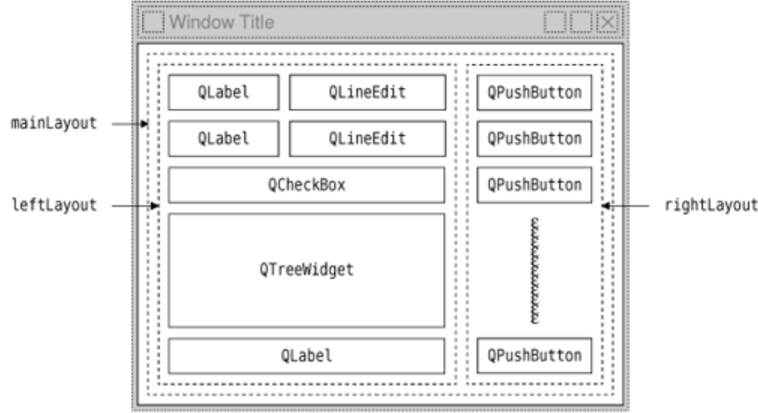
İşte, yerleşim yöneticilerini kullanarak yazılmış `FindFileDialog`:

Kod Görünümü:

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
    leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
    leftLayout->addWidget(messageLabel, 4, 0, 1, 2);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();
    rightLayout->addWidget(helpButton);

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
    setLayout(mainLayout);
```

```
    setWindowTitle(tr("Find Files or Folders"));
}
```

Yerleşim bir QHBoxLayout, bir QGridLayout ve bir QVBoxLayout tarafından idare edilir. Soldaki QGridLayout ve sağdaki QVBoxLayout dıştaki QHBoxLayout tarafından yan yana yerleştirilmiştir. Diyaloğun çevresindeki pay ve çocuk parçacıklar arasındaki boşluk, geçerli parçacık stiline bağlı olan varsayılan değerlere ayarlanır; diğer taraftan QLayout::setContentsMargins() ve QLayout::-setSpacing() kullanılarak değiştirilebilirler.



Şekil 6.3

Aynı diyalog, çocuk parçacıklar benzer konumlara yerleştirilerek, görsel olarak Qt Designer'da da oluşturulabilir. Bu yaklaşımı Bölüm 2'de Spreadsheet uygulamasının Go to Cell ve Sort diyaloglarını oluştururken kullanmıştık.

QHBoxLayout ve QVBoxLayout'u kullanmak oldukça kolaydır, fakat QGridLayout'u kullanmak biraz daha karışıktır. QGridLayout, iki boyutlu bir hücreler ızgarası üzerinde çalışır. Yerleşimin sol üst köşesindeki QLabel (0, 0) konumundadır ve onunla ilişkili QLineEdit (0, 1) konumundadır. QCheckBox iki sütun kaplar; (2, 0) ve (2, 1) konumlarındaki hücrelere oturur. Onun altındaki QTreeWidget ve QLabel da iki sütun kaplar. QGridLayout::addWidget() çağrısının sözdizimi şöyledir:

```
layout->addWidget(widget, row, column, rowSpan, columnSpan);
```

Burada, widget yerleşim içine eklenecek olan çocuk parçacık, (row, column) parçacık tarafından tutulan sol üst hücre, rowSpan parçacık tarafından tutulan satırların sayısı, columnSpan ise parçacık tarafından tutulan sütunların sayısıdır. İhmal edildiğinde, rowSpan ve columnSpan argümanlarının varsayılanı 1'dir.

addStrecth() çağrısı dikey yerleşim yöneticisine yerleşim içinde o noktadaki boşluğun yok edilmesini söyler. Bir genişleme öğesi(stretch item) ekleyerek, yerleşim yöneticisine Close ve Help butonları arasındaki fazlalık boşluğun tüketilmesini söyledik. Qt Designer'da, aynı etkiyi bir ara parça(spacer) ekleyerek meydana getirebiliriz. Ara parçalar Qt Designer'da mavi bir yay gibi görünür.

Yerleşim yöneticilerini kullanmak, şimdiye kadar da söz ettiğimiz gibi, ek avantajlar sağlar. Eğer bir parçacığı bir yerleşime eklersek ya da bir parçacığı bir yerleşimden silersek, yerleşim otomatik olarak yeni duruma uyarlanacaktır. Bir çocuk parçacık üzerinde hide() ya da show()'u çağırırsak da aynısı uygulanır. Eğer bir çocuk parçacığın boyut ipucu değişirse, yerleşim, yeni boyut ipucu dikkate alınarak otomatik olarak yeniden

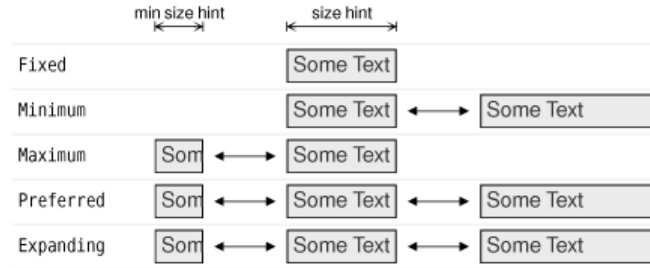
yapılacaktır. Ayrıca, yerleşim yöneticileri, formun çocuk parçacıklarının minimum boyutlarını ve boyut ipuçlarını dikkate alarak, otomatik olarak formun tümü için minimum bir boyut da ayarlar.

Şimdiye kadar ortaya konulan örneklerde, sadece parçacıkları yerleşimlerin içine koyduk ve fazlalık boşluğu tüketmek için de ara parçaları kullandık. Bazı durumlarda, yerleşimin tamamen istediğimiz gibi görünmesinde bu yeterli olmaz. Bu gibi durumlarda, yerleştirmiş olduğumuz parçacıkların boyut politikalarını(size policy) ve boyut ipuçlarını(size hint) değiştirerek, yerleşimi belirleyebiliriz.

Bir parçacığın boyut politikası, yerleşim sistemine nasıl genişlemesi ve küçülmesini gerektiğini söyler. Qt, yerleşik parçacıklarının tümü için mantıklı varsayılan boyut politikaları sağlar, fakat her olası yerleşime tekabül edebilecek tek bir varsayılan olmadığı için, geliştiriciler arasında, form üzerindeki bir ya da iki parçacığın boyut politikalarını değiştirmek hâlâ yaygındır. Bir `QSizePolicy` hem bir yatay hem de bir dikey bileşene sahip olabilir. İşte, en kullanışlı değerler:

- `Fixed`, parçacığın büyüyüp küçülemeyeceği anlamına gelir. Parçacık daima boyut ipucunun belirttiği boyutta kalır.
- `Minimum`, parçacığın minimum boyutunda olduğu anlamına gelir. Parçacık boyut ipucunun daha altında bir boyuta küçültülemez, fakat eğer gerekliyse kullanılabilir boşluğu doldurana kadar genişleyebilir.
- `Maximum`, parçacığın maksimum boyutunda olduğu anlamına gelir. Parçacık minimum boyut ipucuna kadar küçültülebilir.
- `Preferred` parçacığın boyut ipucunun onun tercih edilen boyutu olduğu anlamına gelir, fakat parçacık eğer gerekliyse hâlâ genişletilebilir ya da küçültülebilir.
- `Expanding` parçacığın genişletilebilir ve küçültülebilir olduğu anlamına gelir.

Şekil 6.4, farklı boyut politikalarının anlamlarını özetliyor.



Şekil 6.4

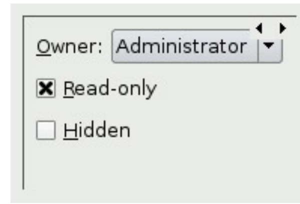
Şekilde, `Preferred` ve `Expanding` aynı tarzda resmedilmiştir. Öyleyse, farklılık nedir? Hem `Preferred` hem de `Expanding` parçacıklar içeren bir form yeniden boyutlandırıldığında, `Preferred` parçacıklar boyut ipuçlarının belirttiği boyutta kalırken, ekstra boşluk `Expanding` parçacıklara verilir.

İki boyut politikası daha vardır: `MinimumExpanding` ve `Ignored`. Birincisi Qt'un daha eski versiyonlarında bazı seyrek karşılaşan durumlarda gerekliydi, fakat fazla kullanışlı değildir; tercih edilen yaklaşım `Expanding` kullanmak ve `mimimumSizeHint()`'i uygun şekilde uyarlamaktır. İkincisi ise, parçacığın boyut ipucunu ve minimum boyut ipucunu görmezden gelmesi dışında, `Expanding` ile benzerdir.

Boyut politikalarının yatay ve dikey bileşenlerine ek olarak, `QSizePolicy` sınıfı bir yatay ve bir de dikey genişleme katsayısı(stretch factor) saklar. Bu genişleme katsayısı, form genişlediğinde farklı çocuk parçacıkların farklı ölçüde genişlemesi istediğinizde, bunu belirtmede kullanılabilir. Örneğin, bir `QTextEdit` üzerinde bir `QTreeWidget`'a sahipsek ve `QTextEdit`'in `QTreeWidget`'ın iki katı uzunluğunda olmasını istiyorsak, `QTextEdit`'in dikey genişleme katsayını 2, `QTreeWidget`'ınkini 1 olarak ayarlayabiliriz.

Yığılı Yerleşimler

`QStackedLayout` sınıfı, çocuk parçacıkların bir setini veya "sayfaları(pages)" yerleştirir ve bir seferde sadece birini gösterir, diğerlerini kullanıcıdan gizler. `QStackedLayout`'un kendisi görünmezdir. Şekil 6.5'teki küçük oklar ve koyu gri çerçeve(frame), Qt Designer tarafından yerleşimi, bu yerleşim yöneticisiyle tasarlamayı daha kolay hale getirmek için sağlanır. Kolaylık olsun diye, Qt, yerleşik bir `QStackedLayout` ile bir `QWidget` sağlayan, `QStackedWidget`'ı da içerir.

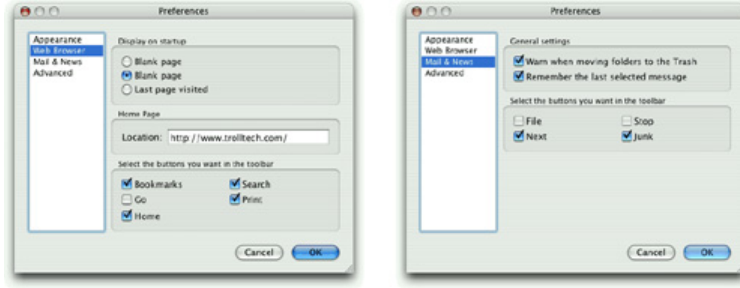


Şekil 6.5

Sayfalar 0'dan başlayarak numaralandırılır. Belirli bir çocuk parçacığı görünür yapmak için, `setCurrentIndex()`'i bir sayfa numarası ile çağırabiliriz. Bir çocuk parçacığın sayfa numarasına `indexOf()` kullanılarak erişilebilir.

Şekil 6.6'da gösterilen Preferences diyaloğu `QStackedLayout`'u kullanan bir örnektir. Diyalog, solda bir `QListWidget` ve sağda bir `QStackedLayout`'tan meydana gelir. `QListWidget`'taki her bir öğe `QStackedLayout`'taki farklı bir sayfaya tekabül eder. İşte, diyaloğun kurucusuyla alâkalı kod:

```
PreferenceDialog::PreferenceDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    listWidget = new QListWidget;
    listWidget->addItem(tr("Appearance"));
    listWidget->addItem(tr("Web Browser"));
    listWidget->addItem(tr("Mail & News"));
    listWidget->addItem(tr("Advanced"));

    stackedLayout = new QStackedLayout;
    stackedLayout->addWidget(appearancePage);
    stackedLayout->addWidget(webBrowserPage);
    stackedLayout->addWidget(mailAndNewsPage);
    stackedLayout->addWidget(advancedPage);
    connect(listWidget, SIGNAL(currentRowChanged(int)),
            stackedLayout, SLOT(setCurrentIndex(int)));
    ...
    listWidget->setCurrentRow(0);
}
```



Şekil 6.6

Bir QListWidget oluştururuz ve onu sayfa isimleriyle doldururuz. Sonra bir QStackedLayout oluştururuz ve her bir sayfa için addWidget()'ı çağırırız. Liste parçacığının currentRowChanged(int) sinyalini -sayfa değiştirmeyi gerçekleştirmek için- yığılı yerleşimin setCurrentIndex(int) yuvasına bağlarız ve kurucunun sonunda, 0. sayfadan başlamak için, liste parçacığı üzerinde setCurrentRow()'u çağırırız.

Bunun gibi formaları Qt Designer kullanarak oluşturmak da çok kolaydır:

1. "Dialog" şablonlarını ya da "Widget" şablonunu baz alan yeni bir form oluşturun.
2. Forma bir QListWidget ve bir QStackedWidget ekleyin.
3. Her bir sayfayı çocuk parçacıklar ve yerleşimlerle doldurun. (Yeni bir sayfa oluşturmak için, sağ tıklayın ve Insert Page'i seçin; sayfaları değiştirmek için, QStackedWidget'ın sağ üstündeki oklara tıklayın.)
4. Parçacıkları bir yatay yerleşim kullanarak yan yana yerleştirin.
5. Liste parçacığının currentRowChanged(int) sinyalini yığılı yerleşimin setCurrentIndex(int) yuvasına bağlayın.
6. Parçacığın currentRow niteliğinin değerini 0 olarak ayarlayın.

Sayfa değiştirmeyi önceden tanımlanmış sinyal ve yuvaları kullanarak geçekleştirdiğimiz için, diyalog Qt Designer'da önizlendiğinde doğru davranışı sergileyecektir.

Sayfaların sayısı küçük olduğu ve büyük ihtimalle küçük kalacağı durumlar için, bir QStackedWidget ve QListWidget kullanmaya daha basit bir alternatif bir QTabWidget kullanmaktır.

Bölücüler

Bir QSplitter diğer parçacıkları içeren bir parçacıktır. Bir bölücü(splitter) içindeki parçacıklar, bölücü kollar(splitter handles) tarafından ayrılmışlardır. Kullanıcılar bölücü kolları sürükleyerek, bölücünün çocuk parçacılarının boyutlarını değiştirebilirler. Bölücüler sıklıkla, kullanıcıya daha fazla kontrol sağlamak için, yerleşimlere bir alternatif olarak da kullanılırlar.

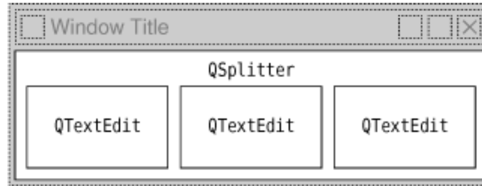
Bir QSplitter'ın çocuk parçacıkları, oluşturulma sıralarına göre yan yana (ya da biri diğerinin altında) komşu parçacıklar arasındaki bölücü kollarla otomatik olarak yerleştirilirler. İşte, Şekil 6.7'de gösterilen pencereyi oluşturmak için kod:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QTextEdit *editor1 = new QTextEdit;
```

```
    QTextEdit *editor2 = new QTextEdit;
    QTextEdit *editor3 = new QTextEdit;

    QSplitter splitter(Qt::Horizontal);
    splitter.addWidget(editor1);
    splitter.addWidget(editor2);
    splitter.addWidget(editor3);
    ...
    splitter.show();
    return app.exec();
}
```

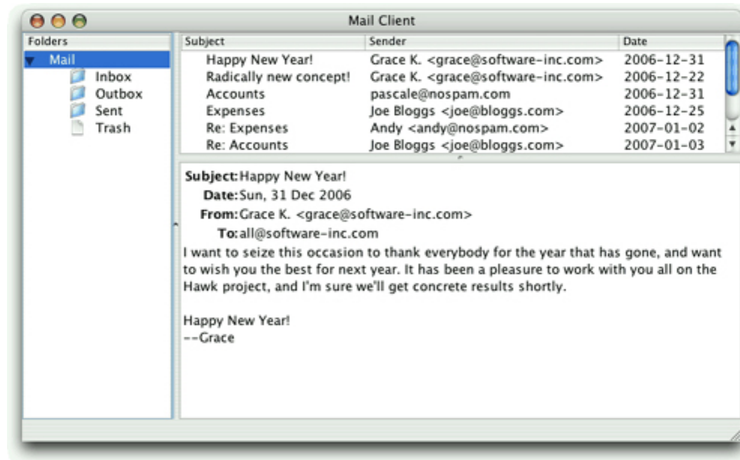
Şekil 6.7

Örnek, bir `QSplitter` parçacığı tarafından yatay olarak yerleştirilmiş üç `QTextEdit`'ten oluşur. Bu, Şekil 6.8'de şematik olarak gösterilmiştir. `QSplitter`, `QWidget`'tan türetilmiştir ve bu nedenle diğer her parçacık gibi kullanılabilir.

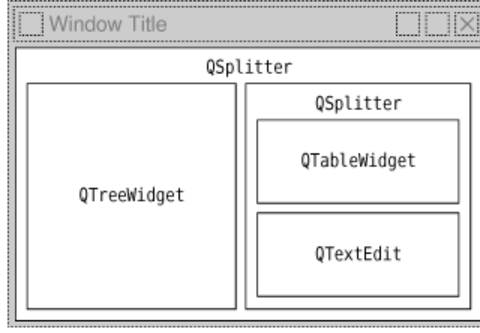


Şekil 6.8

Yatay ve dikey `QSplitter`'ları iç içe koyarak karmaşık yerleşimler meydana getirilebilir. Örneğin, Şekil 6.9'da gösterilen Mail Client uygulaması, dikey bir `QSplitter` içeren yatay bir `QSplitter`'dan meydana gelir. Yerleşimi, Şekil 6.10'da şematik olarak gösteriliyor.



Şekil 6.9



Şekil 6.10

İşte, Mail Client uygulamasının `QMainWindow` altsınıfının kurucusundaki kod:

```
MailClient::MailClient()
{
    ...
    rightSplitter = new QSplitter(Qt::Vertical);
    rightSplitter->addWidget(messagesTreeWidget);
    rightSplitter->addWidget(textEdit);
    rightSplitter->setStretchFactor(1, 1);

    mainSplitter = new QSplitter(Qt::Horizontal);
    mainSplitter->addWidget(foldersTreeWidget);
    mainSplitter->addWidget(rightSplitter);
    mainSplitter->setStretchFactor(1, 1);
    setCentralWidget(mainSplitter);

    setWindowTitle(tr("Mail Client"));
    readSettings();
}
```

Görüntülemek istediğimiz üç parçacığı oluşturduktan sonra, dikey bir bölücü oluştururuz (`rightSplitter`) ve sağda görüntülemek istediğimiz iki parçacığı ona ekleriz. Sonra da yatay bir bölücü oluştururuz (`mainSplitter`) ve solda görüntülemek istediğimiz parçacığı ve `rightSplitter`'ı ona ekleriz. Sonrasında, `mainSplitter`'ı `QMainWindow`'un merkez parçacığı yaparız.

Kullanıcı bir pencereyi yeniden boyutlandırdığında, `QSplitter` normal olarak boşluğu dağıtır, böylece çocuk parçacıkların göreceli boyutları aynı kalır. Mail Client örneğinde, bu davranışı istemeyiz; yerine, `QTreeWidget` ve `QTableWidget`'ın boyutlarının aynı kalmasını ve ekstra boşluğun `QTextedit`'e verilmesini isteriz. Bu, iki `setScrerchFactor()` çağrısı sayesinde başarılır. İlk argüman bölücünün çocuk parçacığının indeksi, ikinci argüman ayarlamak istediğimiz genişleme katsayısıdır (varsayılanı 0'dır).

İlk `setStretchFactor()` çağrısı `rightSplitter` üzerindedir ve 1 konumundaki parçacığın (`textEdit`) genişleme katsayısını 1 olarak ayarlar. İkinci `setStretchFactor()` çağrısı `mainSplitter` üzerindedir ve bir konumundaki parçacığın (`rightSplitter`) genişleme katsayısını 1 olarak ayarlar. Bu, `textEdit`'in kullanılabilir fazlalık boşluğun tümünü almasını sağlar.

Uygulama başlatıldığında, `QSplitter` çocuk parçacıkların ilk boyutlarına bağlı olarak (ya da eğer ilk boyut yoksa boyut ipuçlarına bağlı olarak) uygun boyutları verir. Bölücü kollarını programlanabilir bir biçimde `QSplitter::setSizes()`'ı çağırarak hareket ettirebiliriz. `QSplitter` sınıfı aynı zamanda kendi

durumunun kaydedilmesi ve saklanması için bir araç da sağlar. İşte, Mail Client'ın ayarlarını kaydeden `writeSettings()` fonksiyonu:

```
void MailClient::writeSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("mainSplitter", mainSplitter->saveState());
    settings.setValue("rightSplitter", rightSplitter->saveState());
    settings.endGroup();
}
```

Ve işte, ona uygun bir `readSettings()` fonksiyonu:

```
void MailClient::readSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    restoreGeometry(settings.value("geometry").toByteArray());
    mainSplitter->restoreState(
            settings.value("mainSplitter").toByteArray());
    rightSplitter->restoreState(
            settings.value("rightSplitter").toByteArray());
    settings.endGroup();
}
```

Qt Designer `QSplitter`'ı tamamen destekler. Parçacıkları bir bölücü içine koymak için, parçacıkları arzu edilen konumlarına yaklaşık olarak yerleştirin, onları seçin ve Form > Lay Out Horizontally in Splitter ya da Form > Lay Out Vertically in Splitter'ı seçin.

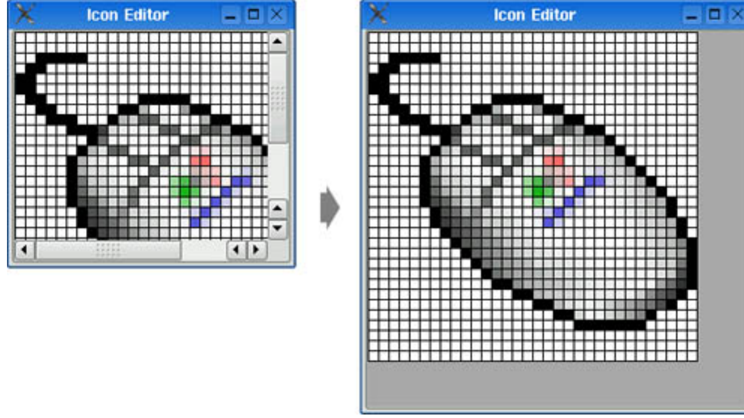
Kaydırılabilir Alanlar

`QScrollArea` sınıfı bir kaydırılabilir görüntü kapısı(scrollable viewport) ve iki kaydırma çubuğu(scroll bar) sağlar. Eğer bir parçacığa kaydırma çubukları eklemek istersek, kendi `QScrollBar`'larımızı somutlaştırmak ve kaydırma işlevlerini kendimiz gerçekleştirmektense, bir `QScrollArea` kullanmak daha basittir.

`QScrollArea` kullanmanın yolu, kaydırma çubuğu eklemek istediğimiz parçacık ile `setWidget()`'ı çağırmaktır. `QScrollArea`, parçacığı otomatik olarak görüntü kapısının (`QScrollArea::viewport()` sayesinde erişilebilir) bir çocuğu yapar, tabi eğer henüz yapılmamışsa. Örneğin, eğer `IconEditor` parçacığı etrafında kaydırma çubukları istiyorsak, şunu yazabiliriz:

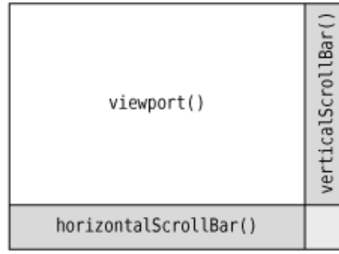
```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));

    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
    scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
    scrollArea.viewport()->setAutoFillBackground(true);
    scrollArea.setWindowTitle(QObject::tr("Icon Editor"));

    scrollArea.show();
```

```
    return app.exec();
}
```



Şekil 6.11

QScrollArea (Şekil 6.12'de şematik olarak gösteriliyor), parçacığı, parçacığın geçerli boyutunda ya da eğer parçacık henüz yeniden boyutlandırılmamışsa boyut ipucunu kullanarak sunar. setWidgetResizable-(true)'yu çağırarak, QSrollArea'ya, boyut ipucu haricindeki ekstra boşluğun avantajını elde etmesi için, parçacığı otomatik olarak yeniden boyutlandırmasını söyleyebiliriz.



Şekil 6.12

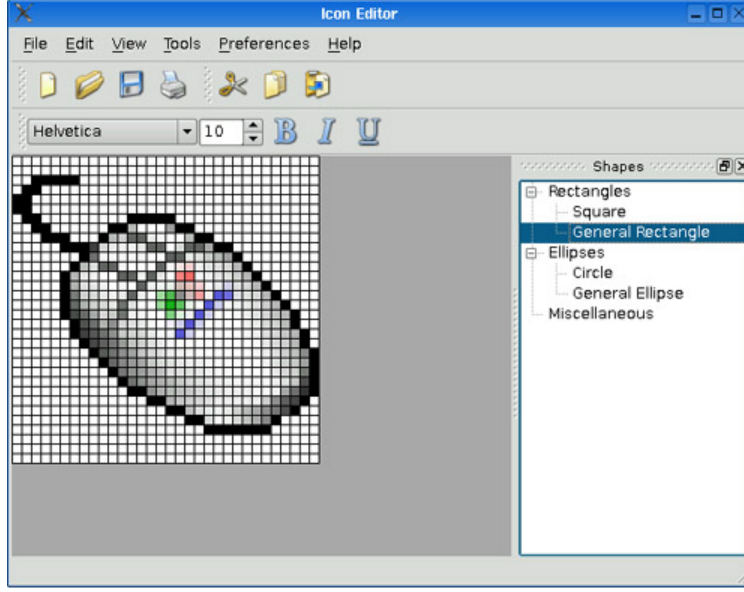
Varsayılan olarak, kaydırma çubukları sadece görüntü kapısı çocuk parçacıktan küçük olduğunda gösterilir. Kaydırma çubuğu politikalarını ayarlayarak, kaydırma çubuklarının her zaman gösterilmesini mecbur kılabiliriz.

```
scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

QScrollArea birçok işlevini QAbstractScrollArea'dan miras almıştır. QTextEdit ve QAbstractItemView (Qt'un öğe görüntüleme sınıflarının temel sınıfı) gibi sınıflar da QAbstract-ScrollArea'dan türetilmiştir, bu nedenle kayırma çubukları edinmek için onları bir QScrollArea içinde paketlememiz gerekmez.

Yapışkan Pencereler ve Araç Çubukları

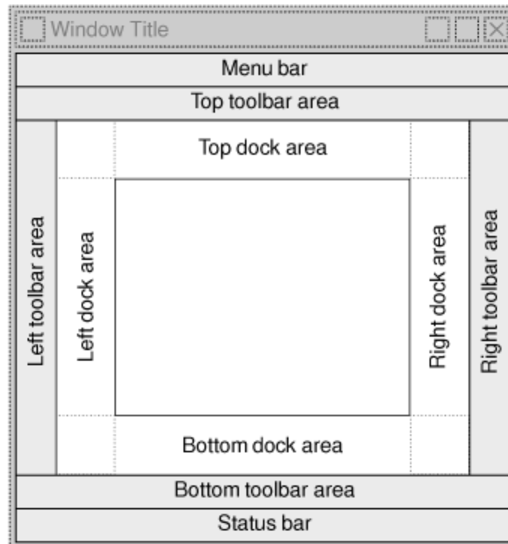
Yapışkan pencereler(dock windows) bir QMainWindow içerisinde, yapışkan pencere alanlarına(dock window area) yapışabilen ya da bağımsız olarak dolaşabilen pencerelerdir. QMainWindow, merkez parçacığın yukarısında, altında, solunda ve sağında olmak üzere toplam dört adet yapışkan pencere alanı sağlar. Microsoft Visual Studio ve Qt Linguist gibi uygulamalar esnek bir kullanıcı arayüzü sağlamak için yapışkan pencereleri geniş ölçüde kullanırlar. Qt'da yapışkan pencereler, QDockWidget'ın örnekleridirler. Şekil 6.13'te araç çubukları ve bir yapışkan penceresi olan bir Qt uygulaması gösteriliyor.



6.13

Her yapışkan pencere, yapıştırıldığında dahi kendi başlık çubuğuna sahiptir. Kullanıcılar yapışkan bir pencereyi başlık çubuğundan sürükleyerek, bir yapışma alanından bir diğerine taşıyabilirler. Ayrıca, yapışkan pencereyi herhangi bir yapışma alanının dışına taşıyarak, yapışkan pencerenin bağımsız bir pencere olarak dolaşması da sağlanılabilir. Serbest yüzen(Free-floating) yapışkan pencereler her zaman ana pencerelerinin en üst seviyesindedirler. Kullanıcılar bir `QDockWidget`'ı pencerenin başlık çubuğundaki kapatma butonuna tıklayarak kapatabilirler. Bu özelliklerin herhangi biri `QDockWidget::setFeatures()` çağrılarak devre dışı bırakılabilir.

Qt'un daha evvelki versiyonlarında, araç çubukları yapışkan pencereler gibi davranırdılar ve aynı yapışma alanlarını kullanırlardı. Qt 4 ile başlayarak, araç çubukları merkez parçacık etrafındaki kendi alanlarına otururlar (Şekil 6.14'te görüldüğü gibi) ve kendi alanlarından kopartılamazlar. Eğer yüzen bir araç çubuğu gerekirse, onu kolaylıkla bir `QDockWidget` içerisine koyabiliriz.



Şekil 6.14

Noktalı çizgilerle gösterilen köşeler, bitişik yapışma alanlarından her ikisine de ait olabilir. Örneğin, QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea)'yı çağırarak sol üst köşeyi sol yapışma alanına ait yapabilirdik.

Sıradaki kod parçası, var olan bir parçacığın (bu durumda bir QTreeWidget) bir QDockWidget içine nasıl paketleneceğini ve sağ yapışma alanı içine nasıl ekleneceğini gösterir:

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setObjectName("shapesDockWidget");
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
                                  | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

setAllowedAreas() çağrısı, yapışma alanlarının yapışkan pencere kabul edebilme kısıtlamalarını açıkça belirtir. Burada, kullanıcının sadece, parçacığın makul bir şekilde görüntülenmesi için yeterli dikey boşluğa sahip olan sol ve sağ yapışma alanlarına yapışkan pencere sürüklemesine izin verdik. Eğer izin verilmeyen alanlar açık bir şekilde belirtilmemişse, kullanıcı yapışkan pencereleri dört alana da sürükleyebilir.

Her QObject'e bir "nesne ismi(object name)" verilebilir. Bu isim hata ayıklamada kullanışlı olabilir ve bazı test araçları tarafından kullanılabilir. Normalde, parçacıklara nesne isimleri verme zahmetine girmeyiz, fakat yapışkan pencere ve araç çubukları oluşturduğumuzda, eğer yapışkan pencere ve araç çubuklarının geometrilerini ve konumlarını kaydetmek ve geri yüklemek için QMainWindow::saveState() ve QMainWindow::restoreState()'i kullanmak istiyorsak, bunu yapmamız gereklidir.

İşte, QMainWindow altsınıfının kurucusundan, bir QComboBox, bir QSpinBox ve birkaç QToolButtons içeren bir araç çubuğunun nasıl oluşturulduğu gösteren kod parçası:

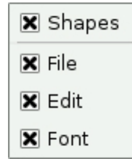
```
QToolBar *fontToolBar = new QToolBar(tr("Font"));
fontToolBar->setObjectName("fontToolBar");
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
fontToolBar->addAction(underlineAction);
fontToolBar->setAllowedAreas(Qt::TopToolBarArea
                             | Qt::BottomToolBarArea);
addToolBar(fontToolBar);
```

Eğer tüm yapışkan pencerelerin ve araç çubuklarının konumlarını kaydedebilmek ve böylece uygulamanın bir sonraki çalışmasında onları geri yükleyebilmek istersek, bir QSplitter'ın konumunu, QMainWindow'un saveState() ve restoreState() fonksiyonlarını kullanarak kaydetmekte kullandığımız koda benzer bir kod yazabiliriz:

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("state", saveState());
    settings.endGroup();
}
```

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    restoreGeometry(settings.value("geometry").toByteArray());
    restoreState(settings.value("state").toByteArray());
    settings.endGroup();
}
```

Son olarak, `QMainWindow`, tüm yapışkan pencereleri ve araç çubuklarını listeleyen bir bağlam menü sağlar. Bu menü Şekil 6.15'te gösteriliyor. Kullanıcı bu menüyü kullanarak yapışkan pencereleri kapatabilir ve geri yükleyebilir ya da araç çubuklarını gizleyebilir ve geri yükleyebilir.



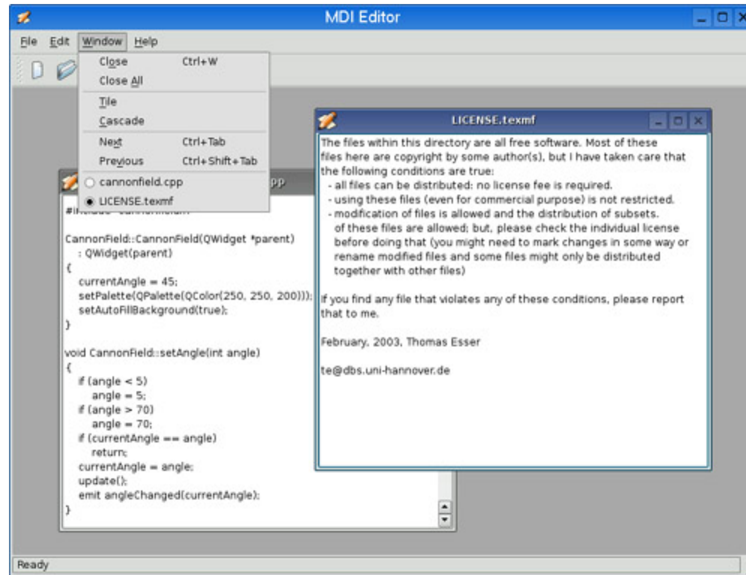
Şekil 6.15

Çoklu Doküman Arayüzü

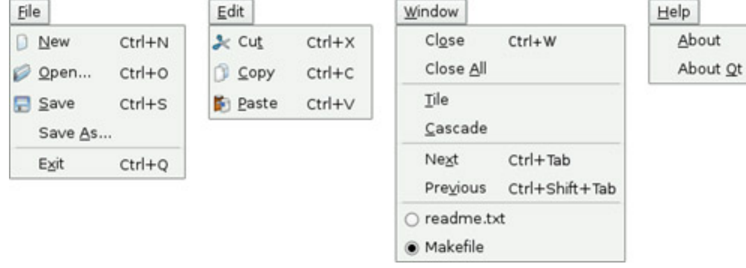
Ana penceresinin merkez alanı içinde çoklu dokümanlar sağlayan uygulamalara çoklu doküman arayüz uygulamaları, ya da kısaca İngilizce karşılığının kısaltması olan MDI(Multiple Document Interface) uygulamaları adı verilir. Qt'da, MDI uygulamaları merkez parçacık olarak `QMdiArea` kullanarak ve her bir doküman penceresini bir `QMdiArea` altpenceresi(subwindow) yaparak oluşturulur.

MDI uygulamaları için, hem pencereleri hem de pencerelerin listesini yönetmek için bazı komutlar içeren bir Window menüsü sağlamak geleneksel bir davranıştır. Aktif pencere bir onay imi(check mark) ile belirtilir. Kullanıcı herhangi bir pencerenin Window menüsündeki girdisine tıklayarak onu aktif edebilir.

Bu kısımda, bir MDI uygulamasının nasıl oluşturulduğunu ve Window menüsünün nasıl gerçekleştirildiğini göstermek için, Şekil 6.16'da görünen MDI Editor uygulamasını geliştireceğiz. Uygulamanın tüm menüleri Şekil 6.17'de gösteriliyor.



Şekil 6.16



Şekil 6.17

Uygulama iki sınıftan meydana gelir: `MainWindow` ve `Editor`. Kodun çoğu Spreadsheet uygulamasınınkinin aynısı ya da benzeri olduğu için, biz sadece MDI ile alâkalı kodu sunacağız.

`MainWindow` sınıfı ile başlayalım.

```
MainWindow::MainWindow()
{
    mdiArea = new QMdiArea;
    setCentralWidget(mdiArea);
    connect(mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow*)),
            this, SLOT(updateActions()));

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();

    setWindowIcon(QPixmap(":/images/icon.png"));
    setWindowTitle(tr("MDI Editor"));
    QTimer::singleShot(0, this, SLOT(loadFiles()));
}
```

`MainWindow` kurucusunda, bir `QMdiArea` oluştururuz ve onu merkez parçacık yaparız. `QMdiArea`'nın `subWindowActivated()` sinyalini, pencere menüsünü güncel tutmada kullanacağımız yuvaya bağlarız ve eylemlerin, uygulamanın durumuna bağlı olarak etkin ya da devre dışı olmasını sağlarız.

Kurucunun sonunda, bir tek atış zamanlayıcısını(single-shot timer) 0 milisaniyelik bir arayla `loadFiles()` fonksiyonunu çağırması için ayarlarız. Bunun gibi zamanlayıcılar, olay döngüsü boşta olduğunda mola verir. Pratikte bunun anlamı; kurucu bitecek ve ana pencere gösterildikten sonra `loadFiles()` çağrılacak. Eğer bunu yapmazsak ve yüklenecek çok büyük dosyalar varsa, kurucu tüm dosyalar yüklenene kadar bitmeyecektir ve bu arada, kullanıcı ekranda hiçbir şey görmeyecektir ve belki de uygulamanın başlamasının başarısız olduğunu düşünecektir.

```
void MainWindow::loadFiles()
{
    QStringList args = QApplication::arguments();
    args.removeFirst();
    if (!args.isEmpty()) {
        foreach (QString arg, args)
            openFile(arg);
        mdiArea->cascadeSubWindows();
    } else {
        newFile();
```

```
    }
    mdiArea->activateNextSubWindow();
}
```

Eğer kullanıcı uygulamayı komut satırında(command line) bir ya da daha fazla dosya ismiyle başlatırsa, bu fonksiyon her bir dosyayı yüklemeyi dener. Öyle ki altpencereler şelâlesinin sonunda kullanıcı onları kolaylıkla görebilir. Qt'un kendine özgü -style ve -font gibi komut satırı seçenekleri `QApplicatin` kurucusu tarafından argüman listesinden otomatik olarak silinir. Bu nedenle, eğer komut satırına şunu yazarsak:

```
mdieditor -style motif readme.txt
```

`QApplication::arguments()` iki öğe("`mdieditor`" ve "`readme.txt`") içeren bir `QStringList` döndürür ve MDI Editor uygulaması, `readme.txt` dokümanı ile birlikte açılır.

Eğer komut satırında hiçbir dosya belirtilmemişse, yeni, boş bir editör altpenceresi oluşturulur, öyle ki kullanıcı hemen yazmaya başlayabilir. `activateNextSubWindow()` çağrısı, bir editör penceresine odaklanıldığı anlamındadır ve `updateActions()` fonksiyonunun Window menüsünü güncellemesini ve uygulamanın durumuna göre eylemlerin etkinleştirilmesini veya devre dışı bırakmasını sağlar.

```
void MainWindow::newFile()
{
    Editor *editor = new Editor;
    editor->newFile();
    addEditor(editor);
}
```

`newFile()` yuvası File > New menü seçeneğine tekabül eder. Bir `Editor` parçacığı oluşturur ve onu `addEditor()` private fonksiyonuna aktarır.

```
void MainWindow::open()
{
    Editor *editor = Editor::open(this);
    if (editor)
    addEditor(editor);
}
```

`open()` fonksiyonu File > Open'a tekabül eder. Bir dosya diyaloğu belirmesini sağlayan `Editor::open()` statik fonksiyonunu çağırır. Eğer kullanıcı bir dosya seçerse, yeni bir `Editor` oluşturulur, dosyanın metni içine okunur ve eğer okuma başarılı ise, `Editor`'a işaret eden bir işaretçi döndürülür. Eğer kullanıcı dosya diyaloğunu iptal ederse, boş bir işaretçi döndürülür ve kullanıcı bundan haberdar edilir. Dosya operasyonlarını `Editor` sınıfı içinde gerçekleştirmek, `MainWindow` sınıfı içinde gerçekleştirmekten daha mantıklıdır, çünkü her bir Editor'ın kendi bağımsız durumunu sürdürmesi gerekir.

```
void MainWindow::addEditor(Editor *editor)
{
    connect(editor, SIGNAL(copyAvailable(bool)),
            cutAction, SLOT(setEnabled(bool)));
    connect(editor, SIGNAL(copyAvailable(bool)),
            copyAction, SLOT(setEnabled(bool)));

    QMdiSubWindow *subWindow = mdiArea->addSubWindow(editor);
    windowMenu->addAction(editor->windowMenuAction());
    windowActionGroup->addAction(editor->windowMenuAction());
    subWindow->show();
}
```

addEditor() private fonksiyonu yeni bir Editor parçacığının ilk kullanıma hazırlanmasını tamamlamak için newFile() ve open()'dan çağrılır. İki sinyal-yuva bağlantısı kurarak başlar. Bu bağlantılar, Edit > Cut ve Edit > Copy menü seçeneklerinin, seçilmiş metin olup olmamasına bağlı olarak etkinleştirmesini ya da devre dışı bırakılmasını sağlar.

MDI kullanmamızdan dolayı, birçok Editor parçacığının kullanımda olacak olması ihtimal dâhilindedir. Biz sadece, aktif Editor penceresinden gelecek copyAvaiable(bool) sinyalinin yanıtıyla ilgilendiğimiz ve diğerlerinden gelecek cevapla ilgilenmediğimiz için, bu kaygılanılacak bir şeydir. Fakat bu sinyaller sadece ve daima aktif pencereler tarafından yayılabilirler; dolayısıyla bu, pratikte bir sorun oluşturmaz.

QMdiArea::addSubWindow() fonksiyonu yeni bir QMdiSubWindow oluşturur, parçacığı altpencere içine bir parametre olarak aktararak yükler ve altpencereyi döndürür. Sonra, Window menüsünde, pencereyi temsil eden bir QAction oluştururuz. Eylem, biraz sonra değineceğimiz Editor sınıfı tarafından sağlanır. Ayrıca eylemi bir QActionGroup nesnesine ekleriz. QActionGroup bir seferde sadece bir Window menüsü öğesinin işaretlenmiş olmasını sağlar. Son olarak, yeni QMdiSubWindow'u görünür yapmak için üzerinde show() fonksiyonunu çağırırız.

```
void MainWindow::save()
{
    if (activeEditor())
        activeEditor()->save();
}
```

save() yuvası, eğer aktif bir editör varsa, o aktif editör üzerinde Editor::save()'i çağırır. Yine, kodun yaptığı gerçek iş Editor sınıfı içine yerleştirilmiştir.

```
Editor *MainWindow::activeEditor()
{
    QMdiSubWindow *subWindow = mdiArea->activeSubWindow();
    if (subWindow)
        return qobject_cast<Editor *>(subWindow->widget());
    return 0;
}
```

activeEditor() private fonksiyonu aktif altpencere içinde tutulan parçacığı bir Editor işaretçisi olarak döndürür. Eğer aktif altpencere yoksa boş bir işaretçi döndürür.

```
void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}
```

cut() yuvası aktif editör üzerinde Editor::cut()'ı çağırır. Burada copy() ve paste() yuvalarını göstermiyoruz çünkü onlar da aynı modeli izliyorlar.

```
void MainWindow::updateActions()
{
    bool hasEditor = (activeEditor() != 0);
    bool hasSelection = activeEditor()
                        && activeEditor()->textCursor().hasSelection();

    saveAction->setEnabled(hasEditor);
```

```
    saveAsAction->setEnabled(hasEditor);
    cutAction->setEnabled(hasSelection);
    copyAction->setEnabled(hasSelection);
    pasteAction->setEnabled(hasEditor);
    closeAction->setEnabled(hasEditor);
    closeAllAction->setEnabled(hasEditor);
    tileAction->setEnabled(hasEditor);
    cascadeAction->setEnabled(hasEditor);
    nextAction->setEnabled(hasEditor);
    previousAction->setEnabled(hasEditor);
    separatorAction->setVisible(hasEditor);

    if (activeEditor())
        activeEditor()->windowMenuAction()->setChecked(true);
}
```

`subWindowActivated()` sinyali, yeni bir altpencere aktif olduğunda ya da son altpencere kapatıldığında (bu durumda, parametresi boş bir işaretçidir) yayılır. Bu sinyal `updateActions()` yuvasına bağlanır.

Menü seçeneklerinin çoğu eğer aktif bir pencere varsa anlam kazanır, bu nedenle, aktif bir pencere olmadığında, onları devre dışı bırakırız. Sonda, aktif pencereyi temsil eden `QAction` üzerinde `setChecked()`'i çağırırız. `QActionGroup`'a şükürler olsun ki, önceki aktif pencerenin onay imini açıkça kaldırmamız gerekmez.

```
void MainWindow::createMenus()
{
    ...
    windowMenu = menuBar()->addMenu(tr("&Window"));
    windowMenu->addAction(closeAction);
    windowMenu->addAction(closeAllAction);
    windowMenu->addSeparator();
    windowMenu->addAction(tileAction);
    windowMenu->addAction(cascadeAction);
    windowMenu->addSeparator();
    windowMenu->addAction(nextAction);
    windowMenu->addAction(previousAction);
    windowMenu->addAction(separatorAction);
    ...
}
```

`createMenus()` private fonksiyonu, Window menüsünü eylemlerle doldurur. Tüm eylemler, menülerin tipik bir örnekleridirler ve `QMdiArea`'nın `closeActiveSubWindow()`, `closeAllSubWindows()`, `tileSubWindows()` ve `cascadeSubWindows()` yuvaları kullanılarak kolayca gerçekleştirilebilirler. Kullanıcının açtığı her yeni pencere Window menüsünün eylemler listesine eklenir. (Bu, `addEditor()` fonksiyonu içinde yapılır.) Kullanıcı bir editör penceresi kapattığında, pencereye ilişkin eylem silinir (eylem, editör penceresi tarafından sahiplenildiğinden dolayı) ve böylece eylem otomatik olarak Window menüsünden de silinmiş olur.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    mdiArea->closeAllSubWindows();
    if (!mdiArea->subWindowList().isEmpty()) {
        event->ignore();
    } else {
        event->accept();
    }
```

```
}
```

`closeEvent()` fonksiyonu, tüm altpencereleri kapatmaya uyarlanır. Eğer alt-pencerelerden biri kendi kapatma olayını "yok sayarsa(ignore)" (çünkü kullanıcı "kaydedilmemiş değişiklikler" mesaj kutusunu iptal etmiştir), `MainWindow` için olan kapatma olayını yok sayarız; aksi halde, kabul ederiz, ve sonucunda da tüm uygulama kapatılır. Eğer `MainWindow` içinde `closeEvent()`'i uyarlamasaydık, kullanıcıya kaydedilmemiş değişiklikleri kaydetme fırsatı verilmeyecekti.

Artık `MainWindow` kritiğini bitirdik; demek ki `Editor`'ın gerçekleştirimine ilerleyebiliriz. `Editor` sınıfı bir altpencereyi temsil eder. Metin düzenleme işlevleri sağlayan `QTextEdit`'ten türetilmiştir.

Her Qt parçacığının bağımsız (stand-alone) bir pencere olarak kullanılabilmesi gibi, her Qt parçacığı bir `QMdiSubWindow` içine yerleştirilebilir ve MDI alan içinde bir altpencere olarak kullanılabilir.

İşte, sınıf tanımı:

Kod Görünümü:

```
class Editor : public QTextEdit
{
    Q_OBJECT

public:
    Editor(QWidget *parent = 0);

    void newFile();
    bool save();
    bool saveAs();
    QSize sizeHint() const;
    QAction *windowMenuAction() const { return action; }

    static Editor *open(QWidget *parent = 0);
    static Editor *openFile(const QString &fileName,
                            QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void documentWasModified();

private:
    bool okToContinue();
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);

    QString curFile;
    bool isUntitled;
    QAction *action;
};
```

Spreadsheet uygulamasının `MainWindow` sınıfındaki private fonksiyonların dördü `Editor` sınıfı içinde de yer alıyor: `okToContinue()`, `saveFile()`, `setCurrentFile()` ve `strippedName()`.

```
Editor::Editor(QWidget *parent)
    : QTextEdit(parent)
{
    action = new QAction(this);
    action->setCheckable(true);
    connect(action, SIGNAL(triggered()), this, SLOT(show()));
    connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));

    isUntitled = true;

    connect(document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));

    setWindowIcon(QPixmap(":/images/document.png"));
    setWindowTitle("[*]");
    setAttribute(Qt::WA_DeleteOnClose);
}
```

Önce, uygulamanın Window menüsünde editörü temsil eden bir `QAction` oluştururuz ve eylemi `show()` ve `setFocus()` yuvalarına bağlarız.

Kullanıcıya istediği kadar editör penceresi oluşturma imkânı verdiğimiz için, onları isimlendirmek için bazı hazırlıklar yapmalıyız. Bunu kıvırmanın yaygın bir yolu, bir sayı içeren isimlerle onları ayırmaktır (`document1.txt` gibi). Kullanıcıdan sağlanan isim ile bizim programlanabilir bir biçimde oluşturduğumuz ismi ayırt etmek için `isUntitled` değişkenini kullanırız.

Metin belgesinin `contentsChanged()` sinyalini, `documentWasModified()` yuvasına bağlarız. Bu yuva sadece `setWindowModified(true)`'yu çağırır.

Son olarak, kullanıcı bir `Editor` penceresini kapattığında, bellek sızıntılarına engel olmak için `Qt::WA_DeleteOnClose` niteliğini ayarlarız.

```
void Editor::newFile()
{
    static int documentNumber = 1;

    curFile = tr("document%1.txt").arg(documentNumber);
    setWindowTitle(curFile + "[*]");
    action->setText(curFile);
    isUntitled = true;
    ++documentNumber;
}
```

`newFile()` fonksiyonu yeni doküman için `document1.txt` gibi bir isim üretir. Kod, kurucu yerine, `newFile()` içinde yer alır, çünkü var olan bir doküman açmak için `open()`'ı çağırdığımızda, numaraları ziyan etmek istemeyiz. `documentNumber` statik bildirildiği için, tüm `Editor` örnekleri arasında paylaşılır.

Yeni dosya oluşturmaya ek olarak, kullanıcılar sıklıkla, bir dosya diyaloğundan ya da son açılan dosyalar listesi gibi bir listeden seçerek, var olan dosyaları açmak ister. İki statik fonksiyon bunları sağlar: `open()` dosya sisteminden bir dosya ismi seçmeye yarar ve `openFile()` bir `Editor` örneği oluşturur ve belirtilen dosyanın içeriğini onun içine okur.

```
Editor *Editor::open(QWidget *parent)
{
    QString fileName =
```

```
            QFileDialog::getOpenFileName(parent, tr("Open"), ".");
    if (fileName.isEmpty())
        return 0;

    return openFile(fileName, parent);
}
```

open() statik fonksiyonu, kullanıcının, içinden bir dosya seçebileceği bir dosya diyaloğu belirmesini sağlar. Eğer bir dosya seçilmişse, bir Editor oluşturması için openFile() çağrılır ve dosyanın içeriği onun içine okunur.

```
Editor *Editor::openFile(const QString &fileName, QWidget *parent)
{
    Editor *editor = new Editor(parent);

    if (editor->readFile(fileName)) {
        editor->setCurrentFile(fileName);
        return editor;
    } else {
        delete editor;
        return 0;
    }
}
```

Statik fonksiyon yeni bir Editor parçacığı oluşturarak başlar ve sonra belirtilen sayfayı içine okumayı dener. Eğer okuma başarılıysa, Editor döndürülür; aksi halde, kullanıcı problem hakkında bilgilendirilir (readFile() içinde), editör silinir ve boş bir işaretçi döndürülür.

```
bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}
```

save() fonksiyonu saveFile()'ımı, yoksa saveAs()'imi çağıracağını belirlemek için isUntitled değişkenini kullanır.

```
void Editor::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        event->accept();
    } else {
        event->ignore();
    }
}
```

closeEvent() fonksiyonu kullanıcıya kaydedilmemiş değişiklikleri kaydetme imkânı vermek için uyarlanır. Mantık, kullanıcıya "Do you want to save your changes?" sorusunu yönelten bir mesaj kutusu belirmesini sağlayan okToContinue() fonksiyonu içinde kodlanır. Eğer okToContinue() true döndürürse, kapatma olayını kabul ederiz; aksi halde, olayı yok sayarız ve pencereye herhangi bir etkide bulunmadan, olaydan vazgeçeriz.

```
void Editor::setCurrentFile(const QString &fileName)
{
```

```
    curFile = fileName;
    isUntitled = false;
    action->setText(strippedName(curFile));
    document()->setModified(false);
    setWindowTitle(strippedName(curFile) + "[*]");
    setWindowModified(false);
}
```

setCurrentFile() fonksiyonu, curFile ve isUntitled değişkenlerini güncellemek, pencere başlığını ve eylem metnini ayarlamak ve dokümanın "modified(değiştirilmiş)" bayrağını false olarak ayarlamak için openFile() ve saveFile()'dan çağrılır. Kullanıcı, editör içindeki metni değiştirdiğinde, söz konusu QTextDocuments, contentsChanged() sinyalini yayar ve iç "modified" bayrağını true olarak ayarlar.

```
QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'),
                 25 * fontMetrics().lineSpacing());
}
```

Son olarak, sizeHint() fonksiyonu, 'x' harfinin genişliğini ve bir metin satırının yüksekliğini baz alan bir boyut döndürür. QMdiArea, boyut ipucunu, pencereye ilk boyutunu vermede kullanır.

BÖLÜM 7: OLAY İŞLEME

Olaylar(events), pencere sistemi ya da Qt'un kendisi tarafından, çeşitli oluşlara karşılık üretilirler. Kullanıcı klavyeden bir tuşa ya da bir fare butonuna bastığında ya da onu serbest bıraktığında, bir tuş olayı(key event) ya da bir fare olayı(mouse event) üretilir; bir pencere ilk kez gösteriliğinde, pencere için bir çiz olayı üretilir. Olayların birçoğu kullanıcı eylemlerine cevaben üretilirler, fakat zamanlayıcı olayları(timer events) gibi bazıları, sistemden bağımsız olarak üretilirler.

Qt ile programlarken, olaylar hakkında nadiren düşünmemiz gerekir, çünkü Qt parçacıkları önemli bir şey olduğunda sinyaller yayarlar. Olaylar, kendi özel parçacıklarınızı yazdığınızda ya da var olan Qt parçacıklarının davranışını değiştirmek istediğinizde kullanışlı olurlar.

Olaylar sinyallerle karıştırılmamalıdır. Bir kural olarak, sinyaller bir parçacığı kullanırken kullanışlıdırlar, oysa olaylar bir parçacık gerçekleştirirken kullanışlıdırlar. Örneğin, QPushButton kullanıyorken, fare ya da klavye olayları yerine, sinyal yayılabilmesi için onun clicked() sinyaliyle ilgileniriz. Fakat QPushButton gibi bir sınıf gerçekleştiriyorsak, fare ve tuş olaylarını işleyen ve gerektiğinde clicked() sinyalini yayan kodu yazmamız gerekir.

Olay İşleyicilerini Uyarlama

Qt'da, bir olay, bir QEvent altsınıfı örneğidir. Qt, her biri bir enum değer olarak tanımlanmış yüzden fazla olay tipini işler. Örneğin, QEvent::type() fare butonuna basma olayları için QEvent::Mouse-ButtonPress'i döndürür.

Olay tiplerinin birçoğu, sade bir QEvent nesnesinin saklayabileceğinden daha fazla bilgi gerektirir; örneğin, fare butonuna basma olayları, olayı hangi fare butonunun tetiklendiğini saklamaya ihtiyaç duyması bir tarafa olay meydana geldiğinde fare işaretçisinin konumunun pozisyonunun neresi olduğunu dahi saklamaya ihtiyaç duyar. Bu ek bilgi, QMouseEvent gibi özel QEvent altsınıflarında saklanır.

Olaylar, nesneleri, QObject'ten miras aldıkları event() fonksiyonu sayesinde bilgilendirir. QWidget içindeki event() gerçekleştirimi, mousePressEvent(), keyPressEvent() ve paintEvent() gibi en yaygın olay tiplerini özel olay işleyicilere(event handler) gönderir.

Biz zaten, daha evvelki bölümlerde MainWindow ve IconEditor'ı gerçekleştirirken, olay işleyicilerin birçoğunu görmüştük. Diğer olay tipleri, QEvent referans dokümantasyonunda listelenmiştir ve ayrıca özel olay tipleri oluşturmak da mümkündür. Burada, daha fazla açıklamayı hak eden yaygın iki olayın kritiğini yapacağız: tuş olayları ve zamanlayıcı olayları.

Tuş olayları, keyPressEvent() ve keyReleaseEvent() uyarlanarak işlenebilir. Normalde, tamamlayıcı tuşlar Ctrl, Shift ve Alt keyPressEvent() içinde QKeyEvent::modifiers kullanılarak kontrol edilebildiği için sadece keyPressEvent()'i uyarlamamız gerekir. Örneğin, bir CodeEditor parçacığı gerçekleştiriyorsak, Home ve Ctrl + Home arasındaki farkı ayırt eden keyPressEvent() şunun gibi görünecektir:

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Home:
        if (event->modifiers() & Qt::ControlModifier) {
            goToBeginningOfDocument();
        } else {
            goToBeginningOfLine();
        }
        break;
    case Qt::Key_End:
        ...
    default:
        QWidget::keyPressEvent(event);
    }
}
```

Tab ve Backtab(Shift + Tab) tuşları özel durumlardır. QWidget::event(), keyPressEvent()'i çağırmadan önce, odağın hangi parçacığa kayacağıyla ilgilenir. Bu davranış genellikle bizim istediğimizdir, fakat bir CodeEditor parçacığında, Tab ile satır başı yapmak isteyebiliriz. event() uyarlanışı artık şöyle görünecektir:

```
bool CodeEditor::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            insertAtCurrentPosition('\t');
            return true;
        }
    }
    return QWidget::event(event);
}
```

Eğer olay bir tuşa basma ise, `QEvent` nesnesini bir `QKeyEvent`'e dönüştürürüz ve hangi tuşa basıldığını kontrol ederiz. Eğer tuş Tab ise, bazı işlemler yaparız ve Qt'a olayı işlediğimizi bildirmek için `true` döndürürüz. Eğer `false` döndürseydik, Qt, olayı ebeveyn parçacığa aktaracaktı.

Tuş bağlantılarını(key bindings) gerçekleştirmek için bir diğer yaklaşım, bir `QAction` kullanmaktır. Örneğin, eğer `goToBeginningOfLine()` ve `goToBeginningOfDocument()`, `CodeEditor` parçacığı içinde public yuvalar ise ve `CodeEditor`, bir `MainWindow` sınıfı içinde merkez parçacık olarak kullanılıyorsa, tuş bağlantılarını şu kodla ekleyebiliriz:

```
MainWindow::MainWindow()
{
    editor = new CodeEditor;
    setCentralWidget(editor);

    goToBeginningOfLineAction =
            new QAction(tr("Go to Beginning of Line"), this);
    goToBeginningOfLineAction->setShortcut(tr("Home"));
    connect(goToBeginningOfLineAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfLine()));

    goToBeginningOfDocumentAction =
            new QAction(tr("Go to Beginning of Document"), this);
    goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
    connect(goToBeginningOfDocumentAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfDocument()));
    ...
}
```

Bu, bir menü veya araç çubuğuna komutlar eklemeyi kolaylaştırır.

Tuş bağlantıları varsayılan olarak, onu içeren pencere aktif olduğunda etkinleşen bir parçacık üzerinde `QAction` veya `QShortcut` kullanılarak ayarlanır. Bu, `QAction::setShortcutContext()` ya da `QShortcut::setContext()` kullanılarak değiştirilebilir.

Bir başka yaygın olay tipi, zamanlayıcı olayıdır. Birçok olay tipi, bir kullanıcı eyleminin sonucu olarak meydana gelirken, zamanlayıcı olayları, uygulamaya düzenli zaman aralıklarında işlem gerçekleştirme imkânı verir. Zamanlayıcı olayları, yanıp sönen imleçler ve diğer animasyonları gerçekleştirmede ya da sadece görüntüyü tazelemede kullanılabilirler.

Zamanlayıcı olaylarını örnekle açıklamak için, Şekil 7.1'de görülen `Ticker` parçacığını gerçekleştireceğiz. Bu parçacık, her 30 milisaniyede bir piksel sola kayan bir ana başlık metni gösterir. Eğer parçacık metinden daha geniş ise, metin parçacığın bütün genişliğini doldurması için gerektiği kadar tekrar edilir.

say ++ How long it lasted was impossible to say ++ Ho

Şekil 7.1

İşte, başlık dosyası:

Kod Görünümü:

```
#ifndef TICKER_H
#define TICKER_H
```

```
#include <QWidget>

class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)

public:
    Ticker(QWidget *parent = 0);

    void setText(const QString &newText);
    QString text() const { return myText; }
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
    void showEvent(QShowEvent *event);
    void hideEvent(QHideEvent *event);

private:
    QString myText;
    int offset;
    int myTimerId;
};

#endif
```

`Ticker`'da dört olay işleyicisini uyarlarız. Bunlardan üçünü daha önce görmemiştik: `timerEvent()`, `showEvent()` ve `hideEvent()`.

Şimdi, gerçekleştirimin kritiğini yapalım:

```
include <QtGui>

#include "ticker.h"

Ticker::Ticker(QWidget *parent)
    : QWidget(parent)
{
    offset = 0;
    myTimerId = 0;
}
```

Kurucu `offset` değişkenini 0'a ilklendirir. Metnin çizildiği x-koordinatı `offset` değişkeninden sağlanır. Zamanlayıcı ID'leri her zaman 0'dan farklıdır, bu nedenle, hiçbir zamanlayıcı başlatılmadığını belirtmek için 0'ı kullanırız.

```
void Ticker::setText(const QString &newText)
{
    myText = newText;
    update();
    updateGeometry();
}
```

`setText()` fonksiyonu, metnin görüntülenişini ayarlar. Bir yeniden boyama istemek için `update()`'i ve `Ticker` parçacığı hakkında bir boyut ipucu değişikliğinden sorumlu yerleşim yöneticisini bilgilendirmek için `updateGeometry()`'i çağırır.

```
QSize Ticker::sizeHint() const
{
    return fontMetrics().size(0, text());
}
```

`sizeHint()` fonksiyonu, parçacığın ideal boyutu olarak, metin ihtiyaç duyduğu boşluğu döndürür. `QWidget::fontMetrics()`, parçacığın yazıtipine ilişkin bilgiyi sağlamada kullanılabilen bir `QFontMetrics` nesnesi döndürür. Bu durumda onu, metnimize uygun boyutu öğrenmede kullanırız. `QFontMetrics::size()`'ın ilk argümanı, basit karakter katarları için gerekmeyen bir bayraktır, bu nedenle sadece 0'ı aktarırız.

```
void Ticker::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    int textWidth = fontMetrics().width(text());
    if (textWidth < 1)
        return;
    int x = -offset;
    while (x < width()) {
        painter.drawText(x, 0, textWidth, height(),
                         Qt::AlignLeft | Qt::AlignVCenter, text());
        x += textWidth;
    }
}
```

`paintEvent()` fonksiyonu, `QPainter::drawText()`'i kullanarak metni çizer. Metnin gerektirdiği yatay boşluğu belirlemek için `fontMetrics()`'i kullanır ve sonra `offset`'i hesaba katarak, metni -gerekirse birkaç sefer- çizer.

```
void Ticker::showEvent(QShowEvent * /* event */)
{
    myTimerId = startTimer(30);
}
```

`showEvent()` fonksiyonu bir zamanlayıcı başlatır. `QObject::startTimer()`'a çağrı, daha sonra zamanlayıcıyı tanımak için kullanabileceğimiz bir ID numarası döndürür. `QObject`, her biri kendi zaman aralığı olan çoklu-bağımsız zamanlayıcıları destekler. `startTimer()` çağrısından sonra, Qt, "yaklaşık olarak" her 30 milisaniyede bir, bir zamanlayıcı olayı üretir; doğruluk, programın çalıştığı işletim sistemine bağlıdır.

`startTimer()`'ı `Ticker` kurucusu içinde çağırabilirdik, fakat bu şekilde Qt'un sadece, parçacık görünür olduğunda zamanlayıcı olayları üretmesini sağlayarak, biraz kaynak tasarrufu yaparız.

```
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        ++offset;
        if (offset >= fontMetrics().width(text()))
            offset = 0;
        scroll(-1, 0);
    } else {
        QWidget::timerEvent(event);
    }
}
```

Sistem timerEvent() fonksiyonunu aralıklarla çağırır. Hareketi simüle etmek için offset'i 1 arttırır. Sonra, QWidget::scroll()'u kullanarak parçacığın içeriğini bir piksel sola kaydırır. scroll() yerine update()'i çağırmak yeterli olacaktır, fakat scroll() daha efektiftir, çünkü ekranda var olan pikselleri hareket ettirir ve sadece parçacığın açığa çıkan yeni alanı için bir çiz olayı üretir (bu durumda 1 piksel genişlik çıkar).

Eğer zamanlayıcı olayı bizim ilgilendiğimiz zamanlayıcı için değilse, onu, temel sınıfa aktarırız.

```
void Ticker::hideEvent(QHideEvent * /* event */)
{
    killTimer(myTimerId);
    myTimerId = 0;
}
```

hideEvent() fonksiyonu, zamanlayıcıyı durdurmak için QObject::killTimer'ı çağırır.

Zamanlayıcı olayları düşük-seviyelidir ve eğer çoklu zamanlayıcılara ihtiyacımız varsa, tüm zamanlayıcı ID'lerinin izini sürmek külfetli olabilir. Bu gibi durumlarda, her zamanlayıcı için bir QTimer nesnesi oluşturmak genellikle daha kolaydır. QTimer her zaman aralığında timeout() sinyalini yayar. Ayrıca, QTimer tek atış zamanlayıcıları için kullanışlı bir arayüz de sağlar.

Olay Filtreleri Kurma

Qt'un olay modelinin gerçekten güçlü bir özelliği, bir QObject örneğinin, başka bir QObject örneğinin olaylarını -sonraki nesneler onları görmeden önce- izlemesi için ayarlanabilinmesidir.

Farz edelim ki, birkaç QLineEdit'ten oluşan bir CustomerInfoDialog parçacığına sahibiz ve Space tuşunu kullanarak odağı sonraki QLineEdit'e kaydırmak istiyoruz. Bu standart dışı davranış bir kurum içi uygulama(in-house application) için uygun olabilir. Kolay bir çözüm, bir QLineEdit altsınıfı türetmek ve keyPressEvent()'i, focusNextChild()'ı çağırmaya uyarlamaktır. Tıpkı şunun gibi:

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) {
        focusNextChild();
    } else {
        QLineEdit::keyPressEvent(event);
    }
}
```

Bu yaklaşım bir dezavantaja sahiptir: Eğer formda parçacıkların birkaç farklı türünü kullanırsak (QComboBox'lar ve QSpinBox'lar gibi), onları da aynı davranışı sergilemeye uyarlamamız gerekir. Daha iyi bir yaklaşım, CustomerInfoDialog'unun, çocuk parçacıklarının tuş basma olaylarını izlemesini sağlamak ve gereken davranışları bir izleme kodu(monitoring code) içinde gerçekleştirmektir. Bir olay filtresi(event filter) kurmak iki adımı gerektirir:

1. İzleme nesnesini(monitoring object), hedef nesne üzerinde installEventFilter()'i çağırarak kayda geçirin.
2. Hedef nesnenin olaylarını, gözlemcinin(monitor) eventFilter() fonksiyonu içinde işleyin.

İzleme nesnesini kayda geçirmek için kurucu uygun iyi bir yerdir:

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
```

```
    : QDialog(parent)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}
```

Olay filtresi bir kez kayda geçirilince, `firstNameEdit`, `lastNameEdit`, `cityEdit` ve `phoneNumber-Edit` parçacıklarına gönderilen olaylar, istenilen hedeflerine gönderilmeden önce, `CustomerInfo-Dialog`'un `eventFilter()` fonksiyonuna gönderilirler.

İşte, olayları alan `eventFilter()` fonksiyonu:

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
            || target == cityEdit || target == phoneNumberEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Space) {
                focusNextChild();
                return true;
            }
        }
    }
    return QDialog::eventFilter(target, event);
}
```

Önce, hedef parçacığın, `QLineEdit`'lerden biri olup olmadığını kontrol ederiz. Eğer olay bir tuş basma olayı ise, onu `QKeyEvent`'e dönüştürürüz ve hangi tuşun basıldığını kontrol ederiz.

Eğer basılan tuş Space ise, odağı, odak zincirindeki sonraki parçacığa kaydırmak için `focusNextChild()`'ı çağırırız ve Qt'a olayı işlediğimizi bildirmek için `true` döndürürüz. Eğer `false` döndürseydik, Qt olayı istenilen hedefe gönderecekti ve bunun sonucu olarak, `QLineEdit` içine sahte bir boşluk karakteri eklenmiş olacaktı.

Eğer hedef parçacık bir `QLineEdit` değilse ya da olay bir Space tuşuna basma değilse, kontrolü `eventFilter()`'in temel sınıfının gerçekleştirimine geçiririz.

Qt, olayların işlenmesinde ve filtrelenmesinde beş seviye sunar:

1. **Belirli bir olay işleyicisini uyarlayabiliriz.**
   `mousePressEvent()`, `keyPressEvent()` ve `paintEvent()` gibi olay işleyicilerini uyarlamak, olayları işlemenin en yaygın yoludur. Zaten bunun birçok örneğini gördük.

2. **`QObject::event()`'i uyarlayabiliriz.**
   `event()` fonksiyonunu uyarlayarak, olaylar, belirli olay işleyicilerine ulaşmadan önce onları işleyebiliriz. Bu yaklaşıma en çok Tab tuşunun varsayılan manasını hükümsüz kılmada ihtiyaç duyulur. Ayrıca, belirli olay işleyicisi var olmayan nadir olay tiplerini işlemede de kullanılır(`QEvent::HoweverEnter` gibi). `event()`'i uyarladığımızda, açıkça işlemediğimiz durumlar için temel sınıfın `event()` fonksiyonunu çağırmalıyız.

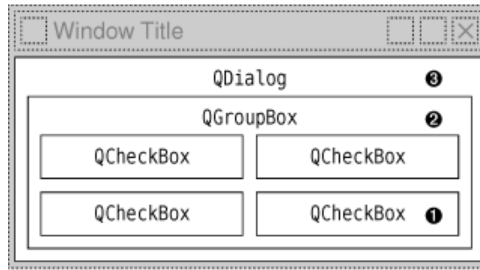
3. **Bir olay işleyicisini tek bir `QObject` üzerine kurabiliriz.**
   Bir nesne, `instalEventFilter()` kullanılarak bir kere kayda geçirildiğinde, hedef nesnenin tüm olayları, önce izleme nesnesinin `eventFilter()` fonksiyonuna gönderilir. Eğer aynı nesne üzerine birçok olay filtresi kurulmuşsa, filtreler, en son kurulandan ilk kurulana doğru, sırayla aktifleştirilirler.

4. **Bir olay işleyicisini bir `QApplication` nesnesi üzerine kurabiliriz.**
   Bir olay filtresi, `qApp` (tek `QApplication` nesnesi) için bir kere kayda geçirildiğinde, uygulama içindeki herhangi bir nesnenin her olayı, diğer olay filtrelerine gönderilmeden önce `eventFilter()`'e gönderilir. Bu yaklaşım hata ayıklama için çok kullanışlıdır. Ayrıca, normalde `QApplication`'ın yok sayacağı, devre dışı olan parçacıklara gönderilen fare olaylarını işlemede de kullanılabilir.

5. **Bir `QApplication` altsınıfı türetebilir ve `notify()`'ı uyarlayabiliriz.**
   Qt, bir olayı yaymak için `QApplication::notify()`'ı çağırır. Bu fonksiyonu uyarlamak, olay işleyicileri olayları elde etmeden önce, tüm olayları elde etmenin tek yoludur. Olay filtreleri genellikle daha kullanışlıdırlar, çünkü birçok eş zamanlı olay filtresi olabilir, fakat sadece bir `notify()` fonksiyonu vardır.

Birçok olay tipi, fare ve tuş olayları da dâhil, nakledilebilirler. Eğer olay, hedef nesneye giderken ya da hedef nesnenin kendi tarafından işlenmemişse, olay işleme süreci tekrar edilir, fakat bu sefer hedef, hedef nesnenin ebeveynidir. Bu böyle, ebeveynden ebeveyne devam eder gider, ta ki olay işleninceye ya da en üst seviyedeki nesneye erişilinceye kadar.

Şekil 7.2, bir tuşa basma olayının, bir diyalog içinde çocuktan ebeveyne nasıl nakledildiğini gösteriyor. Kullanıcı bir tuşa bastığında, olay önce odağın olduğu parçacığa gönderilir, bu durumda sağ alttaki `QCheckBox`'tır. Eğer `QCheckBox` olayı işlemezse, Qt, olayı önce `QGroupBox`'a, son olarak ta `QDialog` nesnesine gönderir.



Şekil 7.2

Yoğun İşlem Sırasında Duyarlı Kalma

`QApplication::exec()`'i çağırdığımızda, Qt'un olay döngüsünü başlatırız. Qt başlangıçta, parçacıkları göstermek ve çizmek için birkaç olay sonuçlandırır. Ondan sonra olay döngüsü çalışır; sıkça, bir olayın meydana gelip gelmediğini ve bu olayların uygulama içinde `QObject`'lere gönderilip gönderilmediğini kontrol eder.

Bir olay işlenirken, ona ek olaylar üretilebilir ve bu olaylar Qt'un olay kuyruğuna(event queue) eklenebilir. Eğer belli bir olayı işlemek için çok fazla zaman harcıyorsak, kullanıcı arayüzü tepkisizleşecektir. Örneğin,

uygulama bir dosyayı kaydederken, pencere sistemi tarafından üretilen hiçbir olay, dosya kaydedilene kadar işlenmeyecektir. Kaydetme sırasında, uygulama, pencere sisteminin kendini yeniden çizme isteklerine cevap vermeyecektir.

Basit bir çözüm, dosya kaydetme kodu içinde sık sık `QApplication::processEvents()`'i çağırmayı sağlamaktır. Bu fonksiyon Qt'a, beklemede olan olayları(pending events) bildirir ve ardından kontrolü çağırana iade eder.

İşte, `processEvents()` kullanarak, kullanıcı arayüzünü nasıl duyarlı tutacağımızın bir örneği (`Spreadsheet`'in dosya kaydetme koduna dayanıyor):

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
        qApp->processEvents();
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

Bu yaklaşımın tehlikesi, kullanıcının, uygulama daha kaydetmeyi bitirmeden, ana pencereyi kapatabilecek olmasıdır, ya da File > Save'e ikinci bir kez daha tıklayabilecek olmasıdır. Bu problem için en kolay çözüm

```
qApp->processEvents();
```

yerine

```
qApp->processEvents(QEventLoop::ExcludeUserInputEvents);
```

yazarak, Qt'a fare ve tuş olaylarını yok saymasını bildirmektir.

Çoğu kez, uzun süren bir işlemin olduğu yerde bir `QProgressDialog` göstermek isteriz. `QProgress–Dialog`, kullanıcıyı uygulamanın ilerlemesi hakkında bilgilendiren bir ilerleme çubuğuna(progress bar) sahiptir. `QProgressDialog` ayrıca, kullanıcıya işlemi iptal etme imkânı veren bir Cancel butonu da sağlar. İşte, bir Spreadsheet dosyasını kaydederken, bu yaklaşımı kullanan örneğimiz:

Kod Görünümü:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QProgressDialog progress(this);
    progress.setLabelText(tr("Saving %1").arg(fileName));
    progress.setRange(0, RowCount);
    progress.setModal(true);

    for (int row = 0; row < RowCount; ++row) {
```

```
        progress.setValue(row);
        qApp->processEvents();
        if (progress.wasCanceled()) {
            file.remove();
            return false;
        }
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    return true;
}
```

Adımların toplam sayısı olarak `NumRows` ile bir `QProgressDialog` oluştururuz. Sonra, her bir satır için `setValue()`'yu çağırarak, ilerleme çubuğunu güncel tutarız. `QProgressDialog` otomatik olarak geçerli ilerleme değerini, adımların toplam sayısına bölerek, bir yüzde hesaplar. Yeniden çiz olaylarını ya da kullanıcı tıklamalarını ya da tuşa basma olaylarını işemesi için `QApplication::processEvents()`'i çağırırız. Eğer kullanıcı Cancel'a tıklarsa, kaydetmeyi iptal ederiz ve dosyayı sileriz.

`QProgressDialog` üzerinde `show()`'u çağırmayız, çünkü ilerleme diyalogları bunu kendileri için yaparlar. Eğer işlem kısa ise, `QProgress` bunu fark eder ve kendini göstermez.

Uzun süren işlemler için tamamen farklı bir yol daha vardır: Kullanıcı istediğinde işlemi icra etmek yerine, işlemi uygulama boşta kalıncaya kadar erteleyebiliriz.

Qt'da bu yaklaşım, bir 0 milisaniye zamanlayıcısı(0-millisecond timer) kullanarak gerçekleştirilebilir. Bu zamanlayıcılar, eğer beklemede olan olaylar yoksa işletilirler. İşte, boştayken işleme(idle processing) yaklaşımını gösteren bir `timerEvent()` gerçekleştirimi örneği:

```
void Spreadsheet::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        while (step < MaxStep && !qApp->hasPendingEvents()) {
            performStep(step);
            ++step;
        }
    } else {
        QTableWidget::timerEvent(event);
    }
}
```

`hasPendingevents()` `true` döndürürse, işlemi durdururuz ve kontrolü Qt'a iade ederiz. Qt, beklemede olan olayların tümünü işlendiğinde, işlem devam edecektir.

BÖLÜM 8: SÜRÜKLE-BIRAK

Sürükle-bırak(Drag and drop), bir uygulama dâhilinde ya da farklı uygulamalar arasında bilgi transfer etmenin modern ve sezgisel bir yoludur. Çoğu kez, veri taşımak ve kopyalamak için ek olarak pano(clipboard) desteği sağlanır.

Bu bölümde, bir uygulamaya sürükle-bırak desteği eklemeyi ve özel biçimler elde etmeyi göreceğiz. Sonra, sürükle-bırak kodunu, pano desteği ekleyerek yeniden kullanmayı göstereceğiz. Bu kodu yeniden kullanmak mümkündür, çünkü her iki mekanizma da veriyi birkaç biçimde temin eden bir sınıf olan `QMimeData`'ya dayanır.

Sürükle-Bırakı Etkinleştirme

Sürükle-bırak iki ayrı eylem içerir: sürükleme ve bırakma. Qt parçacıkları, sürükleme bölgesi(drag site), bırakma bölgesi(drop site) ya da her ikisi olarak hizmet verebilirler.

Bizim örneğimiz, bir Qt uygulamasının başka bir uygulama tarafından sürüklenen öğeyi kabul etmesini gösterir. Qt uygulaması, merkez parçacığı bir `QTextEdit` olan bir ana penceredir. Kullanıcı, masa üstünden ya da bir dosya gezgininden bir metin dosyası sürüklediğinde ve onu uygulama üzerine bıraktığında, uygulama, dosyayı `QTextEdit` içine yükler.

İşte, örneğin `MainWindow` sınıfının tanımı:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);

private:
    bool readFile(const QString &fileName);
    QTextEdit *textEdit;
};
```

`MainWindow` sınıfı, `QWidget`'tan `dragEnterEvent()` ve `dropEvent()`'i uyarlar. Örneğin amacı sürükle-bırakı göstermek olduğu için, işlevlerin çoğunun bir ana pencere sınıfı içinde olduğunu varsayarak, ihmal edeceğiz.

```
MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);
```

```
    textEdit->setAcceptDrops(false);
    setAcceptDrops(true);

    setWindowTitle(tr("Text Editor"));
}
```

Kurucuda, bir QTextEdit oluştururuz ve onu merkez parçacık olarak ayarlarız. Varsayılan olarak, QTextEdit diğer uygulamalardan metinsel sürükle-bırakları kabul eder ve eğer kullanıcı onun üzerine bir dosyayı bırakırsa, dosya ismini metne ekler. Bırakma olayları(drop event) çocuktan ebeveyne nakledildiği için, QTextEdit üzerinde bırakmayı devre dışı bırakarak ve ana pencere üzerinde etkinleştirerek, MainWindow içinde, tüm pencerenin bırakma olaylarını elde ederiz.

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeData()->hasFormat("text/uri-list"))
        event->acceptProposedAction();
}
```

dragEnterEvent(), kullanıcı bir nesneyi bir parçacık üzerine sürüklediğinde çağrılır. Eğer acceptProposedAction()'ı bir olay üzerinde çağırırsak, kullanıcının, sürüklenen nesneyi bu parçacık üzerine bırakabileceğini belirtmiş oluruz. Varsayılan olarak, parçacık sürüklemeyi kabul etmeyecektir. Qt, kullanıcıya parçacığın meşru bırakma alanı olup olmadığını bildirmek için otomatik olarak fare imlecini değiştirir.

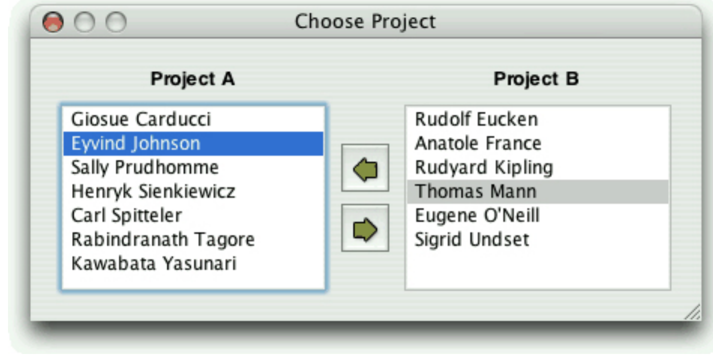
Burada kullanıcının dosya sürüklemesine izin vermek isteriz. Bunu yapmak için, sürüklenenin MIME tipini kontrol ederiz. text/uri-list MIME tipi; dosya isimleri, URL'ler ya da diğer global kaynak tanımlayıcıları olabilen, uniform kaynak tanımlayıcılarının(URI) bir listesini saklamada kullanılır. Standart MIME tipleri, Internet Assigned Numbers Authority(IANA) tarafından tanımlanır. Bir tip ve slash('/') ile ayrılmış bir alttipten(subtype) oluşurlar. Pano ve sürükle-bırak sistemi, verinin farklı tiplerini belirlemek için MIME tiplerini kullanır. MIME tiplerinin resmi listesine http://www.iana.org/assignments/media-types/ adresinden ulaşılabilir.

```
void MainWindow::dropEvent(QDropEvent *event)
{
    QList<QUrl> urls = event->mimeData()->urls();
    if (urls.isEmpty())
        return;

    QString fileName = urls.first().toLocalFile();
    if (fileName.isEmpty())
        return;

    if (readFile(fileName))
        setWindowTitle(tr("%1 - %2").arg(fileName)
                                    .arg(tr("Drag File")));
}
```

dropEvent(), kullanıcı bir nesneyi bir parçacığın üzerine bıraktığında çağrılır. QUrl'lerin bir listesini elde etmek için QMimeData::urls()'i çağırırız. Genellikle, kullanıcılar bir seferde sadece bir dosya bırakırlar, fakat birçok dosyayı aynı anda sürüklemek de olasıdır. Eğer bir URL'den fazlası var ise, ya da URL bir yerel dosya ismi değil ise, derhal geri döneriz.

QWidget ayrıca, dragMoveevent() ve dragLeaveEvent()'i de sağlar, fakat birçok uygulama için onları uyarlamak gerekmez.

İkinci örnek, bir sürüklemeyi sezmeyi ve bir bırakmayı kabul etmeyi gösterir. Sürükle-bırakı destekleyen bir QListWidget altsınıfı oluşturacağız ve onu Şekil 8.1'de görünen Project Chooser uygulamasında bir bileşen olarak kullanacağız.



Şekil 8.1

Project Chooser uygulaması, kullanıcıya isimlerle doldurulmuş iki liste parçacığı sunar. Her bir liste parçacığı bir projeyi temsil eder. Kullanıcı bir kişiyi bir projeden diğerine taşımak için liste parçacığı içindeki isimleri sürükleyip bırakabilir.

Sürükle-bırak kodunun tümü QListWidget altsınıfı içine yerleştirilir. İşte, sınıf tanımı:

```
class ProjectListWidget : public QListWidget
{
    Q_OBJECT

public:
    ProjectListWidget(QWidget *parent = 0);

protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);

private:
    void performDrag();

    QPoint startPos;
};
```

ProjectListWidget sınıfı, QWidget'ta bildirilmiş beş olay işleyicisini uyarlar.

```
ProjectListWidget::ProjectListWidget(QWidget *parent)
    : QListWidget(parent)
{
    setAcceptDrops(true);
}
```

Kurucuda, liste parçacığı üzerinde sürüklemeyi etkinleştiririz.

```
void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        startPos = event->pos();
    QListWidget::mousePressEvent(event);
}
```

Kullanıcı farenin sol butonuna bastığında, farenin konumunu `startPos` private değişkeni içinde saklarız. `QListWidget`'ın, fare butonuna basma olaylarını her zamanki gibi işleme fırsatına sahip olmasını sağlamak için `QListWidget`'ın `mousePressEvent()` gerçekleştirimini çağırırız.

```
void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QListWidget::mouseMoveEvent(event);
}
```

Kullanıcı farenin sol butonuna basıyorken fare imlecini hareket ettirirse, bir sürüklemenin başladığını düşünürüz. Farenin geçerli konumu ile sol fare butonuna basıldığı andaki konumu arasındaki mesafeyi hesaplarız. Eğer mesafe `QApplication`'ın önerilen sürüklemeye başlama mesafesinden (normalde dört piksel) büyükse ya da ona eşitse, sürüklemeyi başlatmak için `performDrag()` private fonksiyonunu çağırırız.

```
void ProjectListWidget::performDrag()
{
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeData = new QMimeData;
        mimeData->setText(item->text());

        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeData);
        drag->setPixmap(QPixmap(":/images/person.png"));
        if (drag->exec(Qt::MoveAction) == Qt::MoveAction)
            delete item;
    }
}
```

`performDrag()`'ta, ebeveyni `this` olan bir `QDrag` nesnesi oluştururuz. `QDrag` nesnesi, veriyi bir `QMimeData` nesnesi içinde saklar. Örneğin, `QMimeData::setText()` kullanarak, veriyi bir `text/plain` karakter katarı olarak sağlarız. `QMimeData`, en yaygın sürüklenen tiplerini (resimler, URL'ler, renkler, vb.) işlemek için birkaç fonksiyon sağlar ve `QByteArray`'ler olarak ifade edilen isteğe bağlı MIME tiplerini işleyebilir. `QDrag::setPixmap()` çağrısı, nesne sürüklenirken onu takip eden fare imlecini ayarlar.

`QDrag::exec()` çağrısı, sürükleme işlemini başlatır. Argüman olarak desteklenen sürükleme eylemlerinin (`Qt::CopyAction`, `Qt::MoveAction` ve `Qt::LinkAction`) bir kombinasyonunu alır ve gerçekleştirilen sürükleme işlemini (ya da gerçekleştirilmemiş ise `Qt::IgnoreAction`) döndürür. Hangi eylemin gerçekleştirileceği, kaynak parçacığın izinlerine, hedefin desteklediklerine ve sürükleme meydana

geldiğinde hangi tamamlayıcı tuşlara basıldığına bağlıdır. `exec()` çağrısından sonra, Qt sürüklenen nesnenin sahipliğini üstlenir ve ona daha fazla ihtiyaç duyulmadığında da siler.

```
void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

`ProjectListWidget` parçacığı, eğer sürüklemeler aynı uygulama içinde başka bir `ProjectList-Widget`'tan geliyorsa, onları da kabul eder. `QDragEnterEvent::source()`, sürüklenen parçacık eğer aynı uygulamanın bir parçası ise, bir işaretçi döndürür; aksi halde, boş bir işaretçi döndürür. Sürüklenenin bir `ProjectListWiget`'tan geldiğinden emin olmak için `qobject_cast<T>()` kullanırız. Eğer hepsi uygun ise, eylemi bir taşıma eylemi olarak kabul etmeye hazır olduğumuzu Qt'a bildiririz.

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

`dragMoveEvent()`'teki kod, `dragEnterEvent()` içindekinin aynıdır. Bu gereklidir çünkü fonksiyonun `QListWidget` (aslında, `QAbstractItemView`) gerçekleştirimini hükümsüz kılmamız gerekir.

```
void ProjectListWidget::dropEvent(QDropEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        addItem(event->mimeData()->text());
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

`dropEvent()`'te, `QMimeData::text()` kullanarak, sürüklenen metne erişiriz ve o metin ile bir öğe oluştururuz. Ayrıca, sürüklenen öğenin orijinal versiyonunu silebilen kaynak parçacığı bildirmek için olayı bir "taşıma eylemi(move action)" olarak almamız gerekir.

Özel Sürükleme Tiplerini Destekleme

Şimdiye kadarki örneklerde, yaygın MIME tipleri için `QMimeData`'nın desteğine güvendik. Nitekim, bir metin sürüklemesi oluşturmak için `QMimeData::setText()`'i çağırdık ve bir `text/uri-list` sürüklemesinin içeriğine erişmek için `QMimeData::urls()`'i kullandık. Eğer düz bir metin, HTML metni, resim, URL ya da renk sürüklemek istersek, başka bir şeye ihtiyaç duymaksızın `QMimeData`'yı kullanabiliriz. Fakat eğer özel veriler sürüklemek istersek, şu alternatifler arasından seçim yapmalıyız:

1. `QMimeData::setData()`'yı kullanarak bir `QByteArray` olarak isteğe bağlı veri sağlayabiliriz ve daha sonra `QMimeData::data()`'yı kullanarak onu elde edebiliriz.
2. Bir `QMimeData` altsınıfı türetebilir ve özel veri tiplerini işlemek için `formats()` ve `retrieveData()`'yı uyarlayabiliriz.
3. Tek bir uygulama dâhilindeki sürükle-bırak işlemleri için, bir `QMimeData` altsınıfı türetebilir ve veriyi istediğimiz herhangi bir veri yapısını kullanarak saklayabiliriz.

İlk yaklaşım altsınıf türetmeyi gerektirmez, fakat bazı sakıncalara sahiptir: Eninde sonunda sürükleme kabul edilmeyecek olsa bile veri yapımızı bir `QByteArray`'e dönüştürmemiz gerekir ve eğer uygulamanın geniş bir alanı ile etkileşmek için birkaç MIME tipi sağlamak istiyorsak, veriyi birkaç kere saklamamız gerekir (her MIME tipini bir kere). Eğer veri büyük ise, bu gereksiz yere uygulamayı yavaşlatabilir. İkinci ve üçüncü yaklaşımlar bu problemi önleyebilir ya da minimize edebilirler. Kontrolü tamamen bize verirler ve birlikte kullanılabilirler.

Uygulamanın nasıl çalıştığını göstermek için, bir `QTableWidget`'a sürükle-bırak yetenekleri eklemeyi göstereceğiz. Sürükleme, şu MIME tiplerini destekleyecek: `text/plain`, `text/html` ve `text/csv`. İlk yaklaşımı kullanarak bir sürüklemeye başlamak şunu gibi görünür:

Kod Görünümü:

```
void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QTableWidget::mouseMoveEvent(event);
}

void MyTableWidget::performDrag()
{
    QString plainText = selectionAsPlainText();
    if (plainText.isEmpty())
        return;

    QMimeData *mimeData = new QMimeData;
    mimeData->setText(plainText);
    mimeData->setHtml(toHtml(plainText));
    mimeData->setData("text/csv", toCsv(plainText).toUtf8());

    QDrag *drag = new QDrag(this);
    drag->setMimeData(mimeData);
    if (drag->exec(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
        deleteSelection();
}
```

`performDrag()` private fonksiyonu, dikdörtgen bir seçimi sürüklemeye başlamak için `MouseMoveEvent()`'ten çağrılır. `setText()` ve `setHtml()`'i kullanarak, `text/plain` ve `text/html` MIME tiplerini ayarlarız ve `setData()`'yı kullanarak, isteğe bağlı bir MIME tipi ve bir `QByteArray` alan `text/csv` tipini ayarlarız.

```
QString MyTableWidget::toCsv(const QString &plainText)
{
```

```
    QString result = plainText;
    result.replace("\\", "\\\\");
    result.replace("\"", "\\\"");
    result.replace("\t", "\", \"");
    result.replace("\n", "\"\n\"");
    result.prepend("\"");
    result.append("\"");
    return result;
}

QString MyTableWidget::toHtml(const QString &plainText)
{
    QString result = Qt::escape(plainText);
    result.replace("\t", "<td>");
    result.replace("\n", "\n<tr><td>");
    result.prepend("<table>\n<tr><td>");
    result.append("\n</table>");
    return result;
}
```

`toCsv()` ve `toHtml()` fonksiyonları, bir "tablar ve yenisatırlar(newlines)" karakter katarını bir CSV (comma-separated values) ya da bir HTML karakter katarına dönüştürür. Örneğin,

```
Red   Green   Blue
Cyan  Yellow  Magenta
```

verisini

```
"Red", "Green", "Blue"
"Cyan", "Yellow", "Magenta"
```

'ya ya da

```
<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>
```

'a dönüştürür.

Dönüştürme en basit yolla yapılır; `QString::replace()` kullanmak. HTML'in özel karakterlerinden kurtulmak için `Qt::escape()`'i kullanırız.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeData()->text();
        ...
        event->acceptProposedAction();
    }
}
```

Veriyi üç farklı biçimde temin ettiğimiz halde, `dropEvent()` içinde sadece onlardan ikisini kabul ederiz. Eğer kullanıcı, hücreleri bir `QTableWidget`'tan bir HTML editörüne sürüklerse, hücrelerin bir HTML tablosuna dönüştürülmesini isteriz. Fakat eğer kullanıcı, keyfi bir HTML'i bir `QTableWidget` içine sürüklerse, kabul etmek istemeyiz.

Bu örneği çalışır yapmak için ayriyeten `MyTableWidget` kurucusu içinde `setAcceptDrops(true)` ve `setSelectionMode(ContiguousSelection)`'ı çağırmamız gerekir.

Şimdi, uygulamayı yeniden yapacağız, fakat bu sefer `QTableWidgetItem` ve `QByteArray` arasındaki dönüştürmeleri ertelemek ya da kaçınmak için, bir `QMimeData` altsınıfı türeteceğiz. İşte altsınıfımızın tanımı:

```
class TableMimeData : public QMimeData
{
    Q_OBJECT

public:
    TableMimeData(const QTableWidget *tableWidget,
                  const QTableWidgetSelectionRange &range);

    const QTableWidget *tableWidget() const { return myTableWidget; }
    QTableWidgetSelectionRange range() const { return myRange; }
    QStringList formats() const;

protected:
    QVariant retrieveData(const QString &format,
                          QVariant::Type preferredType) const;

private:
    static QString toHtml(const QString &plainText);
    static QString toCsv(const QString &plainText);

    QString text(int row, int column) const;
    QString rangeAsPlainText() const;

    const QTableWidget *myTableWidget;
    QTableWidgetSelectionRange myRange;
    QStringList myFormats;
};
```

Asıl veriyi saklamak yerine, hangi hücrelerin sürüklendiği belirten ve `QTableWidget`'a bir işaretçi sağlayan, bir `QTableWidgetSelectionRange` saklarız. `formats()` ve `retrieveData()` fonksiyonları, `QMineData`'dan uyarlanırlar.

```
TableMimeData::TableMimeData(const QTableWidget *tableWidget,
                             const QTableWidgetSelectionRange &range)
{
    myTableWidget = tableWidget;
    myRange = range;
    myFormats << "text/csv" << "text/html" << "text/plain";
}
```

Kurucuda, private değişkenleri ilk kullanıma hazırlarız.

```
QStringList TableMimeData::formats() const
{
    return myFormats;
```

```
}
```

formats() fonksiyonu, MIME veri nesnesi tarafından sağlanan MIME tiplerinin bir listesini döndürür. Biçimlerin sırası genellikle önemsizdir, fakat "en iyi" biçimleri ilk sıraya koymak iyi bir alışkanlıktır. Birçok biçimi destekleyen uygulamalar, bazen biçimlerin ilk eşleşenini kullanırlar.

```
QVariant TableMimeData::retrieveData(const QString &format,
                                     QVariant::Type preferredType) const
{
    if (format == "text/plain") {
        return rangeAsPlainText();
    } else if (format == "text/csv") {
        return toCsv(rangeAsPlainText());
    } else if (format == "text/html") {
        return toHtml(rangeAsPlainText());
    } else {
        return QMimeData::retrieveData(format, preferredType);
    }
}
```

retrieveData() fonksiyonu, verilen MIME tipi için veriyi bir QVariant olarak döndürür. format parametresinin değeri normalde formats() tarafından döndürülen karakter katarlarının biridir, fakat tüm uygulamalar, MIME tipini formats() ile kontrol etmediği için bunu varsayamayız. QMimeData tarafından sağlanan text(), html(), urls(), imageData(), colorData() ve data() tutucu fonksiyonları, retrieveData()'ya dayanarak gerçekleştirilirler.

preferredType parametresi bize, QVariant içine hangi tipi koymamız gerektiğinin ipucunu verir. Burada, onu yok sayarız ve eğer gerekliyse, dönen değeri arzu edilen tipe dönüştürmede QMimeData'ya güveniriz.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    const TableMimeData *tableData =
            qobject_cast<const TableMimeData *>(event->mimeData());

    if (tableData) {
        const QTableWidget *otherTable = tableData->tableWidget();
        QTableWidgetSelectionRange otherRange = tableData->range();
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeData()->text();
        ...
        event->acceptProposedAction();
    }
    QTableWidget::mouseMoveEvent(event);
}
```

dropEvent() fonksiyonu, bu kısımda daha önce yaptığımız ile benzerdir, fakat bu sefer önce, QMimeData nesnesini bir TableMimeData'ya güvenli bir şekilde dönüştürüp dönüştüremeyeceğimizi kontrol ederekten onu optimize ederiz. Eğer qobject_cast<T>() çalışırsa, bunun anlamı; sürükleme

aynı uygulama içinde bir `MyTableWidget`'tan kaynaklanmıştır ve tablo verisine, `QMimeData`'nın `API`'ına sunmadan, doğrudan olarak erişebiliriz. Eğer dönüşüm başarısız olursa, veriyi standart yolla elde ederiz.

Bu örnekte, UTF-8 kodlamasını kullanarak CSV metnini kodladık. Eğer doğru kodlamanın kullanıldığından emin olmak istiyorsak, belli bir kodlama belirtmek için `text/plain` MIME tipinin `charset` parametresini kullanabilirdik. İşte birkaç örnek:

```
text/plain;charset=US-ASCII
text/plain;charset=ISO-8859-1
text/plain;charset=Shift_JIS
text/plain;charset=UTF-8
```

Pano İşleme

Çoğu uygulama Qt'un yerleşik pano işlemesini(clipboard handling) kullanır. Örneğin `QTextEdit` sınıfı hem `cut()`, `copy()` ve `paste()` yuvalarını hem de klavye kısayollarını sağlar, çoğu zaman ek kod gerekmeden.

Kendi sınıflarımızı yazdığımızda, uygulamanın `QClipboard` nesnesine bir işaretçi döndüren `QApplication::clipboard()` sayesinde panoya erişebiliriz. Sistem panosunu işlemek kolaydır: Veriyi panoya yerleştirmek için `setText()`, `setImage()` ya da `setPixmap()`'i çağır, panodan veriye erişmek içinse `text()`, `image()` ya da `pixmap()`'ı çağır. Pano kullanmanın örneğini zaten Bölüm 4'te Spreadheet uygulaması içinde görmüştük.

Bazı uygulamalar için yerleşik işlevler yeterli olmayabilir. Örneğin, sadece metinden ya da sadece bir resimden ibaret olmayan bir veriyi temin etmek isteyebiliriz, ya da veriyi, diğer uygulamalarla maksimum çalışabilirlik için birçok farklı biçimde temin etmek isteyebiliriz. Sorun, daha önce sürükle-bırak ile karşılaştığımız sorun ile çok benzerdir; öyle ki cevabı da benzerdir: Bir `QMimeData` altsınıfı türetebilir ve birkaç sanal fonksiyonu uyarlayabiliriz.

Eğer uygulamalarımız, sürükle-bırakı özel bir `QMimeData` altsınıfı sayesinde destekliyorsa, `QMimeData` altsınıfını yeniden kullanabilir ve `setMimeData()`'yı kullanarak, onu panoya yerleştirebiliriz. Veriye erişmek içinse pano üzerinde `mimeData()`'yı çağırabiliriz.

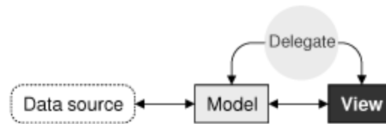
BÖLÜM 9: ÖĞE GÖRÜNTÜLEME SINIFLARI

Birçok uygulama kullanıcıya, bir veri setine ait özel öğeleri arama, görüntüleme ve düzenleme izni verir. Veri, dosyalarda tutuluyor ya da ona veritabanından veya bir ağ sunucusundan erişiliyor olabilir. Bunun gibi veri setleri ile uğraşmanın standart yaklaşımı, Qt'un öğe görüntüleme sınıflarıdır(item view classes).

Qt'un daha önceki versiyonlarında, öğe görüntüleme parçacıkları, bir veri setinin tüm içeriği ile doldurulurdu; kullanıcılar tüm aramalarını ve tutulan veri üzerinde düzenlemeleri parçacık içinde yaparlardı ve bazı noktalarda, değişiklikler veri kaynağına geri yazılırdı. Anlaması ve kullanması basit olmasına rağmen, bu yaklaşım, büyük veri setlerini iyi ölçekleyemez ve aynı verinin iki ya da daha fazla farklı parçacıkta görüntülenmesini istediğimiz durumlarda kendini ödünç veremez.

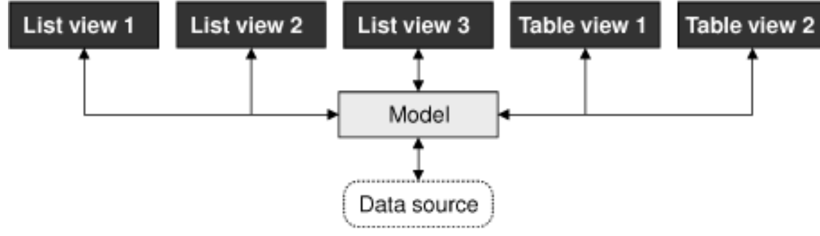
Smalltalk dili, büyük veri setlerinin görselleştirilmesi için esnek bir yaklaşımı popülerleştirmiştir: Model-View-Contoller(MVC yani Model-Görünüş-Kontrolör). MVC yaklaşımında, Model, veri setini temsil eder ve hem görüntüleme için gereken veriyi getirmekten hem de değişiklikleri kaydetmekten sorumludur. Veri setinin her bir tipi kendi modeline sahiptir. Görünüş, veriyi kullanıcıya sunar. Her veri seti ile bir seferde sadece limitli bir miktarda veri görünür olacaktır, bu nedenle bu sadece görüntülenmesi istenen veridir. Kontrolör, kullanıcı ve Görünüş arasında aracılık yapar.

Qt, esin kaynağı MVC yaklaşımı olan bir Model/Görünüş mimarisi sağlar (Şekil 9.1). Qt'da Model, klasik MVC için yaptığının aynını yapar. Fakat Kontrolör yerine, Qt biraz farklı bir soyutlama kullanır: Delege(Delegate). Delege, öğelerin yorumlanmasında ve düzenlenmesinde mükemmel bir kontrol sağlamada kullanılır. Qt, Görünüşün her tipi için varsayılan bir Delege sağlar. Bu pek çok uygulama için yeterlidir, bu nedenle genellikle onunla fazla ilgilenmeyiz.



Şekil 9.1

Qt'un Model/Görünüş mimarisini kullanırken, Modeli sadece Görünüş içinde o an için görüntülenmesi gereken veriyi getirmede kullanabiliriz, ki böyle yapmak performanstan ödün vermeden çok büyük veri setlerini işlemeyi mümkün kılar. Ve bir Modeli iki ya da daha fazla Görünüş ile kayda geçirerek, kullanıcıya, küçük bir yük ile veriyi görüntüleme ve farklı şekillerde onunla etkileşme fırsatı verebiliriz. Qt, çoklu Görünüşleri otomatik olarak, eşzamanlı işletir; birindeki değişiklikler diğer tümüne yansıtılır (Şekil 9.2). Model/Görünüş mimarisinin faydalarına bir ekleme daha: Esas verinin nasıl saklanacağını değiştirmeye karar verdiğimizde, sadece Modeli değiştirmemiz gerekir; Görünüş, aynı davranışı sergilemeye devam edecektir.



Şekil 9.2

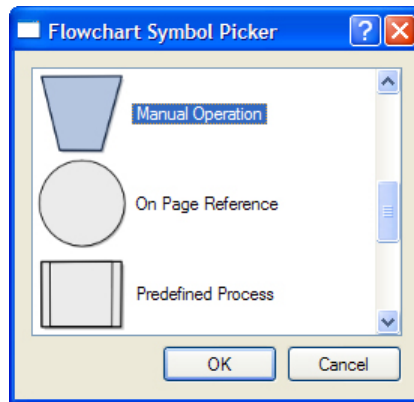
Birçok durumda, kullanıcıya az sayıda öğe sunmamız gerekir. Böyle yaygın durumlarda, Qt'un öğe görüntüleme uygunluk sınıflarını (QListWidget, QTableWidget ve QTreeWidget) kullanabiliriz ve onları direkt olarak öğelerle doldurabiliriz. Bu sınıflar, Qt'un önceki versiyonları tarafından sağlanan öğe görüntüleme sınıfları ile aynı şekilde davranırlar. Verilerini "öğeler(items)" içinde saklarlar (bir QTable-Widget'ın QTableWidgetItem'lar içermesi gibi). Uygunluk sınıfları ayrıca, öğeleri Görünüşlere sağlayan özel modeller kullanır.

Büyük veri setleri için, veriyi kopyalamak çoğu kez bir seçenek değildir. Bu gibi durumlarda, Qt'un Görünüşlerini (QListWidget, QTableWidget ve QTreeWidget), özel bir Model ya da Qt'un önceden tanımlı Modellerinden biri ile birleşim içinde kullanabiliriz. Örneğin, eğer veri seti bir veritabanı(database) içinde tutuluyorsa, bir QTableView ile bir QSqlTableModel'ı birleştirebiliriz.

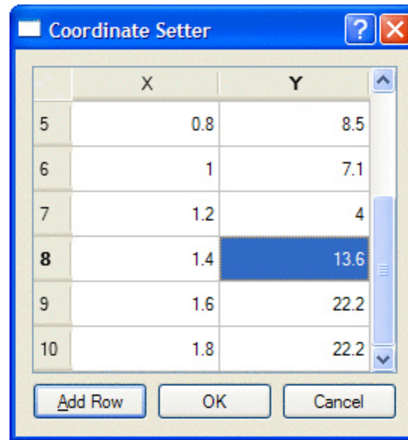
Öğe Görüntüleme Uygunluk Sınıflarını Kullanma

Qt'un öğe görüntüleme uygunluk altsınıflarını kullanmak, özel bir Model tanımlamaktan daha basittir ve Görünüş ve Modelin ayrılmasının faydalarına ihtiyacımız olmadığı zamanlar için biçilmiş kaftandır. Bu tekniği, Bölüm 4'te hesap çizelgesi işlevlerini gerçekleştirmek için QTableWidget ve QTableWidgetItem altsınıfları türetirken kullanmıştık.

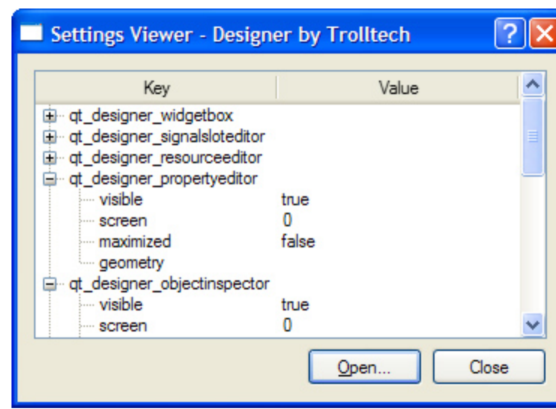
Bu kısımda, öğeleri görüntülemek için öğe görüntüleme uygunluk sınıflarını kullanmayı göstereceğiz. İlk örnek, saltokunur(read-only) bir QListWidget (Şekil 9.3), ikinci örnek, düzenlenebilir bir QTableWidget (Şekil 9.4), ve üçüncü örnek, saltokunur bir QTreeWidget (Şekil 9.5) gösterir.



Şekil 9.3



Şekil 9.4



Şekil 9.5

Kullanıcıya, bir listeden bir akış diyagramı(flowchart) sembolü seçmeye izin veren basit bir diyalogla başlayacağız. Her bir öğe; bir simge, bir metin ve özgün bir ID'den meydana gelir.

Diyaloğun başlık dosyasından bir alıntı ile başlayalım:

```
class FlowChartSymbolPicker : public QDialog
{
    Q_OBJECT

public:
    FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
                          QWidget *parent = 0);

    int selectedId() const { return id; }
    void done(int result);
    ...
};
```

Diyaloğu inşa etiğimiz zaman, ona bir `QMap<int, QString>` aktarmalıyız. Onu aktardıktan sonra, `selectedId()`'yi çağırarak seçilen ID'ye erişebiliriz.

```
FlowChartSymbolPicker::FlowChartSymbolPicker(
        const QMap<int, QString> &symbolMap, QWidget *parent)
    : QDialog(parent)
{
    id = -1;
```

```
    listWidget = new QListWidget;
    listWidget->setIconSize(QSize(60, 60));

    QMapIterator<int, QString> i(symbolMap);
    while (i.hasNext()) {
        i.next();
        QListWidgetItem *item = new QListWidgetItem(i.value(),
                                                     listWidget);
        item->setIcon(iconForSymbol(i.value()));
        item->setData(Qt::UserRole, i.key());
    }
    ...
}
```

`id`'nin ilk değerini -1 olarak atarız. Sonra, bir öğe görüntüleme uygunluk parçacığı olan bir `QListWidget` oluştururuz. Akış diyagramı sembol haritası üzerinde her bir öğeyi tekrarlarız ve her birini temsilen birer `QListWidgetItem` oluştururuz. `QListWidgetItem` kurucusu, görüntülenecek metni ifade eden bir `QString` alır.

Sonra, öğenin simgesini ayarlarız ve isteğe bağlı ID'mizi `QListWidgetItem` içinde saklamak için `setData()`'yı çağırırız. `iconForSymbol()` private fonksiyonu, verilen bir sembol ismi için bir `QIcon` döndürür.

`QListWidgetItem`'lar birkaç role ve birleşmiş bir `QVariant`'a sahiptirler. En yaygın roller, `Qt::DisplayRole`, `Qt::EditRole` ve `Qt::IconRole`'dür, fakat başka roller de vardır. Aynı zamanda, `Qt::UserRole`'un bir nümerik değerini ya da daha yükseğini belirterek, özel roller de tanımlayabiliriz. Bizim örneğimizde, her bir öğenin ID'sini saklamak için `Qt::UserRole` kullanırız.

Kurucunun, butonlar oluşturma, parçacıkları yerleştirme ve pencere başlığını ayarlamayla ilgili bölümleri ihmal edilir.

```
void FlowChartSymbolPicker::done(int result)
{
    id = -1;
    if (result == QDialog::Accepted) {
        QListWidgetItem *item = listWidget->currentItem();
        if (item)
            id = item->data(Qt::UserRole).toInt();
    }
    QDialog::done(result);
}
```

`done()` fonksiyonu `QDialog`'tan uyarlanmıştır. Kullanıcı OK ya da Cancel'a tıkladığında çağrılır. Eğer kullanıcı OK'i tıklamışsa, ilgili öğeye erişiriz ve `data()` fonksiyonunu kullanarak ID bilgisini alırız. Eğer öğenin metni ile ilgileniyor olsaydık, `item->data(Qt::DisplayRole).toString()`'i ya da `item->text()`'i çağırarak ona erişebilirdik.

Varsayılan olarak, `QListWidget` saltokunurdur. Eğer kullanıcının öğeleri düzenlemesini isteseydik, `QAbstractItemView::setEditTriggers()`'ı kullanarak Görünüşün düzenleme tetiklerini(edit triggers) ayarlayabilirdik; örneğin, `QAbstractItemView::AnyKeyPressed` ayarı kullanıcının, klavyeden tuşlamayarak, bir öğeyi düzenlemeye başlayabileceği anlamına gelir. Alternatif olarak, düzenleme

işlemlerini programlanabilir bir şekilde işleyebilmek için, bir Edit butonu (ve belki Add ve Delete butonları) sağlayabilir ve sinyal-yuva bağlantılarını kullanabilirdik.

Hazır, öğe görüntüleme uygunluk sınıflarını görüntüleme ve seçme için kullanmayı görmüşken, şimdi de, veriyi düzenleyebildiğimiz bir örneğe bakalım. Yine bir diyalog kullanıyoruz, fakat bu sefer kullanıcının düzenleyebildiği, bir (x, y) koordinatları seti sunan bir diyalog.

Önceki örnekteki gibi, kurucu ile başlayarak, yine yalnızca öğe görüntüleme ile ilgili kodlara odaklanacağız.

```
CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                   QWidget *parent)
    : QDialog(parent)
{
    coordinates = coords;

    tableWidget = new QTableWidget(0, 2);
    tableWidget->setHorizontalHeaderLabels(
            QStringList() << tr("X") << tr("Y"));

    for (int row = 0; row < coordinates->count(); ++row) {
        QPointF point = coordinates->at(row);
        addRow();
        tableWidget->item(row, 0)->setText(QString::number(point.x()));
        tableWidget->item(row, 1)->setText(QString::number(point.y()));
    }
    ...
}
```

`QTableWidget` kurucusu, görüntüleyeceği satırların ve sütunların ilk sayısını alır. Yatay ve dikey başlık öğeleri de dâhil olmak üzere, bir `QTableWidget` içindeki her öğe, bir `QTableWidgetItem` tarafından temsil edilir. `setHorizontalHeaderLabels()` fonksiyonu, her bir yatay tablo parçacığı öğesinin metnini ayarlar. `QTableWidget` varsayılan olarak, tam da isteğimiz gibi 1'den başlayarak etiketlenmiş satırlar ile bir dikey başlık sunar, böylece dikey başlık etiketlerini el ile ayarlamamız gerekmez.

Sütun etiketlerini oluşturur oluşturmaz, verilen koordinat verisine dayanarak yineleriz. Her (x, y) çifti için yeni bir satır ekleriz (`addRow()` private fonksiyonunu kullanarak) ve her bir satırın sütunlarına uygun metni ayarlarız.

`QTableWidget` varsayılan olarak, düzenlemeye izin verir. Kullanıcı, tablodaki her hücreyi, gezinerek ve sonrasında da F2'ye basarak ya da sadece bir şeyler yazarak düzenleyebilir. Kullanıcının Görünüş içinde yaptığı tüm değişiklikler otomatik olarak `QTableWidgetItem`'lar içine yansıtılacaktır. Düzenlemeye engel olmak için, `setEditTriggers(QAbstractItemView::NoEditTriggers)`'ı çağırırız.

```
void CoordinateSetter::addRow()
{
    int row = tableWidget->rowCount();

    tableWidget->insertRow(row);

    QTableWidgetItem *item0 = new QTableWidgetItem;
    item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 0, item0);

    QTableWidgetItem *item1 = new QTableWidgetItem;
    item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
```

```
    tableWidget->setItem(row, 1, item1);

    tableWidget->setCurrentItem(item0);
}
```

addRow() yuvası, kullanıcı Add Row butonuna tıkladığında çağrılır; ayrıca kurucuda da kullanılır. QTableWidget::insertRow() kullanarak yeni bir satır ekleriz. Sonra, iki QTableWidgetItem oluştururuz ve öğeye ek olarak, bir satır ve sütun alan QTableWidget::setItem()'ı kullanarak, onları tabloya ekleriz. Son olarak, geçerli öğeyi ayarlarız, böylece kullanıcı yeni satırın ilk öğesinden düzenlemeye başlayabilir.

```
void CoordinateSetter::done(int result)
{
    if (result == QDialog::Accepted) {
        coordinates->clear();
        for (int row = 0; row < tableWidget->rowCount(); ++row) {
            double x = tableWidget->item(row, 0)->text().toDouble();
            double y = tableWidget->item(row, 1)->text().toDouble();
            coordinates->append(QPointF(x, y));
        }
    }
    QDialog::done(result);
}
```

Kullanıcı OK'e tıkladığında, diyaloğa aktarılan koordinatları temizleriz ve QTableWidget'ın öğelerindeki koordinatlara dayanarak yeni bir set oluştururuz. Qt'un öğe görüntüleme kolaylık parçacıklarının üçüncü ve son örneği için, bir QTreeWidget kullanarak, bir Qt uygulamasının ayarlarını gösteren bir uygulamadan bazı ufak parçalara bakacağız. QTreeWidget varsayılan olarak saltokunurdur.

İşte, kurucudan bir alıntı:

```
SettingsViewer::SettingsViewer(QWidget *parent)
    : QDialog(parent)
{
    organization = "Trolltech";
    application = "Designer";

    treeWidget = new QTreeWidget;
    treeWidget->setColumnCount(2);
    treeWidget->setHeaderLabels(
            QStringList() << tr("Key") << tr("Value"));
    treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
    treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
    ...
    setWindowTitle(tr("Settings Viewer"));
    readSettings();
}
```

Bir uygulamanın ayarlarına erişmek için, organizasyonun ve uygulamanın adı parametrelerine göre bir QSettings nesnesi oluşturulmalıdır. Varsayılan isimleri ("Designer", "Trolltech") ayarlarız ve sonra yeni bir QTreeWidget inşa ederiz. Ağaç parçacığının(tree widget) başlık Görünüşü(header view), ağacın sütunlarının boyutlarını yönetir. Sütunların yeniden boyutlandırma modunu(resize mode) Stretch'e ayarlarız. Bu, başlık Görünüşünün daima sütunların kullanılabilir boşluklarını doldurmasını emreder. Bu modda, sütunlar kullanıcı tarafından ya da programlanabilir bir şekilde yeniden boyutlandırılamaz. Kurucunun sonunda, ağaç parçacığını doldurmak için readSettings() fonksiyonunu çağırırız.

```
void SettingsViewer::readSettings()
{
    QSettings settings(organization, application);

    treeWidget->clear();
    addChildSettings(settings, 0, "");

    treeWidget->sortByColumn(0);
    treeWidget->setFocus();
    setWindowTitle(tr("Settings Viewer - %1 by %2")
                   .arg(application).arg(organization));
}
```

Uygulama ayarları, anahtarlar ve değerlerin bir hiyerarşisinde saklanır. `addChildSettings()` private fonksiyonu, bir ayar nesnesi(settings object), ebeveyn olarak bir `QTreeWidgetItem` ve geçerli "grup(group)"u alır. Bir grup, bir dosya sistemi dizininin, `QSettings` olarak eşdeğeridir. `addChildSettings()` fonksiyonu, isteğe bağlı bir ağaç yapısına(tree structure) geçiş yapmak için özyinelemeli olarak kendini çağırabilir. `readSettings()` fonksiyonundan ilk çağrı, kökü(root) temsil etmek için ebeveyn öğe olarak boş bir işaretçi aktarır.

```
void SettingsViewer::addChildSettings(QSettings &settings,
        QTreeWidgetItem *parent, const QString &group)
{
    if (!parent)
        parent = treeWidget->invisibleRootItem();
    QTreeWidgetItem *item;

    settings.beginGroup(group);

    foreach (QString key, settings.childKeys()) {
        item = new QTreeWidgetItem(parent);
        item->setText(0, key);
        item->setText(1, settings.value(key).toString());
    }
    foreach (QString group, settings.childGroups()) {
        item = new QTreeWidgetItem(parent);
        item->setText(0, group);
        addChildSettings(settings, item, group);
    }
    settings.endGroup();
}
```

`addChildSettings()` fonksiyonu, tüm `QTreeWidgetItem`'ları oluşturmakta kullanılır. Ayar hiyerarşisinde(settings hierarchy) geçerli seviyedeki bütün anahtarlar üzerinde yinelenir ve her bir anahtar için bir `QTableWidgetItem` oluşturur. Eğer ebeveyn öğe olarak boş bir işaretçi aktarılmışsa, öğeyi, üst seviye(top-level) bir öğe yaparak, `QTreeWidget::invisibleRootItem()`'ın bir çocuğu olarak oluştururuz. İlk sütun anahtarın ismine, ikinci sütun o anahtara ilişkin değere ayarlanır.

Sonra, fonksiyon geçerli seviyedeki her grup üzerinde yinelenir. Her bir grup için, yeni bir `QTreeWidgetItem` oluşturulur ve ilk sütunu gurubun ismine ayarlanır. Fonksiyon daha sonra, `QTreeWidget`'ı grubun çocuk öğeleriyle doldurmak için, grup öğesi ile (ebeveyn olarak) kendini özyinelemeli olarak çağırır.

Bu kısımda gösterilen öğe görüntüleme parçacıkları bize, Qt'un önceki versiyonlarındaki gibi bir programlama stili kullanma imkânı verir: bütün veri setini öğe görüntüleme parçacığı içine okuma, veri elemanlarını temsil

etmede öğe nesnelerini kullanma ve (eğer öğeler düzenlenebilir ise) bunları veri kaynağına geri yazma. Takip eden kısımlarda, bu basit yaklaşımın ötesine geçeceğiz ve Qt'un Model/Görünüş mimarisinin bütün avantajından faydalanacağız.

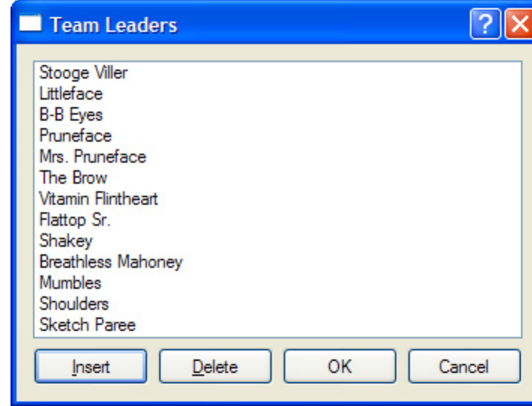
Önceden Tanımlanmış Modelleri Kullanma

Qt, görüntüleme sınıfları ile kullanmak için birkaç model sağlar:

QStringListModel	Karakter katarlarının bir listesini saklar
QStandardItemModel	İsteğe bağlı hiyerarşik veriyi saklar
QDirModel	Yerel dosya sistemini içerir
QSqlQueryModel	Bir SQL sonuç seti içerir
QSqlTableModel	Bir SQL tablosu içerir
QSqlRelationalTableModel	Dış(foreign) anahtarlarla bir SQL tablosu içerir
QSortFilterProxyModel	and/or filtrelerini başka bir Modele ayırır

Bu kısımda, `QStringListModel`, `QDirModel` ve `QSortFilterProxyModel`'ın nasıl kullanıldığına bakacağız.

Her karakter katarının bir takım liderini temsil ettiği bir `QStringList` üzerinde ekleme, silme ve düzenleme işlemlerini yapmakta kullanabileceğimiz basit bir diyalogla başlayalım. Diyalog, Şekil 9.6'da gösteriliyor.



Şekil 9.6

İşte, kurucudan konu ile ilgili bir alıntı:

```
TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,
                                     QWidget *parent)
    : QDialog(parent)
{
    model = new QStringListModel(this);
    model->setStringList(leaders);

    listView = new QListView;
    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
                              | QAbstractItemView::DoubleClicked);
    ...
}
```

Bir QStringListModel oluşturarak ve onu doldurarak başlarız. Sonra, bir QListView oluştururuz ve oluşturduğumuz Modeli onun Modeli olarak ayarlarız. Ayrıca kullanıcıya, bir karakter katarını sadece onun üzerinde yazmaya başlayarak ya da ona çift tıklayarak düzenleme imkânı vermek için bazı düzenleme tetiklerini ayarlarız. Varsayılan olarak, bir QListView üzerinde hiçbir düzenleme tetiği ayarlanmaz.

```
void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);

    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}
```

Kullanıcı Insert butonuna tıkladığında, insert() yuvası çağrılır. Yuva, liste Görünüşünün geçerli öğesi için satır numarasına erişerek başlar. Bir Model içindeki her veri öğesi, bir QModelIndex nesnesi tarafından temsil edilen bir "Model indeksi"ne sahiptir. Model indekslerine detaylıca bir dahaki kısımda bakacağız, fakat şimdilik bir indeksin üç ana bileşene sahip olduğunu bilmeniz yeterli: bir satır, bir sütun ve ait olduğu Modeli işaret eden bir işaretçi. Bir boyutlu(one-dimensional) bir liste Modeli için, sütun daima 0'dır.

Satır numarasını elde eder etmez, o konumda yeni bir satır ekleriz. Ekleme, Model üzerinde de yapılır ve Model otomatik olarak liste Görünüşünü günceller. Sonra, liste Görünüşünün geçerli indeksini az önce eklediğimiz boş satıra ayarlarız. Son olarak, liste Görünüşünü, yeni satır üzerinde düzenleme moduna ayarlarız.

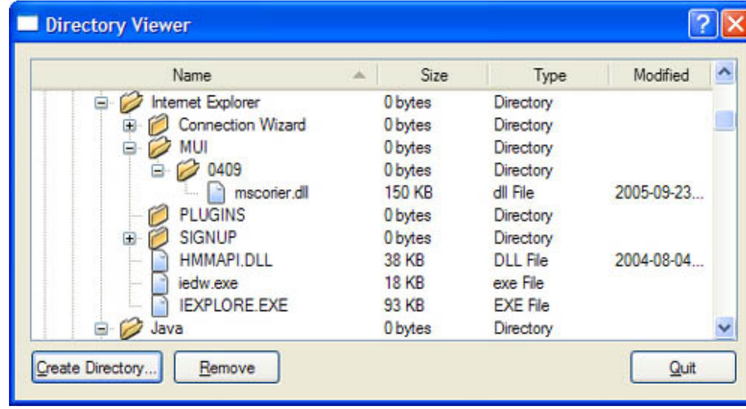
```
void TeamLeadersDialog::del()
{
    model->removeRows(listView->currentIndex().row(), 1);
}
```

Kurucuda, Delete butonunun clicked() sinyali del() yuvasına bağlanır. Sadece geçerli satırı sildiğimiz için, removeRows()'u, geçerli indeks konumu ve bir satır sayısı (burada 1) ile bir kere çağırırız. Tıpkı ekleme gibi, Modelin benzer şekilde Görünüşü güncelleyeceğine güveniriz.

```
QStringList TeamLeadersDialog::leaders() const
{
    return model->stringList();
}
```

Son olarak, leaders() fonksiyonu, diyalog kapandığında, düzenlenmiş karakter katarlarının geriye okunmasını sağlar.

Sıradaki örnek (Şekil 9.7), bilgisayarın dosya sistemini içeren ve çeşitli dosya niteliklerini gösterebilen (ve gizleyebilen) QDirModel sınıfını kullanır. Bu Model, gösterilen dosya sistemi girdilerini kısıtlamak için bir filtre uygulayabilir ve girdileri çeşitli yollarla sıralayabilir.



Şekil 9.7

Directory Viewer diyaloğunun kurucusunda Model ve Görünüşün oluşturulmasına ve kurulmasına bakarak başlayacağız:

```
DirectoryViewer::DirectoryViewer(QWidget *parent)
    : QDialog(parent)
{
    model = new QDirModel;
    model->setReadOnly(false);
    model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    treeView = new QTreeView;
    treeView->setModel(model);
    treeView->header()->setStretchLastSection(true);
    treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
    treeView->header()->setSortIndicatorShown(true);
    treeView->header()->setClickable(true);

    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);
    ...
}
```

Model inşa edilir edilmez, Modeli düzenlenebilir yaparız ve Modelin çeşitli ilk sıralama düzeni niteliklerini ayarlarız. Sonra, Modelin verisini görüntüleyecek olan `QTreeView`'ı oluştururuz. `QTreeView`'ın başlığı, kullanıcı kontrolünde sıralamayı sağlamada kullanılabilir. Başlığı tıklanabilir yaparak, kullanıcının herhangi bir sütun başlığını tıklayarak o sütunu sıralayabilmesi sağlanır. Ağaç Görünüşün başlığı kurulur kurulmaz, geçerli dizinin Model indeksini alırız ve eğer gerekliyse ebeveynini `expand()` kullanarak genişleterek bu dizinin görünür olduğundan emin oluruz, ve `scrollTo()` kullanarak onu kaydırırız. Sonra da, ilk sütunun tüm girdilerini eksilti(…) olmadan gösterecek genişlikte olduğundan emin oluruz.

Kurucunun burada gösterilmeyen parçasında, Create Directory ve Remove butonlarını, eylemleri yerine getirmek için yuvalara bağladık. Kullanıcılar F2'ye basarak ve bir şeyler yazarak yeniden adlandırabildikleri için Rename butonuna ihtiyacımız yok.

```
void DirectoryViewer::createDirectory()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;
```

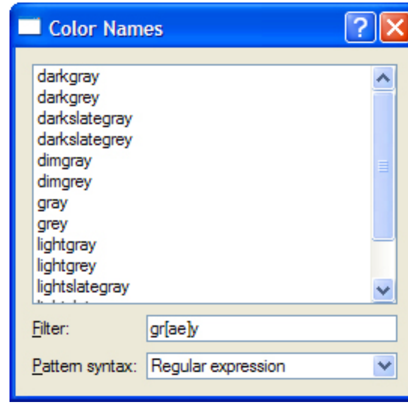
```
    QString dirName = QInputDialog::getText(this,
                                tr("Create Directory"),
                                tr("Directory name"));
    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid())
            QMessageBox::information(this, tr("Create Directory"),
                    tr("Failed to create the directory"));
    }
}
```

Eğer kullanıcı, girdi diyaloğunda bir dizin ismi girerse, geçerli dizinin bir çocuğu olacak şekilde ve bu isimde bir dizin oluşturmayı deneriz. `QDirModel::mkdir()` fonksiyonu, ebeveyn dizinin indeksini ve yeni dizinin ismini alır, ve oluşturduğu dizinin Model indeksini döndürür. Eğer işlem başarısız olursa, geçersiz bir Model indeksi döndürür.

```
void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
                tr("Failed to remove %1").arg(model->fileName(index)));
}
```

Eğer kullanıcı Remove'a tıklarsa, geçerli öğe ile ilişkili dosya ya da dizini silmeyi deneriz. Bunu yapmak için `QDir`'i kullanabilirdik, fakat `QDirModel`, `QModelIndex`'ler üzerinde çalışan uygunluk fonksiyonları sağlar.

Bu kısımdaki son örnek (Şekil 9.8), `QSortFilterProxyModel`'ın kullanılışını örnekle açıklıyor. Diğer önceden tanımlanmış Modellerden farklı olarak, bu Model, var olan bir Modeli içerir ve esas Model ve Görünüş arasında aktarılan veriyi işler. Örneğimizde, esas Model, Qt'un tanıdığı renk isimlerinin bir listesi (`QColor::colorNames()` sayesinde elde edilmiş) ile ilklendirilen bir `QStringListModel`'dır. Kullanıcı, bir `QLineEdit` içine bir filtre karakter katarı yazabilir ve bu karakter katarının nasıl yorumlanacağını bir açılan kutu(combobox) kullanarak belirtebilir.



Şekil 9.8

İşte, `ColorNamesDialog` kurucusundan bir alıntı:

```
ColorNamesDialog::ColorNamesDialog(QWidget *parent)
    : QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());

    proxyModel = new QSortFilterProxyModel(this);
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);

    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

`QStringListModel` alışılmış yolla oluşturulur ve doldurulur. Bunu, `QSortFilterProxyModel`'ın inşası izler. Esas Modeli `setSourceModel()` kullanarak aktarırız ve vekile(proxy) orijinal Modelin 0. sütununa dayanarak filtre etmesini söyleriz. `QComboBox::addItem()` fonksiyonu, `QVariant` tipinde isteğe bağlı bir "veri(data)" argümanı kullanır; bunu, her bir öğenin metnine tekabül eden `QRegExp::PatternSyntax` değerini saklamada kullanırız.

```
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
            QRegExp::PatternSyntax(syntaxComboBox->itemData(
                    syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}
```

`reapplyFilter()` yuvası, kullanıcı, filtre karakter katarını ya da örnek sözdizimi(pattern syntax) açma kutusunu değiştirdiğinde çağrılır. Satır editöründeki metni kullanarak bir `QRegExp` oluştururuz. Sonra, onun örnek sözdizimini, sözdizimi açma kutusunun geçerli öğesinin verisinde saklananlardan biri olarak ayarlarız.

setFilterRegExp()’yi çağırdığımızda, yeni filtre aktif olur. Bunun anlamı; filtre ile eşleşmeyen karakter katarlarını atacak ve Görünüş otomatik olarak güncellenmiş olacak.

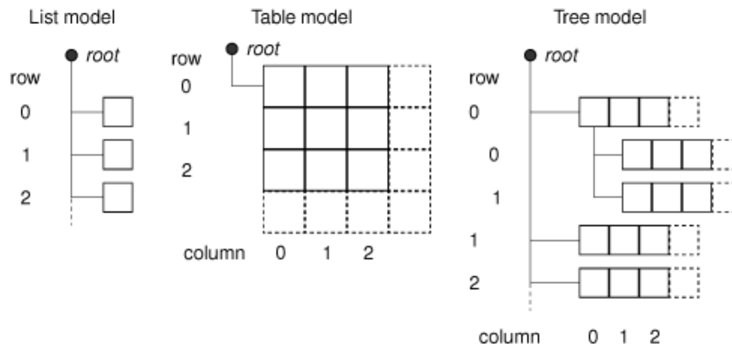
Özel Modeller Gerçekleştirme

Qt’un önceden tanımlanmış Modelleri, veriyi işlemede ve görüntülemede bir pratiklik sunar. Ancak, bazı veri kaynakları önceden tanımlanmış Modelleri kullanarak etkili biçimde kullanılamaz ve bu durumdan dolayı temelini oluşturan veri kaynağı için optimize edilmiş özel Modeller oluşturmak gereklidir.

Özel Modeller oluşturmaya başlamadan önce, Qt’un Model/Görünüş mimarisindeki anahtar kavramlara bir göz atalım. Bir Model, içindeki her veri elemanı bir Model indeksine ve roller(roles) denen ve isteğe bağlı değerler alabilen bir nitelikler setine sahiptir. Daha önceki bölümlerde en yaygın kullanılan rollerden, Qt::DisplayRole ve Qt::EditRole’ü gördük. Diğer roller, tamamlayıcı veriler için (örneğin, Qt::ToolTipRole, Qt::StatusTipRole ve Qt::WhatsThisRole), bazıları da temel görüntüleme niteliklerini kontrol etmek için kullanılırlar (Qt::FontRole, Qt::TextAlignmentRole, Qt::TextColorRole ve Qt::BackgroundColorRole gibi).

Bir liste Modeli için, tek indeks bileşeni, QModelIndex::row()’dan erişilebilen satır numarasıdır. Bir tablo modeli için, indeks bileşenleri, QModelIndex::row() ve QModelIndex::column()’dan erişilebilen, satır ve sütun numaralarıdır. Hem liste hem de tablo Modelleri için, her öğenin ebeveyni, geçersiz bir QModelIndex ile temsil edilen köktür. Bu kısımdaki ilk iki örnek, özel tablo Modelleri gerçekleştirmeyi gösterir.

Bir ağaç Modeli, bazı farlılıklarla birlikte, bir tablo Modeli ile benzerdir. Bir tablo Modeli gibi, üst seviye öğelerinin ebeveyni köktür (geçersiz bir QModelIndex), fakat diğer öğelerinin ebeveynleri, hiyerarşideki bazı diğer öğelerdir. Ebeveynlere, QModelIndex:parent()’dan erişilebilir. Her öğe, kendi rol verisine ve sıfır ya da daha fazla çocuğa sahiptir. Öğeler, diğer öğelere çocuk olarak sahip olabildikleri için, bu kısımda göstereceğimiz son örnekteki gibi özyinelemeli veri yapıları tarif etmek mümkündür. Farklı Modellerin bir şeması Şekil 9.9’da gösteriliyor.



Şekil 9.9

Bu kısımdaki ilk örnek, para birimlerini(currency) birbiriyle ilişkili olarak gösteren, saltokunur bir tablo Modelidir. Uygulama Şekil 9.10’da gösteriliyor.

	NOK	NZD	SEK	SGD	USD
NOK	1.0000	0.2254	1.1991	0.2592	0.1534
NZD	4.4363	1.0000	5.3195	1.1500	0.6804
SEK	0.8340	0.1880	1.0000	0.2162	0.1279
SGD	3.8578	0.8696	4.6258	1.0000	0.5917
USD	6.5200	1.4697	7.8180	1.6901	1.0000

Şekil 9.10

Uygulama, basit bir tablo kullanarak gerçekleştirilebilirdi, fakat özel bir Modelin avantajlarını kullanmak isteriz. Eğer halen işlem gören 162 para birimini bir tabloda saklamak isteseydik, saklamak için 162 x 162 = 26244 değere ihtiyacımız olacaktı; bu kısımda gösterilen özel `CurrencyModel` ile sadece 162 değere ihtiyacımız oluyor (her bir para biriminin değeri Amerikan doları(U.S dollar) ile ilişkili olarak).

`CurrencyModel` sınıfı standart bir `QTableView` ile kullanılacak. `CurrencyModel` bir `QMap<QString, double>` ile doldurulur; her bir anahtar bir para birimi kodu ve her bir değer para biriminin Amerikan doları olarak karşılığı. İşte, haritanın(map) nasıl doldurulduğunu ve Modelin nasıl kullanıldığını gösteren ufak bir kod parçası:

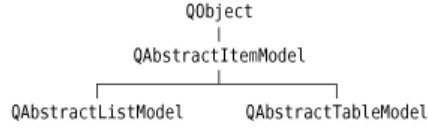
```
    QMap<QString, double> currencyMap;
    currencyMap.insert("AUD", 1.3259);
    currencyMap.insert("CHF", 1.2970);
    ...
    currencyMap.insert("SGD", 1.6901);
    currencyMap.insert("USD", 1.0000);

    CurrencyModel currencyModel;
    currencyModel.setCurrencyMap(currencyMap);

    QTableView tableView;
    tableView.setModel(&currencyModel);
    tableView.setAlternatingRowColors(true);
```

Şimdi, başlığıyla başlayarak Modelin gerçekleştirimine bakabiliriz:

```
class CurrencyModel : public QAbstractTableModel
{
public:
    CurrencyModel(QObject *parent = 0);

    void setCurrencyMap(const QMap<QString, double> &map);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
private:
    QString currencyAt(int offset) const;

    QMap<QString, double> currencyMap;
};
```

Veri kaynağımız ile en çok eşleşen Model olduğu için altsınıfını türetmek için `QAbstractTableModel`'ı seçtik. Qt, `QAbstractListModel`, `QAbstractTableModel` ve `QAbstractItemModel`'ı da içeren

birkaç temel sınıf sağlar; Şekil 9.11'e bakın. QAbstractItemModel sınıfı, özyinelemeli veri yapılarına dayananları da kapsayan, Modellerin birçok çeşidini desteklemede kullanılırken, QAbstractListModel ve QAbstractTableModel sınıfları bir ya da iki boyutlu veri setleri kullanıldığında kolaylık için sağlanırlar.



Şekil 9.11

Saltokunur bir tablo Modeli için, üç fonksiyonu uyarlamalıyız: rowCount(), columnCount() ve data(). Bu durumda, ayrıca headerData()'yı da uyarlarız ve veriyi ilklendirmek için bir fonksiyon sağlarız (setCurrencyMap()).

```
CurrencyModel::CurrencyModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

Kurucuda, parent parametresini temel sınıfa aktarmanın dışında hiçbir şey yapmamız gerekmez.

```
int CurrencyModel::rowCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
}
int CurrencyModel::columnCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
}
```

Bu tablo Modeli için, satır ve sütun sayıları, para birimi haritasındaki para birimlerinin sayısıdır. parent parametresi, bir tablo modeli için hiçbir anlam ifade etmez; vardır, çünkü rowCount() ve coloumnCount(), daha genelleyici olan ve hiyerarşileri destekleyen QAbstractItemModel temel sınıfından miras alınırlar.

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());

        if (currencyMap.value(rowCurrency) == 0.0)
            return "####";

        double amount = currencyMap.value(columnCurrency)
                        / currencyMap.value(rowCurrency);

        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

data() fonksiyonu, bir öğenin rollerinden herhangi birinin değerini döndürür. Öğe, bir QModelIndex olarak belirtilir. Bir tablo Modeli için, bir QModelIndex'in ilgi çekici bileşenleri, row() ve column() kullanarak elde edilebilen satır ve sütun numaralarıdır.

Eğer rol Qt::TextAlignmentRole ise, sayılar için uygun bir hizalanış döndürürüz. Eğer rol Qt::DisplayRole ise, her bir para birimi için değeri arayıp buluruz ve kurunu hesaplarız.

Hesaplanmış değeri bir double olarak döndürebilirdik, fakat daha sonra kaç tane ondalık hane gösterileceği üzerinde kontrole sahip olmayacaktık (özel bir Delege kullanmadıkça). Onun yerine, değeri, isteğimize göre biçimlenmiş bir karakter katarı olarak döndürürüz.

```
QVariant CurrencyModel::headerData(int section,
                                   Qt::Orientation /* orientation */,
                                   int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    return currencyAt(section);
}
```

headerData() fonksiyonu, Görünüş tarafından yatay ve dikey başlıkları doldurmak için çağrılır. section parametresi, satır ya da sütun numarasıdır (yönelime(orientation) bağlı olarak). Satırlar ve sütunlar aynı para birimi kodlarına sahip oldukları için yönelimi önemsemeyiz ve sadece, verilen bölme sayısı için para biriminin kodunu döndürürüz.

```
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    currencyMap = map;
    reset();
}
```

Çağıran, setCurrencyMap()'i kullanarak para birimi haritasını değiştirebilir. QAbstractItem-Model::reset() çağrısı, Model kullanan her Görünüşe, tüm verilerinin geçersiz olduğunu bildirir; bu onları, görünür öğeleri için taze veri istemeye zorlar.

```
QString CurrencyModel::currencyAt(int offset) const
{
    return (currencyMap.begin() + offset).key();
}
```

currencyAt() fonksiyonu, para birimi haritasında verilen göreli konuma(offset), anahtar (para birimi kodu) döndürür. Öğeyi bulmak için bir STL-tarzı yineleyici(STL-style iterator) kullanırız ve üzerinde key()'i çağırırız.

Görüldüğü gibi, esas verinin doğasına bağlı kalarak, saltokunur Modeller oluşturmakta bir zorluk yoktur ve iyi tasarlanmış bir Model ile bellek tasarrufu ve hız sağlamak mümkündür. Sıradaki örneğimiz; Şekil 9.12'de gösterilen Cities uygulaması da tablo temellidir, fakat bu sefer tüm veri kullanıcı tarafından girilir.

Cities

	Arvika	Boden	Eskilstuna	Falun
Arvika	0	1063	280	285
Boden	1063	0	958	830
Eskilstuna	280	958	0	0
Falun	285	830	0	0
Filipstad	122	0	0	0
Halmstad	0	0	0	0

Şekil 9.12

Bu uygulama, herhangi iki şehir(city) arasındaki mesafeyi gösteren değerleri saklamada kullanılır. Bir önceki örnekteki gibi sadece bir `QTableWidget` kullanabilir ve her şehir çifti için bir öğe saklayabilirdik. Ancak, özel bir Model daha etkili olabilir, çünkü herhangi bir A şehrinden herhangi farklı bir B şehrine olan mesafe A'dan B'ye veya B'den A'ya seyahatlerde aynıdır, bu nedenle öğeler ana köşegen boyunca aynıdır.

Özel bir Model ile basit bir tablo arasındaki farkı görmek için üç şehrimiz olduğunu farz edelim, A, B ve C. Eğer her bir kombinasyon için bir değer saklasak, dokuz değer saklamamız gerekecekti. Dikkatlice tasarlanmış bir model sadece şu üç öğeyi gerektirir: (A, B), (A, C) ve (B, C).

İşte, modelin kurgulanması ve kullanılması:

```
    QStringList cities;
    cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun"
           << "Filipstad" << "Halmstad" << "Helsingborg" << "Karlstad"
           << "Kiruna" << "Kramfors" << "Motala" << "Sandviken"
           << "Skara" << "Stockholm" << "Sundsvall" << "Trelleborg";

    CityModel cityModel;
    cityModel.setCities(cities);

    QTableView tableView;
    tableView.setModel(&cityModel);
    tableView.setAlternatingRowColors(true);
```

Önceki örnek için yaptığımız gibi, aynı fonksiyonları yine uyarlamalıyız. Ek olarak, Modeli düzenlenebilir yapmak için `setData()` ve `flags()`'ı da uyarlamalıyız. İşte, sınıf tanımı:

```
class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    CityModel(QObject *parent = 0);

    void setCities(const QStringList &cityNames);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role);
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
```

```
    Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;

    QStringList cities;
    QVector<int> distances;
};
```

Bu model için iki veri yapısı kullanıyoruz: şehir isimlerini tutmada `QStringList` tipinde `cities`, ve her bir benzersiz şehirler çifti arasındaki mesafeyi tutmada `QVektor<int>` tipinde `distances`.

```
CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

Kurucu, temel sınıfa `parent` parametresini aktarmanın ötesinde hiçbir şey yapmaz.

```
int CityModel::rowCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}
int CityModel::columnCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}
```

Şehirlerin kare bir ızgarasına sahip olduğumuz için, satırların ve sütunların sayısı, listemizdeki şehirlerin sayısıdır.

```
QVariant CityModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        if (index.row() == index.column())
            return 0;
        int offset = offsetOf(index.row(), index.column());
        return distances[offset];
    }
    return QVariant();
}
```

`data()` fonksiyonu `CurrencyModel`'da yaptığımızla benzerdir. Eğer sütun ve satır aynı ise 0 döndürür, çünkü iki şehrin aynı olduğu durumlardır; aksi halde, `distances` vektörü(vector) içinde verilen satır ve sütun için girdiyi bulur ve şehir çiftine ait mesafeyi döndürür.

```
QVariant CityModel::headerData(int section,
                               Qt::Orientation /* orientation */,
                               int role) const
{
    if (role == Qt::DisplayRole)
        return cities[section];
    return QVariant();
}
```

headerData() fonksiyonu basittir, çünkü her satırın özdeş bir sütun başlığına sahip olduğu, kare bir tabloya sahibiz. Basitçe, verilen göreli konuma göre cities karakter katarı listesinden şehrin adını döndürürüz.

```
bool CityModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column()
            && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(),
                                                  index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}
```

setData() fonksiyonu, kullanıcı bir öğeyi düzenlediğinde çağrılır. Sağlanan Model indeksi geçerli, iki şehir farklı ve düzenlenecek veri elemanı Qt::EditRole ise, fonksiyon kullanıcının girdiği değeri distances vektörü içinde saklar.

createIndex() fonksiyonu, bir Model indeksi üretmede kullanılır.

Değişen öğenin Model indeksi ile dataChanged() sinyalini yayarız. Sinyal, iki Model indeksi alır. dataChanged() sinyalini ayrıca, Görünüşün öğeyi tazeleyeceğini garantiye almak amacıyla, indeksi aktarmak için de yayarız. Son olarak, düzenlemenin başarılı olup olmadığını bildirmek için true ya da false döndürürüz.

```
Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column())
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

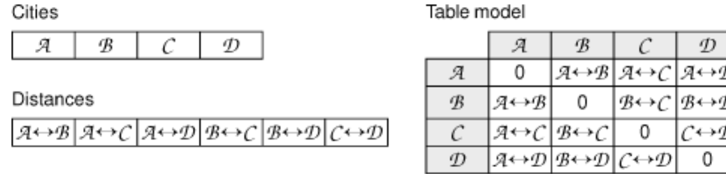
Model, bir öğe ile ne yapılabilineceğini (örneğin, düzenlenebilir olup olmadığını) iletmede flags() fonksiyonunu kullanır. QAbstractTableModel'daki varsayılan gerçekleştirim, Qt::ItemIsSelectable | Qt::ItemIsEnabled döndürür. Biz, köşegen üzerinde bulunanlar dışındaki (her zaman 0 olanlar) tüm öğeler için Qt::ItemIsEditable bayrağını ekleriz.

```
void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}
```

Eğer yeni bir şehirler listesi verilmişse, private QStringList'i yeni listeye ayarlarız; distance vektörünü yeniden boyutlandırır ve temizleriz, ve görünür öğeleri güncellenmesi gereken her Görünüşü haberdar etmek için QAbstractItemModel::reset()'i çağırırız.

```
int CityModel::offsetOf(int row, int column) const
{
    if (row < column)
        qSwap(row, column);
    return (row * (row - 1) / 2) + column;
}
```

offsetOf() private fonksiyonu, verilen şehir çiftinin distances vektörü içindeki indeksini hesaplar. Örneğin, A, B, C ve D şehirlerine sahipsek ve kullanıcı, satır 3, sütun 1'i (B'den D'ye) güncellemişse, göreli konum 3 x (3 - 1)/2 + 1 = 4 olacaktır. Eğer kullanıcı onun yerine satır 1, sütun 3'ü (D'den B'ye) güncellemiş olsaydı, qSwap()'a şükürler olsun ki, tamamen aynı hesaplama yapılacaktı ve aynı göreli konum döndürülecekti. Şekil 9.13, şehirler, mesafeler ve ilgili tablo modeli arasındaki ilişkileri resmeder.

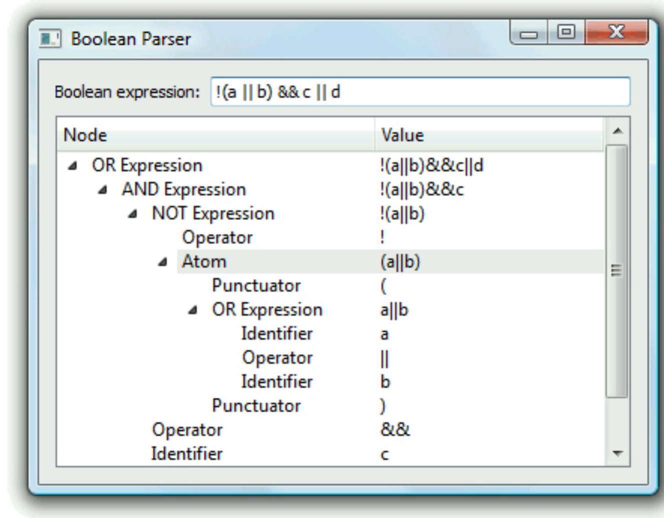


Şekil 9.13

Bu kısımdaki son örnek, verilen bir Boolean deyim için ayrıştırma ağacı(parse tree) gösteren bir modeldir. Bir Boolean deyim, ya "bravo" gibi basit alfanümerik bir belirteçtir ya da "&&", "||", veya "!" operatörleri kullanılarak, daha basit deyimlerden inşa edilmiş karmaşık bir deyimdir. Örneğin, "a || (b && !c)" bir Boolean deyimdir.

Boolean Parser uygulaması(Şekil 9.14), dört sınıftan meydana gelir:

- BooleanWindow, kullanıcının bir Boolean deyim girmesine izin veren ve ilgili ayrıştırma ağacını gösteren bir penceredir.
- BooleanParser, bir Boolean deyimden bir ayrıştırma ağacı üretir.
- BooleanModel, bir ayrıştırma ağacı içeren bir ağaç modelidir.
- Node, bir ayrıştırma ağacındaki bir öğeyi temsil eder.



Şekil 9.14

Node sınıfı ile başlayalım:

```
class Node
{
public:
    enum Type { Root, OrExpression, AndExpression, NotExpression, Atom,
                Identifier, Operator, Punctuator };

    Node(Type type, const QString &str = "");
    ~Node();

    Type type;
    QString str;
    Node *parent;
    QList<Node *> children;
};
```

Her düğüm(node) bir tipe, bir karakter katarına (boş(empty) olabilen), bir ebeveyn (boş(null) olabilen) ve bir çocuk düğümler listesine (boş(empty) olabilen) sahiptir.

```
Node::Node(Type type, const QString &str)
{
    this->type = type;
    this->str = str;
    parent = 0;
}
```

Kurucu sadece düğümün tipine ve karakter katarına ilk değer atar ve ebeveyni boş olarak (ebeveyn yok anlamında) ayarlar. Çünkü tüm veri publictir ve böylece Node sınıfı, tipi, karakter katarını, ebeveyni ve çocukları doğrudan işleyebilir.

```
Node::~Node()
{
    qDeleteAll(children);
}
```

qDeleteAll() fonksiyonu, bir işaretçiler konteyneri(container of pointers) üzerinde yinelenir ve her biri üzerinde delete'i çağırır.

Artık, veri öğelerimizi tanımladık (her biri bir `Node` tarafından temsil edilir) , bir Model oluşturmaya hazırız:

```
class BooleanModel : public QAbstractItemModel
{
public:
    BooleanModel(QObject *parent = 0);
    ~BooleanModel();

    void setRootNode(Node *node);

    QModelIndex index(int row, int column,
                      const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
private:
    Node *nodeFromIndex(const QModelIndex &index) const;

    Node *rootNode;
};
```

Bu sefer ana sınıf olarak, `QAbstractItemModel`'ın uygunluk altsınıfı olan `QAbstractTableModel`'ı kullanmak yerine, kendisini kullandık, çünkü biz hiyerarşik bir Model oluşturmak istiyoruz. Modelin verisini ayarlamak için bir ayrıştırma ağacının kök düğümü(root node) ile çağrılması gereken `setRootNode()` fonksiyonuna sahibiz.

```
BooleanModel::BooleanModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}
```

Modelin kurucusunda sadece, kök düğümü güvenli bir boş(null) değere ayarlamamız ve ana sınıfa parent'ı aktarmamız gerekir.

```
BooleanModel::~BooleanModel()
{
    delete rootNode;
}
```

Yokedicide, kök düğümü sileriz. Eğer kök düğüm çocuklara sahip ise, bunların her biri, özyinelemeli olarak `Node` yokedicisi tarafından silinir.

```
void BooleanModel::setRootNode(Node *node)
{
    delete rootNode;
    rootNode = node;
    reset();
}
```

Yeni bir kök düğüm ayarlandığında, önceki kök düğümü (ve onun tüm çocuklarını) silerek başlarız. Sonra yeni kök düğümü ayarlarız ve her görünür öğesini güncellemesi gereken Görünüşleri haberdar etmek için `reset()`'i çağırırız.

```
QModelIndex BooleanModel::index(int row, int column,
                                const QModelIndex &parent) const
{
    if (!rootNode || row < 0 || column < 0)
        return QModelIndex();
    Node *parentNode = nodeFromIndex(parent);
    Node *childNode = parentNode->children.value(row);
    if (!childNode)
        return QModelIndex();
    return createIndex(row, column, childNode);
}
```

index() fonksiyonu, QAbstractItemModel'dan uyarlanmıştır. Model ya da Görünüş, belirli bir çocuk öğe (ya da eğer parent geçersiz bir QModelIndex ise, bir üst-seviye öğe) için bir QModelIndex oluşturması gerektiğinde çağrılır. Tablo ve liste Modelleri için bu fonksiyonu uyarlamamız gerekmez, çünkü QAbstractListModel'ın ve QAbstractTableModel'ın varsayılan gerçekleştirimleri genellikle yeterli olur.

Bizim index() gerçekleştirimimizde, eğer hiç ayrıştırma ağacı ayarlanmamışsa, geçersiz bir QModelIndex döndürürüz. Aksi halde, verilen satır, sütun ve çocuk için bir Node * ile bir QModelIndex oluştururuz. Hiyerarşik Modeller için, bir öğenin, ebeveynine göre göreli satır ve sütununu bilmek onu tanımak için yeterli değildir; ebeveyninin kim olduğunu da bilmeliyiz. Bunu çözmek için, QModelIndex içinde, dâhili düğüme işaret eden bir işaretçi saklayabiliriz. QModelIndex bize, satır ve sütun numaralarına ek olarak bir void * ya da bir int saklama seçeneği verir.

Çocuk için Node *, ebeveyn düğümün children listesi sayesinde elde edilir. Ebeveyn düğüm, nodeFromIndex() private fonksiyonunu kullanarak, parent Model indeksinden elde edilir:

```
Node *BooleanModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return rootNode;
    }
}
```

nodeFromIndex() fonksiyonu, verilen indeksin void *'ini Node *'a dönüştürür, ya da, eğer indeks geçersiz ise, geçersiz bir Model indeksi bir Modelde kökü temsil etmekte kullanıldığı için, kök düğümü döndürür.

```
int BooleanModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
        return 0;
    return parentNode->children.count();
}
```

Verilen bir öğe için satırların sayısı sadece sahip olduğu çocuklar kadardır.

```
int BooleanModel::columnCount(const QModelIndex & /* parent */) const
{
```

```
    return 2;
}
```

Sütunların sayısı 2'ye sabitlenmiştir. İlk sütun düğüm tiplerini, ikinci sütun düğüm değerlerini tutar.

```
QModelIndex BooleanModel::parent(const QModelIndex &child) const
{
    Node *node = nodeFromIndex(child);
    if (!node)
        return QModelIndex();
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();
    Node *grandparentNode = parentNode->parent;
    if (!grandparentNode)
        return QModelIndex();

    int row = grandparentNode->children.indexOf(parentNode);
    return createIndex(row, 0, parentNode);
}
```

Bir çocuktan ebeveyn `QModelIndex`'e erişmek, bir ebeveynin çocuğunu bulmaktan biraz daha zahmetlidir. `nodeFromIndex()`'i kullanarak kolaylıkla ebeveyn düğüme erişebilir ve `Node`'un ebeveyn işaretçisini kullanarak yükselebiliriz, fakat satır numarasını elde etmek için, büyük-ebeveyne(grandparent) geri gitmemiz ve onun çocuklarının ebeveyn listesinde (örneğin, çocuğun büyük-ebeveyninin) ebeveynin indeks konumunu bulmamız gerekir.

Kod Görünümü:

```
QVariant BooleanModel::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();

    if (index.column() == 0) {
        switch (node->type) {
        case Node::Root:
             return tr("Root");
        case Node::OrExpression:
            return tr("OR Expression");
        case Node::AndExpression:
            return tr("AND Expression");
        case Node::NotExpression:
            return tr("NOT Expression");
        case Node::Atom:
            return tr("Atom");
        case Node::Identifier:
            return tr("Identifier");
        case Node::Operator:
            return tr("Operator");
        case Node::Punctuator:
            return tr("Punctuator");
        default:
            return tr("Unknown");
        }
```

```
    } else if (index.column() == 1) {
        return node->str;
    }
    return QVariant();
}
```

data()'da, istenen öğe için Node *'a erişiriz ve esas veriye erişmek için onu kullanırız. Eğer çağıran, Qt::DisplayRole dışında herhangi bir rol için bir değer isterse ya da verilen Model indeksi için bir Node'a erişemezsek, geçersiz bir QVariant döndürürüz. Eğer sütun 0 ise, düğümün tipinin ismini, eğer sütun 1 ise, düğümün değerini (karakter katarını) döndürürüz.

```
QVariant BooleanModel::headerData(int section,
                                  Qt::Orientation orientation,
                                  int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}
```

Bizim headerData() uyarlamamızda, uygun yatay başlık etiketlerini döndürürüz. Hiyerarşik Modeller için kullanılan QTreeView sınıfı, dikey başlığa sahip değildir, bu nedenle bu olasılığı yoksayarız.

Artık, Node ve BooleanModel sınıflarına değindik; şimdi, kullanıcı satır editöründeki metni değiştirdiğinde kök düğümün nasıl oluşturulduğunu görelim:

```
void BooleanWindow::booleanExpressionChanged(const QString &expr)
{
    BooleanParser parser;
    Node *rootNode = parser.parse(expr);
    booleanModel->setRootNode(rootNode);
}
```

Kullanıcı, uygulamanın satır editöründeki metni değiştirdiğinde, ana pencerenin booleanExpression-Changed() yuvası çağrılır. Bu yuvada, kullanıcının metni ayrıştırılır ve ayrıştırıcı, ayrıştırma ağacının kök düğümüne bir işaretçi döndürür.

BooleanParser sınıfını göstermedik çünkü GUI ya da Model/Görünüş programlama ile ilgili değildir. Yine de örneğin tam kaynağını "Örnekler" dizininde bulabilirsiniz.

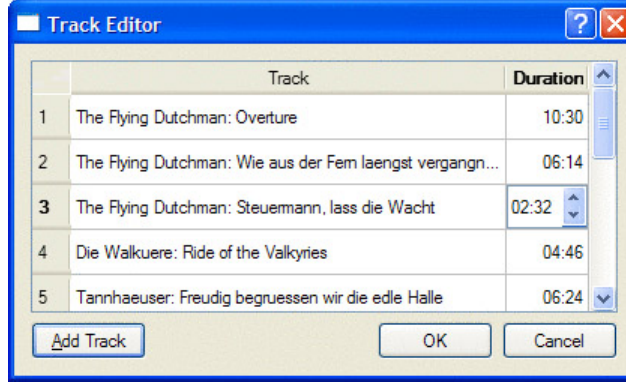
BooleanModel gibi ağaç modelleri gerçekleştirildiğinde, QTreeView'ın tuhaf davranışlar sergilemesiyle sonuçlanabilecek hatalar yapmak oldukça kolaydır. Özel veri modellerinde problemleri bulmaya yardım etmek ve çözmek için bir ModelTest sınıfı mevcuttur.

Özel Delegeler Gerçekleştirme

Görünüşler içindeki özel öğeler, Delegeler(delegates) kullanılarak yorumlanır ve düzenlenirler. Çoğu durumda, bir Görünüş tarafından sağlanan varsayılan Delege yeterlidir. Eğer öğelerin yorumlanması üzerinde daha ince bir kontrole sahip olmak istiyorsak, isteğimizi basitçe özel bir Model kullanarak gerçekleştirebiliriz: data() gerçekleştirimimizde, Qt::FontRole, Qt::TextAlignmentRole,

Qt::TextColorRole ve Qt::BackgroundColorRole'ü işleyebiliriz ve bunlar varsayılan Delege tarafından kullanılabilir. Örneğin, daha önce gördüğümüz Cities ve Currencies örneklerinde, Qt::Text-AlignmentRole'ünü sağa hizalı sayılar elde etmede kullanmıştık.

Eğer daha da büyük derecede kontrol istiyorsak, kendi Delege sınıfımızı oluşturabilir ve onu, kullanmak istediğimiz Görünüşler üzerinde ayarlayabiliriz. Şekil 9.15'te gösterilen Track Editor diyaloğu, özel bir Delege kullanır. Müzik parçalarının başlıklarını(titles) ve sürelerini gösterir. Model tarafından tutulan veri sadece QString'ler (başlıklar) ve int'ler (saniyeler) olacak, fakat süreler dakika ve saniyelere ayrılmış olacak ve bir QTimeEdit kullanılarak düzenlenebilir olacak.



Şekil 9.15

Track Editor diyaloğu, QTableWidgetItem'lar üzerinde işletilen bir öğe görüntüleme uygunluk altsınıfı olan, bir QTableWidget kullanır. Veri, Track'ların bir listesi olarak sağlanır:

```
class Track
{
public:
    Track(const QString &title = "", int duration = 0);

    QString title;
    int duration;
};
```

İşte, kurucudan, tablo parçacığının oluşumunu ve dolduruluşunu gösteren bir alıntı:

```
TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
    : QDialog(parent)
{
    this->tracks = tracks;

    tableWidget = new QTableWidget(tracks->count(), 2);
    tableWidget->setItemDelegate(new TrackDelegate(1));
    tableWidget->setHorizontalHeaderLabels(
            QStringList() << tr("Track") << tr("Duration"));

    for (int row = 0; row < tracks->count(); ++row) {
        Track track = tracks->at(row);

        QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
        tableWidget->setItem(row, 0, item0);

        QTableWidgetItem *item1
             = new QTableWidgetItem(QString::number(track.duration));
```

```
            item1->setTextAlignment(Qt::AlignRight);
            tableWidget->setItem(row, 1, item1);
        }
        ...
}
```

Kurucu, bir tablo parçacığı oluşturur ve varsayılan Delegeyi kullanmak yerine, bizim özel `TrackDelegate` Delegemizi kullanırız, ve onu zaman verisini tutan sütuna aktarırız. Sütun başlıklarını ayarlayarak başlarız ve sonrada veriyi baştan sona yineleriz. Satırları ise her bir parçanın isim ve süresiyle doldururuz.

Kurucunun ve `TrackEditor` sınıfının geri kalanı hiçbir sürpriz barındırmaz, bu nedenle şimdi, gerçeklemeyi ve müzik parçası verisinin düzenlenmesini ele alan `TrackDelegate`'e bakacağız.

```
class TrackDelegate : public QItemDelegate
{
    Q_OBJECT

public:
    TrackDelegate(int durationColumn, QObject *parent = 0);

    void paint(QPainter *painter, const QStyleOptionViewItem &option,
               const QModelIndex &index) const;
    QWidget *createEditor(QWidget *parent,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;
    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const;

private slots:
    void commitAndCloseEditor();

private:
    int durationColumn;
};
```

Ana sınıfımız olarak `QItemDelegate`'i kullanırız, böylece varsayılan Delege gerçekleştiriminden yararlanırız. Ayrıca, eğer en baştan başlamak isteseydik, `QAbstractItemDelegate`'i de kullanabilirdik. Veriyi düzenleyebilen bir Delege ihtiyacını karşılamak için, `createEditor()`, `setEditorData()` ve `setModelData()`'yı da gerçekleştirmemiz gerekir. Ayrıca, süre sütununun gerçeklenmesini değiştirmek için `paint()`'i gerçekleştirmeliyiz.

```
TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
    : QItemDelegate(parent)
{
    this->durationColumn = durationColumn;
}
```

`DurationColumn` parametresi kurucuya, parçanın süresini tutan sütunu bildirir.

```
void TrackDelegate::paint(QPainter *painter,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QString text = QString("%1:%2")
```

```
                         .arg(secs / 60, 2, 10, QChar('0'))
                         .arg(secs % 60, 2, 10, QChar('0'));

        QStyleOptionViewItem myOption = option;
        myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;

        drawDisplay(painter, myOption, myOption.rect, text);
        drawFocus(painter, myOption, myOption.rect);
    } else{
        QItemDelegate::paint(painter, option, index);
    }
}
```

Süreyi "dakika : saniye" biçiminde sunmak istediğimiz için `paint()` fonksiyonunu uyarlarız. `arg()` çağrısı, bir karakter katarı olarak sunmak için bir tamsayı, karakter katarının uzunluğunu, tamsayının tabanını (onlu için 10) ve dolgu(padding) karakterini (`QChar('0')`) alır.

Sağa hizalı metin için, geçerli stil seçeneklerini kopyalarız ve varsayılan hizalanışın üzerine yazarız. Sonra, metni çizmek için, öğe bir odağa sahip olduğunda, dikdörtgen bir odak çizecek, aksi halde hiçbir şey yapmayacak olan `QItemDelegate::drawFocus()`'un ardından `QItemDelegate::drawDisplay()`'i çağırırız. `drawDisplay()`'i kullanmak çok pratiktir, özelliklede kendi stil seçeneklerimizle kullanıldığında. Doğrudan, çizici kullanarak da çizebilirdik.

```
QWidget *TrackDelegate::createEditor(QWidget *parent,
        const QStyleOptionViewItem &option,
        const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()),
                this, SLOT(commitAndCloseEditor()));
        return timeEdit;
    } else {
        return QItemDelegate::createEditor(parent, option, index);
    }
}
```

Sadece parçanın süresinin düzenlenmesini kontrol etmek, parça isminin düzenlenmesini varsayılan Delegeye bırakmak istiyoruz. Bunu, Delegenin hangi sütun için bir editör sağladığını kontrol ederek başarabiliriz. Eğer süre sütunu ise, bir `QTimeEdit` oluştururuz, görüntüleme biçimini(display format) uygun olarak ayarlarız ve onun `editingFinished()` sinyalini, bizim `commitAndCloseEditor()` yuvamıza bağlarız. Diğer herhangi bir sütun için, düzenleme idaresini varsayılan Delegeye aktarırız.

```
void TrackDelegate::commitAndCloseEditor()
{
    QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}
```

Eğer kullanıcı Enter'a basarsa ya da odağı `QTimeEdit`'in dışına taşırsa (fakat Esc'e basmadan), `editingFinished()` sinyali yayılır ve `commitAndCloseEditor()` yuvası çağrılır. Bu yuva, Görünüşe düzenlenmiş verinin varlığını bildirmek ve böylece düzenlenmiş verinin var olan verinin yerini alması için, `commitData()` sinyalini yayar. Ayrıca, Görünüşe bu editöre daha fazla gerek duyulmadığını bildirmek için

closeEditor() sinyalini de yayar; bu durumda Model onu silecektir. Editöre, yuvayı tetikleyen sinyali yayan nesneyi döndüren QObject::sender() kullanılarak erişilir. Eğer kullanıcı iptal ederse (Esc'e basarak), Görünüş sadece editörü silecektir.

```
void TrackDelegate::setEditorData(QWidget *editor,
                                  const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
        QItemDelegate::setEditorData(editor, index);
    }
}
```

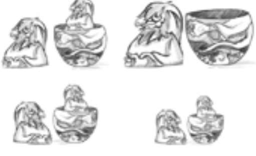
Kullanıcı düzenlemeyi başlattığında, Görünüş bir editör oluşturmak için createEditor()'ı çağırır ve sonra editörü ilk kullanıma hazırlamak için setEditorData()'yı öğenin geçerli verisi ile çağırır. Eğer editör süre sütunu içinse, parçanın süresini saniye olarak sağlarız ve QTimeEdit'in zamanını, dakikaların ve saniyelerin yerini tutan sayılara ayarlarız; aksi halde, ilk kullanıma hazırlamayı varsayılan Delegenin idaresine bırakırız.

```
void TrackDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QItemDelegate::setModelData(editor, model, index);
    }
}
```

Eğer kullanıcı düzenlemeyi iptal etmektense onu tamamlarsa (editör parçacığının dışına tıklayarak ya da Enter veya Tab'a basarak), Model editörün verisi ile güncellenmelidir. Eğer süre düzenlenmişse, QTimeEdit'ten dakikaları ve saniyeleri sağlarız ve veriyi saniyelerin yerini tutan sayılara ayarlarız.

Bu durumda gerekli olmasa da, bir Model içindeki herhangi bir verinin düzenlenmesini ve sunulmasını tümüyle kontrol eden özel bir delege oluşturmak da mümkündür. Biz, belirli bir sütunun kontrolünü elde etmeyi seçtik, fakat QModelIndex, uyarladığımız bütün fonksiyonlara aktarıldığı için, sütunun yanında, satırın, dikdörtgen bölgenin, ebeveynin ya da bunların herhangi bir kombinasyonunun, yani ihtiyacımız olan tüm öğelerin kontrolünü de elde edebiliriz.

BÖLÜM 10: KONTEYNER SINIFLARI



Konteyner sınıfları, verilen bir tipteki öğeleri bellekte saklayan genel amaçlı şablon sınıflarıdır. C++ hâlihazırda, Standart C++ kütüphanesine dâhil olan Standart Şablon Kütüphanesi(Standard Template Library - STL) ile birçok konteyner sunar.

Qt kendi konteyner sınıflarını sunar, fakat Qt programları için hem Qt hem de STL konteynerlerini kullanabiliriz. Qt konteynerlerinin başlıca avantajları, tüm platformlarda aynı şekilde davranmaları ve örtülü olarak paylaşılmalarıdır. Örtük paylaşım(Implicit sharing) ya da "copy on write", kayda değer herhangi bir performans kaybı olmadan, konteynerlerin tümünü değerler gibi aktarmayı mümkün kılan bir optimizasyondur.

Qt, hem `QVector<T>`, `QLinkedList<T>` ve `QList<T>` gibi ardışık(sequential) konteynerler, hem de `QMap<K, T>` ve `QHash<K, T>` gibi birleşik(associative) konteynerler sunar. Kavramsal olarak, ardışık konteynerler öğeleri birbiri ardına saklarken, birleşik konteynerler anahtar-değer(key-value) çiftlerini saklarlar.

Qt ayrıca, isteğe bağlı konteynerler üzerinde işlemler yapan genel algoritmalar(generic algorithms) sağlar. Örneğin, `qSort()` algoritması ardışık bir konteyneri sıralar ve `qBinaryFind()` sıralanmış bir ardışık konteyner üzerinde ikili(binary) arama yapar. Bu algoritmalar, STL tarafından sunulanlarla benzerdirler.

Eğer zaten STL konteynerlerine aşinaysanız ve hedef platformlarınızda STL kullanılabilir durumdaysa, onları, Qt konteynerlerinin yerine ya da onlara ek olarak kullanmak isteyebilirsiniz. STL sınıfları ve fonksiyonları hakkında daha fazla bilgi edinmeye başlamak için SGI'ın STL web sitesi iyi bir yerdir: http://www.sgi.com/tech/stl/.

Bu bölümde, konteynerler ile birçok ortaklıkları olduğu için, `QString`, `QByteArray` ve `QVariant`'a da bakacağız. `QString`, Qt'un API'ının her tarafında kullanılan 16-bit Unicode bir karakter katarıdır. `QByteArray`, ham(raw) ikili veriyi saklamak için faydalı, 8-bitlik `char`'ların bir dizisidir. `QVariant`, birçok C++ ve Qt değer tiplerini saklayabilen bir tiptir.

Ardışık Konteynerler

`QVector<T>`, öğelerini hafızada bitişik konumlarda saklayan, dizi benzeri bir veri yapısıdır (Şekil 10.1). Bir vektör, kendi boyutunu bilmesi ve yeniden boyutlandırılabilmesi ile sade bir C++ dizisinden ayrılır. Bir vektörün sonuna ekstra öğeler eklemek oldukça verimlidir, oysa bir vektörün önüne ya da ortasına öğeler eklemek pahalıya mal olabilir.

0	1	2	3	4
937.81	25.984	308.74	310.92	40.9

Şekil 10.1

Eğer peşinen kaç öğeye ihtiyacımız olacağını biliyorsak, onu tanımlarken bir ilk boyut verebiliriz ve öğelere bir değer atamakta `[]` operatörünü kullanabiliriz; aksi halde, vektörü daha sonra yeniden boyutlandırmalıyız ya da öğeleri sona eklemeliyiz (`append()` fonksiyonunu kullanarak). İşte, ilk boyutunu belirttiğimiz bir örnek:

```
QVector<double> vect(3);
vect[0] = 1.0;
vect[1] = 0.540302;
vect[2] = -0.416147;
```

Bu da aynısı, fakat bu sefer boş bir vektörle başlarız ve `append()` fonksiyonunu kullanarak öğeleri sona ekleriz:

```
QVector<double> vect;
vect.append(1.0);
vect.append(0.540302);
vect.append(-0.416147);
```

`append()` fonksiyonunun yerine << operatörünü de kullanabiliriz:

```
vect << 1.0 << 0.540302 << -0.416147;
```

Vektörün öğelerini yinelemenin bir yolu da, `[]` ve `count()` kullanmaktır:

```
double sum = 0.0;
for (int i = 0; i < vect.count(); ++i)
    sum += vect[i];
```

Belli bir değer ataması olmaksızın oluşturulan vektör girdileri, öğenin varsayılan kurucusu kullanılarak ilk kullanıma hazırlanır. Temel tipler ve işaretçi tipleri 0'a ilklendirilirler.

Bir `QVector<T>`'ın başlangıcına ya da ortasına öğeler eklemek, ya da bu konumlardan öğeler silmek, büyük vektörler için verimsiz olabilir. Bu sebeple Qt, öğelerini hafızada Şekil 10.2'de gösterildiği gibi, bitişik olmayan konumlarda saklayan bir veri yapısı olan `QLinkedList<T>`'i de sunar. Vektörlerden farklı olarak, bağlı listeler(linked lists) rastgele erişimi desteklemezler, fakat "sabit zaman(constant time)" eklemeleri ve silmeleri sağlarlar.

Şekil 10.2

Bağlı listeler `[]` operatörü sağlamazlar, bu nedenle yineleyiciler onların öğelerini çapraz geçiş yaparak kullanmalıdırlar. Yineleyiciler, öğelerin konumlarını belirlemede de kullanılırlar. Örneğin, sıradaki kod, "Tote Hosen" karakter katarını, "Clash" ve "Ramones"in arasına ekler:

```
QLinkedList<QString> list;
list.append("Clash");
list.append("Ramones");

QLinkedList<QString>::iterator i = list.find("Ramones");
list.insert(i, "Tote Hosen");
```

Yineleyicilere bu kısımda daha sonra detaylıca bakacağız.

QList<T> ardışık konteyneri, QVector<T> ve QLinkedList<T>'in en önemli faydalarını tek bir sınıfta birleştiren bir dizi listesidir(array-list). Rastgele erişimi destekler ve arabirimi QVector'lerin ki gibi indeks temellidir. Bir QList<T>'in sonundaki bir öğeyi silmek ya da bir QList<T>'in sonuna bir öğe eklemek çok hızlıdır. Ortaya ekleme ise, bin kadar öğesi olan listeler için yine çok hızlıdır.

Qt'un API'ında da sıkça kullanılan QStringList sınıfı, QList<QString>'in bir altsınıfıdır. Temel sınıfından miras aldığı fonksiyonlara ek olarak, sınıfı, karakter katarı işlemede daha becerikli yapan ekstra fonksiyonlar sağlar. QStringList ile ayrıntılı olarak bu bölümün son kısmında ilgileneceğiz.

QStack<T> ve QQueue<T>, uygunluk altsınıflarının iki örnekleridirler. QStack<T>, push(), pop() ve top() fonksiyonlarını sağlayan bir vektördür. QQueue<T> ise, enqueue(), dequeue() ve head() fonksiyonlarını sağlayan bir listedir.

Şimdiye kadar gördüğümüz tüm konteyner sınıfları için, değer tipi T; int veya double gibi basit bir tip, bir işaretçi tipi, ya da bir varsayılan kurucuya (argüman almayan bir kurucu), bir kopyalama kurucusuna ve bir atama operatörüne sahip olan bir sınıf olabilir. QByteArray, QDataTime, QRegExp, QString ve QVariant sınıflarına da bu hak verilmiştir. Fakat QObject'ten türetilmiş sınıflara bu hak tanınmaz, çünkü onlar bir kopyalama kurucusundan ve bir atama operatöründen yoksundurlar. Fakat biz, nesnelerin kendilerini saklamak yerine, QObject tiplerine işaret eden işaretçileri sakladığımız için, bu, pratikte bir sorun oluşturmaz.

Değer tipi T bir konteyner de olabilir, böyle bir durumda birbirini izleyen köşeli parantezleri boşluklarla ayırmayı unutmamamız gerekir; aksi halde, derleyici onun bir >> operatörü olduğunu sanarak tıkanacaktır. Örneğin:

```
QList<QVector<double> > list;
```

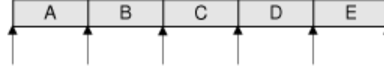
Bahsedilmiş olan tiplere ek olarak, bir konteynerin değer tipi, daha evvel tanımlanmış kriterleri karşılayan herhangi bir özel sınıf da olabilir. İşte, böyle bir sınıfın örneği:

```
class Movie
{
public:
    Movie(const QString &title = "", int duration = 0);

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    void setDuration(int duration) { myDuration = duration; }
    QString duration() const { return myDuration; }

private:
    QString myTitle;
    int myDuration;
};
```

Sınıf, hiç argüman gerektirmeyen (buna rağmen iki adet alabilen) bir kurucuya sahiptir. Her biri dolaylı olarak C++ tarafından sağlanan, bir kopyalama kurucu fonksiyonu ve bir atama operatörüne de sahiptir. Bu sınıf için, üye-üye kopyalama(member-by-member copy) yeterlidir, böylece kendi kopyalama kurucu fonksiyonumuzu ve atama operatörümüzü gerçekleştirmemiz gerekmez.

Qt, bir konteynerde saklanan öğeleri ele almak için yineleyicilerin iki kategorisini sağlar: Java-tarzı yineleyiciler ve STL-tarzı yineleyiciler. Java-tarzı yineleyicilerin kullanımı daha kolayken, STL-tarzı yineleyiciler, Qt'un ve STL'in genel algoritmalarıyla kombine edilebilirler ve daha güçlüdürler.

Her bir konteyner sınıfı için, iki tip Java-tarzı yineleyici vardır: bir sadece-oku(read-only) ve bir de oku-yaz(read-write) yineleyici. Geçerli konumları Şekil 10.3'te gösteriliyor. Sadece-oku yineleyici sınıfları, `QVectorIterator<T>`, `QLinkedListIterator<T>` ve `QListIterator<T>`'dır. Bunlara karşılık gelen oku-yaz yineleyiciler isimlerinde `Mutable`'a sahiptirler (`QMutableVectorIterator<T>` gibi). Burada, `QList`'in yineleyicilerine yoğunlaşacağız.



Şekil 10.3

Java-tarzı yineleyicileri kullanırken aklınızda bulundurmanız gereken ilk şey, onların, öğelere direkt olarak işaret etmedikleridir. Onun yerine, ilk öğeden önce veya son öğeden sonra ya da bu iki öğenin arasında bir yerde yer alırlar. Tipik bir yineleme(iteration) döngüsü şöyle görünür:

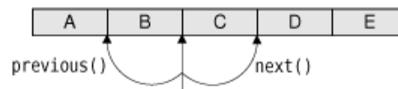
```
QList<double> list;
...
QListIterator<double> i(list);
while (i.hasNext()) {
    do_something(i.next());
}
```

Yineleyici, ele alınacak konteyner ile ilk kullanıma hazırlanır. Bu noktada, yineleyici tam olarak ilk öğeden önce yer alır. Eğer yineleyicinin sağında bir öğe varsa, `hasNext()` çağrısı `true` döndürür. `next()` fonksiyonu yineleyicinin sağındaki öğeyi döndürür ve yineleyiciyi bir sonraki geçerli konuma ilerletir.

Geriye doğru yineleme ise, yineleyiciyi son öğeden sonrasına konumlandırmak için ilk önce `toBack()`'i çağırmak zorunda olmamız dışında benzerdir.

```
QListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    do_something(i.previous());
}
```

`hasPrevious()` fonksiyonu eğer öğenin sol tarafında öğe varsa, `true` döndürür; `previous()`, yineleyicinin solundaki öğeyi döndürür ve yineleyiciyi bir adım geriye taşır. `next()` ve `previous()` yineleyicilerini tasavvur etmenin bir diğer yolu da, üzerlerinden atladıkları öğeleri döndürdüklerini düşünmektir, tıpkı Şekil 10.4'te gösterildiği gibi.



Şekil 10.4

Mutable yineleyiciler, yineleme sırasında öğeleri düzenlemeyi ve silmeyi ve öğeler eklemeyi sağlayan fonksiyonlar sağlarlar. Sıradaki döngü, tüm negatif sayıları bir listeden siler:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    if (i.next() < 0.0)
        i.remove();
}
```

`remove()` fonksiyonu, daima üzerinden atladığı son öğe üstünde işlem yapar. Ayrıca, geriye doğru yinelemede de çalışır:

```
QMutableListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    if (i.previous() < 0.0)
        i.remove();
}
```

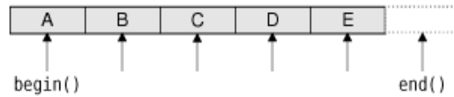
Benzer şekilde, mutable Java-tarzı yineleyiciler, üzerinden atladığı son öğeyi düzenleyen bir `setValue()` fonksiyonu sağlar. İşte, negatif sayıları, onların mutlak değerleriyle değiştiren bir örnek:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    int val = i.next();
    if (val < 0.0)
        i.setValue(-val);
}
```

Aynı zamanda, `insert()`'i çağırarak, geçerli yineleyici konumuna bir öğe eklemek de mümkündür. Ekleme sonrasında, yineleyici yeni öğe ile sonraki öğe arasına ilerletilir.

Java-tarzı yineleyicilere ek olarak, her ardışık konteyner sınıfı `C<T>`, iki STL-tarzı yineleyici tipine sahiptir: `C<T>::iterator` ve `C<T>::const_iterator`. İkisi arasındaki fark, `const_iterator`'ın veriyi düzenlemeye izin vermemesidir.

Bir konteynerin `begin()` fonksiyonu, konteynerin ilk öğesine sevk eden bir STL-tarzı yineleyici döndürürken (`list[0]` gibi), `end()` fonksiyonu, sondan bir sonraki öğeye sevk eden bir yineleyici döndürür (boyutu beş olan bir liste için `list[5]`'tir). Şekil 10.5, STL-tarzı yineleyiciler için geçerli konumları gösteriyor. Eğer bir konteyner boş ise, `begin()`, `end()`'e eşittir. Bu, konteynerde öğe olup olmadığını görmek için kullanılabilir, gerçi `isEmpty()` çağrısı bu amaç için genellikle daha uygundur.



Şekil 10.5

STL-tarzı yineleyicilerin sözdiziminde, C++ işaretçilerin ki örnek alınmıştır. Sonraki ya da önceki öğeye geçmek için `++` ve `--` operatörlerini, geçerli öğeye erişmek için birli(unary) `*` operatörünü kullanabiliriz. `QVector<T>` için, `iterator` ve `const_iterator` tipleri, sırf `T *` ve `const T *` için tanımlanmış tiplerdir. (Bu mümkündür, çünkü `QVector<T>`, öğelerini ardışık bellek konumlarında saklar.)

Sıradaki örnek, bir `QList<double>` içindeki her bir değeri, bu değerlerin mutlak değerleri ile değiştirir:

```
QList<double>::iterator i = list.begin();
while (i != list.end()) {
```

```
    *i = qAbs(*i);
    ++i;
}
```

Az miktarda Qt fonksiyonu bir konteyner döndürür. Eğer STL-tarzı yineleyici kullanan bir fonksiyonun dönen değeri üzerinde yineleme yapmak istiyorsak, konteynerin bir kopyasını almalıyız ve bu kopya üzerinde yineleme yapmalıyız. Örneğin, sıradaki kod, `QSplitter::sizes()` tarafından döndürülen `QList<int>` üzerinde yineleme yapmak için doğru yoldur:

```
QList<int> list = splitter->sizes();
QList<int>::const_iterator i = list.begin();
while (i != list.end()) {
    do_something(*i);
    ++i;
}
```

Şu kod ise yanlıştır:

```
// WRONG
QList<int>::const_iterator i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
    do_something(*i);
    ++i;
}
```

Bunun nedeni, `QSplitter::sizes()` her çağrıldığında, yeni bir `QList<int>`'i değer olarak döndürmesidir. Eğer dönen değeri saklamazsak, C++ onu yinelemeye başlamadan önce otomatik olarak yok eder, bizi işe yaramaz bir yineleyici ile karşı karşıya bırakır. Daha da kötüsü, döngünün çalıştığı her zaman, `QSplitter::sizes()`, listenin yeni bir kopyasını üretmek zorundadır, çünkü `splitter->sizes().end()` çağrılır. Özet olarak: STL-tarzı yineleyicileri kullanırken, her zaman, değer olarak döndürülen bir konteynerin bir kopyası üzerinde yineleme yapın.

Sadece-oku Java-tarzı yineleyicilerle, bir kopya almamıza gerek yoktur. Yineleyici bizim için, perde arkasında bir kopya alır ve bu, her zaman fonksiyondan ilk dönen veri üzerinde yineleme yapmamızı sağlar. Örneğin:

```
QListIterator<int> i(splitter->sizes());
while (i.hasNext()) {
    do_something(i.next());
}
```

Bir konteyneri kopyalamak pahalıya mal olabilecek gibi görünür, fakat -örtülü paylaşım sayesinde- öyle değildir. Bunun anlamı, bir Qt konteynerini kopyalamak, tek bir işaretçiyi kopyalamak kadar hızlıdır. Yalnız, eğer kopyalardan birinin verisi değiştirilirse gerçekten kopyalanır ve tüm işlemler otomatik olarak perde arkasında yapılır.

Örtülü paylaşımın güzelliği, programcı müdahalesine gerek duymaksızın basitçe çalışan bir optimizasyon olmasıdır. Örtülü paylaşım aynı zamanda, nesneler değer olarak döndürüldüğü yerde temiz bir programlamayı teşvik eder. Şu fonksiyonu dikkat edin:

```
QVector<double> sineTable()
{
    QVector<double> vect(360);
    for (int i = 0; i < 360; ++i)
```

```
        vect[i] = std::sin(i / (2 * M_PI));
    return vect;
}
```

Fonksiyona çağrı aşağıdaki gibidir:

```
QVector<double> table = sineTable();
```

STL, bizi, fonksiyonun dönüş değeri bir değişkende saklandığında, kopyalamayı önlemek için, vektörü sabit olmayan bir referans olarak aktarmaya teşvik eder:

```
void sineTable(std::vector<double> &vect)
{
    vect.resize(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = std::sin(i / (2 * M_PI));
}
```

Çağrı, ondan sonra, yazması ve okuması daha can sıkıcı bir hal alır:

```
std::vector<double> table;
sineTable(table);
```

Qt, örtülü paylaşımı, tüm konteynerleri için ve `QByteArray`, `QBrush`, `QFont`, `QImage`, `QPixmap` ve `QString` de dâhil birçok sınıfı için kullanır. Bu, bu sınıfları değer olarak aktarmada çok verimli yapar.

Örtülü paylaşım, Qt'dan, bir verinin biz onu düzenlemek istemediğimiz takdirde kopyalanmayacağına dair bir garantidir.

Qt, ardışık bir konteyner içinde yineleme yapmak için son bir metot daha sağlar: `foreach` döngüsü.

```
QLinkedList<Movie> list;
...
foreach (Movie movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

`foreach` sözde anahtar kelimesi, standart `for` döngüsünce gerçekleştirilir. Döngünün her bir yinelenmesinde, yineleme değişkeni (`movie`) yeni bir öğeye yerleştirilir, konteynerdeki ilk öğeden başlanır ve sonrakine ilerlenir. `foreach` döngüsü, döngüye girildiğinde, otomatik olarak konteynerin bir kopyasını alır ve bu nedenle eğer konteyner yineleme sırasında düzenlenirse, döngü bundan etkilenmez.

`break` ve `continue` döngü ifadeleri desteklenir. Eğer gövde tek bir ifadeden oluşuyorsa, süslü parantezler gereksizdir. Tıpkı bir `for` ifadesi gibi, yineleme değişkeni döngü dışında tanımlanabilir:

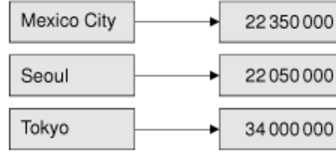
```
QLinkedList<Movie> list;
Movie movie;
...
foreach (movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
```

```
}
```

Birleşik Konteynerler

Birleşik konteynerler, aynı tip öğelerden isteğe bağlı sayıda kadarını tutar, bir anahtar ile indeksler. Qt, iki ana birleşik konteyner sınıfı sağlar: `QMap<K, T>` ve `QHash<K, T>`.

Bir `QMap<K, T>`, anahtar-değer çiftlerini artan anahtar sırası içinde saklayan bir veri yapısıdır (Şekil 10.6). Bu düzen, iyi bir arama ve ekleme performansı ve anahtar-sıra yinelemesi sağlar. Aslen, `QMap<K, T>` bir sıçrama listesi(skip-list) olarak gerçekleştirilmiştir.



Şekil 10.6

Bir haritaya(map) öğeler eklemenin basit bir yolu insert()'ü çağırmaktır:

```
QMap<QString, int> map;
map.insert("eins", 1);
map.insert("sieben", 7);
map.insert("dreiundzwanzig", 23);
```

Alternatif olarak, aşağıdaki gibi basitçe bir değeri, verilen bir anahtara atayabiliriz:

```
map["eins"] = 1;
map["sieben"] = 7;
map["dreiundzwanzig"] = 23;
```

`[]` operatörü, hem ekleme hem de erişim için kullanılabilir. Eğer `[]` operatörü, mutable bir harita içerisinde, var olmayan bir anahtardaki bir veriye erişmede kullanılırsa, verilen anahtar ve boş bir değer ile yeni bir öğe oluşturulacaktır. Kazara boş değerler oluşturmayı önlemek için, öğelere erişirken `value()` fonksiyonunu kullanabiliriz.

```
int val = map.value("dreiundzwanzig");
```

Eğer anahtar bulunmuyorsa, değer tipinin varsayılan kurucusu kullanılarak varsayılan bir değer döndürülür ve yeni bir değer oluşturulmaz. Temel ve işaretçi tipleri için, sıfır döndürülür. `value()`'ya ikinci bir argüman aktararak başka bir varsayılan değer belirtebiliriz, örneğin:

```
int seconds = map.value("delay", 30);
```

Bu, şuna denktir

```
int seconds = 30;
if (map.contains("delay"))
    seconds = map.value("delay");
```

Bir `QMap<K, T>`'ın `K` ve `T` veri tipleri, `int` ve `double` gibi temel veri tipleri, işaretçi tipleri ya da bir varsayılan kurucuya, bir kopyalama kurucu fonksiyonuna ve bir atama operatörüne sahip sınıflar olabilir. Ek

olarak, K tipi, QMap<K, T>,“<” operatörünü öğeleri artan anahtar sırası içinde saklamada kullandığı için, bir operator<() sağlamalıdır.

QMap<K, T>, bir çift uygunluk fonksiyonuna sahiptir. keys() ve values(), özellikle küçük veri setleriyle ilgilenirken çok kullanışlıdırlar. Haritanın anahtar ve değerlerinin QList’lerini döndürürler.

Haritalar normalde tek değerlidirler(singled-valued): Eğer var olan bir anahtara yeni bir değer atanırsa, yeni değer eski değerin yerine konur; bunu sağlamak için de, hiçbir öğe başka bir öğe ile aynı anahtarı paylaşmaz. Bunun yanında, insertMulti() fonksiyonunu ya da QMultiMap<K, T> uygunluk altsınıfını kullanarak, çoklu anahtar-değer çiftlerine sahip olmak mümkündür. QMap<K, T>, verilen anahtarın tüm değerlerinin bir QList’ini döndüren aşırı yüklenmiş bir values(const K &) fonksiyonuna da sahiptir. Örneğin:

```
QMultiMap<int, QString> multiMap;
multiMap.insert(1, "one");
multiMap.insert(1, "eins");
multiMap.insert(1, "uno");

QList<QString> vals = multiMap.values(1);
```

Bir QHash<K, T>, anahtar-değer çiftlerini bir anahtarlı tablo(hash table) içinde saklayan bir veri yapısıdır. Arayüzü hemen hemen QMap<K, T>’ın ki ile aynıdır, fakat K şablon tipi için farklı gerekliliklere sahiptir ve genellikle QMap<K, T>’ın başarabildiğinden daha hızlı aramalar sağlar. Bir başka farklılık ise, QHash<K, T> sıralı değildir.

Bir konteyner içinde saklanan her veri tipi için istenen standart gerekliliklere ek olarak, bir QHash<K, T>’ın K tipi, bir operator==() sağlamalıdır ve bir anahtar için bir hash değeri döndüren global bir qHash() tarafından destekleniyor olmalıdır. Qt, hâlihazırda, tamsayı tipleri, işaretçi tipleri, QChar, QString ve QByteArray için qHash() fonksiyonları sağlamaktadır.

reserve()’ü, hash içinde saklanması beklenen öğelerin sayısını belirtmek için, squeeze()’i de geçerli öğelerin sayısına bağlı olarak anahtarlı tabloyu küçültmek için çağırarak, performans ince-ayarı yapmak mümkündür. Yaygın bir kullanım şekli, reserve()’ü -beklediğimiz- maksimum öğe sayısı ile çağırmak, sonra verileri eklemek ve son olarak, eğer beklenenden daha az öğe varsa, hafıza kullanımı azaltmak için squeeze()’i çağırmaktır.

Anahtarlı tablolar normalde tek değerlidirler, fakat aynı anahtara çoklu değerler atama, insertMulti() fonksiyonunu ya da QMultiHash<K, T> uygunluk altsınıfını kullanmak suretiyle yapılabilmektedir.

QHash<K, T>’ten başka, Qt, önbellek(cache) nesnelerini bir anahtar ile birleştirmede kullanılabilen bir QCache<K, T> sınıfı ve sadece anahtarları saklayan bir QSet<K> konteyneri de sağlar. Aslen, ikisi de QHash<K, T>’e dayanırlar ve ikisi de K tipi için QHash<K, T> ile aynı gerekliliklere sahiptirler.

Birleşik bir konteyner içinde saklanan tüm anahtar-değer çiftleri boyunca yineleme yapmanın en kolay yolu Java-tarzı bir yineleyici kullanmaktır. Çünkü yineleyiciler hem bir anahtara hem de bir değere erişmelidirler, ve Java-tarzı yineleyiciler, birleşik konteynerler için, onların ardışık emsallerine göre, bir parça farklı çalışırlar. Ana farklılık, next() ve previous() fonksiyonlarının, bir anahtar-değer çiftini temsil eden bir nesne ya da tercihen sadece bir değer döndürmesidir. Anahtar ve değer bileşenlerine, bu nesneden, key() ve value() kullanılarak erişilebilir. Örneğin:

```
QMap<QString, int> map;
...
int sum = 0;
QMapIterator<QString, int> i(map);
while (i.hasNext())
    sum += i.next().value();
```

Eğer hem anahtara hem de değere erişmemiz gerekiyorsa, açıkça `next()` ya da `previous()`'ın dönüş değerlerini yoksayabilir ve yineleyicinin `key()` ve `value()` fonksiyonlarını kullanabiliriz.

```
QMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() > largestValue) {
        largestKey = i.key();
        largestValue = i.value();
    }
}
```

Mutable yineleyiciler, geçerli öğe ile ilgili değeri düzenleyen `setValue()` fonksiyonuna sahiptir:

```
QMutableMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() < 0.0)
        i.setValue(-i.value());
}
```

STL-tarzı yineleyiciler de `key()` ve `value()` fonksiyonlarını sağlarlar. Sabit olmayan(non-const) yineleyici tipleriyle, `value()` sabit olmayan bir referans döndürür ve bize, yineleme yaparken değeri değiştirme imkânı verir. Bu yineleyiciler "STL-tarzı" olarak adlandırılsalar da, `std::pair<K, T>`'ı esas alan `std::map<K, T>` yineleyicilerinden önemli derecede ayrılırlar.

`foreach` döngüsü, birleşik konteynerler üzerinde de çalışır, fakat anahtar-değer çiftlerinin sadece değer bileşeni üzerinde. Eğer hem anahtar hem de değer bileşenlerine ihtiyacımız varsa, aşağıdaki gibi, iç içe `foreach` döngüleri içinde `keys()` ve `values(const K &)` fonksiyonlarını kullanabiliriz:

```
QMultiMap<QString, int> map;
...
foreach (QString key, map.keys()) {
    foreach (int value, map.values(key)) {
        do_something(key, value);
    }
}
```

Genel Algoritmalar

`<QtAlgorithms>` başlığı, konteynerler üzerinde temel algoritmaları gerçekleştiren global şablon fonksiyonlarının bir setini bildirir. Bu fonksiyonların çoğu, STL-tarzı yineleyiciler üzerinde çalışırlar.

STL `<algorithm>` başlığı, genel algoritmaların daha eksiksiz bir setini sağlar. Bu algoritmalar, Qt konteynerleri üzerinde, STL konteynerleriymiş gibi kullanılabilirler. STL gerçekleştirimleri tüm platformlarında mevcutsa, Qt denk bir algoritma sunmadığında, STL algoritmalarını kullanmayı engelleyecek bir neden muhtemelen yoktur. Burada, en önemli Qt algoritmalarını tanıtacağız.

`qFind()` algoritması, bir konteyner içinde belirli bir değeri arar. Bir "begin" ve bir "end" yineleyicisi alır ve eşleşen ilk öğeye işaret eden bir yineleyici, ya da hiç eşleşen yok ise, "end" döndürür. Aşağıdaki örnekte, `i`, `list.begin() + 1` olarak belirlenirken, `j`, `list.end()` olarak belirlenir:

```
QStringList list;
list << "Emma" << "Karl" << "James" << "Mariette";

QStringList::iterator i = qFind(list.begin(), list.end(), "Karl");
QStringList::iterator j = qFind(list.begin(), list.end(), "Petra");
```

`qBinaryFind()` algoritması, tıpkı `qFind()` gibi bir arama gerçekleştirir, ancak `qFind()` gibi doğrusal arama yapmaktansa öğelerin artan düzende sıralandığını varsayarak, hızlı ikili aramayı(fast binary searching) kullanır.

`qFill()` algoritması, bir konteyneri belirli bir değer ile doldurur:

```
QLinkedList<int> list(10);
qFill(list.begin(), list.end(), 1009);
```

Diğer yineleyici temelli algoritmalar gibi, `qFill()`'i farklı argümanlar kullanarak, konteynerin bir parçası üzerinde de kullanabiliriz. Aşağıdaki kod parçası, bir vektörün ilk beş öğesini 1009'a, son 5 öğesini de 2013'e ilklendiriyor:

```
QVector<int> vect(10);
qFill(vect.begin(), vect.begin() + 5, 1009);
qFill(vect.end() - 5, vect.end(), 2013);
```

`qCopy()` algoritması, değerleri bir konteynerden bir diğerine kopyalar:

```
QVector<int> vect(list.count());
qCopy(list.begin(), list.end(), vect.begin());
```

`qCopy()`, kaynağın menzili ile hedefin menzili çakışmadığı sürece, değerleri aynı konteyner içinde kopyalamakta da kullanılabilir. Sıradaki kod parçasında, onu, bir listenin ilk iki öğesini, listenin son iki öğesi üzerine yazmada kullanıyoruz:

```
qCopy(list.begin(), list.begin() + 2, list.end() - 2);
```

`qSort()` algoritması, konteynerin öğelerini artan düzende sıralar:

```
qSort(list.begin(), list.end());
```

`qSort()` varsayılan olarak, öğeleri karşılaştırmak için `<` operatörünü kullanır. Öğeleri azalan düzende sıralamak için, üçüncü argüman olarak `qGreater<T>`'ı (`T` yerine konteynerin değer tipini yazarak) aktarın:

```
qSort(list.begin(), list.end(), qGreater<int>());
```

Üçüncü parametreyi, özel sıralama kriteri tanımlamada da kullanabiliriz. Örneğin burada, `QString`'leri büyük/küçük harf duyarsız bir yöntemle karşılaştıran, bir karşılaştırma fonksiyonu:

```
bool insensitiveLessThan(const QString &str1, const QString &str2)
{
    return str1.toLower() < str2.toLower();
}
```

Öyleyse qSort() çağrısı şu hali alır:

```
QStringList list;
...
qSort(list.begin(), list.end(), insensitiveLessThan);
```

qStableSort() algoritması ise, karşılaştırılan öğeler arasında eşit görünenlerin sıralama sonrasında, sıralama öncesindeki gibi aynı sırada olmalarını garanti eder. Bu, eğer sıralama kriteri sadece değerin parçalarını hesaba katıyorsa ve sonuçlar kullanıcı tarafından görülüyorsa, kullanışlıdır. qStableSort()'u Bölüm 4'te, Spreadsheet uygulaması içinde sıralamayı gerçekleştirirken kullanmıştık.

qDeleteAll() algoritması, bir konteyner içinde saklanan her işaretçi için delete'i çağırır. qDeletAll()'u kullanmak, sadece, değer tipi bir işaretçi tipi olan konteynerler üzerinde mantıklıdır. Çağrıdan sonra, öğeler konteyner içinde asılı kalmış işaretçiler olarak varlıklarını sürdürürler ve bu nedenle normal olarak, konteyner üzerinde clear()'ı da çağırmamız gerekir. Örneğin:

```
qDeleteAll(list);
list.clear();
```

qSwap() algoritması, iki değişkenin değerini değiş tokuş eder. Örneğin:

```
int x1 = line.x1();
int x2 = line.x2();
if (x1 > x2)
    qSwap(x1, x2);
```

Son olarak, tüm diğer Qt başlıkları tarafından dâhil edilen <QtGlobal> başlığı, argümanının mutlak değerini döndüren qAbs() fonksiyonu ve iki değerin minimum ya da maksimumunu döndüren qMin() ve QMax() fonksiyonları da dâhil olmak üzere, birkaç kullanışlı tanım sağlar.

Karakter Katarları, Byte Dizileri ve Variantlar

QString, QByteArray ve QVariant, konteynerlerle benzer birçok şeye sahip olan ve bazı durumlarda konteynerlere alternatif olarak kullanılabilen üç sınıftır. Ayrıca, konteynerler gibi bu sınıflar da, bir bellek ve hız optimizasyonu olarak örtülü paylaşımı kullanırlar.

QString ile başlayacağız. Her GUI programı karakter katarlarını(string) kullanır. C++, iki çeşit karakter katarı sağlar: geleneksel C-tarzı karakter katarları ve std::string sınıfı. Bunlardan farklı olarak, QString, 16-bit Unicode değerleri tutar. Unicode, bir altküme olarak, ASCII ve Latin-1'i de kapsar. Fakat QString 16-bit olduğu için, dünya dillerinin çoğunu yazabilmek için diğer binlerce karakteri de ifade edebilir.

QString kullandığımızda, yeterli belleğin ayrılması ya da verinin '\0' ile sonlandırılmasını sağlamak gibi detaylar hakkında endişelenmemiz gerekmez. Kavramsal olarak, QString'ler, QChar'ların bir vektörü olarak düşünülebilir. Bir QString, '\0' karakterleri içerebilir. length() fonksiyonu tüm karakter katarının boyutunu -içerdiği '\0' karakterleri de dâhil olmak üzere- döndürür.

QString, iki karakter katarını birleştirmek için bir ikili + operatörü, bir karakter katarını bir diğerine iliştirmek için bir += operatörü sağlar. Çünkü QString otomatik olarak, karakter katarı verisinin sonunda belek ayırır ve karakter katarını, karakterleri çok hızlıca ve aralıksız olarak ona iliştirerek oluşturur. İşte, + ve += operatörlerini birleştiren bir örnek:

```
QString str = "User: ";
```

```
str += userName + "\n";
```

Ayrıca, += operatörü ile aynı işi yapan bir `QString::append()` fonksiyonu da vardır:

```
str = "User: ";
str.append(userName);
str.append("\n");
```

Karakter katarlarını birleştirmenin tamamen farklı bir yolu `QString`'in `sprintf()` fonksiyonunu kullanmaktır:

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

Bu fonksiyon, C++ kütüphanesindeki `sprintf()` fonksiyonu ile aynı format belirteçlerini destekler. Önceki örnekte, `str`, "perfect competition 100.0%" değerini alır.

Diğer karakter katarlarından ya da sayılardan bir karakter katarı oluşturmanın bir başka yolu ise, `arg()` fonksiyonunu kullanmaktır:

```
str = QString("%1 %2 (%3s-%4s)")
      .arg("permissive").arg("society").arg(1950).arg(1970);
```

Bu örnekte, "permissive", "%1"in yerini, "society", "%2"nin yerini, "1950" "%3"ün yerini, "1970" de "%4"ün yerini alır. Sonuç, "permissive society (1950s-1970s)"tir. Çeşitli veri tiplerini işlemek için, `arg()` fonksiyonunun aşırı yüklenmiş versiyonları da vardır. `arg()`'ın aşırı yüklenmiş bazı versiyonları alan genişliğini, sayısal tabanı ya da kayan nokta hassasiyetini kontrol etmek için ekstra parametrelere sahiptirler. Genelde `arg()`, `sprintf()`'ten daha iyi bir çözümdür, çünkü tip güvenliği sağlar, Unicode'u tam destekler ve çevirmenlere, "%n" parametrelerini yeniden düzenleme imkânı verir.

`QString`, `QString::number()` statik fonksiyonunu kullanarak, sayılar karakter katarlarına dönüştürebilir:

```
str = QString::number(59.6);
```

Ya da `setNum()` fonksiyonunu kullanarak:

```
str.setNum(59.6);
```

Bir karakter katarından bir sayıya dönüşüm ise, `toInt()`, `toLongLong()`, `toDouble()`, vs kullanılarak gerçekleştirilir. Örneğin:

```
bool ok;
double d = str.toDouble(&ok);
```

Bu fonksiyonlar, isteğe bağlı olarak, bir `bool` değişkene işaret eden bir işaretçi alır ve dönüşümün başarısına bağlı olarak değişkeni `true` ya da `false` olarak ayarlarlar. Eğer dönüşüm başarısız olursa, bu fonksiyonlar sıfır döndürür.

Bir karakter katarına sahip olduğumuzda, sık sık onun parçalarını seçerek almak isteriz. `mid()` fonksiyonu, verilen konumdan (ilk argüman) başlayarak, verilen uzunluğa (ikinci argüman) kadar olan altkarakter katarını(substring) döndürür. Örneğin, aşağıdaki kod, konsola "pays" yazdırır:

```
QString str = "polluter pays principle";
```

```
qDebug() << str.mid(9, 4);
```

Eğer ikinci argümanı ihmal edersek, `mid()` verilen konumdan başlayarak, karakter katarının sonuna kadar olan altkarakter katarını döndürür. Örneğin, aşağıdaki kod, konsola "pays principle" yazdırır:

```
QString str = "polluter pays principle";
qDebug() << str.mid(9);
```

Benzer işi yapan `left()` ve `right()` fonksiyonları da vardır. Her ikisi de, karakterlerin sayısını(n) alır ve karakter katarının ilk ya da son n sayıda karakterini döndürür. Örneğin, aşağıdaki kod, konsola "pollutter principle" yazdırır:

```
QString str = "polluter pays principle";
qDebug() << str.left(8) << " " << str.right(9);
```

Eğer bir karakter katarının, belirli bir karakteri, altkarakter katarını ya da düzenli ifadeyi(regular expression) içerip içermediğini öğrenmek istersek, `QString`'in `indexOf()` fonksiyonlarından birini kullanabiliriz:

```
QString str = "the middle bit";
int i = str.indexOf("middle");
```

Burada `i`, 4 olarak belirlenir. Başarısızlık durumunda, `indexOf()` -1 döndürür ve isteğe bağlı olarak, başlama konumu ve büyük/küçük harf duyarlılık bayrağı(case-sensitivity flag) alabilir.

Eğer sadece, bir karakter katarının bir şeyle başlayıp başlamadığını ya da bitip bitmediğini kontrol etmek istiyorsak, `startsWith()` ve `endsWith()` fonksiyonlarını kullanabiliriz:

```
if (url.startsWith("http:") && url.endsWith(".png"))
    ...
```

Önceki kod, aşağıdakinden hem basit hem de daha hızlıdır:

```
if (url.left(5) == "http:" && url.right(4) == ".png")
    ...
```

`==` operatörü ile yapılan karakter katarı karşılaştırmaları büyük/küçük harf duyarlıdır. Eğer kullanıcı tarafından görülebilir karakter katarlarını karşılaştırıyorsak, `localeAwareCompare()` genellikle doğru seçimdir ve eğer karşılaştırmaları büyük/küçük harf duyarlı yapmak istiyorsak, `toUpper()` ya da `toLower()` kullanabiliriz. Örneğin:

```
if (fileName.toLower() == "readme.txt")
    ...
```

Eğer bir karakter katarının belirli bir parçasını, başka bir karakter katarı ile değiştirmek istiyorsak, `replace()` fonksiyonunu kullanabiliriz:

```
QString str = "a cloudy day";
str.replace(2, 6, "sunny");
```

Sonuç "a sunny day" olacaktır. Kod, `remove()` ve `insert()` kullanılarak yeniden yazılabilir:

```
str.remove(2, 6);
str.insert(2, "sunny");
```

Önce, 2 konumundan başlayarak 6 karakter sileriz, karakter katarı “a day” halini alır, ve sonrada 2 konumuna “sunny”yi ekleriz.

`replace()` fonksiyonunun, birinci argümana eşit olan her şeyi, ikinci argüman ile değiştiren aşırı yüklenmiş versiyonları vardır. Örneğin burada, bir karakter katarı içindeki tüm “&”ların, “&” ile değiştirilişi gösteriliyor:

```
str.replace("&", "&amp;");
```

Çok sık görülen bir ihtiyaç da, bir karakter katarından, alfabe dışı karakterleri (boşluk, tab, yeni satır karakterleri gibi) çıkarmaktır. `QString`, bir karakter katarının başındaki ve sonundaki alfabe dışı karakterleri çıkaran bir fonksiyona sahiptir:

```
QString str = "   BOB \t THE \nDOG \n";
qDebug() << str.trimmed();
```

`str` karakter katarı şu şekilde tasvir edilebilir:

			B	O	B		\t		T	H	E			\n	D	O	G		\n

`trimmed()` tarafından döndürülen karakter katarı ise şöyledir:

B	O	B		\t		T	H	E			\n	D	O	G

Kullanıcı girdisi işlenirken, karakter katarının başındaki ve sonundaki alfabe dışı karakterleri çıkarmaya ek olarak, sık sık, karakter katarı içindeki bir ya da daha fazla alfabe dışı karakterler dizisini, birer boşluk ile değiştirmek de isteriz. Bu isteğimizi `simplified()` fonksiyonu yapar:

```
QString str = "   BOB \t THE \nDOG \n";
qDebug() << str.simplified();
```

`simplified()` tarafından döndürülen karakter katarı şöyledir:

B	O	B		T	H	E		D	O	G

Bir karakter katarı, `QString:.split()` kullanılarak, altkarakter katarlarının bir `QStringList`’ine ayrılabilir:

```
QString str = "polluter pays principle";
QStringList words = str.split(" ");
```

Önceki örnekte, “polluter pays principle” karakter katarını, üç altkarakter katarına ayırdık: “polluter”, “pays” ve “principle”.

Bir `QStingList` içindeki öğeler, `join()` kullanılarak, tek bir karakter katarında birleştirilebilirler. `join()`’e aktarılan argüman, karakter katarı çiftleri arasına eklenir.

```
words.sort();
str = words.join("\n");
```

Karakter katarlarıyla ilgilenirken, sık sık, bir karakter katarının boş olup olmadığını bilmeye ihtiyaç duyarız. Bu, `isEmpty()` kullanılarak ya da `length()` ile karakter katarının uzunluğunun 0 olup olmadığı kontrol edilerek yapılır.

const char * karakter katarlarından QString'e dönüşüm, çoğu durumda otomatiktir, örneğin:

```
str += " (1870)";
```

Burada, bir const char * 'ı bir QString'e, formalite olmaksızın ekledik. Bir const char *'ı bir QString'e açık bir şekilde dönüştürmek için, basitçe bir QString çevirim(cast) kullanabilir veya fromAscii() ya da fromLatin1() çağrılabilir.

Bir QString'i bir const char *'a dönüştürmek içinse toAscii() ya da toLatin1() kullanılır. Bu fonksiyonlar, QByteArray::data() ya da QByteArray::constData() kullanılarak bir const char *'a dönüştürülebilen, bir QByteArray döndürürler.

```
printf("User: %s\n", str.toAscii().data());
```

Kolaylık olsun diye, Qt, toAscii().constData() ikilisiyle aynı işi yapan qPrintable() makrosunu sağlar:

```
printf("User: %s\n", qPrintable(str));
```

Bir QByteArray üzerinde data() ya da constData()'yı çağırdığımızda, QByteArray nesnesi tarafından sahip olunan bir karakter katarı döndürür. Bunun anlamı, bellek sızıntısı hakkında endişelenmememiz gerektiğidir; Qt belleğin iadesini bizim için isteyecektir. Bir taraftan da, işaretçiyi uzun süre kullanmama konusunda dikkatli olmalıyız. Eğer QByteArray bir değişkende saklanmıyorsa, ifade sonunda otomatik olarak silinecektir.

QByteArray sınıfı, QString ile çok benzer bir API'a sahiptir. left(), right(), mid(), toLower(), toUpper(), trimmed() ve simplified() gibi fonksiyonlar, QString sınıfındaki emsalleri ile aynı anlamda, QByteArray sınıfında da vardır. QByteArray, ham ikili verileri ve 8-bit ile kodlanmış metinleri/karakter katarlarını saklamak için oldukça kullanışlıdır. Genelde, metinleri saklamak için, QByteArray yerine QString kullanmayı öneririz, çünkü QString Unicode destekler.

Kolaylık olması için, QByteArray otomatik olarak, sondan bir sonraki karakterin '\0' olmasını sağlar, böylece bir QByteArray'in bir fonksiyona bir const char * alarak aktarılması kolaylaşır. QByteArray de gömülü '\0' karakterlerini destekler ve bize, isteğe bağlı ikili veriler saklamada kullanma imkânı verir.

Bazı durumlarda, verilerin farklı tiplerini aynı değişken içerisinde saklamamız gerekebilir. Bir yaklaşım, veriyi bir QByteArray ya da bir QString olarak kodlamaktır. Örneğin bir karakter katarı, bir metinsel değeri ya da bir nümerik değeri, karakter katarı formunda tutabilir. Bu yaklaşımlar tam bir esneklik verir, fakat bazı C++ yararlarını ortadan kaldırır. Qt, farklı tipleri tutabilen değişkenleri işlemenin daha iyi bir yolunu sağlar: QVariant.

QVariant sınıfı, QBrush, QColor, QCursor, QDateTime, QFont, QKeySequence, QPalette, QPen, QPixmap, QPoint, QRect, QRegion, QSize ve QString dâhil olmak üzere birçok Qt tipinin değerlerinin tutabildiği gibi, double ve int gibi temel C++ nümerik tiplerini de tutabilir. QVariant sınıfı, konteynerleri de tutabilir: QMap<QString, QVariant>, QStringList ve QList<QVariant>.

Öğe görüntüleme sınıfları, veritabanı modülleri ve QSettings, yaygın olarak variantları kullanırlar ve bu bize, öğe verisini, veritabanı verisini ve kullanıcı seçeneklerini her QVariant uyumlu tip için okuma ve yazma imkânı verir. Bunun bir örneğini Bölüm 3'te görmüştük.

Konteyner tiplerinin iç içe değerleri ile QVariant kullanarak, isteğe bağlı karmaşık veri yapıları oluşturmak mümkündür:

```
QMap<QString, QVariant> pearMap;
pearMap["Standard"] = 1.95;
pearMap["Organic"] = 2.25;

QMap<QString, QVariant> fruitMap;
fruitMap["Orange"] = 2.10;
fruitMap["Pineapple"] = 3.85;
fruitMap["Pear"] = pearMap;
```

Burada, karakter katarı anahtarları (ürün adı) ve kayan noktalı sayılar (fiyatlar) ya da haritalar olabilen değerler ile bir harita oluşturduk. Üst seviye harita üç anahtar içerir: "Orange", "Pear" ve "Pineapple". "Pear" anahtarı ile birleştirilen değer bir haritadır ve iki anahtar içerir ("Standart" ve "Organic"). Variant değerler taşıyan bir harita üzerinde yineleme yaparken, tipi kontrol etmek için type() fonksiyonunu kullanmamız gerekir, böylece ona uygun şekilde davranabiliriz.

Bunun gibi veri yapıları oluşturmak, veriyi istediğimiz her yolla organize edebildiğimiz için, çok çekici olabilir. Fakat QVariant'ın bu kolaylığı, okunurluk ve verimlilik açısından zorlukları da beraberinde getirir. Bir kural olarak, genellikle verilerimizi saklamayı, eğer mümkünse uygun bir C++ sınıfı tanımlayarak gerçekleştirmek daha uygundur.

QVariant, Qt'un meta-object sistemi tarafından kullanılır ve bu nedenle de QtCore modülünün parçasıdır. Bununla beraber, QtGui modülünü bağladığımızda, QVariant, GUI'ye ilişkin tipleri de saklayabilir (QColor, QFont, QIcon, QImage ve QPixmap gibi):

```
QIcon icon("open.png");
QVariant variant = icon;
```

Bir QVariant'tan, GUI'ye ilişkin bir tipin değerine erişmek için, QVariant::value<T>() şablon üye fonksiyonu, aşağıdaki gibi, kullanılabilir:

```
QIcon icon = variant.value<QIcon>();
```

value<T>() fonksiyonu aynı zamanda, GUI'ye ilişkin olmayan veri tipleri ile QVariant arasında dönüşüm gerçekleştirmek için de kullanılır, fakat pratikte, biz zaten GUI'ye ilişkin olmayan tipler için to...() dönüşüm fonksiyonlarını kullanmaktayız.

QVariant ayrıca, özel veri tiplerini saklamak için de kullanılır, ancak onların bir varsayılan kurucu fonksiyon ve bir kopyalama kurucu fonksiyonu sağladıklarını varsayarak. Bunun çalışması için önce özel tipimizi, Q_DECLARE_METATYPE() makrosunu kullanarak kayda geçirmeliyiz (genelde bir başlık dosyası içinde sınıf tanımının altında):

```
Q_DECLARE_METATYPE(BusinessCard)
```

Bu bize, aşağıdaki gibi kod yazma olanağı verir:

```
BusinessCard businessCard;
QVariant variant = QVariant::fromValue(businessCard);
...
if (variant.canConvert<BusinessCard>()) {
    BusinessCard card = variant.value<BusinessCard>();
    ...
}
```

Derleyici kısıtlaması nedeniyle, bu şablon üye fonksiyonları MSVC 6 için kullanılır değildir. Eğer bu derleyiciyi kullanmanız gerekiyorsa, `qVariantFromValue()`, `qVariantValue<T>()` ve `qVariantCanConvert<T>()` global fonksiyonlarını kullanabilirsiniz.

Eğer özel veri tipi, bir `QDataStream`'dan yazma ve okuma için `<<` ve `>>` operatörlerine sahipse, onları `qRegisterMetaTypeStreamOperators<T>()` kullanarak kayda geçirebiliriz. Bu, özel veri tiplerinin tercihlerini, `QSettings` kullanarak diğer şeylerin arasında saklamayı mümkün kılar. Örneğin:

```
qRegisterMetaTypeStreamOperators<BusinessCard>("BusinessCard");
```

BÖLÜM 11: GİRDİ/ÇIKTI

Bir dosyadan okuma ya da bir dosyaya yazma ihtiyacı, hemem hemen her uygulama için geçerlidir. Qt, QIODevice sayesinde, I/O(Input/Output yani Girdi/Çıktı) için mükemmel bir destek sağlar. Qt, aşağıdaki QIODevice altsınıflarını içerir:

QFile	Yerel dosya sisteminde ve gömülü kaynaklar içindeki dosyalara erişir
QTemporaryFile	Yerel dosya sisteminde geçici dosyalar oluşturur ve geçici dosyalara erişir
QBuffer	Bir QByteArray'e veri yazar ya da bir QByteArray'den veri okur
QProcess	Harici programlar çalıştırır ve süreçler arası iletişimi işler
QTcpSocket	Bir veri akışını, TCP kullanarak ağ üzerinden transfer eder
QUdpSocket	Ağ üzerinden UDP veri iletileri(datagrams) gönderir ya da alır
QSslSocket	Şifrelenmiş bir veri akışını, ağ üzerinden SSL/TSL kullanarak transfer eder

QProcess, QTcpSocket, QUdpSocket ve QSslSocket, ardışık(sequential) aygıtlardır(device), bu, veriye sadece bir kere erişilebileceği anlamına gelir; ilk byte'tan başlanır ve seri bir şekilde son byte'a ilerlenir. QFile, QTemporaryFile ve QBuffer, rastgele erişimli(random access) aygıtlardır, böylece byte'lar, herhangi bir konumdan, istenilen defa erişilebilirdirler; dosya işaretçisini yeniden konumlandırmak için QIODevice::seek() fonksiyonunu sağlarlar.

Aygıt sınıflarına ek olarak, Qt, herhangi bir I/O aygıtından okumada ve herhangi bir I/O aygıtına yazmada kullanabildiğimiz, iki tane yüksek seviyeli akış sınıfı daha sağlar: ikili veriler için QDataStream, metinler için QTextStream. Bu sınıflar, byte düzenleme ve metin kodlama gibi meselelerle de ilgilenirler ve bu, farklı platformlarda ya da farklı ülkelerde çalışan Qt uygulamalarının, diğerlerinin dosyalarını okuyabilmelerini ve onlara yazabilmelerini sağlar. Bu, Qt'un I/O sınıflarını, emsal Standart C++ sınıflarından (biraz önce bahsedilen meselelerin çözümünü uygulama programcısına yükleyen) daha kullanışlı yapar.

QFile, ayrı ayrı dosyalara erişimi kolaylaştırır, tabi eğer dosya sistemi içinde ya da uygulamanın yürütülebilir dosyası içinde gömülülerse. Üzerinde çalışılan dosya setlerinin kimliğini saptaması gereken uygulamalar için, Qt, dizinleri işleyebilen ve içerisindeki dosyalar hakkında bilgi temin edebilen QDir ve QFileInfo sınıflarını sağlar.

QProcess sınıfı bize, harici programlar başlatma ve onlarla, onların standart girdi, çıktı ve hata kanalları (cin, cout ve cerr) sayesinde haberleşme imkânı verir. Harici programın kullanacağı ortam değişkenlerini ve çalışma dizinini ayarlayabiliriz.

İkili Veri Okuma ve Yazma

Qt ile ikili veri yüklemenin ve kaydetmenin en basit yollu; bir QFile somutlaştırmak, dosyayı açmak ve ona bir QDataStream nesnesi sayesinde erişmektir. QDataStream, int ve double gibi temel C++ tiplerini ve birçok Qt veri tipini destekleyen, platformdan bağımsız bir depolama formatı sağlar.

İşte, bir QImage, bir QMap<QString, QColor> ve bir tamsayının(int), facts.dat adlı bir dosyada saklanması:

```
QImage image("philip.png");

QMap<QString, QColor> map;
map.insert("red", Qt::red);
map.insert("green", Qt::green);
map.insert("blue", Qt::blue);

QFile file("facts.dat");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);

out << quint32(0x12345678) << image << map;
```

Eğer dosyayı açamazsak, kullanıcıyı bilgilendirir ve döneriz(`return`). `qPrintable()` makrosu, bir `QString` için bir `const char *` döndürür. (Bir başka yaklaşım da, aşırı yüklenmiş << operatörüne sahip `<iostream>` için bir `std::string` döndüren `QString::toStdString()` kullanmaktır.)

Eğer dosya başarıyla açılmışsa, bir `QDataStream` oluştururuz ve versiyon numarasını ayarlarız. Qt 4.3'te, en kapsamlı format, versiyon 9'dur. Ya 9 sabitini elle kodlarız ya da `QDataStream::Qt_4_3` sembolik adı kullanırız.

`0x12345678` sayısının tüm platformlar üzerinde, bir işaretsiz(unsigned) 32-bit tamsayı olarak yazıldığını garantiye almak için, onu, kesin olarak 32-bit olduğunu garanti eden bir veri tipine (`quint32`) dönüştürürüz. Birlikte çalışabilirliği sağlamak için, `QDataStream` varsayılan olarak, düşük son haneliye(big-endian) ayarlanır; bu, `setByteOrder()` çağrılarak değiştirilebilir.

`QFile` değişkeni, kapsam(scope) dışına çıktığında otomatik olarak silindiği için, dosyayı açıkça kapatmamız gerekmez. Eğer verinin gerçekten yazıldığını doğrulamak istiyorsak, `flush()` fonksiyonunu çağırabiliriz ve döndürdüğü değeri kontrol edebiliriz (başarılı ise `true`).

Veri yansımalarını(data mirrors) okumak için yazdığımız kod:

```
quint32 n;
QImage image;
QMap<QString, QColor> map;

QFile file("facts.dat");
if (!file.open(QIODevice::ReadOnly)) {
    std::cerr << "Cannot open file for reading: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_3);

in >> n >> image >> map;
```

`QDataStream` versiyonu, okumak için kullandığımızın aynısıdır. Her zaman bu şekilde olmalıdır. Versiyonu belirterek, uygulamanın veriyi her zaman okuyabilmesini ve yazabilmesini garantiye almış oluyoruz.

QDataStream, veriyi, onu kesintisiz bir şekilde geri okuyabilmemizi sağlayan bir yolla saklar. QDataStream aynı zamanda, ham byte'ları okumada ve yazmada da kullanılabilir (readRawBytes() ve writeRawBytes() kullanarak).

Bir QDataStream'den okurken, hataları işlemek oldukça kolaydır. Akış(stream), QDataStream::Ok, QDataStream::ReadPastEnd ya da QDataStream::ReadCorruptData olabilen bir status() değerine sahiptir. Bir hata ortaya çıktığında, >> operatörü daima sıfır ya da boş değerler okur. Bunun anlamı, potansiyel hatalar hakkında endişelenmeksizin ve sonda, okumanın geçerliliğini hakkında bilgi edinmek için status() değerini kontrol etmeksizin, bir dosyayı okumaya kalkışmamamız gerektiğidir.

QDataStream çeşitli C++ ve Qt veri tiplerini işler. Ayrıca << ve >> operatörlerini aşırı yükleyerek, kendi özel tiplerimiz için destek de ekleyebiliriz. İşte, QDataStream ile kullanılabilen özel bir veri tipinin tanımı:

```
class Painting
{
public:
    Painting() { myYear = 0; }
    Painting(const QString &title, const QString &artist, int year) {
        myTitle = title;
        myArtist = artist;
        myYear = year;
    }

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    ...

private:
    QString myTitle;
    QString myArtist;
    int myYear;
};

QDataStream &operator<<(QDataStream &out, const Painting &painting);
QDataStream &operator>>(QDataStream &in, Painting &painting);
```

İşte, << operatörünün gerçekleştirilmesi:

```
QDataStream &operator<<(QDataStream &out, const Painting &painting)
{
    out << painting.title() << painting.artist()
        << quint32(painting.year());
    return out;
}
```

Bir Painting çıktısı için, sadece iki QString ve bir quint32 çıktısı alırız. Fonksiyonun sonunda akışı döndürürüz. Bu, bir çıktı akışı ile bir << operatörler zincirini kullanma imkânı veren, yaygın bir C++ deyimidir. Örneğin:

```
out << painting1 << painting2 << painting3;
```

operator>>()'ın gerçekleştirimi, operator<<()'ınkine benzerdir:

```
QDataStream &operator>>(QDataStream &in, Painting &painting)
{
    QString title;
```

```
    QString artist;
    quint32 year;

    in >> title >> artist >> year;
    painting = Painting(title, artist, year);
    return in;
}
```

Akış operatörlerini(streaming operators) özel veriler için de sağlamanın birkaç faydası vardır. Onlardan biri, özel tipleri kullanan konteynerleri akışa katma imkânı vermesidir. Örneğin:

```
QList<Painting> paintings = ...;
out << paintings;
```

Konteynerlerin içine okumayı da kolaylıkla yapabiliriz:

```
QList<Painting> paintings;
in >> paintings;
```

Eğer `Painting` tipi, `<<` ya da `>>` operatörlerinden birini desteklemiyorsa, bu işlemlerin sonucunda derleyici hata verecektir. Özel tipler için akış operatörleri sağlamanın bir başka faydası, bu tiplerin değerlerini `QVariant`'lar olarak saklayabilmemizdir. Bu, Bölüm 10'de açıklandığı gibi, önceden `qRegisterMeta-TypeStreamOperators<T>()` kullanılarak kayda geçirilmiş tipleri için geçerlidir.

`QDataStream` kullandığımızda, Qt, istenilen sayıda öğe içeren konteynerler de dâhil olmak üzere, her bir tipin okunması ve yazılması ile ilgilenir. Bu bizi, neyi yazacağımızı planlamaktan ve okuduğumuz üzerinde ayrıştırmanın herhangi bir türünü icra etmekten kurtarır. Tek yükümlülüğümüz, tüm tipleri, yazdığımız sırada okumaktır, detaylarla başa çıkmayı Qt'a bırakırız.

`QDataStream`, hem özel uygulamalarımızın dosya formatları için hem de standart ikili formatlar için oldukça kullanışlıdır. Standart ikili formatları, temel tipler üzerinde (`quint16` ya da `float` gibi) akış operatörleri ya da `readRawBytes()` ve `writeRawBytes()` kullanarak okuyabilir ve yazabiliriz. Eğer `QDataStream` sadece temel C++ veri tiplerini okumakta ve yazmakta kullanılıyorsa, `setVersion()` çağrısına bile ihtiyaç duymayız.

Şimdiye kadar, verileri akışın versiyonunu elle kodlayarak (`QDataStream::Qt_4_3` gibi) yükledik ve kaydettik. Bu yaklaşım basit ve güvenilirdir, fakat bir sakıncası vardır: yeni ya da güncellenmiş formatların avantajından yararlanamayız. Örneğin, eğer Qt'un daha sonraki bir versiyonu `QFont`'a yeni bir nitelik eklemişse ve biz versiyon numarasını elle kodlamışsak, bu nitelik kaydedilemeyecek ya da yüklenemeyecektir. İki çözüm vardır. İlk yaklaşım, `QDataStream` versiyon numarasını dosya içine gömmektir:

```
QDataStream out(&file);
out << quint32(MagicNumber) << quint16(out.version());
```

(`MagicNumber`, dosya tipini tanımlayan özgün bir sabittir.) Bu yaklaşım, veriyi daima `QDataStream`'in en son versiyonunu kullanarak yazmayı garanti eder. Dosyayı okuyacağımız zaman, akış versiyonunu da okuruz:

```
quint32 magic;
quint16 streamVersion;

QDataStream in(&file);
in >> magic >> streamVersion;
```

```
if (magic != MagicNumber) {
    std::cerr << "File is not recognized by this application"
              << std::endl;
} else if (streamVersion > in.version()) {
    std::cerr << "File is from a more recent version of the "
              << "application" << std::endl;
    return false;
}

in.setVersion(streamVersion);
```

Akış versiyonu, uygulama tarafından kullanılanın versiyonuna eşit ya da ondan küçük olduğu sürece, veriyi okuyabiliriz; aksi halde, bir hata ile karşılaşırız.

Eğer bir dosya formatı kendine ait bir versiyon numarası içeriyorsa, akış versiyonunu açıkça saklamak yerine onu kullanabiliriz. Örneğin, dosya formatının, uygulamamızın 1.3 versiyonu için olduğunu varsayalım. Veriyi aşağıdaki gibi yazabiliriz:

```
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);
out << quint32(MagicNumber) << quint16(0x0103);
```

Geri okurken de, kullanılacak `QDataStream` versiyonunu uygulamanın versiyon numarasına bağlı olarak belirleyebiliriz:

```
QDataStream in(&file);
in >> magic >> appVersion;

if (magic != MagicNumber) {
    std::cerr << "File is not recognized by this application"
              << std::endl;
    return false;
} else if (appVersion > 0x0103) {
    std::cerr << "File is from a more recent version of the "
              << "application" << std::endl;
    return false;
}

if (appVersion < 0x0103) {
    in.setVersion(QDataStream::Qt_3_0);
} else {
    in.setVersion(QDataStream::Qt_4_3);
}
```

Özet olarak, `QDataStream` versiyonlarını ele almak için üç yaklaşım vardır: versiyon numarasını elle kodlama, versiyon numarasını açıkça yazma ve okuma, ve elle kodlamanın farklı bir versiyonu olarak versiyon numarasını uygulamanın versiyonuna bağlı olarak belirleme. `QDataStream` versiyonlarını ele almak için bu yaklaşımlardan herhangi birini seçtikten sonra, Qt kullanarak ikili verileri yazma ve okuma hem basit hem de güvenilirdir.

Eğer bir dosyayı "bir seferde" okumak ya da yazmak istersek, `QDataStream`'i kullanmaktan kaçınabilir ve yerine `QIODevice`'ın `write()` ve `readAll()` fonksiyonlarını kullanabiliriz. Örneğin:

```
bool copyFile(const QString &source, const QString &dest)
{
```

```
    QFile sourceFile(source);
    if (!sourceFile.open(QIODevice::ReadOnly))
        return false;

    QFile destFile(dest);
    if (!destFile.open(QIODevice::WriteOnly))
        return false;

    destFile.write(sourceFile.readAll());

    return sourceFile.error() == QFile::NoError
           && destFile.error() == QFile::NoError;
}
```

`readAll()`'u çağırdığımız satırda, girdi dosyasının tümü -sonradan çıktı dosyasına yazılması için `write()` fonksiyonuna aktarılacak olan- bir `QByteArray` içine okunur. Tüm veriyi bir `QByteArray` içinde tutmak, öğe öğe okumaktan daha fazla bellek gerektirir, fakat bazı avantajları vardır. Örneğin, daha sonra veriyi sıkıştırmak ya da sıkıştırılmış veriyi açmak için `qCompress()` ve `qUncompress()` fonksiyonlarını çağırabiliriz.

Metin Okuma ve Yazma

İkili dosya formatları genellikle, metin tabanlı formatlardan daha derli toplu oldukları halde, insan tarafından okunabilir ve düzenlenebilir halde değildirler. Bunun bir sorun olduğu durumlarda, ikili formatlar yerine metin formatlarını kullanabiliriz. Qt, düz metin dosyalarını okumak ya da düz metin dosyalarına yazmak ve HTML, XML ve kaynak kod gibi diğer metin formatlarını kullanan dosyalar için `QTextStream` sınıfını sağlar.

`QTextStream`, sistemin yerel kodlaması ya da diğer herhangi bir kodlama ile Unicode arasındaki dönüşümle ilgilenir ve farklı işletim sistemleri tarafından kullanılan farklı satır sonlandırma kurallarını elden geçirir (Windows'ta "\r\n", Unix ve Mac OS X'te "\n"). `QTextStream`, verinin temel birimi olarak 16-bitlik `QChar` tipini kullanır. Karakterlere ve karakter katarlarına ek olarak, `QTextStream`, karakter katarlarına dönüştürülen ve karakter katarlarından dönüştürülen, C++'ın temel nümerik tiplerini de destekler. Örneğin aşağıdaki kod, `sf-book.txt` dosyasına "Thomas M. Disch: 334\n" yazar:

```
QFile file("sf-book.txt");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QTextStream out(&file);
out << "Thomas M. Disch: " << 334 << endl;
```

Metin yazma çok kolaydır, fakat metin okuma zorlayıcı olabilir, çünkü metinsel veri (`QDataStream` kullanılarak yazılan ikili veriden farklı olarak) temelde belirsizdir. Şimdi aşağıdaki örnek üzerinde düşünelim:

```
out << "Denmark" << "Norway";
```

Eğer `out` bir `QTextStream` ise, veri "DenmarkNorway" şeklinde yazılacaktır. Aşağıdaki kodun bu veriyi doğru okumasını kesinlikle bekleyemeyiz:

```
in >> str1 >> str2;
```

Gerçek şu ki, olan şey şudur: `str1` tüm kelimeyi ("DenmarkNorway") alır ve `str2` hiçbir değer alamaz. Bu problem, `QDataStream` ile ortaya çıkmaz çünkü `QDataStream` her bir karakter katarının uzunluğunu karakter verisinin önünde saklar.

Karmaşık dosya formatları için, bir full-blown ayrıştırıcı(parser) gerekebilir. Bu tür ayrıştırıcılar, veriyi, bir `QChar` üzerinde `>>` kullanarak, karakter karakter ya da `QTextStream::readLine()` kullanarak, satır satır okuyabilirler. Bu kısmın sonunda, iki küçük örnek göreceğiz; biri, bir girdi dosyasını satır satır okur, diğeri, onu karakter karakter okur. Bütün bir metin üzerinde çalışan ayrıştırıcılar için, eğer bellek kullanımı hakkında kaygılı değilsek, dosyanın tamamını tek seferde `QTextStream::readAll()` kullanarak okuyabiliriz.

`QTextStream` varsayılan olarak, okuma ve yazma için sistemin yerel kodlamasını kullanır (ISO 8859-1 ya da ISO 8859-15 gibi). Bu, aşağıdaki gibi `setCodec()` kullanılarak değiştirilebilir:

```
stream.setCodec("UTF-8");
```

Örnekte kullanılan UTF-8 kodlaması, tüm Unicode karakter setini sunan, popüler bir ASCII uyumlu kodlamadır.

`QTextStream` çeşitli seçeneklere sahiptir. Bunlar, akış manipülatörleri(stream manipulators) denilen özel nesnelere aktarılarak ya da aşağıda listelenen fonksiyonlar çağrılarak ayarlanabilirler. Aşağıdaki örnek, 12345678 tamsayısının çıktısını almadan önce `showbase`, `uppercasedigits` ve `hex` seçeneklerini ayarlamak suretiyle, "0xBC614E" metinini üretir:

```
out << showbase << uppercasedigits << hex << 12345678;
```

`setIntegerBase(int)`	
0	Öneke bağlı olarak otomatik belirlenir (okuma sırasında)
2	İkili
8	Sekizli
10	Onlu
16	Onaltılı

`setNumberFlags(NumberFlags)`	
`ShowBase`	Bir önek gösterir (eğer taban 2 ise ("0b"), 8 ise ("0"), 16 ise ("0x"))
`ForceSign`	Gerçek sayılarda daima işareti gösterir
`ForcePoint`	Sayılarda daima ondalık ayırıcı gösterir
`UppercaseBase`	Daima taban öneklerinin büyük harfli formatlarını kullanır ("0B", "0X")
`UppercaseDigits`	Onaltılı sayılarda büyük harf kullanır

`setRealNumberNotation(RealNumberNotation)`	
`FixedNotation`	Sabit noktalı gösterim (örneğin, "0.000123")
`ScientificNotation`	Bilimsel gösterim (örneğin, "1.234568e-04")
`SmartNotation`	Sabit noktalı ya da bilimsel gösterimden hangisi daha uygunsa

`setRealNumberPrecision(int)`
Gerçek sayıların duyarlılığını ayarlar (varsayılan olarak 6'dır)

`setFieldWidth(int)`
Bir alanın minimum boyutunu ayarlar (varsayılan olarak 0'dır)

setFieldAlignment(FieldAlignment)	
AlignLeft	Alanın sağ tarafını doldurur
AlignRight	Alanın sol tarafını doldurur
AlignCenter	Alanın her iki tarafını da doldurur
AlignAccountingStyle	İşaret ve sayı arasını doldurur

setPadChar(QChar)
Doldurma için kullanılacak karakteri ayarlar (varsayılan olarak boşluk karakteridir)

Seçenekler, üye fonksiyonlar kullanılarak da ayarlanabilirler:

```
out.setNumberFlags(QTextStream::ShowBase
                   | QTextStream::UppercaseDigits);
out.setIntegerBase(16);
out << 12345678;
```

QDataStream gibi, QTextStream de bir QIODevice altsınıfı (QFile, QTemporaryFile, QBuffer, QProcess, QTcpSocket, QUdpSocket ya da QSslSocket) ile işlem yapar. Ek olarak, direkt olarak bir QString ile de kullanılabilir. Örneğin:

```
QString str;
QTextStream(&str) << oct << 31 << " " << dec << 25 << endl;
```

Bu, onlu 31 sayısı, sekizli 37 sayısına denk geldiği için, str'nin içeriğini "37 25\n" yapar. Bu durumda, QString daima Unicode olduğu için, akış üzerinde bir kodlama ayarlamamız gerekmez.

Şimdi, bir metin tabanlı dosya formatının basit bir örneğine bakalım. Spreadsheet uygulamasında, Spreadsheet verisini saklamak için bir ikili format kullanmıştık. Veri satır, sütun ve formül üçlüsünün bir dizisinden oluşuyordu. Veriyi metin olarak yazmak basittir; burada Spreadsheet::writeFile()'ın revize edilmiş bir versiyonunun bir kısmı var:

```
QTextStream out(&file);
for (int row = 0; row < RowCount; ++row) {
    for (int column = 0; column < ColumnCount; ++column) {
        QString str = formula(row, column);
        if (!str.isEmpty())
            out << row << " " << column << " " << str << endl;
    }
}
```

Her bir satırın bir hücreyi temsil ettiği ve satır ile sütün, sütun ile formül arasında boşlukların olduğu basit bir format kullandık. Formül, boşluklar içerebilir, fakat hiç "\n" karakteri (satırları sonlandırmada kullandığımız karakter) içermediğini varsayabiliriz. Şimdi de, okuma kodunun revize edilmiş versiyonuna bakalım:

```
QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    QStringList fields = line.split(' ');
    if (fields.size() >= 3) {
        int row = fields.takeFirst().toInt();
        int column = fields.takeFirst().toInt();
        setFormula(row, column, fields.join(' '));
    }
}
```

Spreadsheet verisi içinden bir seferde bir satır okuruz. `readLine()` fonksiyonu, ardı ardına gelen '\n' karakterlerinden birini siler. `QString::split()` bir karakter katarı listesi döndürür, karakter katarını ayırırken verilen ayırıcıyı kullanır. Örneğin, "5 19 Total value" satırı için sonuç dört öğeli bir listedir: ["5", "19", "Total", "value"].

Eğer en az üç alana(field) sahipsek, veri çekmeye hazırızdır. `QStringList::takeFirst()` fonksiyonu, bir listedeki ilk öğeyi siler ve sildiği öğeyi döndürür. Onu, satır ve sütun numaralarını elde etmede kullanırız. Hata kontrolü yapmayız; eğer tamsayı olmayan bir satır ya da sütun değeri okuduysak, `QString::toInt()` 0 döndürür. `setFormula()` fonksiyonunu çağırırken, kalan alanları tek bir karakter katarı içine eklemeliyiz.

İkinci `QTextStream` örneğimizde, bir metin dosyasını okuyan ve çıktı olarak aynı metni fakat satırlardan ardı ardına gelen boşluklar silinmiş ve tüm tablar yerine boşluklar yerleştirilmiş şekilde üreten bir programı gerçekleştirmek için bir karakter karakter okuma yaklaşımı kullanacağız. Programın işini `tidyFile()` fonksiyonu yapar:

Kod Görünümü:

```
void tidyFile(QIODevice *inDevice, QIODevice *outDevice)
{
    QTextStream in(inDevice);
    QTextStream out(outDevice);

    const int TabSize = 8;
    int endlCount = 0;
    int spaceCount = 0;
    int column = 0;
    QChar ch;

    while (!in.atEnd()) {
        in >> ch;

        if (ch == '\n') {
            ++endlCount;
            spaceCount = 0;
            column = 0;
        } else if (ch == '\t') {
            int size = TabSize - (column % TabSize);
            spaceCount += size;
            column += size;
        } else if (ch == ' ') {
            ++spaceCount;
            ++column;
        } else {
            while (endlCount > 0) {
                out << endl;
                --endlCount;
                column = 0;
            }
            while (spaceCount > 0) {
                out << ' ';
                --spaceCount;
                ++column;
            }
            out << ch;
            ++column;
```

```
        }
    }
    out << endl;
}
```

Fonksiyona aktarılan QIODevice'lara bağlı olarak bir QTextStream girdisi ve bir QTextStream çıktısı oluştururuz. Geçerli karaktere ek olarak, üç adet durum izleme değişkenine(state-tracking variable) bakarız: biri yenisatırları sayar, biri boşlukları sayar, ve biri de geçerli satırdaki geçerli sütunu işaretler (tabları doğru sayıda boşluğa dönüştürmek için).

Ayrıştırma, girdi dosyasındaki her karakter üzerinde (bir seferde biri üzerinde) yinelenen bir while döngüsü içinde yapılır.

```
int main()
{
    QFile inFile;
    QFile outFile;

    inFile.open(stdin, QFile::ReadOnly);
    outFile.open(stdout, QFile::WriteOnly);

    tidyFile(&inFile, &outFile);

    return 0;
}
```

Bu örnek için, bir QApplication nesnesine ihtiyacımız yoktur, çünkü sadece Qt'un araç sınıflarını(tool classes) kullanıyoruz. Programın bir filtre olarak kullanıldığını varsaydık, örneğin:

```
tidy < cool.cpp > cooler.cpp
```

Programı, komut satırı üzerinde verilen dosya isimlerini işleyebilecek şekilde genişletmek kolay olacaktır.

Bu bir konsol uygulaması olduğu için, GUI uygulamaları için olanlardan biraz farklı bir .pro dosyasına sahiptir:

```
TEMPLATE        = app
QT              = core
CONFIG         += console
CONFIG         -= app_bundle
SOURCES         = tidy.cpp
```

Hiçbir GUI işlevi kullanmadığımız için sadece QtCore'a bağlarız. Sonra, Windows üzerinde konsol çıktısına izin vermek istediğimizi belirtiriz.

Düz ASCII ya da ISO 8859-1 (Latin-1) dosyalarını okumak ve yazmak için, QTextStream kullanmak yerine direkt olarak QIODevice'ın API'ını kullanmak da mümkündür. Bu, nadiren kullanılan bir yöntemdir. Eğer yinede bir QIODevice'a direkt olarak metin yazmak istiyorsanız, open() fonksiyonu çağrısında QIODevice::Text bayrağını açıkça belirtmek zorundasınız, örneğin:

```
file.open(QIODevice::WriteOnly | QIODevice::Text);
```

Yazarken, bu bayrak QIODevice'a, Windows üzerinde '\n' karakterlerini "\r\n" dizisine dönüştürmesini söyler. Okurken, bu bayrak aygıta, tüm platformlar üzerinde '\r' karakterlerini yok saymasını söyler.

Dizinler

QDir sınıfı, dizinleri platformdan bağımsız bir şekilde ele almayı sağlar ve dosyalar hakkında bilgilere erişir. QDir'in nasıl kullanıldığını görmek için, belli bir dizin ve onun tüm altdizinlerindeki tüm görseller tarafından kullanılan alanı hesaplayan küçük bir konsol uygulaması yazacağız.

Uygulamanın kalbi, verilen bir dizindeki görsellerin toplam boyutunu özyinelemeli olarak hesaplayan imageSpace() fonksiyonudur:

```
qlonglong imageSpace(const QString &path)
{
    QDir dir(path);
    qlonglong size = 0;

    QStringList filters;
    foreach (QByteArray format, QImageReader::supportedImageFormats())
        filters += "*." + format;

    foreach (QString file, dir.entryList(filters, QDir::Files))
        size += QFileInfo(dir, file).size();

    foreach (QString subDir, dir.entryList(QDir::Dirs
                                           | QDir::NoDotAndDotDot))
        size += imageSpace(path + QDir::separator() + subDir);

    return size;
}
```

Verilen yollu(path) kullanarak bir QDir nesnesi oluşturarak başlarız. entryList() fonksiyonuna iki argüman aktarırız. İlki, dosya isim filtrelerinin bir listesidir. Bunlar, '*' ve '?' genel arama karakterlerini içerebilirler. Bu örnekte, sadece QImage'ın okuyabileceği dosya formatlarını filtreleriz. İkincisi,istediğimiz girdilerin(entry) çeşidini (normal dosyalar, dizinler, sürücüler, vs) belirtir.

Dosya listesi üzerinde yineleme yapar ve dosyaların boyutlarını toplarız. QFileInfo sınıfı, bir dosyanın boyutu, izinleri, sahibi gibi özniteliklerine erişme imkânı verir.

İkinci entryList() çağrısı bu dizindeki tüm altdizinlere erişir. Onların üzerinde yineleme yapar ve toplam boyutlarını öğrenmek için özyinelemeli olarak imageSpace() fonksiyonunu çağırırız.

Her bir altdizinin yolunu şu şekilde oluştururken, geçerli dizinin yolunu -araya bir slash koyarak- altdizinin ismi ile birleştiririz. Yolları kullanıcıya sunarken, slashları platforma özgü ayırıcıya dönüştürmek için QDir::toNativeSeparators() statik fonksiyonunu çağırabiliriz.

Şimdide küçük programımıza bir main() fonksiyonu ekleyelim:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    QString path = QDir::currentPath();
    if (args.count() > 1)
        path = args[1];

    std::cout << "Space used by images in " << qPrintable(path)
              << " and its subdirectories is "
```

```
                << (imageSpace(path) / 1024) << " KB" << std::endl;

    return 0;
}
```

Yolu, geçerli dizine ilklendirmek için `QDir::currentPath()` fonksiyonunu kullanırız. Alternatif olarak, yolu, kullanıcının ev dizinine(home directory) ilklendirmek için `QDir::homePath()` fonksiyonunu kullanabilirdik. Eğer kullanıcı, komut satırında açıkça belirtilmiş bir yola sahipse, ilki yerine bunu kullanırız. Son olarak, görsellerin ne kadar alan kapladıklarını hesaplamak için `imageSpace()` fonksiyonunu çağırırız.

`QDir` sınıfı, `entryInfoList()` (`QFileInfo` nesnelerinin bir listesini döndürür), `rename()`, `exists()`, `mkdir()` ve `rmdir()` gibi başka dosya (ve dizin) ilişkili fonksiyonlar da sağlar. `QFile` sınıfı, `remove()` ve `exists()` gibi statik uygunluk fonksiyonları sağlar. Ve `QFileSystemWatcher` sınıfı, `directoryChanged()` ve `fileChanged()` sinyallerini yayarak, bir dizin ya da bir dosyada bir değişiklik meydana geldiğinde bizi bilgilendirir.

Gömülü Kaynaklar

Bu bölümde şimdiye kadar, harici aygıtlar içindeki veriye erişmek hakkında konuştuk, fakat Qt ile, uygulamanın çalıştırılabilir dosyası içine ikili veri ya da metin gömmek de mümkündür. Bu, Qt'un kaynak sistemi(resource system) kullanılarak başarılır. Diğer bölümlerde, kaynak dosyalarını, çalıştırılabilir dosya içine görseller gömmek için kullandık, fakat her türlü dosyayı gömmek mümkündür. Gömülü dosyalar, tıpkı dosya sistemindeki normal dosyalar gibi `QFile` kullanılarak okunabilirler.

Kaynaklar, Qt'un kaynak derleyicisi `rcc` tarafından C++ koduna dönüştürülürler. `.pro` dosyasına bu satırı ekleyerek `qmake`'e `rcc`'yi yürütmede özel kurallar dâhil edebiliriz:

```
RESOURCES = myresourcefile.qrc
```

`myresourcefile.qrc` dosyası, çalıştırılabilir dosya içine gömülü dosyaları listeleyen bir XML dosyasıdır.

İletişim bilgilerini tutan bir uygulama yazdığımızı hayal edelim. Kullanıcılarımıza kolaylık olması için, uluslar arası telefon kodlarını çalıştırılabilir dosya içine gömmek istiyoruz. Eğer dosya, uygulamanın inşa edildiği dizindeki `datafiles` dizininin içinde olursa, kaynak dosyası(resource file) şuna benzeyecekti:

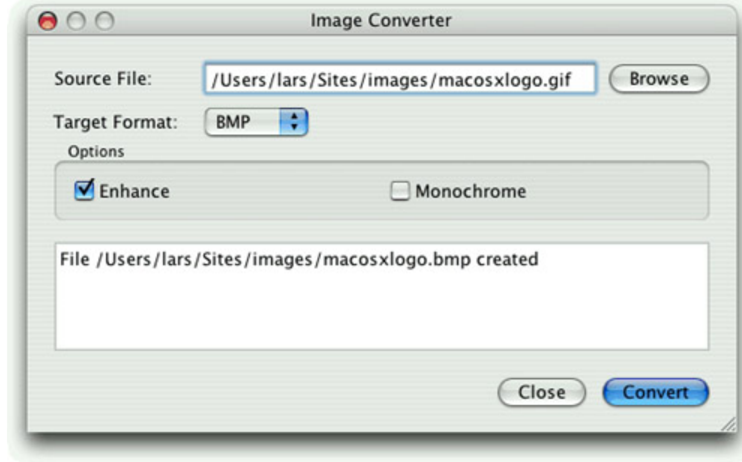
```
<RCC>
<qresource>
    <file>datafiles/phone-codes.dat</file>
</qresource>
</RCC>
```

Kaynaklar, uygulama içinde `:/` yol öneki ile tanınırlar. Bu örnekte, telefon kodları dosyası `:/datafiles/phone-codes.dat` yoluna sahiptir ve diğer herhangi bir dosya gibi `QFile` kullanılarak okunabilirler.

Süreçler Arası İletişim

`QProcess` sınıfı bize, harici programlar çalıştırma ve onlar ile etkileşme olanağı verir. Sınıf, zaman uyumsuz çalışır fakat işlerini arka planda yaptığı için, kullanıcı arayüzü uyumu sürdürür. `QProcess`, harici işlem bir veriye sahip olduğunda ya da sonlandığında, bizi bilgilendirmek için sinyaller yayar.

Bir harici resim dönüştürme programına, bir kullanıcı arayüzü sağlayan küçük bir uygulamanın kodlarını inceleyeceğiz. Bu örnek için, tüm platformlar için mevcut olan `ImageMagick` dönüştürme programına bel bağlıyoruz. Kullanıcı arayüzümüz Şekil 11.1'de gösteriliyor.



Şekil 11.1

Kullanıcı arayüzü Qt Designer'da oluşturulmuştur. Biz burada sadece başlık dosyası ile başlayarak, `uic` tarafından üretilen `Ui::ConvertDialog` sınıfından türetilmiş altsınıfa odaklanacağız:

```
#ifndef CONVERTDIALOG_H
#define CONVERTDIALOG_H

#include <QDialog>
#include <QProcess>

#include "ui_convertdialog.h"

class ConvertDialog : public QDialog, private Ui::ConvertDialog
{
    Q_OBJECT

public:
    ConvertDialog(QWidget *parent = 0);

private slots:
    void on_browseButton_clicked();
    void convertImage();
    void updateOutputTextEdit();
    void processFinished(int exitCode, QProcess::ExitStatus exitStatus);
    void processError(QProcess::ProcessError error);

private:
    QProcess process;
    QString targetFile;
};

#endif
```

Başlık dosyası, Qt Designer formlarının altsınıfları için olan modele yakın bir modeli takip eder. Diğer örneklerin bazılarından küçük bir fark olarak, `Ui::ConvertDialog` sınıfı için private kalıtım kullandık. Bu, form parçacıklarına formun fonksiyonları haricindeki erişimlerin önüne geçer. Qt Designer'ın otomatik

bağlama mekanizmasına şükürler olsun ki, `on_browseButton_clicked()` yuvası, Browse butonunun `clicked()` sinyaline otomatik olarak bağlanır.

```
ConvertDialog::ConvertDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    QPushButton *convertButton =
            buttonBox->button(QDialogButtonBox::Ok);
    convertButton->setText(tr("&Convert"));
    convertButton->setEnabled(false);

    connect(convertButton, SIGNAL(clicked()),
            this, SLOT(convertImage()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
    connect(&process, SIGNAL(readyReadStandardError()),
            this, SLOT(updateOutputTextEdit()));
    connect(&process, SIGNAL(finished(int, QProcess::ExitStatus)),
            this, SLOT(processFinished(int, QProcess::ExitStatus)));
    connect(&process, SIGNAL(error(QProcess::ProcessError)),
            this, SLOT(processError(QProcess::ProcessError)));
}
```

`setupUi()` çağrısı, formun tüm parçacıklarını oluşturur, yerleştirir ve `on_browseButton_clicked()` yuvası için sinyal-yuva bağlantısını kurar. Buton kutusunun(Button Box) OK butonuna bir işaretçi ediniriz ve ona daha uygun bir metin veririz. Ayrıca başlangıçta dönüştürmek için resim mevcut olmadığından, butonu devredışı bırakırız ve `convertImage()` yuvasına bağlarız. Sonra, buton kutusunun `rejected()` sinyalini (Close butonu tarafından yayılan), diyaloğun `reject()` yuvasına bağlarız. Bundan sonra, `QProcess` nesnesinden üç sinyali, üç private yuvaya bağlarız. Harici işlem, `cerr` üzerinde veriye sahip olduğunda, onu `updateOutputTextEdit()` içinde işleyeceğiz.

```
void ConvertDialog::on_browseButton_clicked()
{
    QString initialName = sourceFileEdit->text();
    if (initialName.isEmpty())
        initialName = QDir::homePath();
    QString fileName =
            QFileDialog::getOpenFileName(this, tr("Choose File"),
                                         initialName);
    fileName = QDir::toNativeSeparators(fileName);
    if (!fileName.isEmpty()) {
        sourceFileEdit->setText(fileName);
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
}
```

`setupUi()` tarafından, Browse butonunun `clicked()` sinyali, `on_browseButton_clicked()` yuvasına otomatik olarak bağlanmıştı. Eğer kullanıcı önceden bir dosya seçmişse, dosya diyaloğunu dosyanın ismi ile başlatırız; aksi halde, kullanıcının ev dizinini kullanırız.

```
void ConvertDialog::convertImage()
{
    QString sourceFile = sourceFileEdit->text();
    targetFile = QFileInfo(sourceFile).path() + QDir::separator()
                 + QFileInfo(sourceFile).baseName() + "."
                 + targetFormatComboBox->currentText().toLower();
```

```
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);
    outputTextEdit->clear();

    QStringList args;
    if (enhanceCheckBox->isChecked())
        args << "-enhance";
    if (monochromeCheckBox->isChecked())
        args << "-monochrome";
    args << sourceFile << targetFile;

    process.start("convert", args);
}
```

Kullanıcı, Convert butonuna tıkladığında, kaynak dosyanın ismini kopyalarız ve uzantısını hedef dosya formatının ki ile değiştiririz. Slashları elle kodlamak yerine platforma özgü dizin ayırıcısı kullanırız, çünkü dosya ismi kullanıcıya görünür olacak.

Sonra, kullanıcının kazayla çoklu dönüştürmeler başlatmasını önlemek için Convert butonunu devredışı bırakırız ve durum bilgisini göstermede kullandığımız metin editörünü temizleriz.

Harici süreci başlatmak için, çalıştırmak istediğimiz programın ismi (`convert`) ve ihtiyacı olan argümanlar ile `QProcess::start()`'ı çağırırız. Bu durumda, eğer kullanıcı uygun seçenekleri işaretlediyse, `–enhance` ve `–monochrome` bayraklarını, ardından da kaynak ve hedef dosya isimlerini aktarırız. `convert` programı, gerekli dönüşümü dosya uzantılarına bakarak anlar.

```
void ConvertDialog::updateOutputTextEdit()
{
    QByteArray newData = process.readAllStandardError();
    QString text = outputTextEdit->toPlainText()
                   + QString::fromLocal8Bit(newData);
    outputTextEdit->setPlainText(text);
}
```

Harici süreç `cerr`'e yazdığında, `updateOutputTextEdit()` yuvası çağrılır. Hata metnini okur ve onu `QTextEdit`'in varolan metnine ekleriz.

```
void ConvertDialog::processFinished(int exitCode,
                                    QProcess::ExitStatus exitStatus)
{
    if (exitStatus == QProcess::CrashExit) {
        outputTextEdit->append(tr("Conversion program crashed"));
    } else if (exitCode != 0) {
        outputTextEdit->append(tr("Conversion failed"));
    } else {
        outputTextEdit->append(tr("File %1 created").arg(targetFile));
    }
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
}
```

Süreç sonlandığında, kullanıcının bunu bilmesini sağlarız ve Convert butonunu etkinleştiririz.

```
void ConvertDialog::processError(QProcess::ProcessError error)
{
    if (error == QProcess::FailedToStart) {
        outputTextEdit->append(tr("Conversion program not found"));
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
```

```
}
```

Eğer süreç başlatılamazsa, QProcess, finished() yerine error() sinyalini yayar. Hatayı rapor ederiz ve Click butonunu etkinleştiririz.

Bu örnekte, zaman uyumsuz dosya dönüştürmeleri yaptık, QProcess ile convert programını çalıştırmaktan ve ardından kontrolü tekrar uygulamaya iade etmekten bahsettik. Bu, süreç arkaplanda işlerken, kullanıcı arayüzünü cevap verebilir halde tutar. Fakat bazı durumlarda, uygulamamızda başka işlemler yapabilmek için harici sürecin sonlanması, bazı durumlarda ise QProcess'i eşzamanlı işletmemiz gerekir.

Eşzamanlı davranmanın hoş olduğu yaygın bir örnek, düz metinleri düzenlemede kullanılan metin editörü uygulamalarıdır. Bu, apaçık bir QProcess uygulamasıdır. Örneğin, diyelim ki bir QTextEdit içinde düz metne sahibiz ve kullanıcının tıklayabildiği ve bir edit() yuvasına bağlanmış bir Edit butonu sağlıyoruz:

```
void ExternalEditor::edit()
{
    QTemporaryFile outFile;
    if (!outFile.open())
        return;

    QString fileName = outFile.fileName();
    QTextStream out(&outFile);
    out << textEdit->toPlainText();
    outFile.close();

    QProcess::execute(editor, QStringList() << options << fileName);

    QFile inFile(fileName);
    if (!inFile.open(QIODevice::ReadOnly))
        return;

    QTextStream in(&inFile);
    textEdit->setPlainText(in.readAll());
}
```

QTemporaryFile'ı, benzersiz(unique) bir isimle boş bir dosya oluşturmakta kullanırız. Varsayılan olarak oku-yaz moduna uygun açıldığı için QTemporaryFile::open()'a argüman aktarmayız. Metin editörünün içeriğini geçici dosyaya(temporary file) yazarız ve sonra dosyayı kapatırız, çünkü bazı metin editörleri açık olan dosyalar üzerinde çalışamazlar.

QProcess::execute() statik fonksiyonu, harici bir süreci yürütür ve süreç sonlanana kadar uygulamayı bloke eder. editor argümanı, çalıştırılabilir bir editörün ismini (örneğin gvin) tutan bir QString'dir. options argümanı ise, bir QStringList'tir (eğer gvim kullanıyorsak, bir öğe içerir, "-f").

Kullanıcı, metin editörünü kapattıktan sonra, süreç sonlanır ve execute() sonuçları çağırır. Sonra, geçici dosyayı açarız ve içeriğini QTextEdit içine okuruz. QTemporaryFile, nesne kapsamın dışına çıktığında, geçici dosyayı otomatik olarak siler.

QProcess eşzamanlı kullanıldığında, sinyal-yuva bağlantılarına gerek yoktur. Eğer execute() statik fonksiyonunun sağladığından daha iyi bir kontrol gerekliyse, alternatif bir yaklaşım kullanabiliriz. Bu, bir QProcess nesnesi oluşturmayı ve üstünde start()'ı çağırmayı gerektirir, sonra QProcess::wait-ForStarted()'ı çağırarak uygulamayı bloke etmeye zorlarız, ve eğer bu başarılı olursa,

`QProcess::waitForFinished()` çağrılır. Bu yaklaşımı kullanan bir örnek için, `QProcess` referans dokümantasyonuna bakabilirsiniz.

BÖLÜM 12: VERİTABANLARI

QtSql modülü, SQL veritabanlarına erişmek için, platformdan (ve veritabanından) bağımsız bir arayüz sağlar. Bu arayüz, kullanıcı arayüzü ile veritabanı bütünleşmesini sağlayan Qt Model/Görünüş mimarisini kullanan bir sınıf seti tarafından desteklenir. Bu bölüm, Bölüm 9'da üzerinde durulan Qt Model/Görünüş sınıflarına aşina olduğunuzu varsayar.

Bir veritabanı bağlantısı, bir QSqlDatabase nesnesi ile ifade edilir. Qt, çeşitli veritabanı API'ları ile haberleşmek için sürücüler(drivers) kullanır. Qt Desktop Edition aşağıdaki sürücüleri içerir:

Driver	Database
QDB2	IBM DB2 versiyon 7.1 ve sonrası
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle (Oracle Call Interface)
QODBC	ODBC (Microsoft SQL Server'ı içerir)
QPSQL	PortgreSQL 7.3 ve sonrası
QSQLITE	SQLite versiyon 3
QSQLITE2	SQLite versiyon 2
QTDS	Sybase Adaptive Server

Lisans kısıtlamaları nedeniyle, Qt Open Source Edition ile tüm sürücüler sağlanmaz.

SQL sözdizimi ile rahat olan kullanıcılar için, QSqlQuery sınıfı, keyfi SQL ifadelerini doğrudan çalıştırmayı ve sonuçlarını ele almayı sağlar. SQL sözdiziminden kaçınarak daha yüksek seviyeli bir veritabanı arayüzünü tercih eden kullanıcılar için QSqlTableModel ve QSqlRelationalTableModel uygun soyutlamayı sağlar. Bu sınıflar, Qt'un diğer Model sınıfları ile aynı şekilde, bir SQL tablosunu temsil ederler. Veriyi kod içinde travers etmede ve düzenlemede bağımsız olarak kullanılabilirler ya da son kullanıcıların veriyi görüntüleyebildiği ve düzenleyebildiği Görünüşlere doğrudan bağlı olabilirler.

Qt ayrıca, master-detail ve drill-down gibi yaygın veritabanı deyimlerini programlamayı basitleştirir ve veritabanını görüntülemek için formlar ya da bu bölümde örneklendirileceği gibi GUI tabloları kullanır.

Bağlanma ve Sorgulama

SQL sorgularını çalıştırmak için, ilk önce bir veritabanı ile bağlantı kurmalıyız. Veritabanı bağlantıları genellikle, uygulamanın başlangıcında çağırdığımız ayrı bir fonksiyon içinde ayarlanır. Örneğin:

```
bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("mozart.konkordia.edu");
    db.setDatabaseName("musicdb");
```

```
    db.setUserName("gbatstone");
    db.setPassword("T17aV44");
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                              db.lastError().text());
        return false;
    }
    return true;
}
```

Öncelikle, bir `QSqlDatabase` nesnesi oluşturmak için `QSqlDatabase::addDatabase()`'i çağırırız. `addDatabase()`'in ilk argümanı, Qt'un veritabanına erişmede kullanacağı veritabanı sürücüsüdür. Bu örnekte, MySQL'i kullanıyoruz.

Sonra, veritabanı ana makine adını(host name), veritabanı adını(database name), kullanıcı adını(user name) ve şifreyi(password) ayarlar, bağlantıyı açarız. Eğer `open()` başarısız olursa, bir hata mesajı gösteririz.

Genellikle, `main()` içinde `createConnection()`'ı çağırırdık:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!createConnection())
        return 1;
    ...
    return app.exec();
}
```

Bağlantı bir kere kurulduğunda, kullandığımız veritabanının desteklediği herhangi bir SQL ifadesini çalıştırmada `QSqlQuery`'yi kullanabiliriz. Örneğin burada bir `SELECT` ifadesinin çalıştırılması gösteriliyor:

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

`exec()` çağrısından sonra, sorgunun sonuç seti boyunca dolaşabiliriz:

```
while (query.next()) {
    QString title = query.value(0).toString();
    int year = query.value(1).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

`QSqlQuery`'yi sonuç setinin ilk kaydına(record) konumlandırmak için `next()`'i bir kere çağırırız. Sonraki `next()` çağrıları, en sona ulaşana kadar, kayıt işaretçisini her seferinde bir kayıt ilerletir, sona geldiğinde ise `next()`, `false` döndürür. Eğer sonuç seti boş ise (ya da sorgulama başarısız olduysa), ilk `next()` çağrısında `false` dönecektir.

`value()` fonksiyonu bir alanın değerini bir `QVaiant` olarak döndürür. Alanlar 0'dan başlayarak, `SELECT` ifadesinde verilen sırayla numaralandırılırlar. `QVariant` sınıfı, `int` ve `QString` de dâhil olmak üzere birçok C++ ve Qt tipini tutabilir. Bir veritabanında saklanabilecek farklı tiplerde veriler, uygun C++ ve Qt tipleri ile eşleştirilirler ve `QVariant`'lar içinde saklanırlar. Örneğin, bir `VARCHAR` bir `QString` olarak, bir `DATETIME` bir `QDateTime` olarak ifade edilir.

QSqlQuery, sonuç seti boyunca dolaşmak için başka fonksiyonlar da sağlar: first(), last(), previous() ve seek(). Bu fonksiyonlar kullanışlıdır, fakat bazı veritabanları için next()'ten daha yavaş ve bellek canavarı olabilirler. Büyük veri setleri üzerinde işlemler yaparken, basit bir optimizasyon olarak, exec()'i çağırmadan önce QSqlQuery::setForwardOnly(true)'yu çağırabilir ve sonuç seti boyuca dolaşmak için sadece next()'i kullanabiliriz.

Daha evvel, SQL sorgusunu QSqlQuery::exec()'e bir argüman olarak aktarmıştık, fakat onu doğrudan kurucusuna da aktarabiliriz:

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

Sorgu üzerinde isActive()'i çağırarak hata kontrolü yapabiliriz:

```
if (!query.isActive())
    QMessageBox::warning(this, tr("Database Error"),
                         query.lastError().text());
```

Eğer hiç hata oluşmamışsa, sorgu aktifleşecek ve next()'i kullanarak sonuç seti boyunca dolaşabileceğiz.

Bir INSERT işlemi yapmak, hemen hemen bir SELECT sorgusu gerçekleştirmek kadar kolaydır:

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
                "VALUES (203, 102, 'Living in America', 2002)");
```

Bundan sonra, numRowsAffected(), SQL ifadesi tarafından etkilenmiş satırların sayısını döndürür (ya da hata olduğunda -1).

Eğer birçok kayıt eklememiz(insert) gerekiyorsa ya da değerlerin karakter katarlarına dönüşmesinin önüne geçmek istiyorsak, sorgunun, yer tutucular(placeholders) içeren bir sorgu olduğunu belirtmek için prepare() kullanabilir, sonrasında da eklemek istediğimiz değerleri tutturabiliriz(bind). Qt, tüm veritabanlarına, yer tutucular için hem Oracle tarzı hem de ODBC tarzı sözdizimi desteği verir. İşte, Oracle tarzı sözdizimi kullanan bir örnek:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", "Living in America");
query.bindValue(":year", 2002);
query.exec();
```

Bu da aynı örneğin ODBC tarzı sözdizimi kullanan versiyonu:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue("Living in America");
query.addBindValue(2002);
query.exec();
```

exec() çağrısından sonra, yeni değerler tutturmak için bindValue() ya da addBindValue()'yu çağırabiliriz, sonrada sorguyu yeni değerler ile çalıştırmak için exec()'i tekrar çağırırız.

Yer tutucular sıklıkla, ikili veriler ya da ASCII veya Latin-1 karakterler içermeyen karakter katarlarını belirtmede kullanılırlar.

Qt, kullanılabilir oldukları yerlerde, veritabanları üzerinde SQL etkileşimlerini(transactions) destekler. Bir etkileşim başlatmak için, veritabanı bağlantısını temsil eden QSqlDatabase nesnesi üzerinde transaction()'ı çağırırız. Etkileşimi sonlandırmak içinse, `commit()` ya da `rollback()`'i çağırırız. Örneğin burada, bir dış anahtarı nasıl arayacağımız ve bir etkileşim içinde bir `INSERT` ifadesini nasıl çalıştıracağımız gösteriliyor:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
    int artistId = query.value(0).toInt();
    query.exec("INSERT INTO cd (id, artistid, title, year) "
               "VALUES (201, " + QString::number(artistId)
               + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

`QSqlDatabese::database()` fonksiyonu, `createConnection()` içinde oluşturduğumuz bir bağlantıyı temsil eden bir `QSqlDatabase` nesnesi döndürür. Eğer etkileşim başlatılamazsa, `QSqlDatabase::transaction()`, `false` döndürür. Bazı veritabanları etkileşimleri desteklemezler. Onlar için, `transaction()`, `commit()` ve `rollback()` fonksiyonları hiçbir şey yapmazlar. Bir veritabanının etkileşimleri destekleyip desteklemediğini anlamak için, veritabanına ilişkin `QSqlDriver` üzerinde `hasFeature()`'ı çağırabiliriz:

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
    ...
```

BLOB(binary large object), Unicode ve hazır sorgular gibi farklı veritabanı özellikleri de test edilebilir.

Şimdiye kadarki örneklerde, uygulamanın tek bir veritabanı bağlantısı kullandığını varsaydık. Eğer çoklu bağlantılar oluşturmak istersek, `addDatabase()`'e ikinci argüman olarak bir isim aktarabiliriz. Örneğin:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL", "OTHER");
db.setHostName("saturn.mcmanamy.edu");
db.setDatabaseName("starsdb");
db.setUserName("hilbert");
db.setPassword("ixtapa7");
```

Sonra, `QSqlDatabase::datebase()`'e ismi aktararak `QSqlDatabase` nesnesine bir işaretçi edinebiliriz:

```
QSqlDatabase db = QSqlDatabase::database("OTHER");
```

Diğer bağlantıyı kullanarak sorgular çalıştırmak için, `QSqlQuery` kurucusuna `QSqlDatabase` nesnesini aktarabiliriz:

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

Çoklu bağlantılar, her bir bağlantı sadece bir etkileşimi işleyebildiği için, aynı anda birden fazla etkileşim gerçekleştirmek istediğimizde çok kullanışlıdırlar. Çoklu veritabanı bağlantılarını kullanırken, isimsiz bir bağlantıya sahip olabiliriz, öyle ki `QSqlQuery`, eğer hiçbiri belirtilmemişse, bu bağlantıyı kullanır.

`QSqlQuery`'ye ek olarak, Qt, bizi en yaygın SQL işlemlerini (`SELECT`, `INSERT`, `UPDATE` ve `DELETE`) gerçekleştirmek için saf SQL kullanmaktan kaçınma imkânı veren, daha yüksek seviyeli bir arayüz olarak `QSqlTableModel` sınıfını sağlar. Bu sınıf, bir veritabanını GUI'den bağımsız olarak değiştirmede, ya da bir veritabanını `QListView` veya `QTableView` için bir veri kaynağı olarak ayarlamada kullanılabilir.

İşte, bir `SELECT` sorgusu gerçekleştirmede `QSqlTableModel`'ın kullanıldığı bir örnek:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("year >= 1998");
model.select();
```

Bu, şu sorguya denktir:

```
SELECT * FROM cd WHERE year >= 1998
```

Sonuç seti boyunca dolaşma, verilen bir kayda `QSqlTableModel::record()` kullanarak, alanlara ise `value()` kullanarak erişerek yapılır:

```
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

`QSqlRecord::value()` fonksiyonu, bir alan ismi ya da bir alan indeksi alır. Büyük veri setleri üzerinde çalışırken, bu alanlara, onları belirten indeksler ile erişilmesi önerilir. Örneğin:

```
int titleIndex = model.record().indexOf("title");
int yearIndex = model.record().indexOf("year");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value(titleIndex).toString();
    int year = record.value(yearIndex).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

Bir veritabanı tablosuna bir kayıt eklemek için, yeni, boş bir satır (kayıt) oluşturması için `insertRow()`'u çağırırız ve her bir sütunun (alan) değerini ayarlamak için `setData()`'yı kullanırız:

```
QSqlTableModel model;
model.setTable("cd");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 113);
model.setData(model.index(row, 1), "Shanghai My Heart");
model.setData(model.index(row, 2), 224);
model.setData(model.index(row, 3), 2003);
model.submitAll();
```

submitAll() çağrısından sonra kayıt, tablonun nasıl sıralandığına bağlı olarak, farklı bir satır konumuna taşınabilir. Eğer ekleme başarılı olmazsa, submitAll() çağrısı false döndürecektir.

Bir SQL Modeli ile standart bir Model arasındaki önemli bir fark, bir SQL Modelinde, değişikliklerin veritabanına yazılması için submitAll()'u çağırmak zorunda olmamızdır.

Bir kaydı güncellemek için ilk önce QSqlTableModel'ı, düzenlemek istediğimiz kayda konumlandırmalıyız (örneğin select()'i kullanarak). Sonra, değiştirmek istediğimiz alanları günceller ve değişikliklerimizi veritabanına geri yazarız:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    QSqlRecord record = model.record(0);
    record.setValue("title", "Melody A.M.");
    record.setValue("year", record.value("year").toInt() + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

Eğer belirtilen filtreyle eşleşen bir kayıt varsa, QSqlTableModel::record()'u kullanarak ona erişiriz. Değişikliklerimizi uygular ve düzenlenmiş kaydımızı orijinal kayıt üzerine yazarız.

SQL Modeli olmayan Modeller için yaptığımız gibi, setData() kullanarak da bir güncelleme gerçekleştirme mümkündür:

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), "Melody A.M.");
    model.setData(model.index(0, 3),
                  model.data(model.index(0, 3)).toInt() + 1);
    model.submitAll();
}
```

Bir kaydı silmek, güncellemekle benzerdir:

```
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

removeRows() çağrısı, silinecek ilk kaydın satır numarasını ve silinecek kayıtların sayısını alır. Sıradaki örnek, filtre ile eşleşen tüm kayıtları siler:

```
model.setTable("cd");
model.setFilter("year < 1990");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

`QSqlQuery` ve `QSqlTableModel` sınıfları, Qt ve bir SQL veritabanı arasında bir arayüz sağlar. Bu sınıfları kullanarak, kullanıcılara veriler sunan ve kullanıcılara kayıt ekleme, güncelleme ve silme izni veren formlar oluşturabiliriz.

SQL sınıfları kullanan projeler için, `.pro` dosyamıza şu satırı eklemeliyiz:

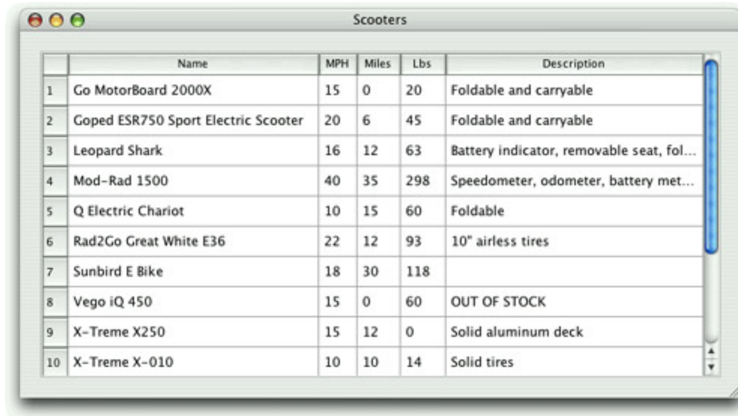
```
QT += sql
```

Tabloları Görüntüleme

Önceki kısımda, bir veritabanı ile `QSqlQuery` ve `QSqlTableModel` kullanarak nasıl etkileşeceğimizi gördük. Bu kısımda, bir `QSqlTableModel`'ı bir `QTableView` parçacığı içinde sunmayı göreceğiz.

Şekil 12.1'de görünen Scooters uygulaması, `scooter` Modellerinin bir tablosunu sunar. Örnek, aşağıdaki gibi tanımlanan tek bir tabloya (scooter) dayanır:

```
CREATE TABLE scooter (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    maxspeed INTEGER NOT NULL,
    maxrange INTEGER NOT NULL,
    weight INTEGER NOT NULL,
    description VARCHAR(80) NOT NULL);
```



	Name	MPH	Miles	Lbs	Description
1	Go MotorBoard 2000X	15	0	20	Foldable and carryable
2	Goped ESR750 Sport Electric Scooter	20	6	45	Foldable and carryable
3	Leopard Shark	16	12	63	Battery indicator, removable seat, fol...
4	Mod-Rad 1500	40	35	298	Speedometer, odometer, battery met...
5	Q Electric Chariot	10	15	60	Foldable
6	Rad2Go Great White E36	22	12	93	10" airless tires
7	Sunbird E Bike	18	30	118	
8	Vego iQ 450	15	0	60	OUT OF STOCK
9	X-Treme X250	15	12	0	Solid aluminum deck
10	X-Treme X-010	10	10	14	Solid tires

Şekil 12.1

`id` alanının değeri, veritabanı (bu örnekte SQLite) tarafından otomatik üretilir. Diğer veritabanları, bunun için farklı bir sözdizimi kullanabilir.

Bakım kolaylığı için, sütun indekslerine anlamlı isimler vermede bir enum kullanırız:

```
enum {
    Scooter_Id = 0,
    Scooter_Name = 1,
    Scooter_MaxSpeed = 2,
    Scooter_MaxRange = 3,
    Scooter_Weight = 4,
    Scooter_Description = 5
};
```

İşte, bir `QSqlTableModel`'ı, `scooter` tablosunu görüntülemek için ayarlamada gerekli tüm kodlar:

```
model = new QSqlTableModel(this);
model->setTable("scooter");
model->setSort(Scooter_Name, Qt::AscendingOrder);
model->setHeaderData(Scooter_Name, Qt::Horizontal, tr("Name"));
model->setHeaderData(Scooter_MaxSpeed, Qt::Horizontal, tr("MPH"));
model->setHeaderData(Scooter_MaxRange, Qt::Horizontal, tr("Miles"));
model->setHeaderData(Scooter_Weight, Qt::Horizontal, tr("Lbs"));
model->setHeaderData(Scooter_Description, Qt::Horizontal,
                     tr("Description"));
model->select();
```

Modeli oluşturma, bir önceki kısımda gördüğümüzün benzeridir. Tek farklılık, kendi sütun başlıklarımızı sağlamamızdır. Eğer böyle yapmazsak, ham alan isimleri kullanılacaktır. Ayrıca `setSort()` kullanarak, bir sıralama düzeni belirtiriz; perde arkasında bu, bir `ORDER_BY` cümleciği tarafından gerçekleştirilir.

Böylece bir Model oluşturduk ve `select()` kullanarak onu veri ile doldurduk. Şimdi, onu sunmak için bir Görünüş oluşturabiliriz:

```
view = new QTableView;
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
view->setColumnHidden(Scooter_Id, true);
view->resizeColumnsToContents();
view->setEditTriggers(QAbstractItemView::NoEditTriggers);

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

Bölüm 9'da, `QTableView`'ı bir `QAbstractItemModel`'daki veriyi bir tablo içinde sunmada kullanmayı görmüştük. `QSqlTableModel`, `QAbstractItemModel`'dan türetildiği için, kolayca bir `QTableView`'ın veri kaynağı olarak kullanılabilir. `setModel()` çağrısı, tamamen Görünüşü Modele bağlamak için gereklidir. Geri kalan kod, sadece tabloyu daha kullanıcı dostu yapmak için tabloyu isteğimize göre uyarlar.

Seçme modu(selection mode) kullanıcının neyi seçebileceğini belirtir; burada, hücreleri (alanları) seçilebilir yaptık. Bu seçim genellikle seçili hücrenin etrafında kesikli çizgi ile gösterilir. Seçme davranışı(selection behavior) seçimlerin görsel olarak nasıl işleneceğini belirtir; bu örnekte, tüm satır görselleştirilecektir. Bu seçim genellikle farklı bir arkaplan rengi kullanılarak gösterilir. ID sütununu gizlemeyi seçtik, çünkü ID'ler kullanıcı için pek anlamlı değildirler. Ayrıca, tablo görüntülemesini saltokunur yapmak için `NoEditTriggers`'ı ayarlarız.

Saltokunur tablolar sunmak için bir alternatif de `QSqlTableModel`'ın ana sınıfı `QSqlQueryModel`'ı kullanmaktır. Bu sınıf, `setQuery()` fonksiyonunu sağlar, böylece kompleks SQL sorguları yazmak mümkün olur.

Scooters veritabanından farklı olarak, çoğu veritabanı, birçok tabloya ve dış anahtar ilişkilerine(foreign key relationships) sahiptir. Qt, dış anahtarlar ile tabloları görüntülemede ve düzenlemede kullanılabilen, `QSqlTableModel`'ın bir altsınıfı olan `QSqlRelationalTableModel`'ı sağlar. Bir `QSqlRelationalTableModel`, bu Modele her bir dış anahtar için bir `QSqlRelation` ekleyebilmemiz dışında, bir `QSqlTableModel`'a çok benzer.

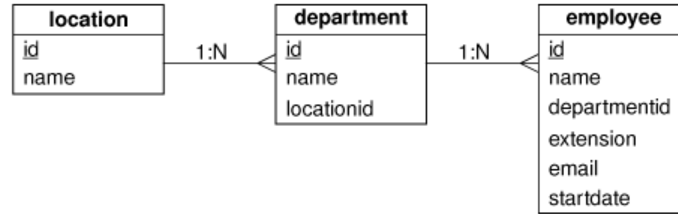
Formları Kullanarak Kayıtları Düzenleme

Bu kısımda, bir seferde bir kayıt görüntüleyen bir diyalog form oluşturmayı göreceğiz. Diyalog, kayıt eklemede, silmede ve düzenlemede ve bir tablo içindeki tüm kayıtlar boyunca dolaşmada kullanılabilir.

Bu konseptleri Staff Manager uygulaması kapsamında örneklendireceğiz. Uygulama, hangi departman personelinin(employee) içerde olduğunu takip eder, departmanların nerede olduklarını ve personelin dâhili telefon kodları gibi personel hakkındaki bazı temel bilgileri saklar. Uygulama aşağıdaki üç tabloyu kullanır:

```
CREATE TABLE location (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL));

CREATE TABLE department (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    locationid INTEGER NOT NULL,
    FOREIGN KEY (locationid) REFERENCES location));

CREATE TABLE employee (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    departmentid INTEGER NOT NULL,
    extension INTEGER NOT NULL,
    email VARCHAR(40) NOT NULL,
    startdate DATE NOT NULL,
    FOREIGN KEY (departmentid) REFERENCES department));
```

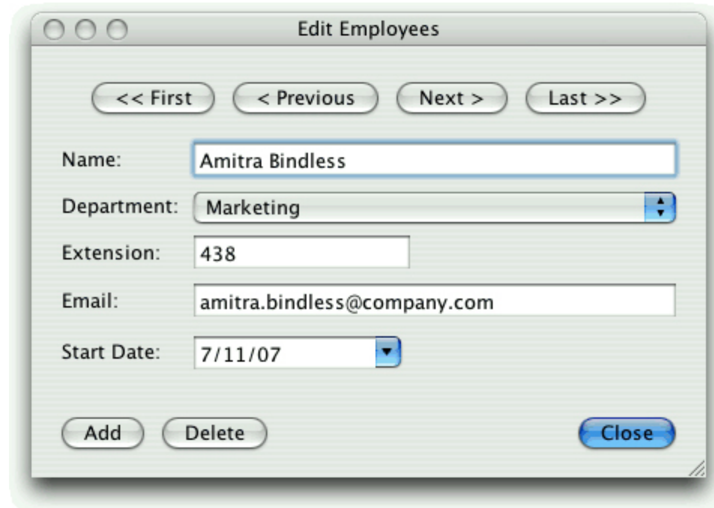
Tablolar ve onların ilişkileri Şekil 12.2'de şematik olarak gösteriliyor. Her bir mevki(location) herhangi bir sayıda departmana sahip olabilir, ve her bir departman herhangi bir sayıda personele sahip olabilir. Dış anahtarlar için belirtilen sözdizimi SQLite 3 içindir ve belki diğer veritabanları için farklı olabilir.



Şekil 12.2

Bu kısımda, personeli yönetmek için kullanılan `EmployeeForm` diyaloğunun üzerine odaklanacağız. Bir sonraki kısımda, departmanların ve personelin master-detail görünüşünü sağlayan `MainForm` ile ilgileneceğiz.

Form çağrıldığında, eğer geçerli bir personel ID'si verilmişse, onu; aksi halde, ilk personeli gösterir. (Form, Şekil 12.3'te gösteriliyor.) Kullanıcılar, personel kayıtları boyunca dolaşabilir, personel bilgilerini düzenleyebilir, var olan personelleri silebilir ya da yeni personeller ekleyebilirler.



Şekil 12.3

Aşağıdaki enum'ı, `employeeform.h` içinde, sütun indekslerine anlamlı isimler vermek için tanımladık:

```
enum {
    Employee_Id = 0,
    Employee_Name = 1,
    Employee_DepartmentId = 2,
    Employee_Extension = 3,
    Employee_Email = 4,
    Employee_StartDate = 5
};
```

Başlık dosyasının kalanında `EmployeeForm` sınıfı tanımlanır:

```
class EmployeeForm : public QDialog
{
    Q_OBJECT
public:
    EmployeeForm(int id, QWidget *parent = 0);

    void done(int result);

private slots:
    void addEmployee();
    void deleteEmployee();

private:
    QSqlRelationalTableModel *tableModel;
    QDataWidgetMapper *mapper;
    QLabel *nameLabel;
    ...
    QDialogButtonBox *buttonBox;
};
```

Veritabanına erişmek için, yalın bir `QSqlTableModel` yerine bir `QSqlRelationalTableModel` kullanırız, çünkü dış anahtarları çözmemiz gerekir. `QDataWidgetMapper`, bir form içindeki parçacıkları uygun sütunlar ile eşleştirme imkânı veren bir sınıftır.

Formun kurucusu oldukça uzundur, bu nedenle onu parçalar halinde ve yerleşim kodu alakasız olduğu için yerleşim kodunu atlayarak inceleyeceğiz.

Kod Görünümü:

```
EmployeeForm::EmployeeForm(int id, QWidget *parent)
    : QDialog(parent)
{
    nameEdit = new QLineEdit;

    nameLabel = new QLabel(tr("Na&me:"));
    nameLabel->setBuddy(nameEdit);

    departmentComboBox = new QComboBox;

    departmentLabel = new QLabel(tr("Depar&tment:"));
    departmentLabel->setBuddy(departmentComboBox);

    extensionLineEdit = new QLineEdit;
    extensionLineEdit->setValidator(new QIntValidator(0, 99999, this));

    extensionLabel = new QLabel(tr("E&xtension:"));
    extensionLabel->setBuddy(extensionLineEdit);

    emailEdit = new QLineEdit;

    emailLabel = new QLabel(tr("&Email:"));
    emailLabel->setBuddy(emailEdit);

    startDateEdit = new QDateEdit;
    startDateEdit->setCalendarPopup(true);
    QDate today = QDate::currentDate();
    startDateEdit->setDateRange(today.addDays(-90), today.addDays(90));

    startDateLabel = new QLabel(tr("&Start Date:"));
    startDateLabel->setBuddy(startDateEdit);
```

Her bir alan için bir düzenleme parçacığı(editing widget) oluşturarak başlarız. Ayrıca, uygun alanları belirlemek için her bir düzenleme parçacığının yanına bir etiket(label) yerleştiririz.

Extension satır editörünün sadece geçerli telefon kodlarını kabul etmesini sağlamak için bir `QIntValidator` kullanırız, bu örnekte sayılar 0-99999 aralığındadır. Ayrıca, Start Date editörü için bir zaman aralığı ayarlarız, ve editörü bir takvim(calendar) sağlamaya ayarlarız. Açılır kutuyu(Combo box) doğrudan doldurmayız; daha sonra ona, kendi kendini doldurabilmesi için bir Model vereceğiz.

```
    firstButton = new QPushButton(tr("<< &First"));
    previousButton = new QPushButton(tr("< &Previous"));
    nextButton = new QPushButton(tr("&Next >"));
    lastButton = new QPushButton(tr("&Last >>"));

    addButton = new QPushButton(tr("&Add"));
    deleteButton = new QPushButton(tr("&Delete"));
    closeButton = new QPushButton(tr("&Close"));

    buttonBox = new QDialogButtonBox;
    buttonBox->addButton(addButton, QDialogButtonBox::ActionRole);
    buttonBox->addButton(deleteButton, QDialogButtonBox::ActionRole);
    buttonBox->addButton(closeButton, QDialogButtonBox::AcceptRole);
```

Diyaloğun üst kısmında, birlikte gruplanmış navigasyon butonları (<< First, < Previous, Next > ve Last >>) oluştururuz. Sonra, diğer butonları (Add, Delete ve Close) oluştururuz ve onları diyaloğun alt kısmına

yerleşmiş bir QDialogButtonBox içine koyarız. Yerleşimleri oluşturan kod basittir, o yüzden onu incelemeyeceğiz.

Buraya kadar, kullanıcı arayüzünün parçacıklarını ayarladık, bu nedenle artık dikkatimizi temel işlevselliğe verebiliriz.

```
tableModel = new QSqlRelationalTableModel(this);
tableModel->setTable("employee");
tableModel->setRelation(Employee_DepartmentId,
                        QSqlRelation("department", "id", "name"));
tableModel->setSort(Employee_Name, Qt::AscendingOrder);
tableModel->select();

QSqlTableModel *relationModel =
        tableModel->relationModel(Employee_DepartmentId);
departmentComboBox->setModel(relationModel);
departmentComboBox->setModelColumn(
        relationModel->fieldIndex("name"));
```

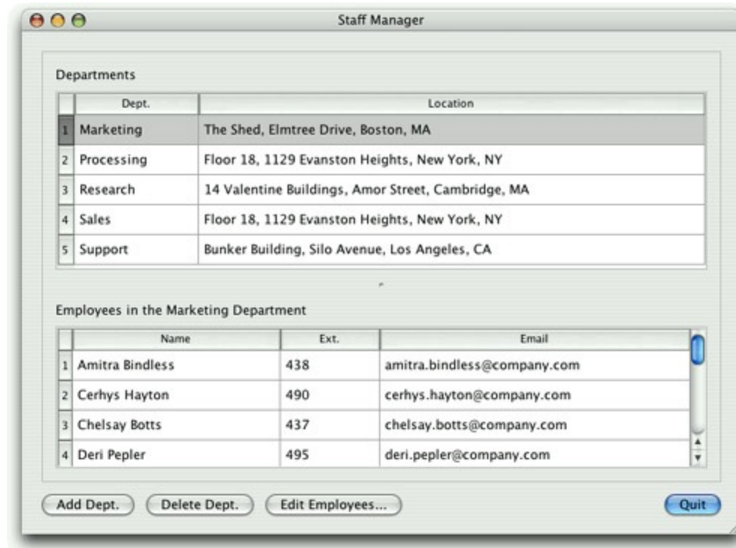
Model, daha önce gördüğümüz QSqlTableModel ile hemen hemen aynı şekilde inşa edilir ve ayarlanır, fakat bu sefer bir QSqlRelationalTableModel kullanırız ve bir dış anahtar ilişkisi ayarlarız. setRelation() fonksiyonu, bir dış anahtar alanının indeksini ve bir QSqlRelation alır. QSqlRelation kurucusu, bir tablo ismi (dış anahtarın tablosu), dış anahtar alanının adını ve dış anahtar alanının değerini temsil ederken görüntüleyeceği alanın adını alır.

Bir QComboBox, bir QListWidget gibi veri öğelerini tutmak için dâhili bir Modele sahiptir. Bu Model yerine kendi Modelimizi yerleştirebiliriz ve burada yaptığımız da tam olarak budur; ona, bir QSqlRelationalTableModel tarafından kullanılan İlişki Modeli(Relation Model) vermek. İlişki, iki sütuna sahiptir, bu nedenle açılır kutunun hangisini göstermesi gerektiğini belirtmeliyiz. İlişki Modeli, setRelation()'ı çağırdığımızda oluşturulmuştu, bu nedenle "name" sütununun indeksini bilmiyoruz. Bu nedenle de açılır kutunun departman isimlerini gösterebilmesi için doğru indekse ulaşmak amacıyla fieldIndex() fonksiyonunu alan ismi ile çağırırız. QSqlRelationalTableModel'a şükürler olsun ki, açılır kutu departman ID'leri yerine departman isimlerini gösterecek.

```
mapper = new QDataWidgetMapper(this);
mapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
mapper->setModel(tableModel);
mapper->setItemDelegate(new QSqlRelationalDelegate(this));
mapper->addMapping(nameEdit, Employee_Name);
mapper->addMapping(departmentComboBox, Employee_DepartmentId);
mapper->addMapping(extensionLineEdit, Employee_Extension);
mapper->addMapping(emailEdit, Employee_Email);
mapper->addMapping(startDateEdit, Employee_StartDate);
```

Veriyi Tablo Biçiminde Formlar İçinde Sunma

Bu kısımda, Staff Maneger uygulamasının, bir master-detail ilişki içinde iki QTableView içeren ana formunu inceleyeceğiz. (Form, Şekil 12.4'te gösteriliyor.) Master Görünüş, departmanların bir listesidir. Detail Görünüş, geçerli departman içindeki personelin bir listesidir. Sundukları iki veritabanı tablosu da dış anahtar alanlarına sahip oldukları için, her iki Görünüş de QSqlRelationalTableModel kullanır. Konu ile ilgili CREATE TABLE ifadeleri bir önceki kısımda gösterilmiştir.



Şekil 12.4

Her zamanki gibi, sütun indekslerine anlamlı isimler vermek için bir enum kullanırız:

```
enum {
    Department_Id = 0,
    Department_Name = 1,
    Department_LocationId = 2
};
```

`MainForm` sınıfının, başlık dosyası içindeki tanımlanmasına bakarak başlayacağız:

```
class MainForm : public QWidget
{
    Q_OBJECT

public:
    MainForm();

private slots:
    void updateEmployeeView();
    void addDepartment();
    void deleteDepartment();
    void editEmployees();

private:
    void createDepartmentPanel();
    void createEmployeePanel();

    QSqlRelationalTableModel *departmentModel;
    QSqlRelationalTableModel *employeeModel;

    QWidget *departmentPanel;
    ...
    QDialogButtonBox *buttonBox;
};
```

Bir master-detail ilişki ayarlamak için, kullanıcının farklı bir kaydı (satır) seçtiğinden emin olmalıyız. Detail tablosunu sadece ilgili kayıtları gösterecek şekilde güncelleriz. Bu, `updateEmployeeView()` private yuvası ile başarılır. Diğer üç yuva, kurucu için yardımcılardır.

Kurucuya ait kodun çoğu kullanıcı arayüzünü oluşturmakla ve uygun sinyal-yuva bağlantılarını ayarlamakla ilgilidir. Biz sadece veritabanı programlama ile ilgili parçalarıyla ilgileneceğiz.

```
MainForm::MainForm()
{
    createDepartmentPanel();
    createEmployeePanel();
```

Kurucu, iki yardımcı fonksiyonu çağırmakla başlar. İlki, departman Modeli ve Görünüşünü oluşturur ve ayarlar, ikincisi, aynılarını personel(employee) Modeli ve Görünüşü için yapar. Bu fonksiyonların konu ile ilgili parçalarına kurucuyu incelemeyi bitirdikten sonra bakacağız.

Kurucunun sıradaki parçası, iki tablo Görünüşünü içeren bir bölücü(splitter) ve formun butonlarını ayarlar. Bunların tümünü atlayacağız.

```
    ...
    connect(addButton, SIGNAL(clicked()), this, SLOT(addDepartment()));
    connect(deleteButton, SIGNAL(clicked()),
            this, SLOT(deleteDepartment()));
    connect(editButton, SIGNAL(clicked()), this, SLOT(editEmployees()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    ...
    departmentView->setCurrentIndex(departmentModel->index(0, 0));
}
```

Butonları diyalog içindeki yuvalara bağlarız ve geçerli öğenin ilk departman olmasını sağlarız.

Böylece kurucuyu görmüş olduk. Şimdi, departman Modelini ve Görünüşünü ayarlayan `create-DepartmanPanel()` yardımcı fonksiyonunun kodlarına bakacağız:

Kod Görünümü:

```
void MainForm::createDepartmentPanel()
{
    departmentPanel = new QWidget;

    departmentModel = new QSqlRelationalTableModel(this);
    departmentModel->setTable("department");
    departmentModel->setRelation(Department_LocationId,
            QSqlRelation("location", "id", "name"));
    departmentModel->setSort(Department_Name, Qt::AscendingOrder);
    departmentModel->setHeaderData(Department_Name, Qt::Horizontal,
                                   tr("Dept."));
    departmentModel->setHeaderData(Department_LocationId,
                                   Qt::Horizontal, tr("Location"));
    departmentModel->select();

    departmentView = new QTableView;
    departmentView->setModel(departmentModel);
    departmentView->setItemDelegate(new QSqlRelationalDelegate(this));
    departmentView->setSelectionMode(
            QAbstractItemView::SingleSelection);
    departmentView->setSelectionBehavior(QAbstractItemView::SelectRows);
    departmentView->setColumnHidden(Department_Id, true);
    departmentView->resizeColumnsToContents();
    departmentView->horizontalHeader()->setStretchLastSection(true);

    departmentLabel = new QLabel(tr("Depar&tments"));
```

```
    departmentLabel->setBuddy(departmentView);

    connect(departmentView->selectionModel(),
            SIGNAL(currentRowChanged(const QModelIndex &,
                                     const QModelIndex &)),
            this, SLOT(updateEmployeeView()));
    ...
}
```

Kod, bir önceki kısımda `employee` tablosu için bir Model ayarlarken gördüğümüz ile benzer bir şekilde başlar. Görünüş, standart bir `QTableView`'dır, fakat bir dış anahtara sahip olduğumuz için `QSqlRelationalDelegate` kullanmalıyız. Böylece dış anahtarın metni Görünüş içinde görünür ve ID yerine bir açılan kutu ile değiştirilebilir.

Kullanıcı için anlamlı olmadığı için departmanın ID alanını gizlemeyi tercih ettik.

Departman Görünüşü, `QAbstractItemView::SingleSelection`'a ayarlanmış kendi seçme moduna sahiptir ve seçme davranışı `QAbstractItemView::SelectRows`'a ayarlanmıştır. Mod ayarı, kullanıcının tablodaki hücreler boyunca dolaşabileceği anlamına, davranış ayarı ise kullanıcı hücreler boyunca dolaşırken tüm satırın vurgulanacağı anlamına gelir.

Görünüşün seçme modelinin `currentRowChanged()` sinyalini, `updateEmployeeView()` yuvasına bağlarız. Bu bağlantı, master-detail ilişkisinin yaptığı iştir, ve personel Görünüşünde daima departman Görünüşünde vurgulanan departmana ait personeli göstermeyi sağlar.

```
void MainForm::createEmployeePanel()
{
    employeePanel = new QWidget;
    employeeModel = new QSqlRelationalTableModel(this);
    employeeModel->setTable("employee");
    employeeModel->setRelation(Employee_DepartmentId,
            QSqlRelation("department", "id", "name"));
    employeeModel->setSort(Employee_Name, Qt::AscendingOrder);
    employeeModel->setHeaderData(Employee_Name, Qt::Horizontal,
                                 tr("Name"));
    employeeModel->setHeaderData(Employee_Extension, Qt::Horizontal,
                                 tr("Ext."));
    employeeModel->setHeaderData(Employee_Email, Qt::Horizontal,
                                 tr("Email"));

    employeeView = new QTableView;
    employeeView->setModel(employeeModel);
    employeeView->setSelectionMode(QAbstractItemView::SingleSelection);
    employeeView->setSelectionBehavior(QAbstractItemView::SelectRows);
    employeeView->setEditTriggers(QAbstractItemView::NoEditTriggers);
    employeeView->horizontalHeader()->setStretchLastSection(true);
    employeeView->setColumnHidden(Employee_Id, true);
    employeeView->setColumnHidden(Employee_DepartmentId, true);
    employeeView->setColumnHidden(Employee_StartDate, true);

    employeeLabel = new QLabel(tr("E&mployees"));
    employeeLabel->setBuddy(employeeView);
    ...
}
```

Personel Görünüşünün düzenleme tetikleri(edit triggers) `QAbstractItemView::NoEditTriggers`'a ayarlanır ve sonuç olarak Görünüş saltokunur yapılır. Bu uygulamada, kullanıcı, bir önceki kısımda geliştirdiğimiz `EmployeeForm`'u çağıran Edit Employees butonuna tıklayarak, personel kaydı ekleyebilir, düzenleyebilir ve silebilir.

Bu sefer, bir değil, üç sütun gizleriz. `id` sütununu gizleriz, çünkü yine kullanıcı için bir anlam ifade etmez. Ayrıca, `departmanid` sütununu da gizleriz, çünkü seçili departman içinde sadece personeller gösterilir. Son olarak, `startdate` sütununu gizleriz, çünkü konu ile az ilgilidir ve zaten Edit Employees butonuna tıklayarak erişilebilir.

```
void MainForm::updateEmployeeView()
{
    QModelIndex index = departmentView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = departmentModel->record(index.row());
        int id = record.value("id").toInt();
        employeeModel->setFilter(QString("departmentid = %1").arg(id));
        employeeLabel->setText(tr("E&mployees in the %1 Department")
                               .arg(record.value("name").toString()));
    } else {
        employeeModel->setFilter("departmentid = -1");
        employeeLabel->setText(tr("E&mployees"));
    }
    employeeModel->select();
    employeeView->horizontalHeader()->setVisible(
            employeeModel->rowCount() > 0);
}
```

Her ne zaman departman değişirse (başlangıç dâhil), bu yuva çağrılır. Eğer geçerli departman uygunsa, fonksiyon, departmanın ID'sine erişir ve personel Modeli üzerinde bir filtre ayarlar. Bu, eşleşen bir departman ID dış anahtarının personellerinin gösterilmesini mecbur kılar. (Bir filtre sadece, `WHERE` anahtar kelimesi olmayan bir `WHERE` cümleciğidir.) Ayrıca, `employee` tablosunun üzerinde gösterilen etiketi, personelin içinde bulunduğu departmanın ismini gösterecek şekilde güncelleriz.

Eğer geçerli departman uygun değilse (mesela veritabanı boşsa), herhangi bir kayıtla eşleşmemesini sağlamak için filtreyi var olmayan bir departman ID'sine ayarlarız.

Sonra, filtreyi Modele uygulamak için Model üstünde `select()`'i çağırırız. Bu, sırayla, Görünüşün kendisini güncellemesiyle sonuçlanacak sinyaller yayacak. Son olarak, personel olup olmamasına bağlı olarak, `employee` tablosunun sütun başlıklarını gösterir ya da gizleriz.

```
void MainForm::addDepartment()
{
    int row = departmentModel->rowCount();
    departmentModel->insertRow(row);
    QModelIndex index = departmentModel->index(row, Department_Name);
    departmentView->setCurrentIndex(index);
    departmentView->edit(index);
}
```

Eğer kullanıcı Add Dept. butonuna tıklarsa, `department` tablosunun sonuna bir satır ekler, bu satırı geçerli satır yapar ve kullanıcı sanki F2'ye basmış ya da ona çift tıklamış gibi, "department name" sütununun düzenlenmesini başlatırız. Eğer bazı varsayılan değerler sağlamamız gerekseydi, `insertRow()` çağrısından hemen sonra `setData()`'yı çağırarak yapabilirdik.

Kendimizi, yeni kayıtlar için eşsiz(unique) anahtarlar oluşturmaya mecbur etmemiz gerekmiyordu, çünkü sütunları ele alırken, bunu bizim için gerçekleştirmesi için otomatik artan(auto-incrementing) bir sütun kullanmıştık. Eğer bu yaklaşım mümkün ya da uygun değilse, Modelin `beforeInsert()` sinyalini bağlayabiliriz. Bu, kullanıcının düzenlemesinden sonra, ekleme, veritabanında yerini almadan hemen önce yayılır. Bu, ID'ler ilave etmek ya da kullanıcı verilerini işlemek için en ideal zamandır. Benzer şekilde, `beforeDelete()` ve `beforeUpdate()` sinyalleri de vardır; bunlar denetim izleri(audit trails) oluşturmak için kullanışlıdırlar.

Kod Görünümü:

```
void MainForm::deleteDepartment()
{
    QModelIndex index = departmentView->currentIndex();
    if (!index.isValid())
        return;

    QSqlDatabase::database().transaction();
    QSqlRecord record = departmentModel->record(index.row());
    int id = record.value(Department_Id).toInt();
    int numEmployees = 0;

    QSqlQuery query(QString("SELECT COUNT(*) FROM employee "
                            "WHERE departmentid = %1").arg(id));
    if (query.next())
        numEmployees = query.value(0).toInt();
    if (numEmployees > 0) {
        int r = QMessageBox::warning(this, tr("Delete Department"),
                    tr("Delete %1 and all its employees?")
                    .arg(record.value(Department_Name).toString()),
                    QMessageBox::Yes | QMessageBox::No);
        if (r == QMessageBox::No) {
            QSqlDatabase::database().rollback();
            return;
        }

        query.exec(QString("DELETE FROM employee "
                           "WHERE departmentid = %1").arg(id));
    }

    departmentModel->removeRow(index.row());
    departmentModel->submitAll();
    QSqlDatabase::database().commit();

    updateEmployeeView();
    departmentView->setFocus();
}
```

Eğer kullanıcı bir departmanı silmek isterse ve eğer departmanda hiç personel yoksa, bir formalite olmaksızın yapmasına izin veririz. Fakat departmanda personel varsa, kullanıcıdan silme işlemini onaylamasını isteriz ve eğer onaylarsa, veritabanının ilişkisel bütünlüğünü sağlamak için basamak basamak silme(cascading delete) yaparız. Bunu başarmak için, en azından SQLite 3 gibi bizi ilişkisel bütünlüğü için zorlamayan veritabanları için, bir etkileşim(transaction) kullanmak zorundayız.

Etkileşim başlatılır başlatılmaz, departman içinde kaç personel olduğunu bulmak için bir sorgu çalıştırırız. Eğer en az bir tane varsa, onay almak için bir mesaj kutusu gösteririz. Eğer kullanıcı hayır derse, etkileşimi keseriz ve geri döneriz. Aksi halde, departmandaki tüm personeli ve departmanın kendini siler ve etkileşimi işleriz.

```
void MainForm::editEmployees()
{
    int employeeId = -1;
    QModelIndex index = employeeView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = employeeModel->record(index.row());
        employeeId = record.value(Employee_Id).toInt();
    }

    EmployeeForm form(employeeId, this);
    form.exec();
    updateEmployeeView();
}
```

`editEmployee()` yuvası, kullanıcı Edit Employee butonuna tıkladığında çağrılır. Geçersiz bir personel ID'si atayarak başlarız. Sonra, eğer mümkünse, bunun üzerine geçerli bir personel ID'si yazarız. Sonra, `EmployeeForm`'u kurarız ve onu gösteririz. Son olarak, değiştirilmiş personeller olabileceğinden, ana formun detail tablosunun Görünüşünün kendisini yenilemesini sağlamak için `updateEmployeeView()` yuvasını çağırırız.