**Sveučilište u Zagrebu**

**PMF - Matematički odjel**

Josip Paić

# Programska biblioteka Qt 4

Diplomski rad

Zagreb, rujan 2010.

**Sveučilište u Zagrebu**

**PMF - Matematički odjel**

Josip Paić

# Programska biblioteka Qt 4

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, rujan 2010.

Ovaj diplomski rad obranjen je dana _____ pred nastavničkim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

# Sadržaj

# 1. Uvod

Qt[1] je višeplatformski radni okvir (*framework*[2]) najčešće korišten za razvoj grafičkih sučelja (GUI[3]), ali i programa namijenjenih za rad u konzoli i na poslužiteljima. Proizveden od strane norveškoga Trølltecha, danas je u vlasništvu Nokia-e (Qt Development Frameworks).

Qt koristi C++ kao standardni programski jezik, iako ga je moguće povezati (*language binding*) i sa mnogim drugim programskim jezicima kao što su: JAVA, C#, Python, Ada, Pascal, Perl, PHP i Ruby. Radi na svim većim platformama, npr. Linux, Mac OS X, MS Windows, Symbian i ima veliku internacionalnu podršku. Od zanimljivijih mogućnosti, ne vezanih za GUI, tu su pristup SQL bazi podataka, XML analiza, podrška za rad s dretvama, podrška za mrežno programiranje i zajednički više-platformski API[4] za baratanje datotekama. Kod izgradnje pojedine aplikacije za određenu platformu, dovoljno je prevesti isti kod na željenoj platformi. Qmake alat za izgradnju (*building tool*) stvara *make* datoteke (*makefiles*) ili .dsp datoteke prikladne za određenu platformu.

U ovom ću se radu najviše bazirati na programskoj biblioteci i elementima vezanima za GUI, pošto C++ ne posjeduje standardnu GUI biblioteku, što mu mnogi zamjeraju kao najveći nedostatak. Što se tiče *frameworka,* za Windows platformu dolaze komercijalne verzije GUI *frameworka* i gotovo uvijek ovise o proizvođaču aplikacija. Ista je stvar i s Mac OS X. S druge strane Linux koristi većinom C orijentirane GUI *frameworkove.* Višeplatformskih alternativa, osim Qt, gotovo da i nema. Upravo zato, postoji mogućnost stavljanja Qt-a u standardnu C++ biblioteku.

U prvom dijelu (poglavalja 2. i 3.) pokušati ću objasniti arhitekturu i samu prirodu Qt objekata te prednosti nad standardnom C++ bibliotekom, ulogu *meta-compilera* i proces izgradnje Qt aplikacije. U drugom se dijelu (poglavlja 4., 5., 6., 7. i 8.) baziram na GUI, odnosno funkcije i klase vezane za pravilno funkcioniranje sučelja kao što su obrada i filtriranje događaja, komunikacija između objekata, elementi sučelja i raspored elemenata, prozori i njihov međusobni odnos te sustav za crtanje koji omogućava vizualni prikaz svih elemenata grafičkog sučelja. U trećem dijelu (poglavlje 10.) obrađujem Model/View programiranje koje određuje prikaz i obradu podataka neke baze ili strukture preko grafičkog sučelja.

Qt još omogućava izradu aplikacija za rad s bazama podataka nezavisnih o platformi. Koristi *drivere* za Oracle, Microsoft, SQL Server, Sybase Adaptive Server, IBM DB2, PostgreSQL, MySql, Borlan Interbase, SQLite i ODBC baze podataka. Sadrži i neke značajke specifične samo za određenu platformu kao što su ActiveX za Windows ili Motif za Unix. Koristi Unicode i ima znatnu podršku za internacionalizaciju. Sadrži alat Qt Linquist, koji omogućava lako prevođenje i prilagođavanje aplikacije. Aplikacije mogu jednostavno koristiti i miješati tekstove na raznim jezicima koje podržava Unicode.

1 izgovara se kao engleska riječ "*cute*"- simpatično
2 učestalo korišten dio programske potpore koji implementira generičko rješenje generičkog problema
3 *Graphic User Interface*
4 *Aplication Programming Interface*

Također, Qt ima i razne klase specifične za određeno područje, kao npr. XML() modul koji sadrži SAX[1] i DOM[2] klase za čitanje i rad sa podacima spremljenim u XML[3] formate. Objekti se mogu smjestiti u memoriju koristeći STL[4] kompatibilni skup klasa i baratati s njima koristeći načine korištenja iteratora u Javi i C++ standardnoj biblioteci. Za lokalno ili udaljeno baratanje datotekama koriste se standardni protokoli koje pružaju I/O (*Input/Output*) i mrežne klase. QtScript modul omogućava aplikacijama da koriste jezik baziran na ECMAScriptu sličan JavaScriptu. U svakom trenutku možemo proširiti funkcionalnost pojedine Qt aplikacije pomoću raznih dodataka (*plugins*) i dinamičkih biblioteka. Iako su i to važni dijelovi Qt biblioteke, izlaze iz okvira ovoga rada.

## 1.1. Povijest

Haavard Nord i Eirik Chambe-Eng započeli su razvoj Qt-a 1991. godine. Dobio je ime Qt po uzoru na Xt[5]. Q je uzeto zato što im je izgledalo privlačno u Haavardovom editoru Emacs.

1998. izlazi KDE[6] i postaje jasno da će postati jedan od vodećih *desktop environmenta* za Linux. Kako je KDE bio baziran na Qt-u, *open source* (otvoreni kôd) zajednica se zabrinula za opstanak KDE-a kao softvera otvorenog koda. Zato kreću nastojanja da se to spriječi. Jedna strana stvara Harmony toolkit, odnosno duplicira Qt pod *open source* licencom. Druga strana stvara GNOME[7] Desktop, koji namjerava zamijeniti KDE u cijelosti. GNOME koristi GTK+ toolkit, originalno pisan za GIMP[8], i primarno koristi C programski jezik.

Prve dvije verzije Qt-a bile su Qt/X11 za Unix i Qt/Windows za Windows platformu. Kako je Windows platforma bila dostupna jedino pod komercijalnom, vlasničkom licencom, besplatne *open source* aplikacije pisane u Qt-u za X11 nisu se mogle izvršavati na Windows platformama bez kupnje vlasničkog izdanja. Krajem 2001. Trølltech objavljuje Qt 3.0 s podrškom za Mac Os X platformu. Podrška za Mac OS X je bila dostupna samo u komercijalnoj verziji sve do 2003. kada je objavljena pod GPL licencom.

Kao odgovor na Trolltechovo odbijanje da objavi Qt/Windows pod GPL licencom, 2002. god. članovi "KDE on Cygwin" projekta počeli su modifikaciju Qt/X11 za Windows platformu. Iako projekt nije dostigao kvalitetno izdanje, Trølltech je, ipak, 2005. objavio Qt4/Windows pod GPL licencom.

Nokia kupuje Trølltech 2008. i mijenja ime, prvo u Qt Software, a zatim u Qt Development Frameworks. Od tada razvija Qt kako bi ga pretvorila u glavnu razvojnu platformu za svoje uređaje.

1 *Simple* API *for* XML
2 Document Object Model
3 EXtensible Markup Language
4 Standard Template Library
5 X-w*indow system toolkit*
6 K Desktop Enviroment; igra riječi prema CDE-Common Desktop Enviroment, postojećem grafičkom okruženju
7 GNU Network Object Model Environment
8 GNU Image Manipulation Program

# 2. Qt-ov objektni model

Standardni C++ objektni model omogućuje brzo i vrlo efikasno izvođenje objektnih paradigmi. Ali ponekad je problem njegova statičnost i nefleksibilnost. Programiranje grafičkog korisničkog sučelja je domena koja zahtijeva i vrijeme izvođenja i visoku razinu fleksibilnosti. Qt daje oboje, kombiniranjem brzine C++-a s fleksibilnošću Qt-ovog objektnog modela.

Qt dodaje ove značajke C++-u:

- Vrlo snažan mehanizam za sigurnu komunikaciju između objekata zvanu *signals and slots*
- Raspored i uređivanje objektnih postavki
- Veliku mogućnost baratanja događajima (*events*) i filtriranja događaja
- Kontekstualne prijevode *stringova* za internacionalizaciju
- Sofisticirane mjeritelje vremena (QTimer) koji omogućavaju elegantnu integraciju mnogih zadataka u npr. animiranom grafičkom sučelju
- Hijerarhiju i raspored objektnih stabala koje prirodno organiziraju vlasništvo objekata
- Sigurnije pokazivače (QPointer) koji su automatski postavljeni na 0 kada je objekt na koji pokazuju uništen, za razliku od standardnih C++ pokazivača koji postaju neodređeni kada su njihovi objekti uništeni
- dinamičku eksplicitnu konverziju koja funkcionira preko granica biblioteke.

Mnoge od ovih značajki Qt provodi pomoću standardnih C++ tehnika, nasljeđivanjem QObject klase. Ostale, kao npr. komunikaciju između objekata i sustav dinamičkih svojstava (*dynamic properties system*), zahtijevaju meta-objektni sustav (*Meta-object system*) osiguran od strane Qt-ovog vlastitog meta-objektong prevoditelja, MOC[1]-a.

Meta-objektni sustav je C++ proširenje koje prilagođava jezik pravom komponentnom GUI programiranju. Iako se i predlošci (*templates*) mogu koristiti za proširenje C++-a, meta-objektni sustav osigurava prednosti koristeći standardni C++ koje se ne mogu postići s predlošcima.

## 2.1. Qt-ovi objekti: identitet nasuprot vrijednosti

Neki od gore navedenih značajki Qt-ovoga objektnog modela, zahtijevaju da se Qt objekti zamišljaju kao identiteti, a ne vrijednosti. Vrijednosti su kopirane ili dodijeljene. Identiteti su klonirani. Kloniranje znači stvaranje novog identitet, a ne identično kopiranje starog.

Na primjer, blizanci imaju različite identitete. Oni mogu izgledati sasvim identično, ali

1 *Meta-Object Compiler*

imaju različita imena, različite lokacije, a mogu imati i potpuno različitu društvenu okolinu.

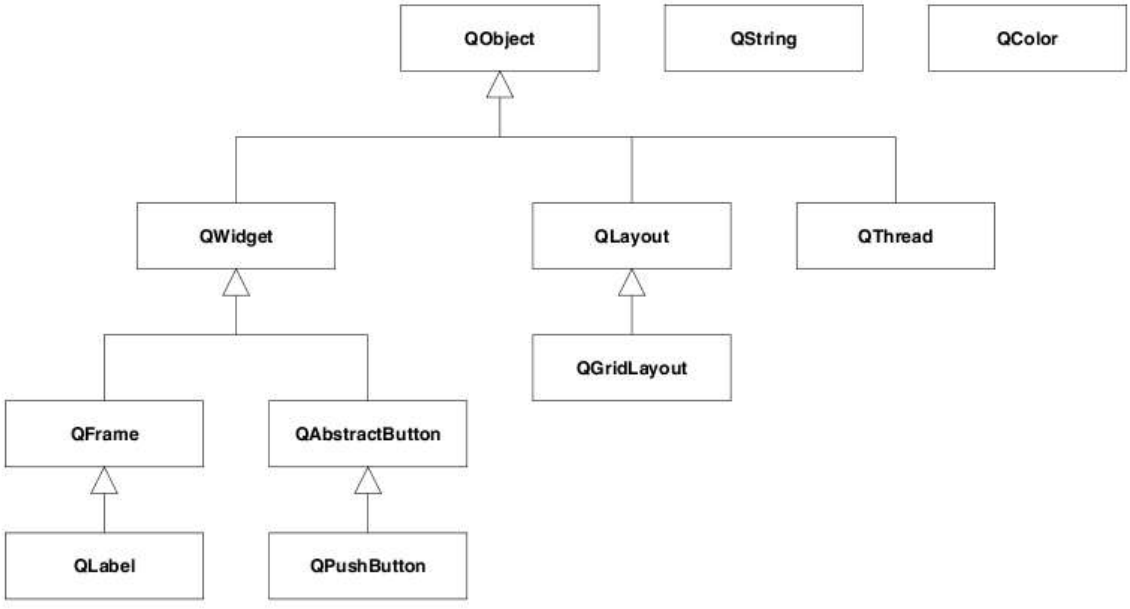
Kloniranje identiteta je mnogo složeniji posao nego što je kopiranje ili dodjeljivanje vrijednosti. Možemo vidjeti što to znači u Qt-ovom objektnom modelu.

Qt objekt:

- može imat jedinstveno ime, QObject::objectName().
- ima mjesto u hijerarhiji objekata.
- može biti povezan s drugim Qt objektima da im emitiraju ili od njih primaju signale.
- mogu imati nova svojstva dodana za vrijeme izvođenja, koja nisu deklarirana u C++ klasi.

Iz tih razloga, Qt objekte treba tretirati kao identitete, a ne kao vrijednosti. Identiteti su klonirani, a ne kopirani ili dodijeljeni i kloniranje identiteta je složeniji od kopiranja ili operacije dodjele vrijednosti. Stoga, QObject i sve podklase QObjecta (direktne ili indirektne) imaju onemogućen svoj *copy*-konstruktor i operator dodjeljivanja.

## 2.2. Hijerarhija nasljeđivanja




*Slika 2-1: Hijerarhija nasljeđivanja*

Slika 2-1 prikazuje mali izvadak iz hijerarhije nasljeđivanja u Qt-u. QLabel ne nasljeđuje samo sva obilježja QObject i QWidget klase, nego i one od QFrame klase. To je osnovna klasa svih *widgeta* koji mogu imati vizualni okvir.

Klasa QAbstractButton je također naslijeđena od QWidget klase. Ona služi kao bazna klasa za sve klase koje prikazuju gumb (QPushButton). Osim QPushButton klase, ona

uključuje i QCheckBox i QRadioButton.

*Layout* klase nalaze se u odvojenoj grani, koja ne vodi natrag u QWidget, a za koje je QLayout bazna klasa. QGridLayout izravno nasljeđuje QLayout, dok su QVBoxLayout i QHBoxLayout izvedene iz QBoxLayout klase.

Klase QFrame, QAbstractButton, QLayout i QBoxLayout koriste se izravno samo u vrlo malo slučajeva. Oni su jednostavno sažetak zajedničkih svojstava i funkcija klasa koje ih proširuju. Klasa QString i QColor, s druge strane, nemaju bazne klase (osim sebe).

Ako je potrebno implementirati vlastiti *widget*, to se obično čini proširenjem QWidget klase.

## 2.3. Objektna stabla i vlasništvo objekata

QObject klase organiziraju se u objektna stabla. Kada stvorite QObject s drugim objektom kao roditeljem, on je dodan u roditeljsku children() listu (listu djece), i obrisan je kada i roditelj. Takav pristup vrlo dobro odgovara potrebama GUI objekata.

Na primjer, QShortcut (tipkovni prečac) je dijete relevantnih prozora pa kada korisnik zatvori prozor, prečac se također briše.

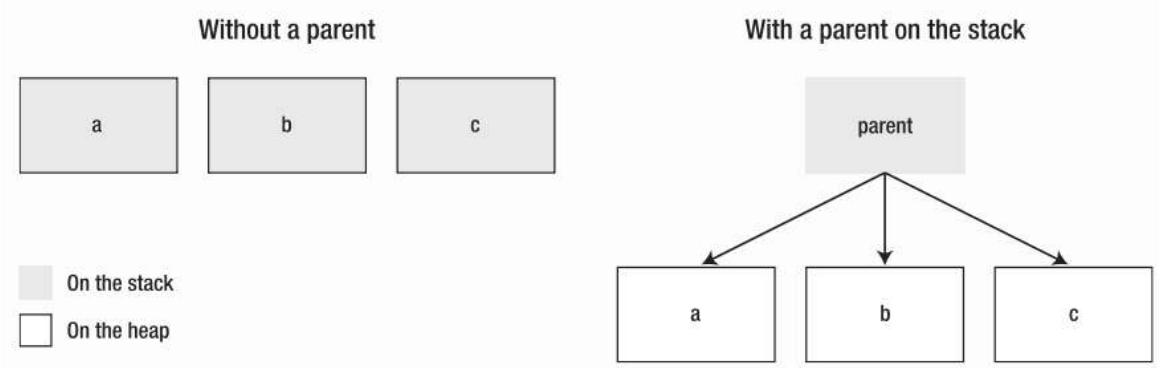
QWidget, osnovna klasa svega što se pojavljuje na ekranu, proširuje odnos roditelj-dijete. Dijete obično postaje dijete *widget*, odnosno prikazano je u roditeljskom koordinatnom sustavu i grafički je određen granicama roditelja. Na primjer, kada program izbriše prozor s porukom, nakon što je on zatvoren, gumb i labela tog prozora također se brišu, kao što smo željeli, jer su gumbi i labele djeca tog prozora.

Također, ako sami izbrišemo dijete, ono će se automatski ukloniti od svojih roditelja.

Na primjer, kada korisnik uklanja alatnu traku (*toolbar*), aplikacija može obrisati jedan od svojih QToolBar objekata. U tom će slučaju QMainWindow, kao roditelj alatne trake, otkriti promjene i prilagoditi svoj prostor u skladu s tim.

### 2.3.1. Raspored izgradnje i uništenja QObjecta

Kad su QObject objekti stvoreni na hrpi (*heap*), tj. kreiran s new, stablo može biti izgrađeno od njih u bilo kojem redoslijedu, a objekti u stablu, kasnije, mogu biti i uništeni u bilo kojem redoslijedu.




*Slika 2-2: Razlika između objekata sa i bez roditelja*

Kada je bilo koji QObject u stablu izbrisan, ako objekt ima roditelja, destruktor automatski uklanja taj objekt iz svojih roditelja. Ako objekt ima djecu, destruktor automatski briše svako dijete. Nijedan QObject nije izbrisana dva puta, bez obzira na poredak uništenja.

Isto se događa i kada se QObjecti stvore na stogu (*stack*). Tada, u principu, raspored uništavanja još uvijek ne predstavlja problem.

Ipak, postoji nedostatak:

```
int main()
{
      QWidget prozor;
      QPushButton ugasi("Quit", &prozor);
      ...
}
```

I *prozor* i *ugasi* (roditelj i dijete) su oboje QObjecti. QPushButton nasljeđuje QWidget, a QWidget nasljeđuje QObject. Kod je ispravan: destruktor od *ugasi* ne zove se dva puta zbog C++ standarda (ISO / IEC 14882:2003) koji navodi da se destruktori lokalnih objekata pozivaju u obrnutom redoslijedu od njihovih konstruktora. Stoga se destruktor od djeteta *ugasi* zove prvi, te sebe uklanja iz svojeg roditelja *prozor* prije nego je pozvan destruktor od *prozora*.

Problem se javlja ako zamijenimo redoslijed konstruktora kao u sljedećem primjeru:

```
int main (){
      QPushButton ugasi("Quit");
      QWidget prozor;

      ugasi.setParent (&prozor);
      ...
}
```

U ovom slučaju redoslijed uništavanja uzrokuje problem. Destruktor roditelja je pozvan prvi, pošto je stvoren posljednji. Zatim se poziva destruktor od ugasi, njegova djeteta, što je pogrešno jer je ugasi lokalna varijabla. Kada ugasi kasnije ode van dosega, njegov destruktor se ponovo zove, ovaj put ispravno, ali šteta je već učinjena.

### 2.3.2. Upravljanje memorijom

C++ ne nudi automatsko upravljanje memorijom. Općenito, programer aplikacije sam mora voditi brigu o tome. Međutim, Qt može preuzeti dio posla. Objekti klasa koje su izvedene iz QObject klase čine strukturu stabla kao što je objašnjeno u prethodnom odjeljku. Objekti mogu držati djecu objekte. Ako se takav objekt obriše, Qt automatski briše i svu djecu objekte, a ta djeca opet brišu svoje "potomstvo" i tako dalje.

Drugim riječima, ako je korijen objektnog stabla izbrisan, Qt automatski briše cijelo stablo. To oslobađa programera potrebe praćenja potomaka objekta i oslobađa memoriju koju zauzima. Međutim, kako bi to automatsko upravljanje memorijom funkcioniralo, sva djeca (i djeca djece, i...) moraju ležati na hrpi, odnosno, moraju biti kreirana pomoću *new*. Objekti koji su stvoreni korištenjem *new* određeni su pokazivačem na hrpi.

Ne stavljanje objekata na hrpu česta je pogreška početnika. *Widgeti* koji su stvoreni samo na stogu, na primjer, u konstruktoru klase, brišu se nakon obrade. Iako aplikacija nakratko generira *widget*, on nikada nije vidljiv.

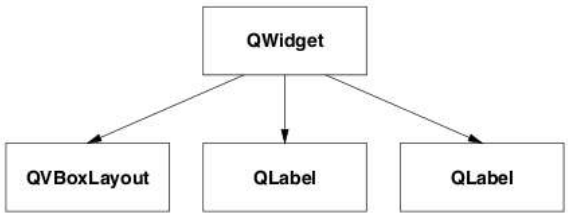
Za kreiranje *layouta* radije koristimo

```
QVBoxLayout* mainLayout = new QVBoxLayout(&window);
```

nego

```
QVBoxLayout mainLayout(&window);
```

Ta deklaracija ne stvara samo QVBoxLayout objekt, nego ga postavlja i kao dijete QWidget objekta, pomoću konstruktora QVBoxLayout klase. Nasuprot tome, ako stvorimo dvije labele, one u početku nemaju roditelja. QLabel konstruktor inicijalizira samo tekst:



*Slika 2-3*

```
QLabel* label1 = new QLabel("One");
```

Tada se koristi QVBoxLayout::addWidget() funkcija kako bi se osiguralo da QWidget objekt pretpostavlja porijeklo svake nove labele[1]. Time se stvara struktura stabla prikazana na slici 2-3. *Layout* objekt i dvije labele, koje su podobjekti prozora kreiranog preko QWidgeta, moraju biti stvoreni na hrpi pomoću *new*. S druge strane, *widget* se može stvoriti i na stogu, tako da ga ne moramo ručno brisati kada program završi izvođenje[2]. Dakle, u većini slučajeva, QObject objekte trebalo bi stvarati na hrpi korištenjem *new*.

1 U stvari, GUI elementi sadržani u *widgetu* moraju biti djeca tom *widgetu*. Iz tog razloga, QWidget objekt postaje roditelj QLabel objekta, a ne QVBoxLayout objekta, kao što bi se moglo pretpostaviti

2 To se može učiniti samo sa objektima koji nemaju roditelja

# 3. Meta-objektni sustav

Meta-objektni sustav temelji se na tri stvari:

1. QObject klasa je baza za objekte koji mogu koristiti meta-objektni sustav.
2. Q_OBJECT makronaredba, unutar privatnog dijela deklaracije klase, koristi se za uključivanje značajki meta-objektnog sustava, kao što su dinamička svojstva, signali i slotovi.
3. Meta-objektni prevoditelj (MOC) svakoj QObject podklasi dodaje potreban kod za provedbu meta-objektnih značajki.

MOC alat čita C++ *source* datoteku. Ako se utvrdi jedna ili više deklaracija klase koje sadrže Q_OBJECT makro, stvara još jednu C++ *source* datoteku koja sadrži meta-objektni kod za svaku od tih klasa. Ta se generirana *source* datoteka u klasu može ili uključiti pomoću #include ili, češće, prevesti i povezati (*compile and link*).

Pored mogućnosti korištenja *signals and slots* mehanizma za komunikaciju između objekata (glavni razlog za uvođenje sustava), meta-objektni kod pruža i sljedeće mogućnosti:

- QObject::metaObject() vraća meta-objekt za određenu klasu.
- QMetaObject::className() vraća ime klase kao *string* za vrijeme izvršavanja programa, bez zahtjeva za RTTI[1] podrškom.
- QObject::inherits() funkcija vraća (bool) da li je objekt instanca klase koja nasljeđuje određenu klasu čije je ime zadano kao parametar funkcije.
- QObject::tr() i QObject::trUtf8() prevode *stringove* za internacionalizaciju.
- QObject::setProperty() i QObject::property() dinamički postavljaju i traže svojstva prema imenu.
- QMetaObject::newInstance() konstruira novu instancu klase.

Na QObject klasama je moguće izvesti i dinamičku eksplicitnu konverziju (dynamic cast) pomoću qobject_cast(). Operator qobject_cast() ponaša se slično kao i standardni C++ dynamic_cast() operator, s prednosti da on ne zahtijeva RTTI podršku. Operator pokušava konvertirati svoj argument u pokazivač ili referencu tipa navedenog u kutnim zagradama (< >) i vraća ne-null pokazivač ako se radi o kompatibilnom tipu (koji je određen za vrijeme izvođenja) ili null ako vrsta objekta nije kompatibilna.

Na primjer, pretpostavimo da MojWidget nasljeđuje QWidget te da je deklariran s Q_OBJECT makrom:

```
QObject *obj = new MojWidget;
```

Varijabla obj tipa QObject* pokazuje na MojWidget objekt, stoga je možemo eksplicitno konvertirati na odgovarajući način:

1 *Run-Time Type Information*

```
QWidget *widget = qobject_cast<QWidget *>(obj);
```

Konverzija od QObject u QWidget je uspješna, jer je objekt zapravo MojWidget koji proširuje QWidget. Budući da znamo da je obj klase MojWidget, možemo ga konvertirati i direktno u MojWidget*:

```
MojWidget *mojWidget = qobject_cast<MojWidget *>(obj);
```

Konverzija u MojWidget je uspješna jer qobject_cast() ne razlikuje tipove koji su ugrađeni u Qt biblioteku od posebno napravljenih tipova, kao što je MojWidget te ih točno prepoznaje.

```
QLabel *labela = qobject_cast<QLabel *>(obj);  // labela je 0
```

S druge strane, konverzija u QLabel ne uspijeva, pošto MojWidget ne proširuje QLabel. Pointer je tada je postavljen na 0. To pomaže da se objekti različitih tipova obrađuju različito za vrijeme izvršavanja programa:

```
if (QLabel *labela = qobject_cast<QLabel *>(obj)){
      labela->setText(tr("Ping"));
}
else if (QPushButton *button = qobject_cast<QPushButton *>(obj)){
      button->setText(tr("Pong!"));
}
```

Iako je moguće koristiti QObject kao baznu klasu bez Q_OBJECT makronaredbe i bez meta-objektnog koda, nijedne značajke opisane ovdje neće biti dostupane, kao ni *signals and slots* mehanizam. Iz gledišta meta-objektnog sustava, podklasa QObjecta bez meta koda jednaka je svom najbližem pretku s meta-objektnim kodom. To znači, na primjer, da QMetaObject::className() neće vratiti stvarni naziv klase, već ime klase tog pretka.

Stoga je iznimno preporučljivo da sve podklase QObjecta koristite Q_OBJECT makro bez obzira da li koriste signale, slotove i ostala svojstva.

## 3.1. Korištenje meta-objektnog prevoditelja (MOC)

Meta-objektni prevoditelj, MOC, je program koji barata Qt-ovim C++ ekstenzijama.

MOC alat čita C++ header datoteku. Ako nađe deklaracije klasa koje sadrže Q_OBJECT makro, kreira C++ *source* datoteku koja sadrži C++ kod za te klase. Između ostalog, meta-objektni kod je potreban za *signals and slots* mehanizam, RTTI i sustav dinamičkih svojstava.

C++ *source* datoteka generirana od strane MOC-a mora biti prevedena i povezana s implementacijom klase. Ako se koristi qmake za stvaranje make datoteka, ako treba, pravila izgradnje koja pozivaju MOC će biti uključena, tako da se ne mora koristiti MOC izravno.

Korištenje MOC-a je najčešće korišten sa ulaznom daotekom koja sadrži deklaraciju klase poput ove:

```
class MyClass : public QObject{
      Q_OBJECT

public:
      MyClass(QObject *parent = 0);
      ~MyClass();

signals:
      void mySignal();

public slots:
      void mySlot();
};
```

Uz *signals and slots* mehanizam, MOC, također, implementira objektna svojstva kao u sljedećem primjeru. Q_PROPERTY() makronaredba deklarira objektno svojstvo, dok Q_ENUMS() deklarira listu enumeratora tipova unutar klase da bi se mogli koristiti u sustavu svojstava (property system).
U sljedećem primjeru, deklariramo vrstu enumeratora Priority sa pripadajućim get i set funkcijama priority() i setPriority().

```
class MyClass : public QObject{
      Q_OBJECT
      Q_PROPERTY(Priority priority READ priority WRITE setPriority)
      Q_ENUMS(Priority)

public:
      enum Priority { High, Low, VeryHigh, VeryLow };
      MyClass(QObject *parent = 0);
      ~MyClass();

      void setPriority(Priority priority);
      Priority priority() const;
};
```

Q_FLAGS() makro deklarira enumeratore koji će biti korišteni kao zastavice (flags). Makro Q_CLASSINFO() dozvoljava da se meta-objektnoj klasi doda par ime/vrijednost (name/value).

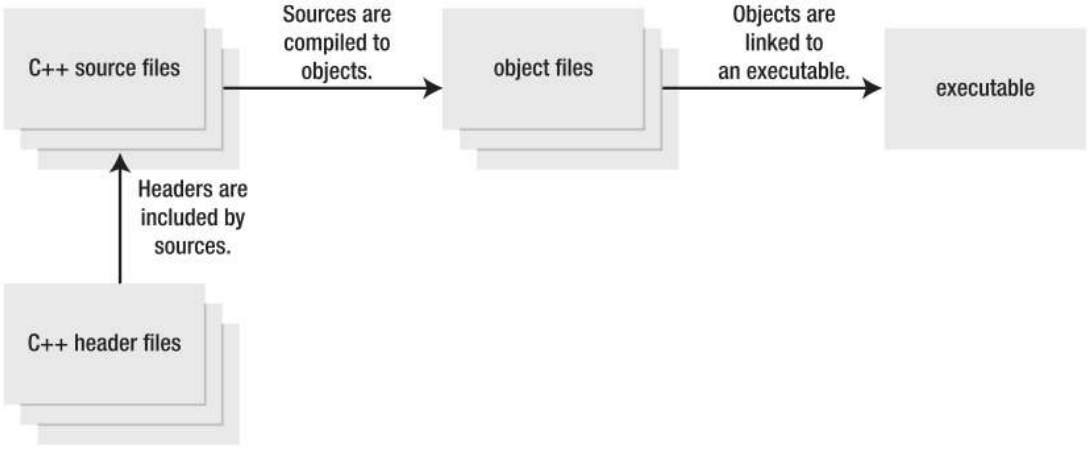
```
class MyClass : public QObject{
      Q_OBJECT
      Q_CLASSINFO("Author", "Oscar Peterson")
      Q_CLASSINFO("Status", "Active")

public:
      MyClass(QObject *parent = 0);
      ~MyClass();
};
```

Izlazne datoteke koje generira MOC moraju se prevoditi i povezati kao i drugi C++ kod. U suprotnom, prevođenje će biti neuspješno u završnoj fazi. Ako se koristi qmake, to se obavlja automatski.

Ukoliko se deklaracija klase nalazi u datoteci myclass.h, MOC generira datoteku zvanu moc_myclass.cpp. Ova datoteka se zatim treba prevesti kao i obično, što kreira objektnu datoteku, npr., moc_myclass.obj na Windows platformi. Taj objekt tada treba uključiti u popis objektnih datoteka koje su povezane zajedno u završnoj fazi izgradnje programa.

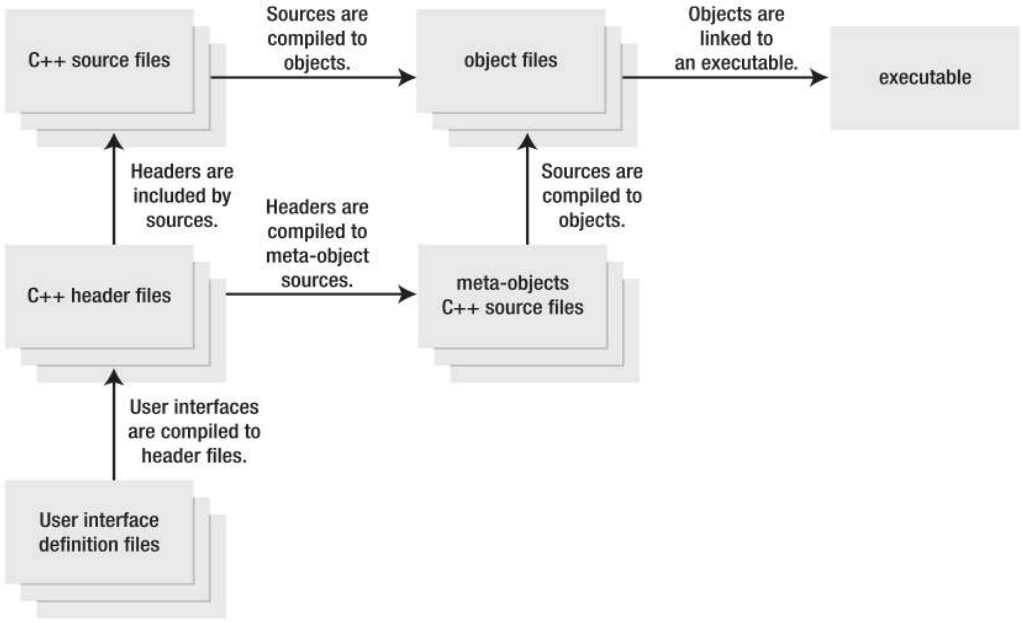
## 3.2. Proces izgradnje

Jedan od najvažnijih razloga korištenja QMake alata je dobivanje neovisnosti o platformi. Još jedan veliki razlog je da QMake generira meta-objekte te ih uključuje u konačnu aplikaciju.




*Slika 3-1: Izgradnja standardnog C++ projekta*

Kod korištenja QMake alata, sve header datoteke su pregledane pomoću meta-objektnog prevoditelja MOC-a. MOC traži klase koje sadrže Q_OBJECT i generira meta-objekte za te klase. Generirani meta-objekti su tada automatski povezani u konačni program.




*Slika 3-2: Izgradnja Qt projekta*

# 4. Događaji i filtriranje događaja

Događaji (*events*) su u Qt-u objekti, izvedeni iz apstraktne klase QEvent, koji predstavljaju stvari koje su se dogodile bilo unutar programa ili kao rezultat vanjske aktivnosti o kojoj aplikacija treba znati. Događaji se mogu primati i obrađivati od bilo koje instance QObject podklase, ali su posebno važni za *widgete*.

Kada se neki događaj dogodi, Qt kreira *event* objekt da ga predstavlja konstruirajući instancu odgovarajuće QEvent podklase i dostavlja je određenoj instanci QObjecta (ili jednoj od njegovih podklasa) pozivajući svoju event() funkciju.

Ta funkcija ne obrađuje sam događaj, već na temelju tipa dostavljenog događaja, ona poziva *event handler* za taj određeni tip događaja i šalje odgovor da li je događaj bio prihvaćen ili ignoriran.

Neki događaji, kao što su QMouseEvent i QKeyEvent, dolaze od sustava za prozore (*window system*), dok neki drugi, kao što je QTimerEvent, dolaze iz drugih izvora.

## 4.1. Tipovi događaja

Vrste događaja većinom imaju posebne klase, posebice QResizeEvent, QPaintEvent, QMouseEvent, QKeyEvent i QCloseEvent. Svaka od tih klasa proširuje QEvent i dodaje funkcije specifične za taj događaj. Na primjer, QResizeEvent dodaje size() i oldSize() kako bi omogućila da *widgeti* otkriju promijene njihovih dimenzija.

Neke klase podržavaju i više nego samo jedan stvarni tip događaja. QMouseEvent podržava pritisak na tipku miša, dupli-klik, pomak miša i druge srodne aktivnosti.

Svaki događaj ima tip koji ga povezuje, definiran u QEvent::Type enumeratoru, a to može biti iskorišteno kao zgodan način kako bi se u realnom vremenu utvrdilo od koje je podklase dati objekt događaja konstruiran.

Budući da programi moraju reagirati na različite i složene načine, Qt-ovi mehanizmi za dostavljanje događaja moraju biti izuzetno fleksibilni.

## 4.2. *Event handler*

Uobičajen je način da se događaji isporučuju pozivom virtualne funkcije. Na primjer, QPaintEvent se dohvaća pozivom QWidget::paintEvent(). Ta je virtualna funkcija odgovorna za reagiranje na odgovarajući način, najčešće, radi ponovnog iscrtavanja *widgeta*. Ako se te virtualne funkcije ne reimplementiraju, potrebno je pozvati

implementaciju bazne klase.

Na primjer, sljedeći kod obrađuje klik lijeve tipke miša u prilagođenu *widget* kućicu (*checkbox*), dok se svi ostali klikovi prosljeđuju QCheckBox baznoj klasi:

```
 void MyCheckBox::mousePressEvent(QMouseEvent *event){
      if (event->button() == Qt::LeftButton) {
      // obrada lijeve tipke miša
      }
      else{
            // za ostale tipke odradi zadano
            QCheckBox::mousePressEvent(event);
      }
}
```

Ponekad, ne postoji funkcija za neki specifični događaj, odnosno funkcija nije dovoljna za neki događaj. Najčešći je primjer pritisak na Tab tipku. Normalno, QWidget presretne taj događaj da bi pomaknuo fokus ulaznih elemenata koje korisnik najčešće popunjuje tipkovnicom, ali nekoliko *widgeta* treba tipku Tab za posebne radnje.

Ti objekti mogu reimplementirati QObject::event() funkciju i obraditi svoj događaj prije ili poslije uobičajenog djelovanja funkcije ili mogu zamijeniti funkciju u potpunosti. Funkcija event(), nekog *widgeta* koji interpretira Tab tipku i neki posebno kreiran tip događaja, može izgledati ovako:

```
bool MyWidget::event(QEvent *event){
      if (event->type() == QEvent::KeyPress){
            QKeyEvent *ke = static_cast<QKeyEvent *>(event);
            if (ke->key() == Qt::Key_Tab){
                  // posebno djelovanje tab tipke
                  return true;
            }
      }
      else if (event->type() == MyCustomEventType){
            MyCustomEvent *myEvent = static_cast<MyCustomEvent *>(event);
            // posebno djelovanje za MyCustomType
            return true;
      }
      return QWidget::event(event);
}
```

Na kraju se ipak koristi QWidget::event() za sve slučajeve koji nisu pritisak Tab tipke ili MyCustomEvent. Funkcija event() vraća vrijednost koja ukazuje da li objekt obrađuje taj događaj. True vrijednost sprečava da se događaj pošalje drugim objektima.

## 4.3. Filtriranje događaja

Objekt, ponekad, treba vidjeti, a vjerojatno i presresti, događaje koji su dostavljeni drugom objektu. Na primjer, dijalog često želi filtrirati pritiske na tipke za neke *widgete*, npr. da bi modificirao djelovanje Return tipke.

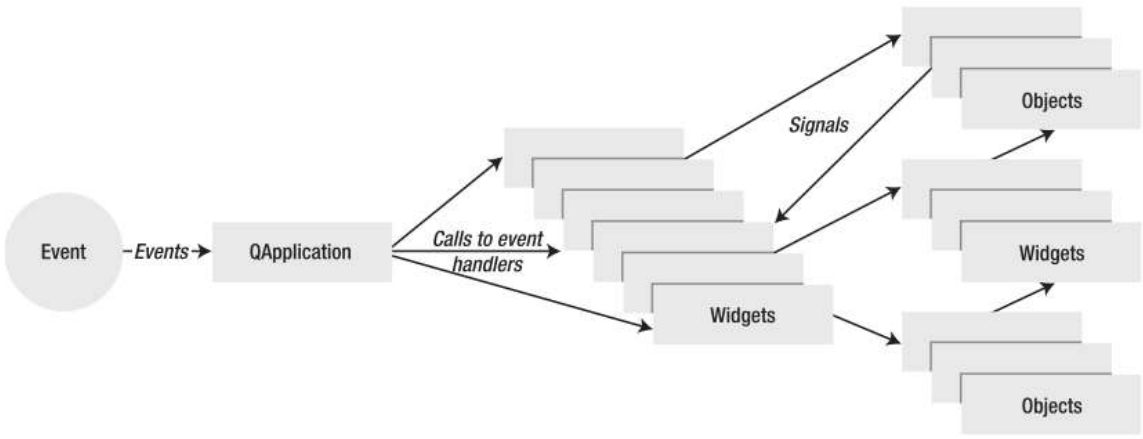
To omogućuje QObject::installEventFilter() funkcija postavljanjem filtra događaja, zbog čega osnovni objekt prima događaje namjenjene nekom drugom objektu u svojoj QObject::eventFilter() funkciji. Filtar prima događaje na obradu prije objekta kojemu su ti događaji namijenjeni, omogućujući mu da ih, po potrebi, pregleda i odbaciti. Postojeći filtar se može ukloniti pomoću QObject::removeEventFilter() funkcije.

Kada je pozvana eventFilter() funkcija filtra, on može prihvatiti ili odbiti događaj i dopustiti ili zabraniti daljnju obradu događaja. Ako svi filtri dopuste obradu događaja (svaki vraća *false*), događaj se šalje ciljanom objektu. Ako jedan od njih zabrani daljnju obradu (vrati *true*), ciljani objekt ili bilo koji naknadni filtar neće vidjeti taj događaj.

```
bool FilterObject::eventFilter(QObject *object, QEvent *event){
      if (object == target && event->type() == Qevent::KeyPress){
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Tab){
                  // posebna obrada tab tipke
                  return true;
            } else
            return false;
            }
      return false;
}
```

Gornji kod pokazuje jedan način presretanja događaja pritiska tipke Tab poslanog određenom ciljanom *widgetu* (`target`). U tom slučaju, filtar obrađuje relevantne događaje i vraća true kako bi zaustavio daljnju obradu. Svi ostali događaji se ignoriraju i filtar vraća false da im dozvoli da budu poslani ciljanom objektu preko bilo kojeg drugog filtra događaja koji je instaliran.

Također je moguće filtrirati sve događaje za cijelu aplikaciju instalacijom filtra događaja na QApplication ili QCoreApplication objektu. Takvi globalni filtri događaja su pozvani prije filtra specifičnih za pojedini objekt. To je vrlo moćan sustav, ali usporava dostavljanje svakog događaja u cijeloj aplikaciji. Općenito, ne treba koristiti takve filtre.




*Slika 4-1: Obrada događaja*

## 4.4. Slanje događaja

Mnoge aplikacije žele stvoriti i poslati svoje vlastite događaje. Događaji se mogu poslati na isti način kao i Qt-ova vlastita petlja događaja (*event loop*) konstruirajući odgovarajuće event objekte i slanjem istih pomoću QCoreApplication::sendEvent() i QCoreApplication::postEvent() funkcija.

Funkcija sendEvent() odmah procesuira događaj. Kada vrati povratnu vrijednost (bool), događaj je već obrađen od strane filtra i/ili samog objekta. Za mnoge klase događaja (*event classes*) postoji funkcija isAccepted() koja govori da li je događaj bio prihvaćen ili odbijen od posljednjeg objekta koji je događaj obrađivao.

Funkcija postEvent() stavlja događaj u red za kasnije slanje. Sljedeći put, kada se pokrene, Qt-ova glavna petlja događaja pošalje, uz optimizaciju, sve događaje spremljene u redu. Na primjer, ako postoji više događaja vezanih za promjenu veličine (*resize events*), oni su komprimirani u jedan. Isto vrijedi i događaje crtanja *(paint events*). QWidget::update() poziva postEvent(), što eliminira titranje i povećava brzinu izbjegavanjem višestrukih ponovnih iscrtavanja. Ta se funkcija koristi i za vrijeme inicijalizacije objekata, pošto bi, uobičajeno, spremljeni događaji bili poslani vrlo brzo nakon što je inicijalizacija objekta završena.

Kod implementacije *widgeta*, važno je shvatiti da se događaji mogu isporučiti rano u njihovom životnom ciklusu, već u njihovom konstruktoru. Zato varijable treba inicijalizirati vrlo rano, prije nego što postoji šansa da se primi određeni događaj.

Da bi se stvorili događaji prilagođenih tipova, potrebno je definirati broj događaja, koji mora biti veći od QEvent::User i po potrebi proširiti QEvent kako bi slali određene informacije o tom događaju.

# 5. *Signals and slots* mehanizam

*Signals and slots* mehanizam koristi se za komunikaciju između objekata. To je centralno obilježje Qt-a, a vjerojatno, i dio koji se ponajviše razlikuje od mogućnosti koje pružaju ostali radni okviri.

Kada u GUI programmiranju napravimo neku promjenu na jednom *widgetu*, vrlo često želimo da o toj promjeni zna i neki drugi *widget*. Općenito, želimo da objekti bilo koje vrste mogu međusobno komunicirati. Na primjer, ako korisnik klikne na gumb Close, vjerojatno želi da se pozove funkcija close() nekog prozora.

Stariji alati obavljali su ovu vrstu komunikacije pomoću tehnike zvane *callback*. *Callback* je ustvari pokazivač na funkciju, pa ako želimo da nas funkcija koja se izvodi obavijesti o nekom događaju, proslijedimo joj pokazivač na drugu funkciju (*callback*). Tada, funkcija koja se izvodi poziva *callback* funkciju kada je potrebno.

*Callback* ima dva nedostatka:

1. nije *type-safe*: ne možemo biti sigurni da će funkcija koja se izvodi pozvati *callback* funkciju s točnim argumentima.
2. *callback* je ovisan o funkciji koja se izvodi, pošto ona mora znati koji *callback* treba pozvati.

U Qt-u imamo alternativu *callback* metodi: koristimo *signals and slots* mehanizam. Kada se dogodi određeni događaj, emitira se signal. Qt-ovi *widgeti* imaju mnoge unaprijed definirane signale, ali uvijek možemo proširiti *widget* i dodati mu vlastite signale. Slot (prijemnik, utor) je funkcija koja se poziva kao odgovor na određeni signal. Qt-ovi *widgeti*, također, imaju mnoge unaprijed definirane slotove, ali je uobičajena praksa da se klase proširuju i da im se dodaju vlastiti slotovi koji su potrebni.

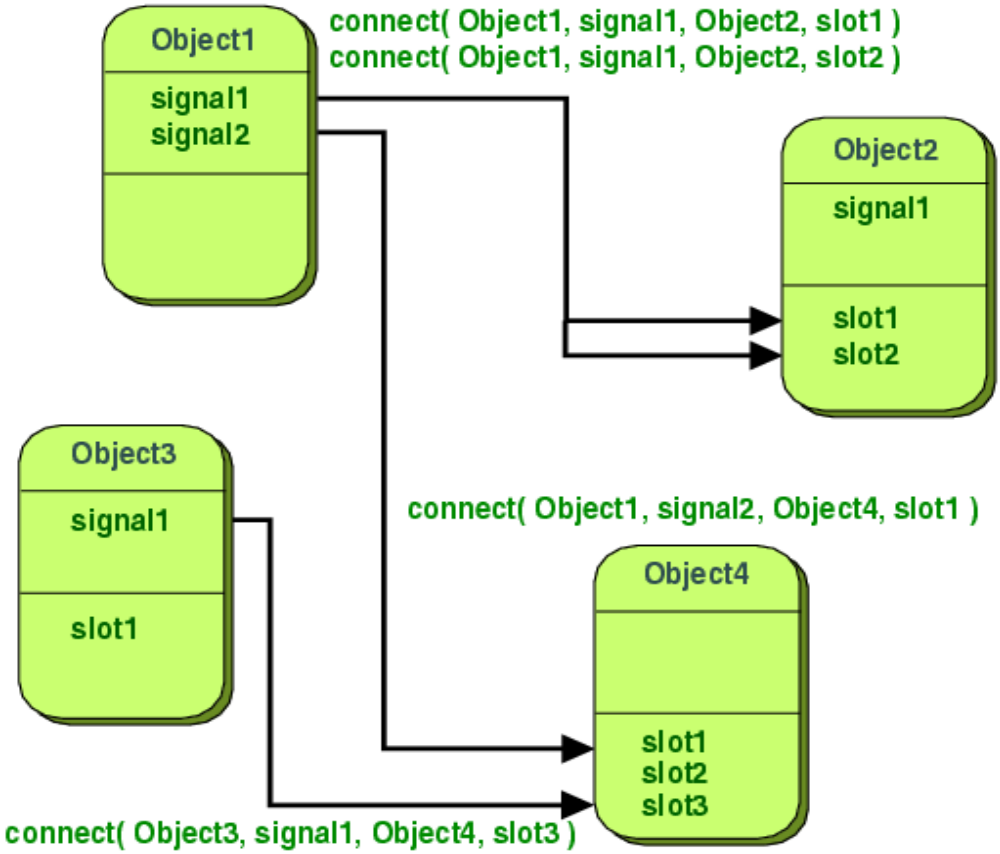
*Signals and slots* mehanizam je *type-safe*: argumenti signala moraju odgovarati argumentima slota. (Zapravo, slot može imati manji broj argumenata od signala kojeg prima, jer slot može ignorirati dodatne argumente.) Pošto su argumenti kompatibilni, prevodilac (kompajler) nam može pomoći otkriti nepoklapanja. Signali i slotovi su potpuno neovisni: klasa, koja emitira signal ne zna, a u principu joj to nije ni važno, tko prima signal. Taj Qt-ov mehanizam osigurava da se slot pozove u pravo vrijeme sa parametrima signala. Signali i slotovi mogu imati bilo koliko argumenata bilo koje vrste.

Sve klase koje nasljeđuju QObject ili neku od njegovih podklasa (npr. QWidget) mogu sadržavati signale i slotove. Objekti emitiraju signale kada promijene svoje unutarnje stanje na način koji može biti zanimljiv drugim objektima. To je sve što objekt čini pri komunikaciji i ne zna, a nije mu ni važno, da li neki drugi objekt prima njegov signal. To je potpuna informacijska enkapsulacija koja osigurava korištenje objekta kao softversku komponentu.

Slotovi se mogu koristiti za primanje signala, ali to su, također, normalne funkcije. Kao što objekt ne zna da li netko prima njegov signal, slot ne zna da li je neki signal spojen na

njega. To osigurava potpunu neovisnost komponenti kreiranih pomoću Qt-a.

Na jedan slot može se spojiti signala koliko god se želi, a signal može biti spojen sa onoliko slotova koliko je potrebno. Čak je moguće spojiti signal izravno na drugi signal. (To će se emitirati drugi signal odmah nakon prvoga.)




*Slika 5-1: Signals and slots mehanizam*

Na taj je način, *signals and slots* snažan mehanizam za komponentno programiranje.

**Primjer:** Deklaracija C++ klase može izgledati ovako:

```
class Counter{

      public:
            Counter() { m_value = 0; }

            int value() const { return m_value; }
            void setValue(int value);

      private:
            int m_value;
};
```

Mala klasa bazirana na QObject može izgledati ovako:

```
#include <QObject>

class Counter : public QObject{
      Q_OBJECT

      public:
            Counter() { m_value = 0; }
            int value() const { return m_value; }

      public slots:
            void setValue(int value);

      signals:
            void valueChanged(int newValue);

      private:
            int m_value;
};
```

Verzija koja koristi QObject ima isto unutarnje stanje, te pruža javne metode pristupa stanju, ali osim toga ima i podršku za komponentno programiranje pomoću *signals and slots* mehanizma.

Emitirajući signal valueChanged(int) ta klasa može obavijestiti ostale kada joj se promijeni stanje, ali posjeduje i slot preko kojega može primati signale drugih objekata.

Sve klase koje sadrže *signals and slots* moraju imati Q_OBJECT na početku deklaracije. Također moraju proširivati (direktno ili indirektno) QObject.

Za razliku od signala, slotovi mogu biti implementirani od strane programera. Ovdje je moguća implementacija Counter::setValue() slota:

```
void Counter::setValue(int value){
      if (value != m_value){
            m_value = value;
            emit valueChanged(value);
      }
}
```

Pomoću naredbe emit šalje se signal valueChanged(int) s novom vrijednosti stanja.

U sljedećem su isječku koda stvorena dva Counter objekta i povezan je valueChanged(int) signal prvoga sa setValue(int) slotom drugoga pomoću QObject::connect():

```
      Counter A, B;
      QObject::connect(&A, SIGNAL(valueChanged(int)),&B,
SLOT(setValue(int)));

      A.setValue(12);   // A.value() == 12, b.value() == 12
      B.setValue(48);   // B.value() == 12, b.value() == 48
```

Pozivanjem **A**.setValue(12) **A** emitira signal valueChanged(12) koji prima **B** sa setValue(int) slotom, odnosno poziva se **B**.setValue(12). Tada **B**, također emitira isti

valueChanged(int) signal, ali kako ne postoji slot spojen na taj signal, signal se ignorira.

Treba uočiti da setValue(int) funkcija postavlja vrijednost i emitira signal samo ako je value != m_value. Na taj način sprječavamo beskonačnu petlju u slučaju cikličkih spojeva (što bi se dogodilo da je npr. i **B**.valueChanged(int) povezan s **A**.setValue ()).

Signal se emitira za svaku vezu koja se napravi. Ako se duplicira veza, emitiraju se dva signala. Veza se može uništiti pomoću QObject::disconnect().

Primjer pokazuje da objekti mogu raditi zajedno, a da pritom ne moraju znati nikakve podatke jedni o drugima. Da bi se to omogućilo, objekte samo treba spojiti zajedno, a to se može postići s jednostavnom QObject::connect() funkcijom.

## 5.1. Signali

Signali su emitirani od objekta prilikom promjene njegova unutarnjeg stanja, koja bi na neki način mogla biti zanimljiva ostalim objektima. Samo klasa u kojoj je deklariran signal i njegove podklase mogu emitirati taj signal.

Kada se signal emitira, svi se slotovi spojeni na njega obično izvršavaju odmah, baš kao i normalni pozivi funkcija. Kada se to dogodi, *signals and slots* mehanizam je potpuno neovisan o prikazivanju GUI-a i procesima vezanim uz njega. Slijedno izvršenje kôda koji dolazi nakon emit nastaviti će se kada svi slotovi vrate povratnu vrijednost. Situacija je malo drugačija kada se koristi Qt::QueuedConnection, u tom slučaju, kod koji slijedi iza ključne riječi emit nastaviti će s izvođenjem odmah, a slotovi će se izvršiti naknadno.

Ako je nekoliko slotova povezano na jedan signal, slotovi će se izvršavati jedan za drugim u proizvoljnom redoslijedu nakon što se signal emitira.

Signali se automatski generiraju u MOC-u i ne smiju biti implementirani u .cpp datoteci, tako da programer ne treba brinuti o njihovoj implementaciji, već ih samo deklarira i emitira pomoću naredbe emit na željenom mjestu. Također, ne smiju imati povratni tip (to jest, koriste void).

## 5.2. Slotovi

Slotovi se pozivaju kada se signali spojeni na njih emitiraju. To su normalne C++ funkcije i mogu se pozivati normalno, a njihova jedina posebna značajka je da se signali mogu spojiti na njih.

Budući da su slotovi normalne funkcije, kada su pozvani direktno, slijede normalna C++ pravila. Međutim, kao slotovi, može ih pozivati bilo koja komponenta, bez obzira na razinu pristupa, preko signal-slot veze. To znači da signal emitiran iz instance proizvoljne klase može pozvati privatni slot instance neke nevezane klase.

Također, slotovi se mogu definirati kao virtualni, što može biti vrlo korisno u praksi.

U usporedbi s *callback* metodom, *signals and slots* je malo sporija zbog povećane fleksibilnosti koju ona pruža, iako je ta razlika praktično beznačajna. Općenito, emitiranje signala koji je spojen na neki slot, je oko deset puta sporije nego poziv tog slota izravno s ne-virtualnim pozivima funkcije. To je usporenje potrebno radi pronalaska konektiranog objekta, sigurne iteracije po svim konekcijama (tj. provjeru da, kasnije, tijekom emisije, prijemnici nisu uništeni) i uobičajenog *marshallinga*[1] parametara. Iako deset ne-virtualnih poziva funkcije izgleda mnogo, to je npr. znatno manje nego bilo koja *new* ili *delete* operacija. Čim se koriste *string*, vektor ili lista, operacije koje zahtijevaju *new* ili *delete*, *signals and slots* mehanizam odgovoran je za samo vrlo mali dio cjelovita troška poziva funkcija.

Isto vrijedi kada se god napravi sistemski poziv slota, ili indirektno više od deset poziva funkcije. Na i586-500, može se emitirati oko 2 000 000 signala u sekundi povezanih na jedan slot, odnosno oko 1 200 000 u sekundi povezanih na dva slota. Jednostavnost i fleksibilnost *signals and slots* mehanizma vrijedno je usporenja, koje korisnici neće ni primjetiti.

Druge biblioteke koje definiraju varijable sa imenom *signals* ili *slots* mogu uzrokovati upozorenja i greške prevodioca kada se prevode zajedno s Qt aplikacijom. Kako bi riješio taj problem, koristi se #undef simbol.

## 5.3. Meta-objektne informacije

Meta-objekni prevoditelj (MOC) obrađuje deklaraciju klase u C++ datoteci i generira C++ kod koji inicijalizira meta-objekt. Meta-objekt sadrži imena svih signala i slotova, kao i pokazivače na te funkcije.

Meta-objekt sadrži i dodatne podatke, kao što su ime klase objekta ili da li objekt nasljeđuje određenu klasu. Na primjer, ako želimo provjeriti da li neki *widget* nasljeđuje QAbstractButton:

```
if (widget->inherits("QAbstractButton")){
      QAbstractButton *button = static_cast<QAbstractButton *>(widget);
      button->toggle();
}
```

Tu informaciju meta-objekta također koristi qobject_cast <T> (), koji je sličan QObject::inherits(), ali manje sklon greškama:

```
if(QAbstractButton *button=qobject_cast<QAbstractButton *>(widget))
      button->toggle();
```

Ovdje provjeravamo da li se *widget* može eksplicitno konvertitrati u QAbstratctButton, jer

1 *Marshalling* - proces transformacije memorijske reprezentacije objekta u podatkovni format (sekvence bitova) pogodan za pohranu u datoteku ili memorijski spremnik i slanje preko mreže.

ako može, tada je ili sam QAbstractButton, ili ga nasljeđuje.

**Primjer:**

```
#ifndef LCDNUMBER_H
#define LCDNUMBER_H

#include <QFrame>

class LcdNumber : public QFrame{    //LcdNumber nasljeđuje QObject,
                                    //preko QFramea i QWidgeta.

      Q_OBJECT    //Q_OBJECT makronaredba govori predprocesoru da
                  //deklarira nekoliko funkcija koje implementira MOC.

      public:
      LcdNumber(QWidget *parent = 0);
```

Ako klasa nasljeđuje QWidget gotovo sigurno želimo da ima roditelja u svom konstruktoru i da ga proslijedi do konstruktora bazne klase.

```
      signals:
            void overflow();
```

LcdNumber treba emitirati signal kada je od njega zatraženo da pokaže nemoguću vrijednost.

```
public slots:
      void display(int num);
      void display(double num);
      void display(const QString &str);
      void setHexMode();
      void setDecMode();
      void setOctMode();
      void setBinMode();
      void setSmallDecimalPoint(bool point);
};

#endif
```

Funkciju koju prima, slot  koristi kako bi dobio informaciju o promjeni stanja u drugim *widgetima*. LcdNumber se koristi, kao što to prikazuje kod iznad, za postavljanje broja koji se prikazuje. Budući da je display() dio sučelja klase s ostatkom programa, taj je slot javan. Često se koristi spajanje valueChanged() signala QScrollBara sa display() slotom, tako da LCD broj stalno pokazuje vrijednosti klizača (*scrollbar*).

LcdNumber koristi display() slot za prikaz zadanog broja. Taj je slot preopterećen, a Qt će odabrati odgovarajuću verziju kada se poveže odgovarajući signal.

## 5.4. Signali i slotovi sa zadanim vrijednostima argumenata

Deklaracije signala i slotova mogu sadržavati argumente, a argumenti mogu imati zadane vrijednosti. Na primjer QObject::destroyed():

```
void destroyed(QObject* = 0);
```

Kada je QObject obrisan, emitira QObject::destroyed() signal. Taj je signal potrebno uloviti kada imamo nevažeću referencu na obrisani objekt, kako bi je mogli obrisati. Pogodna deklaracija slota bila bi:

```
void objectDestroyed(QObject* obj = 0);
```

Kako bi povezali signal i slot, koristimo QObject::connect() i SIGNAL() i SLOT() makronaredbe. Ako argumeti imaju zadane vrijednosti argumenata, pravilo je da deklaracija proslijeđena u SIGNAL() ne smije imati manje argumenata nego deklaracija u SLOT() makronaredbi

**Primjer:**

```
connect(sender,SIGNAL(destroyed(QObject*)),this,SLOT(objectDestroyed(Qbje
ct*)));
connect(sender, SIGNAL(destroyed(QObject*)), this,
SLOT(objectDestroyed()));
connect(sender, SIGNAL(destroyed()), this, SLOT(objectDestroyed()));

connect(sender, SIGNAL(destroyed()), this,
SLOT(objectDestroyed(QObject*)));
```

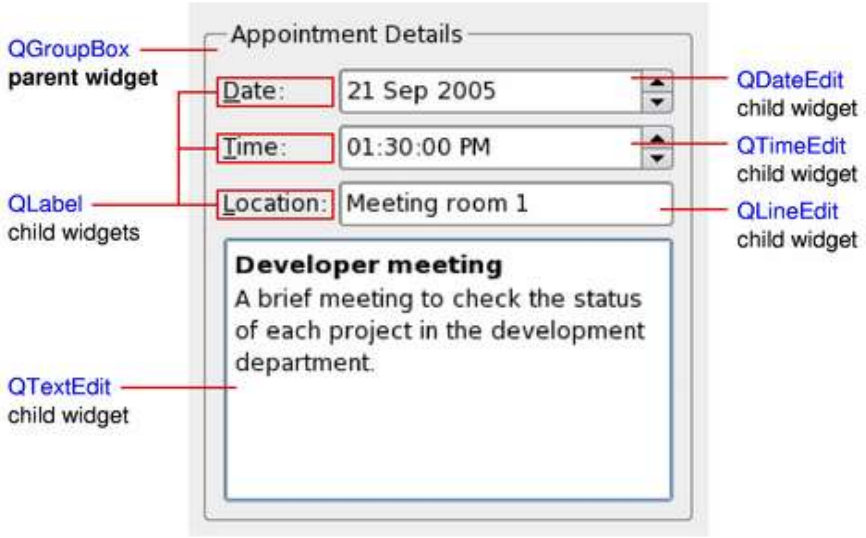
Posljednji connect ne radi, pošto slot očekuje QObject koji signal ne šalje. Ta će veza izbaciti *runtime* grešku.

# 6. *Widgeti* i *Layout Management*

Glavni elementi u kreiranju korisničkih sučelja u Qt-u su *widgeti* i *layout manageri*.

## *6.1. Widgeti*

*Widgeti* mogu prikazivati podatke i informacije o statusu, primaju korisničke akcije i osiguravaju kontejner za druge *widgete* koji bi se trebali zajedno grupirati. *Widget* koji nije ugrađen u roditelja naziva se prozor.




QWidget klasa pruža osnovne mogućnosti za prikaz na ekranu i obrađuje događaje inicirane od korisnika. Svi UI elementi koje Qt daje su podklase QWidget klase, ili se koriste u vezi s QWidget podklasom. Stvaranje prilagođenog *widgeta* obavlja se proširivanje QWidget klase ili odgovarajuće podklase i reimplementiranjem virtualnog *event handlera*.

## 6.2. *Layout Management*

Sustav *Qt layouta* pruža jednostavan i moćan način automatskog uređenja djece *widgeta* osiguravajući da optimalno iskoriste raspoloživi prostor.

Qt uključuje skup *layout* klasa za opisivanje položaja i međusobnog odnosa *widgeta* u korisničkom sučelju aplikacije. Te klase automatski određuju položaj i veličinu *widgeta* prilikom promjene količine dostupnog prostora namjenjenog njima, čime se osigurava dosljedan raspored, a korisničko sučelje kao cjelina ostaje upotrebljivo.

Sve QWidget podklase mogu koristiti *layout* za svoju djecu. Funkcija QWidget::setLayout() koristi se za postavljanje *layouta* nekog *widgeta*. Kada je tako postavljen, *layout* postaje zadužen za sljedeće zadatke:

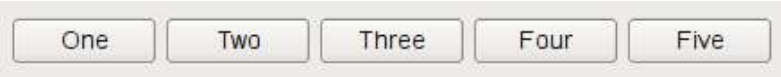
- Pozicioniranje *widgeta* koji su djeca.
- Smisleno određivanje zadane veličine prozora.
- Smisleno određivanje minimalne veličine prozora.
- Promjenu veličine.
- Automatsko ažuriranje pri promjeni sadržaja:
    - veličine slova, teksta ili drugog sadržaja *widgeta*.
    - skrivanje ili prikazivanje *widgeta*.
    - uklanjanje *widgeta*.

Kako bi bile što lakše za razumijevanje i jednostavnije ručno pisanje koda, *layout* klase omogućuju korištenje piksela u svim mjerama i dimenzijama. Kôd generiran za obradu formi koje stvara Qt Designer također korisiti *layout* klase. Qt Designer je korisno koristiti prilikom eksperimentiranja s dizajnom forme, budući da tada ne treba prevoditi, povezivati, niti pokretati aplikaciju kako bi se vidio prikaz, kao što je to uobičajeno kod razvoja.

### 6.2.1. Vrste ugrađenih *layout managera*

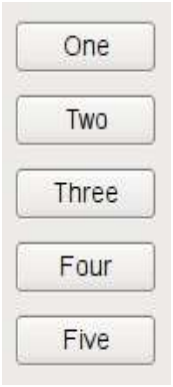
Najlakši način rasporeda *widgeta* je korištenje ugrađenih *layout managera*: QHBoxLayout, QVBoxLayout, QGridLayout i QFormLayout. Ove klase nasljeđuju QLayout, koja nasljeđuje QObject (ne QWidget!). Oni se brinu za upravljanje geometrijom skupa *widgeta*. Za izradu složenijih shema, *layout manageri* se mogu međusobno gnijezditi jedan unutar drugog.

- QHBoxLayout postavlja *widgete* u horizontalnom nizu, slijeva nadesno (ili zdesna nalijevo za pripadajuće jezike).




*Slika 6-1: QHBoxLayout*

- QVBoxLayout postavlja *widgete* u vertikalni niz, od vrha prema dnu.

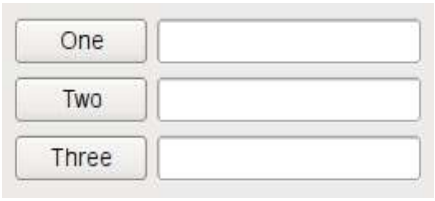



*Slika 6-2: QVBoxLayout*

- QGridLayout postavlja *widgete* u dvodimenzionalnu mrežu. *Widgeti* mogu zauzeti i više ćelija.


*Slika 6-3: QGridLayout*

- QFormLayout postavlja *widgete* u formu od dva stupca tipa labela – unos (*label-input*).




*Slika 6-4: QFormLayout*

### 6.2.2. Postavljanje *layouta*

Sljedeći kod stvara QHBoxLayout koji upravlja geometrijom pet QPushButtona, kao što je prikazano na slici 6-1 gore:

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QHBoxLayout *layout = new QHBoxLayout;
```

```
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
layout->addWidget(button4);
layout->addWidget(button5);

window->setLayout(layout);
window->show();
```

Kod za QVBoxLayout je identičan, osim linije gdje se stvara *layout*, dok je kod za QGridLayout malo drugačiji, jer je potrebno navesti broj retka i stupca:

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);  // (1)
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);

window->setLayout(layout);
window->show();
```

(1) `button3` koristi 2 stupca.To je postavljeno petim argumentom u QGridLayout::addWidget()

Kada se koristi *layout*, pri konstrukciji djeteta ne mora se proslijediti roditelj. *Layout* će automatski odrediti roditeljstvo *widgeta* (interno koristeći QWidget::setParent()), tako da su svi *widgeti layouta* djeca *widgeta* na kojem je postavljen taj *layout*.

**Napomena**: *Widgeti* u *layoutu* su djeca *widgeta* na kojem je postavljen *layout*, a ne od samog *layouta*. *Widgeti* mogu imati samo druge *widgete* kao roditelje, a ne *layout*.

*Layouti* se mogu gnijezditi pomoću addLayout(), unutrašnji *layout* onda postaje dijete *layouta* u koji je umetnut.

### 6.2.3. Dodavanje *widgeta* u *layout*

Kada se *widgeti* dodaju u *layout*, *layout* proces funkcionira na sljedeći način:

1. Svim *widgetima* se u početku dodjeljuje određena količina prostora u skladu s njihovim QWidget::sizePolicy() i QWidget::sizeHint() postavkama.
2. Ukoliko bilo koji od *widgeta* ima postavljenu mogućnost širenja, rastezanja (*stretch factors*), s vrijednosti većom od nule, tada mu se dodijeljuje prostor razmjerno njegovom faktoru rastezanja.

3. Ukoliko bilo koji od *widgeta* ima faktore rastezanja postavljene na nulu, dobiti će više prostora samo ako nema drugih *widgeta* koji žele taj prostor. Od njih, prostor će se prvo dodijeliti *widgetu* sa zadanim QSizePolicy::Expanding svojstvom.
4. Bilo kojem *widgetu* kojemu je dodijeljeno manje prostora od njegove minimalne veličine (ili hinta minimalne veličine, ako niti jedna minimalna veličina nije određena) dodijeljuje se prostor minimalne veličine koju zahtijeva.[1]
5. Bilo koji *widget* kojemu je dodijeljeno više prostora od njegove maksimalne veličine dodijeljuje se maksimalna veličina prostora koju zahtijeva[2].

### 6.2.4. Faktori rastezanja

*Widgeti* se obično stvaraju bez zadanih faktora rastezanja. Kada su položeni u *layout widgetima* su dani udjeli prostora u skladu s njihovim QWidget::sizePolicy() svojstvom ili hintom njihove minimalne veličine, odnosno većim od tog dvoje. Faktori rastezanja koriste se za određivanje promjene količine prostora koja se dodijeljuje *widgetima* u omjeru jedan naprema drugom.

Ako imamo tri *widgeta* postavljenih pomoću QHBoxLayout bez faktora rastezanja dobiti ćemo ovakav izgled kao na slici 6-5:

*Slika 6-5*

Ako primijenimo faktor rastezanja za svaki *widget*, oni će biti određeni u omjeru (ali nikada manje od hinta njihovih minimalnih veličina), npr.


*Slika 6-6*

### 6.2.5. Pisanje vlastitog *layout managera*

Drugi način kreiranja vlastitog *layouta* je napraviti svoj vlastiti *layout manager* proširivanjem QLayout klase.

**Primjer**: CardLayout je klasa inspirirana istoimenim Java *layout managerom*. Ona postavlja svaki element (*widgete* ili ugniježđene *layoute*) preko prethodnog pomaknut za QLayout::spacing().

1 *Widgeti* ne moraju imati zadanu minimalnu veličinu ili hint minimalne veličine i u tom slučaju je faktor rastezanja njihov odlučujući faktor.
2 *Widgeti* ne moraju imati zadanu maksimalnu veličinu i u tom slučaju je faktor rastezanja njihov odlučujući faktor.

Da bi se napisala vlastita *layout* klasa, potrebno je definirati sljedeće:

- Strukturu podataka za pohranu elemenata u *layoutu*. Svaki element je QLayoutItem. (U ovom primjeru to će biti QList)
- addItem(), kako dodavati elemente u *layout*.
- setGeometry(), kako *layout* prikazuje elemente. (pozicioniranje elemenata i međusobni odnos)
- sizeHint(), željena veličina *layouta*.
- itemAt(), kako pristupiti pojedinom elementu *layouta*.
- takeAt(), kako ukloniti element iz *layouta*.
- minimumSize(), minimalna veličina *layouta*.

Header datoteka (`card.h`)

```
#ifndef CARD_H
#define CARD_H

#include <QtGui>
#include <QList>

class CardLayout : public QLayout{

      public:
            CardLayout(QWidget *parent=0): QLayout(parent){}
            CardLayout(QWidget *parent, int dist):QLayout(parent{
                  setSpacing(dist)
            }
            ~CardLayout();

            void addItem(QLayoutItem *item);
            QSize sizeHint() const;
            QSize minimumSize() const;
            int count();
            QLayoutItem *itemAt(int) const;
            QLayoutItem *takeAt(int);
            void setGeometry(const QRect &rect);

      private:
            QList<QLayoutItem*> list;
};

#endif
```

Implementacija (`card.cpp`)

```
#include "card.h"
```

Prvo definiramo count() koja vraća broj elemenata u *layoutu*

```
int CardLayout::count(){
      return list.size();
      //QList::size() vraća broj QLayoutItema u listi
}
```

Tada definiramo dvije funkcije pomoću kojih pristupamo elementima *layouta*: itemAt() i takeAt(). *Layout* sustav koristi interno ove funkcije da bi se omogućilo brisanje *widgeta*. One su, također, dostupne i za programere.

Funkcija itemAt() vraća element koji se nalazi u *layoutu* na danom indeksu. Funkcija takeAt() vraća element iz *layouta* za dani indeks te isti uklanja. U ovom se slučaju elementi nalaze u QListi pa indeks određuje poziciju u listi. U nekim drugim slučajevima, gdje se koriste složenije strukture podataka, kompleksnost implementacije može biti znatno veća.

```
QLayoutItem *CardLayout::itemAt(int idx) const{
	return list.value(idx);
//QList::value() vrši provjeru indeksa i vraća 0 ako smo izvan granica
}

QLayoutItem *CardLayout::takeAt(int idx)
{	return idx >= 0 && idx < list.size() ? list.takeAt(idx) : 0;
	// Qlist::takeAt() ne vrši provjeru

}
```

Funkcija addItem() sprema element u određenu strukturu podataka. Ova funkcija mora biti implementirana. Koriste je QLayout::add() i QLayout konstruktor. Ako je *layout* napredniji s dodatnim parametrima, npr. sa mogučnošću dijeljenja redaka ili stupaca potrebno je preopteretit QGridLayout::addItem(), QGridLayout::addWidget() i QGridLayout::addLayout().

```
void CardLayout::addItem(QLayoutItem *item){
	list.append(item);
}
```

*Layout* preuzima odgovornost za svaki dodan element. Budući QLayoutItem ne nasljeđuje QObject, elementi se moraju izbrisati ručno. U destruktoru se svaki element uklanja iz liste pomoću takeAt(), a zatim se briše.

```
CardLayout::~CardLayout(){

	QLayoutItem *item;
	while ((item = takeAt(0)))
		delete item;
}
```

Funkcija setGeometry() obavlja stvarni prikaz *layouta*. Pravokutnik dostavljen kao argument ne uključuje margin(). Ako je potrebno, koristi se spacing() za definiranje pomaka između elemenata.

```
void CardLayout::setGeometry(const QRect &r){
	QLayout::setGeometry(r);

	if (list.size() == 0)
		return;

	int w = r.width() - (list.count() - 1) * spacing();
	int h = r.height() - (list.count() - 1) * spacing();
```

```
      int i = 0;
      while (i < list.size()) {
            QLayoutItem *o = list.at(i);
            QRect geom(r.x() + i * spacing(), r.y() + i * spacing(), w,
h);
            o->setGeometry(geom);
            ++i;
      }
}
```

Funkcije sizeHint() i minimumSize() obično imaju vrlo slične implementacije. Veličine vraćene od obje funkcije sadrže spacing(), ali ne i margin().

```
QSize CardLayout::sizeHint() const{
      QSize s(0,0);
      int n = list.count();
      if (n > 0)
            s = Qsize(100,70);      //početi sa prikladnom veličinom
      int i = 0;
      while (i < n) {
            QLayoutItem *o = list.at(i);
            s = s.expandedTo(o->sizeHint());
            ++i;
      }
      return s + n*QSize(spacing(), spacing());
}

QSize CardLayout::minimumSize() const{
      QSize s(0,0);
      int n = list.count();
      int i = 0;
      while (i < n) {
            QLayoutItem *o = list.at(i);
            s = s.expandedTo(o->minimumSize());
            ++i;
      }
      return s + n*QSize(spacing(), spacing());
}
```

# 7. Prozori aplikacije i dijalozi

*Widget* koji nije dijete nekog drugog *widgeta* naziva se prozor. Prozori, obično, imaju okvir i naslov, iako je moguće stvoriti i prozor bez tih detalja koristeći pogodne tzv. zastavice prozora[1] (window flags). Najčešći tipovi prozora su QMainWindow i razne podklase QDialoga.

U aplikacijama, prozori određuju prostor na zaslonu u kojem je izgrađeno korisničko sučelje. Prozori aplikacija vizualno su odvojeni jedni od drugih i obično imaju detalje koji korisniku omogućuju promjenu veličine i položaja aplikacije prema njegovim željama. Prozori su obično integrirani u radnu površinu (*desktop enviroment*), a do neke mjere njima upravlja sustav za upravljanje prozorima (*window manager*) koji pruža ta radna površina. Na primjer, odabrani prozori aplikacija su prikazani u traci sa zadacima (*taskbar*).

## 7.1. Primarni i sekundarni prozori

Svaki QWidget koji nema roditelja postati će prozor i na većini će platforma biti uvršten u traku sa zadacima radne površine. To je obično poželjno samo za jedan prozor aplikacije, primarni prozor.

Osim toga, QWidget koji ima roditelje može, također, postati prozor postavljanjem Qt::WA_Window zastavice. Ovisno o sustavu za upravljanje prozorima takvi sporedni prozori su najčešće ispred svojih roditelja te nisu prikazani u traci sa zadacima radne površine.

QMainWindow i QDialog klase postavljaju Qt::WA_Window zastavicu u svom konstruktoru, pošto su oni dizajnirani da se koriste kao prozori.

## 7.2. Glavni prozori i dijalozi

Izgradnju glavnog korisničkog sučelja (glavnog prozora) aplikacije vrši se proširivanjem QMainWindow klase. QMainWindow ima svoj *layout* na koji se može dodati traka izbornika, alatne trake, *dock widgeti* i statusna traka. Centralno područje može zauzimati bilo koji QWidget.

Dijalog prozori koriste se kao sekundarni prozori koji korisniku nude opcije i izbor. Dijalozi se stvaraju kao podklase QDialoga i standardno koriste *widgete* i *layoute* za izradu samog sučelja. Osim toga, Qt pruža niz gotovih standardnih dijaloga koji se mogu koristiti za standardne zadatke kao što su odabir datoteke ili fonta.

1 enumeratori korišteni za specifikaciju različitih postavki sustava prozora (*window-system*) za *widget*.

Glavni prozori i dijalozi mogu biti kreirani pomoću Qt Designera, vizualnog alata koji je dio Qt-a. Korištenje Qt Designer je puno brže nego ručno kodiranje i olakšava testiranje različitih ideja dizajna. Vizualna izrada dizajna i čitanje koda generiranog od strane UIC[2]-a () je odličan način učenja Qt-a.

### 7.2.1. Glavni prozor u Qt aplikaciji

Qt 4 pruža sljedeće klase za upravljanje glavnim prozorima i pripadajuće komponente korisničkog sučelja:

- QMainWindow je centralna klasa oko koje mogu biti izgrađene aplikacije.
- QMenuBar osigurava traku s izbornicima.
- QDockWidget pruža *widget* koji se može koristiti za stvaranje odvojivih paleta alata ili prozora za pomoć (*help window*). *Dock widget* pamti svoje postavke, koje mogu biti *moved*, *closed* i *floated* kao vanjski prozori.
- QToolBar je *widget* općenite alatne trake koja može sadržavati više različitih akcija vezanih uz widget, kao što su gumbi, padajućim izbornici, *comboboxovi* i *spinboxovi*.
- QStatusBar je traka koja prikazuje statusne informacije aplikacije.


*Slika 7-1: Glavni prozor*

2 *User Interface Compiler*

Korištenje QMainWindow klase je jednostavno. Općenito, proširujemo QMainWindow vlastitom klasom i postavljamo izbornike, alatne trake, i *dock widgete* unutar QMainWindow konstruktora.

Da bi se u glavni prozor dodala traka s izbornicima (*menu bar*) treba jednostavno kreirati izbornike i dodati ih u traku s izbornicima glavnog prozora. Funkcija QMainWindow::menuBar() automatski stvara traku ako ne postoji. Također može se pozvati QMainWindow::setMenuBar() za korištenje prilagođene trake s izbornicima u glavnom prozoru.

```
MainWindow::MainWindow(QWidget *parent) : QmainWindow(parent){
      ...
      newAct = new QAction(tr("&New"), this);
      newAct->setShortcuts(QKeySequence::New);       //prečac je Ctrl+n
      newAct->setStatusTip(tr("Create a new file"));
      connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

      openAct = new QAction(tr("&Open..."), this);
      openAct->setShortcuts(QKeySequence::Open);     //pečac je Ctrl+o
      openAct->setStatusTip(tr("Open an existing file"));
      connect(openAct, SIGNAL(triggered()), this, SLOT(open()));
      ...
```

Nakon što su kreirane akcije, dodamo ih u pripadajuće komponente glavnog prozora:

```
      fileMenu = menuBar()->addMenu(tr("&File"));
      fileMenu->addAction(newAct);
      fileMenu->addAction(openAct);
      ...
      fileMenu->addSeparator();
      ...
```

QToolBar i QMenu klase koriste Qt-ov sustav akcija (*action system*) kako bi se osigurao dosljedan API. U prethodnom kodu, akcije su dodane u izbornik sa QMenu::addAction() funkcijom. QToolBar, također, omogućuje ovu funkciju, što omogućava lako iskorištavanje iste funkcije u različitim dijelovima glavnog prozora. Time se izbjegavaju nepotrebna dupliciranja.

Alatna traka se kreira kao dijete glavnog prozora te joj se dodaju željene akcije:

```
      fileToolBar = addToolBar(tr("File"));
      fileToolBar->addAction(newAct);
      fileToolBar->addAction(openAct);
      ...
      fileToolbar->setAllowedAreas(Qt::TopToolBarArea |
Qt::BottomToolBarArea);
      addToolBar(Qt::TopToolBarArea, fileToolbar);
```

U ovom primjeru, traka s alatima je ograničena na vrh i na dno glavnog prozora te se na početku nalazi na vrhu. Možemo vidjeti da će akcije newAct i openAct biti prikazani na alatnoj traci i u traci s izbornicima.

QDockWidget se koristi na sličan način kao i QToolBar. Stvorimo ga kao dijete glavnog prozora i dodamo različite *widgete* kao djecu *dock widgeta*:

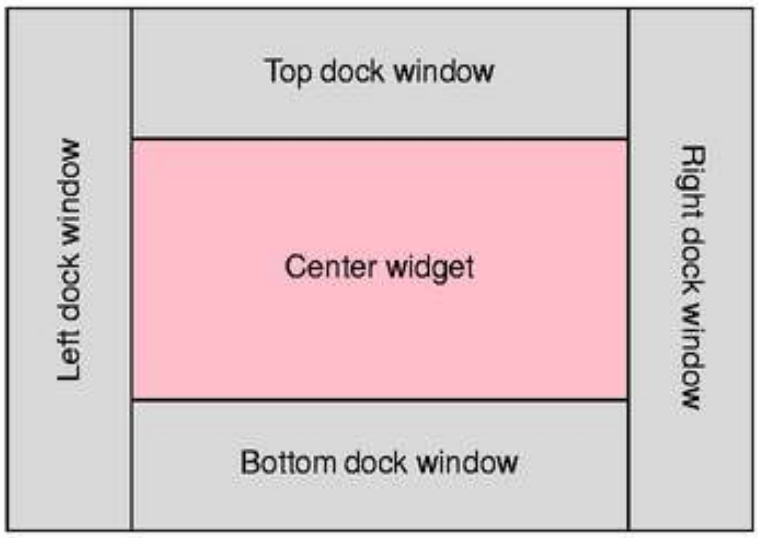
```
contentsWindow = new QDockWidget(tr("Table of Contents"), this);
contentsWindow->setAllowedAreas(Qt::LeftDockWidgetArea |
Qt::RightDockWidgetArea );
addDockWidget(Qt::LeftDockWidgetArea, contentsWindow);

headingList = new QListWidget(contentsWindow);
contentsWindow->setWidget(headingList);
```

U ovom primjeru, *dock widget* može biti smješten u lijevom i desnom području, a inicijalno je smješten u lijevo *dock* područje.

QMainWindow API programeru dopušta odabir jednog od četiri kutova koje će neko od dock *widget* područja zauzeti. Ako je potrebno, postavljeno se može mijenjati s QMainWindow::setCorner() funkcijom:

```
setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea);
setCorner(Qt::BottomLeftCorner, Qt::LeftDockWidgetArea);
setCorner(Qt::TopRightCorner, Qt::RightDockWidgetArea);
setCorner(Qt::BottomRightCorner, Qt::RightDockWidgetArea);
```

Sljedeći dijagram 7-2 prikazuje konfiguraciju gornjeg koda. Važno je uočiti da u ovom primjeru lijevi i desni *dock widget* zauzimaju gornji i donji kut glavnog prozora, a što bi po pretpostavljenim postavkama zauzimali gornji i donji *dock widget*.




*Slika 7-2: Dock widget područja*

Nakon što su postavljene sve glavne komponente prozora, kreira se i postavi središnji *widget* pomoću sljedećeg koda:

```
QWidget * centralWidget = new QWidget (this);
setCentralWidget (centralWidget);
```

Središnji *widget* može biti bilo koja podklasa QWidgeta.

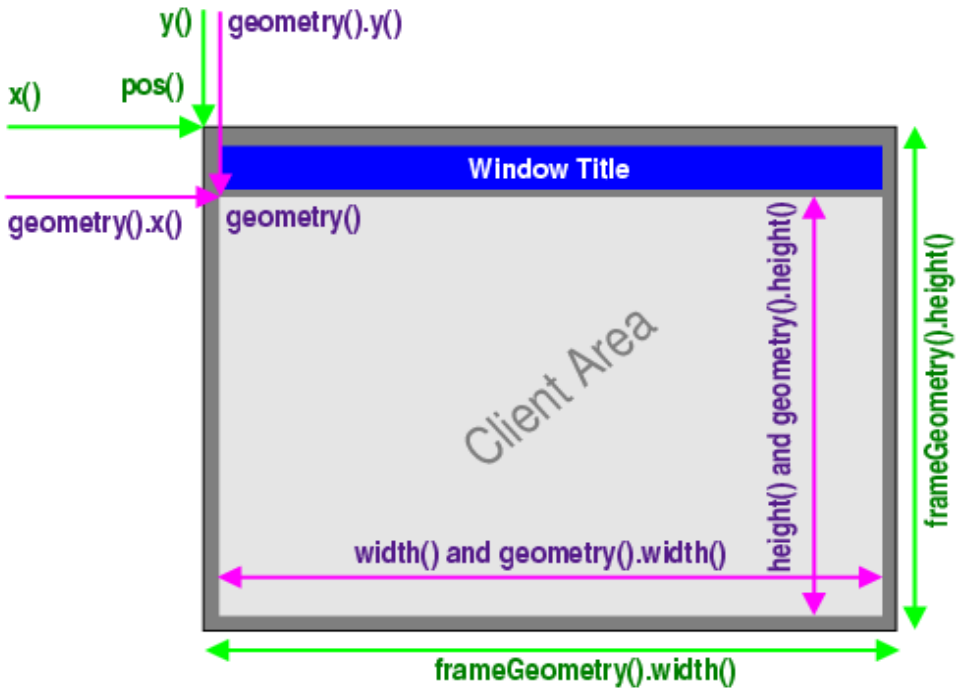
## 7.3. Geometrija prozora

QWidget nudi nekoliko funkcija koje se bave geometrijom *widgeta*. Neke od tih funkcija djeluju na području bez okvira prozora, dok drugi na području s okvirom. Razdjeljuju se prema načinu na koji se najčešće koriste.

- oni koji uključuju prozorski okvir: x(), y(), frameGeometry(), pos(), i move().
- oni bez okvira prozora: geometry(), width(), height(), rect(), i size().

Treba znati da se razlika odnosi samo na *top-level widgete*. Za svu djecu, geometrija se odnosi relativno s obzirom na roditelja i bez okvira.

Ovaj dijagram 7-3 pokazuje većinu funkcija u upotrebi:




*Slika 7-3: Geometrija prozora*

# 8. Sustav za crtanje

Qt-ov sustav za crtanje omogućuje crtanje i ispis koristeći isti API, a u osnovi je baziran na QPainter, QPaintDevice i QPaintEngine klasama.

QPainter se koristi za crtanje, QPaintDevice je apstrakcija dvodimenzionalnog prostora po kojem se može crtati, koristeći QPainter, a QPaintEngine pruža sučelje koje QPainter koristi za crtanje na različitim objektima po kojima se može crtati (*devices*). QPaintEngine klasa koristi se interno od QPainter i QPaintDevice klasa te ga programer ne koristi, osim ako ne izrađuje vlastiti objekt po kojem se može crtati.

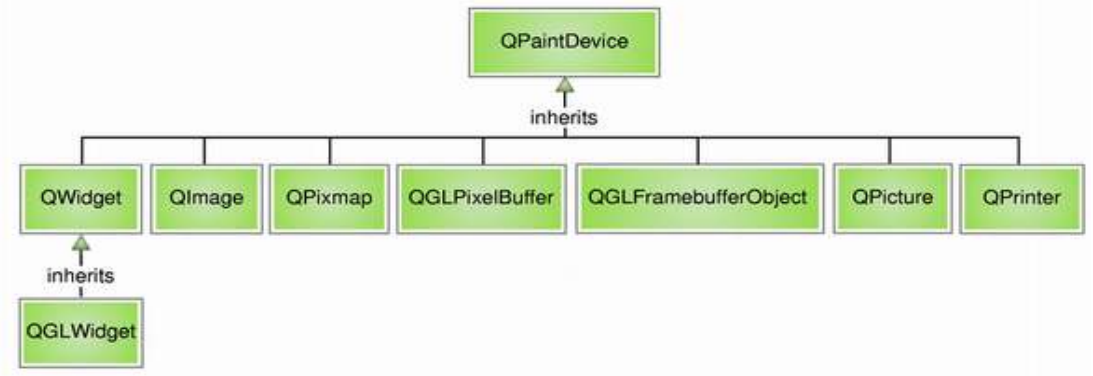

Glavna prednost ovog pristupa je da svo crtanje ima isti slijed što olakšava dodavanje novih mogućnosti i pružanje implementacije za one koje nisu podržane.

Alternativno, Qt daje QtOpenGL modul, nudeći klase koje olakšavju uporabu OpenGL-a u Qt aplikacijama. Među ostalima, modul pruža OpenGL *widget* klasu koja se može koristiti kao bilo koji drugi *widget*, osim što otvara OpenGL spremnik prikaza (*display buffer*), gdje se OpenGL API može koristiti za prikaz sadržaja.

## 8.1. Objekti za crtanje

### 8.1.1. Stvaranje objekta za crtanje

QPaintDevice klasa je osnovna klasa objekata na kojima se može crtati, odnosno QPainter može crtati na bilo kojoj QPaintDevice podklasi. Sposobnosti crtanja QPaintDevice klase trenutno su implementirane od QWidget, QImage, QPixmap, QGLWidget, QGLPixelBuffer, QPicture i QPrinter podklasa.



- **Widget**
  QWidget klasa je osnovna klasa svih objekata korisničkog sučelja. *Widget* prima događaje miša, tipkovnice i druge događaje iz sustava prozora, te crta svoj vlastiti prikaz na zaslonu.

- **Image**
  QImage klasa predstavlja reprezentaciju slike neovisnu o hardveru koja je dizajnirana i optimizirana za I/O i za neposredan pristup i manipulaciju pikselima. QImage podržava nekoliko slikovnih formata, uključujući monochrome, 8-bitni, 32-bitni i alfa-blended slike.
  Jedna od prednosti korištenja QImage kao objekta za crtanje je zajamčena točnost piksela bilo koje operacije crtanja neovisno o platformi. Još jedna prednost je da slika može biti prikazana ili obrađivana u drugoj dretvi od trenutne GUI dretve.

- **Pixel Buffer**
  QtOpenGL modul također osigurava QGLPixelBuffer klasu koja izravno nasljeđuje QPaintDevice.
  QGLPixelBuffer encapsulira OpenGL pbuffer. Iscrtavanje u pbuffer obično se obavlja pomoću potpune hardver akceleracije koja može biti znatno brža od iscrtavanja u QPixmap.

- **Framebuffer Object**
  QtOpenGL modul također osigurava QGLFramebufferObject klase koja izravno nasljeđuje QPaintDevice.
  QGLFramebufferObject encapsulira OpenGL framebuffer objekt. Framebuffer objekti mogu biti korišteni i za off-screen renderiranje i nude nekoliko prednosti nad memorijskim spremnicima za piksele. One su opisane u dokumentaciji QGLFramebufferObject klase.

- **Picture**
  QPicture klasa je objekt za crtanje koji bilježi i ponovo izvodi QPainter naredbe. Slika raspoređuje niz naredbi za crtanje koje šalje I/O objektu u formatu neovisnom o platformi. QPicture je neovisna o rezoluciji, tj. QPicture prikazana na različitim objektima (npr. svg, pdf, ps, pisač i zaslon) izgleda jednako.
  Postoje QPicture::load() i QPicture::save() funkcije, kao i streaming operatori za učitavanje i spremanje slike.

- **Printer**
  QPrinter klasa je objekt za crtanje koji crta na pisač. Na Windowsima ili Mac OS X-u, QPrinter koristi ugrađene *drivere* za pisač. Na X11, QPrinter generira PostScript i šalje ga LPR[1]-u, LP[2]-u ili nekom drugom programu za ispis. QPrinter također može ispisati na bilo koji drugi QPrintEngine objekt.

1 *Line Printer Remote*
2 *Line Printer*

QPrintEngine klasa definira sučelje za interakciju QPrintera sa datim podsustavom za ispis. Čest slučaj pri stvaranju vlastitog mehanizma za ispis, je koristiti i QPaintEngine i QPrintEngine.
Izlazni format pretpostavljeno je određen platformom na kojoj pisač radi, ali eksplicitnim postavljanjem izlaznog formata na QPrinter::PdfFormat, QPrinter će generirati njegov izlaz kao PDF datoteku.

# 8.2. Crtanje i ispunjavanje površina

## 8.2.1. Crtanje

QPainter nudi visoko optimizirane funkcije za većinu crtanja koja GUI programi zahtijevaju. Može nacrtati sve od jednostavnih grafičkih elemenata (kao što su QPoint, QLine, QRect, QRegion i QPolygon klase) do složenih oblika kao što su vektorske putanje (*vector path*). U Qt-u su vektorske putanje predstavljene QPainterPath klasom. QPainterPath pruža kontejner za operacije crtanja omogućavajući da se grafički oblik konstruira i ponovno koristi.

**QPainterPath**

Putanja crtanja (*painter path*) je objekt sastavljen od linija i krivulja. Na primjer, pravokutnik se sastoji od linija, a elipsa se sastoji od krivulje.

Glavna prednost putanje crtanja nad običnim crtanjem je što crtanje složenih oblika, pomoću putanje, treba kreirati samo jednom te oni mogu biti korišteni mnogo puta pozivajući jedino QPainter::drawPath() funkciju.

QPainterPath objekt može se koristiti za ispunjenje (*filling*), crtanje obrisa (*outlining*) i crtanje vidljivih dijelova (*clipping*). Da bi se generirao obris koji se može ispuniti za danu putanju crtanja, koristi se QPainterPathStroker klasa.

Linije i obrisi su nacrtani koristeći QPen klasu. Olovka (*pen*) je definirana svojim stilom (tj. svojim tipom linije), širinom, kistom, kako su nacrtane krajnje točke (*cape-style*) i kako su povezane dvije linije (*join-style*). Kist je QBrush objekt koji se koristi za popunjuje crte generirane olovkom, odnosno QBrush klasa definira obrazac za popunjavanje. QPainter također može crtati poravnani tekst i pixmape.

Kod crtanja teksta, font je definiran uporabom QFont klase. Qt će koristiti font s određenim atributima ili, ako ne postoji odgovarajući font, Qt će koristiti najbliži odgovarajući font koji je instaliran. Atributi fonta koji se koriste mogu se naći pomoću QFontInfo klase. Osim toga, QFontMetrics klasa daje mjere fonta, a QFontDatabase klasa pruža informacije o fontovima koji su na raspolaganju.

Uobičajeno, QPainter crta u "prirodnom" koordinatnom sustavu, ali je moguće obavljati pregled i transformacije pomoću QTransform klase.

### 8.2.2. Ispunjavanje površina

Oblici se ispunjavaju korištenjem QBrush klase. Kist (*brush*) je definiran svojom bojom i svojim stilom (tj. načinom ispunjavanja). Svaka boja u Qt-u opisana je QColor klasom koja podržava RGB[1], HSV[2] i CMYK[3] modele boja. QColor podržava alfa miješanje ocrtavanja i ispunjavanja (omogućavajući efekat transparentnosti-prozirnosti) i ta je klasa neovisna o platformi i objektu na koji se crta.

Prilikom stvaranja novog *widgeta*, preporučljivo je korištenje boje iz palete *widgeta* umjesto samostalnog kodiranja određene boje. Svi *widgeti* u Qt-u sadrže palete i koriste svoju paletu kako bi se nacrtali. Paleta *widgeta* dio je QPalette klase koja sadrži skupine boja za svako stanje *widgeta.*

Dostupni načini ispunjavanja opisani su u Qt::BrushStyle enumeratoru. To su osnovni obrasci u rasponu od jednolike boje do vrlo oskudnih uzoraka, razne kombinacije linija, gradijent ispunjavanja i teksture. Qt ima i QGradient klasu koja definira prilagođeni gradijent ispunjavanja, dok su uzorci tekstura navedeni pomoću QPixmap klase.

## 8.3. Koordinatni sustav

Koordinatni sustav kontrolira QPainter klasa zajedno s QPaintDevice i QPaintEngine klasama. Zadani koordinatni sustav objekta za crtanje ima svoje ishodište u gornjem lijevom kutu. X vrijednost raste u desno, a y vrijednost raste prema dolje. Zadana mjerna jedinica je jedan piksel na piksel-baziranim objektima, odnosno jedna točka (1/72 incha) na pisačima.

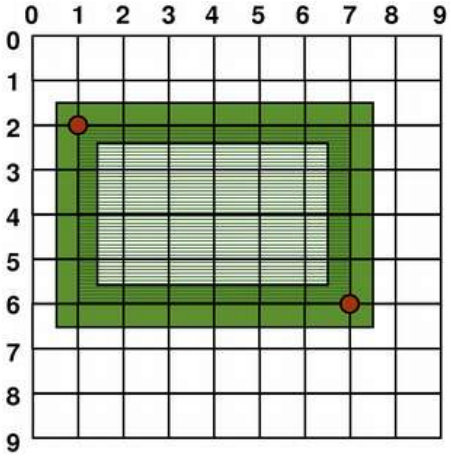
Pretvaranjem logičkih QPainter koordinata u fizičke QPaintDevice koordinate upravljaju funkcije QPainter::worldTransform(), QPainter::viewport() i QPainter::widow(). Logički i fizički koordinatni sustavi pretpostavljeno se podudaraju. QPainter podržava i koordinatne transformacije (npr. rotaciju i skaliranje).

### 8.3.1. Iscrtavanje

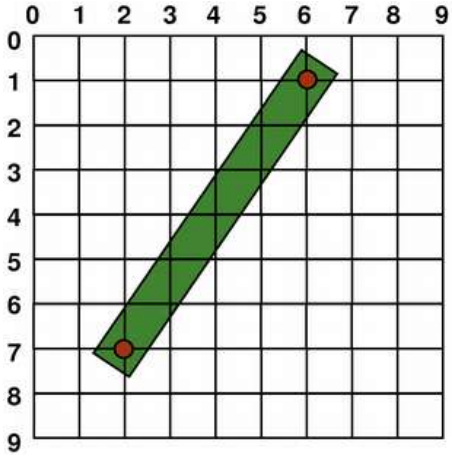
#### Logički prikaz

Veličina (širina i visina) grafičkih elemenata uvijek odgovara svom matematičkom modelu, ignorirajući širinu olovke koja ih iscrtava:

1 *Red, Green, Blue*
2 *Hue, Saturation, Value*
3 *Cyan, Magenta, Yellow, Key black*




*Slika 8-1: QRect(1,2,6,4)*

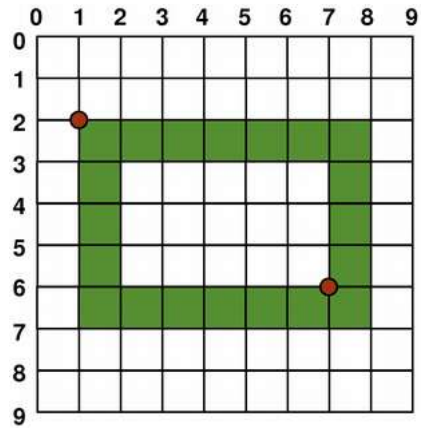



*Slika 8-2: QLine(2,7,6,1)*

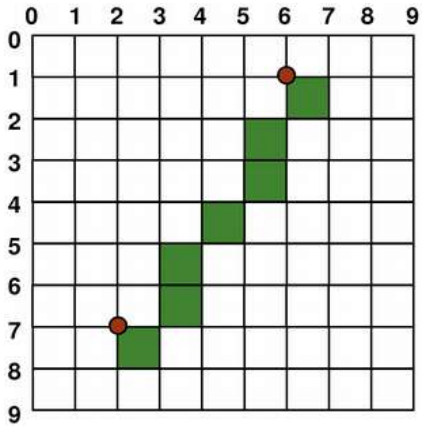
**Nezaglađeno crtanje (*Aliased Painting*)**

Kod crtanja, QPainter::Antialiasing kontrolira iscrtavanje piksela. RenderHint enumerator koristi se za određivanje zastavice QPaintera koja može, ali ne mora, biti poštivana od strane bilo kojeg mehanizma. QPainter::Antialiasing vrijednost ukazuje na to da bi mehanizam trebao zagladiti rubove grafičkih elemenata, ako je moguće.

Ipak pretpostavljeno je da crtanje nije zaglađeno, tako da se koristi pravilo: Kada se crta sa olovkom veličine jednoga piksela, piksel će biti nacrtan desno i ispod matematički definirane točke. Na primjer:




*Slika 8-3:*

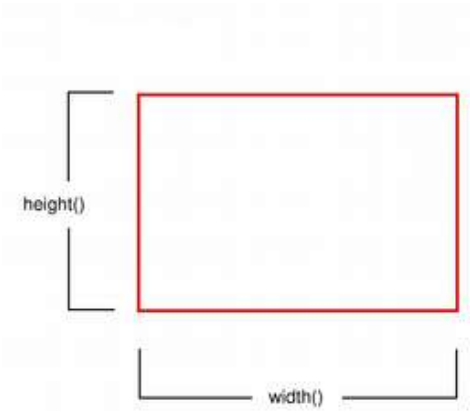



*Slika 8-4:*

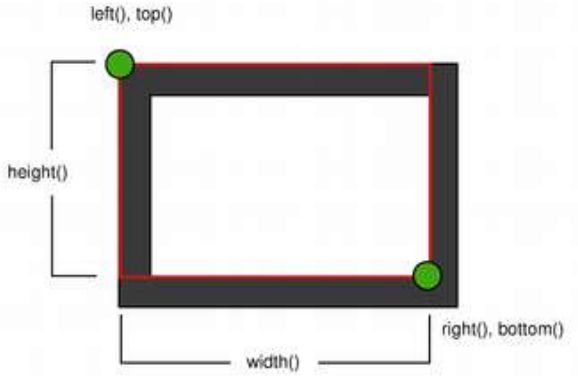
```
 QPainter painter(this);
painter.setPen(Qt::darkGreen);
 painter.drawRect(1, 2, 6, 4);
```

```
QPainter painter(this);
 painter.setPen(Qt::darkGreen);
 painter.drawLine(2, 7, 6, 1);
```

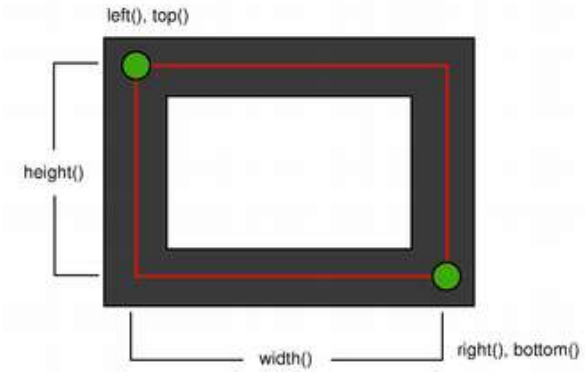
Kad crtanja olovkom s parnim brojem piksela, pikseli će se biti iscrtanih simetrično oko matematički definiranih točaka, dok će kod iscrtavanja olovkom s neparnim brojem piksela, suvišan piksel od parnog broja biti iscrtan desno i ispod matematičke točke kao što je slučaj s jednim pikselom. Sljedeće slike pokazuju QRectF dijagrame kao konkretne primjere.



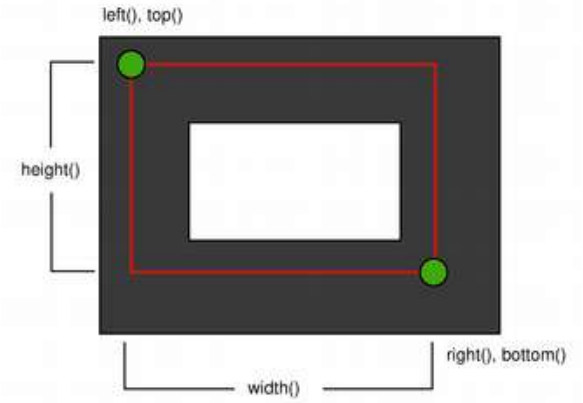

*Slika 8-5: Logički prikaz*




*Slika 8-6: Olovka širine jednog piksela*




*Slika 8-7: Olovka širine dva piksela*




*Slika 8-8: Olovka širine tri piksela*

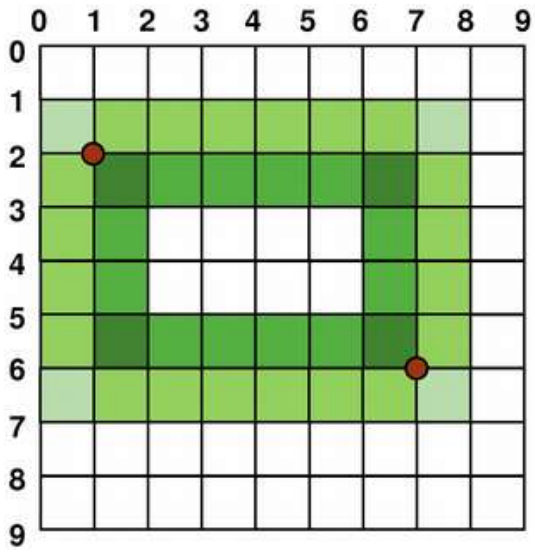
Iz povijesnih razloga povratne vrijednosti funkcija QRect::right() i QRect::bottom() odstupaju od stvarnog donjeg desnog kuta pravokutnika.

QRect::right() funkcija vraća left() + width() - 1, a QRect::bottom() funkcija vraća na top() + height() - 1. Donja desna zelena točka u dijagramima pokazuje povratne koordinate tih funkcija.
Preporučljivo je umjesto toga jednostavno koristiti QRectF. QRectF klasa definira pravokutnik u ravnini pomoću koordinata s pomičnim zarezom kako bi se povećala točnost (QRect koristi cjelobrojne koordinate), te QRectF::right() i QrectF::bottom() funkcija vraća stvarni donji desni kut.

Alternativno, kod korištenja QRect, sa x() + width() i y() + height() može se dobiti donji desni kut i tako izbjeći right() i bottom() funkcije.

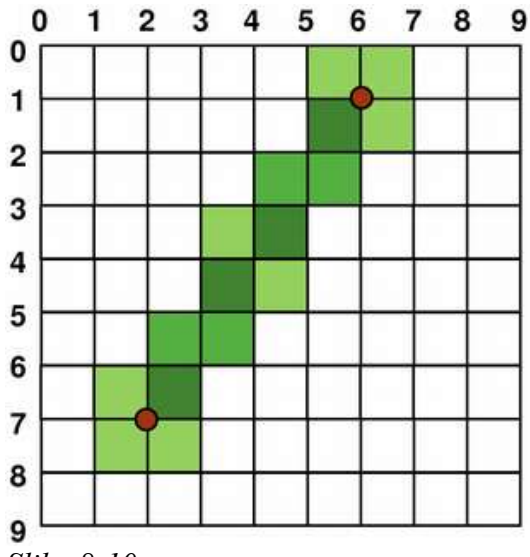
**Zaglađeno crtanje (*Anti-aliased Painting*)**

Ako se postavi QPainter::Antialiasing, pikselai će biti iscrtani simetrično na obje strane matematički definirane točke:




*Slika 8-9:*

```
QPainter painter(this);
painter.setRenderHint(
    QPainter::Antialiasing);
painter.setPen(Qt::darkGreen);
painter.drawRect(1, 2, 6, 4);
```



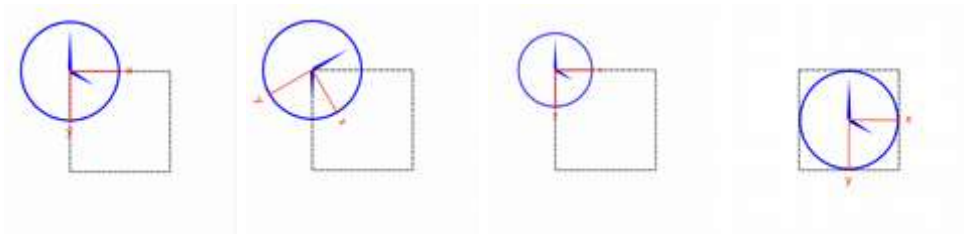

*Slika 8-10:*

```
QPainter painter(this);
 painter.setRenderHint(
     QPainter::Antialiasing);
 painter.setPen(Qt::darkGreen);
 painter.drawLine(2, 7, 6, 1);
```

### 8.3.2. Transformacije

QPainter, zadano, radi na objektu za crtanje koristeći objektov vlastiti koordinatni sustav, ali ima i potpunu podršku za afine transformacije koordinata.

Koordinatnom sustavu može se promijeniti veličina koristeći QPainter::scale() funkciju, može se rotirati u smjeru kazaljke na satu koristeći QPainter::rotate() funkciju i može se translatirati (tj. dodati pomak) pomoću QPainter::translate() funkcije.



nop rotate() scale() translate()

Također, koordinatni sustav se može okretati oko originalnog pomoću QPainter::shear() funkcije. Sve operacije transformacije rade na QPainter transformacijskoj matrici do koje se može preko QPainter::worldTransform() funkcije. Matrica transformira točku u ravnini u drugu točku.

Ako vam je potrebna ista transformacija više puta, mogu se koristiti QTransform objekti i QPainter::worldTransform() i QPainter::setWorldTransform() funkcije. QPainter transformacijska matrica pozivom QPainter::save() funkcije sprema matricu na interni stog. QPainter::restore() funkcija je ponovno vraća.

Jedna od čestih potreba za transformacijskom matricom je potrebno ponovno korištenje istog koda za crtanje na različitim objektima za crtanje. Bez transformacije, rezultati su usko vezani za rezoluciju objekta za crtanje. Pisači imaju visoke rezolucije, na primjer 600 točaka po inču, dok ekrani često imaju između 72 i 100 točaka po inču.

## 8.4. Čitanje i pisanje slika

Najčešći način za čitanje slika je preko QImage i QPixmap konstruktora ili pozivom QImage::load() i QPixmap::load() funkcija. Osim toga, Qt pruža QImageReader klasu koja daje više kontrole nad procesom. Ovisno o formatu slike, funkcije te klase mogu sačuvati memoriju i ubrzati učitavanje slika.

Isto tako, Qt daje QImageWriter klasu koja podržava određene opcije za postavljanje formata, kao što su gama razina, razina i kvaliteta kompresije, prioritet spremanja slike. Ako takve mogućnosti nisu potrebne, mogu se koristiti QImage::save() ili QPixmap::save().

QImageReader i QImageWriter klase oslanjaju se na QImageIOHandler klasu, koja je zajedničko I/O sučelje slike za sve formate slika u Qt-u. QImageIOHandler objekti su korišteni interno od strane QImageReader i QImageWriter klasa kako bi omogućile podršku za različite formate slika.

Popis podržanih formata datoteka dostupne su putem QImageReader::supportedImageFormats() i QImageWriter::supportedImageFormats() funkcije. Qt zadano podržava nekoliko formata datoteka, a novi formati mogu biti dodani kao dodatak (*plugin*).

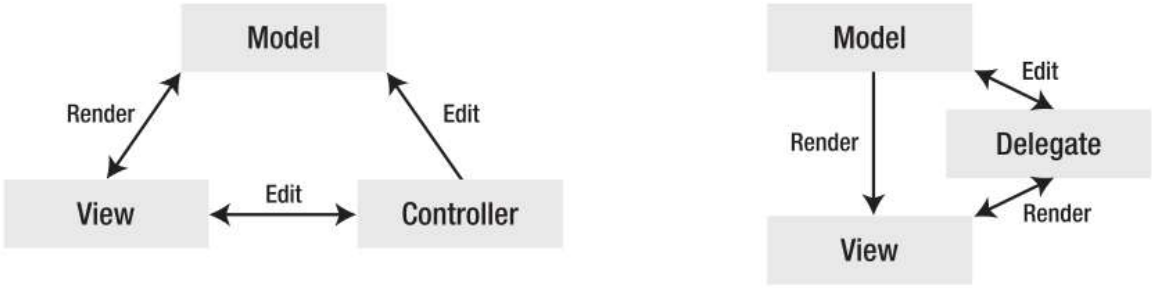
# 9. Model/View programiranje

Qt 4 uvodi novi skup klasa za pregled elemenata (*item view*) koje koriste model/view arhitekturu kako bi se omogućilo upravljanje odnosima između podataka i načinom na koji su predstavljeni korisniku. Razdvajanje funkcionalnosti pomoću ove arhitekture daje programerima veću fleksibilnost za prikaz elemenata i pruža standardno sučelje modela kako bi se omogućio širok raspon izvora podataka koji će se koristiti sa postojećim pregledima elemenata.

## 9.1. Model/View arhitektura

Model-View-Controller (MVC) je oblikovni uzorak (*design pattern*) podrijetlom iz Smalltalk programskog jezika koji se često koristi prilikom izgradnje korisničkih sučelja.

MVC se sastoji od tri vrste objekata. Model je objekt aplikacije, pregled (*view*) je prezentacija modela na ekranu, a kontroler (*controller*) određuje način na koji korisničko sučelje reagira na korisničke akcije. Prije MVC-a, dizajni korisničkog sučelja bili su skloni spajanju tih objekata zajedno. MVC ih odvaja kako bi povećao fleksibilnost i ponovno korištenje.




*Slika 9-1: MVC vs Model/View*

Arhitekturu model/view čini spoj view i model objekata. To još uvijek odvaja način na koji su podaci spremljeni od način na koji su predstavljeni korisniku, ali pruža jednostavniji radni okvir temeljen na istim načelima MVC-a. Ovo razdvajanje omogućava prikaz istih podataka u nekoliko različitih pregleda te izradu novih vrsta pregleda bez promjene temeljne strukture podataka. Da bi se omogućilo fleksibilno rukovanje korisničkim unosima, uvodi se pojam delegata (*delegate*). Prednost delegata je u tome što dopušta način na koji su elementi podataka prikazani i uređeni kako bi se mogli promijeniti.

Model komunicira sa izvorom podataka, pružajući sučelje za ostale komponente u arhitekturi. Priroda komunikacije ovisi o vrsti izvora podataka te načinu na koji je model implementiran.

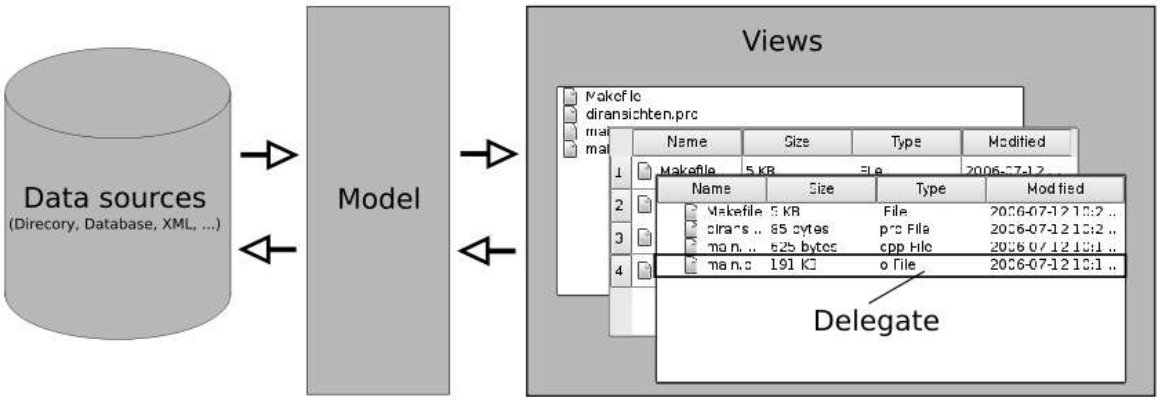
View dobiva od modela indekse modela. To su reference na elemente podataka. Slanjem tih indeksa modelu, view može dohvatiti elemente podataka iz izvora podataka.

U standardnom pregledu, delegat prikazuje elemente podataka. Kad je element promijenjen, delegat komunicira s modelom neposredno koristeći indexe modela.

Općenito, model/view klase mogu se podijeliti u tri skupine: modeli, pregledi i delegati. Svaka od ovih komponenti je definirana apstraktnim klasama koje osiguravaju uobičajena sučelja i, u nekim slučajevima, zadane implementacije raznih mogućnosti. Apstraktne klase su zamišljene kako bi bile proširene da se osigura puna funkcionalnost koju ostale komponente očekuju. To omogućuje pisanje specijaliziranih komponenti.

Modeli, pregledi i delegati komuniciraju jedni s drugima pomoću *signals and slots* mehanizma:

- Signali iz modela obavještavaju pregled o promjenama u izvoru podataka.
- Signali iz pregleda pružaju informacije o interakciji korisnika s elementima koji se prikazuju.
- Signali iz delegata se koriste tijekom uređivanja kako bi obavijestili model i pregled o stanju editora.




*Slika 9-2: Prikaz komunikacije u Model/View arhitekturi*

### 9.1.1. Modeli

Svi modeli elemenata temelje se na QAbstractItemModel klasi. Ta klasa definira sučelje koje se koriste pregledi i delegati za pristup podacima. Sami podaci ne moraju biti pohranjeni u modelu. Mogu se držati u nekoj strukturi podataka, odnosno repozitoriju zasebne klase, datoteke, baze podataka ili neke druge komponente.

QAbstractItemModel pruža sučelje za podatke koje je dovoljno fleksibilno da obrađuje preglede koji prikazuju podatke u obliku tablica, lista i stabala. Međutim, kod implementacije novih modela za strukture podataka tipa lista i tablica bolje je koristiti QAbstractListModel i QAbstractTableModel klase, jer one pružaju odgovarajuće zadane

implementacije uobičajenih funkcija. Svaka od tih klasa može biti proširena kako bi pružila modele koji podržavaju specijalizirane vrste lista i tablica.

Qt daje neke gotove modele koji se mogu koristiti za obradu elemenata podataka:

- QStringListModel se koristi za pohranu jednostavne liste QString elemenata.
- QStandardItemModel upravlja složenijim stablom elemenata, od kojih svaki može sadržavati proizvoljne podatake.
- QDirModel pruža informacije o datotekama i mapama na lokalnom sustavu.
- QSqlQueryModel, QSqlTableModel i QSqlRelationalTableModel se koriste za pristup bazama podataka koristeći model/view konvencije.

Za stvaranje vlastitog prilagođenog modela može se proširiti QAbstractItemModel, QAbstractListModel ili QAbstractTableModel klasa.

### 9.1.2. Pregledi

Potpune implementacije različitih vrsta pregleda su: QListView prikazuje listu elemenata, QTableView prikazuje podatke iz modela u tablici, a QTreeView prikazuje elemente podataka modela u hijerarhijskoj listi. Svaka od tih klasa se temelji na QAbstractItemView apstraktnoj klasi. Iako su te klase spremne za korištenje, one se također mogu proširiti kako bi pružile prilagođene prikaze.

### 9.1.3. Delegati

QAbstractItemDelegate je osnovna apstraktna klasa delegata u model/view radnom okviru. Od verzije 4.4, implementaciju zadanog delegata osigurava QStyledItemDelegate klasa, a ona se koristi i kao zadani delegat od strane Qt-ovog standardnog pregleda. Međutim, QStyledItemDelegate i QItemDelegate neovisne su alternative za crtanje i pružanje editora za uređivanje elemenata u pregledima. Razlika između njih je što QStyledItemDelegate koristi trenutni stil za crtanje svojih elemenata. Zato je QStyledItemDelegate preporučena bazna klasa prilikom implementacije prilagođenih delegata ili prilikom rada s Qt stilovima.

### 9.1.4. Sortiranje

Postoje dva načina sortiranje u model/view arhitekturi. Koji pristup odabrati ovisi o modelu.

Ako model dopušta sortiranje, odnosno, ako reimplementira QAbstractItemModel::sort() funkciju, i QTableView i QTreeView osiguravaju API koji omogućuje programatsko sortiranje modela podataka. Osim toga, može se omogućiti interaktivno sortiranje (tj. ono

koje korisniku dozvoljava sortiranje podatka ako klikne na zaglavlje pregleda), koje povezuje QHeaderView::sortIndicatorChanged() signal i QTableView::sortByColumn() , odnosno QTreeView::sortByColumn() slot.

Alternativno, ako model nema potrebno sučelje ili ako se koristiti pregled liste za prikaz podatka, potrebno je koristiti proxy model za transformaciju strukture modela prije prikazivanja podataka u pregledu.

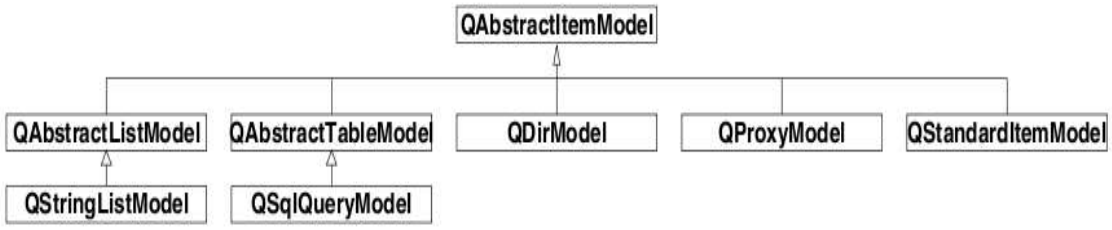
### 9.1.5. Korištenje modela i pregleda

Dva Qt-ova sandardna modela su QStandardItemModel i QDirModel. QStandardItemModel je višenamjenski model koji se može koristiti za prikaz različitih struktura podataka potrebnih za liste, tablice, stabla i preglede. Taj model, također, drži i elemente podataka. QDirModel je model koji održava informacije o sadržaju direktorija. Kao rezultat toga, sam ne drži bilo koje elemente podataka, već jednostavno predstavlja datoteke i direktorije na lokalnim sustavima.

## 9.2. Modeli

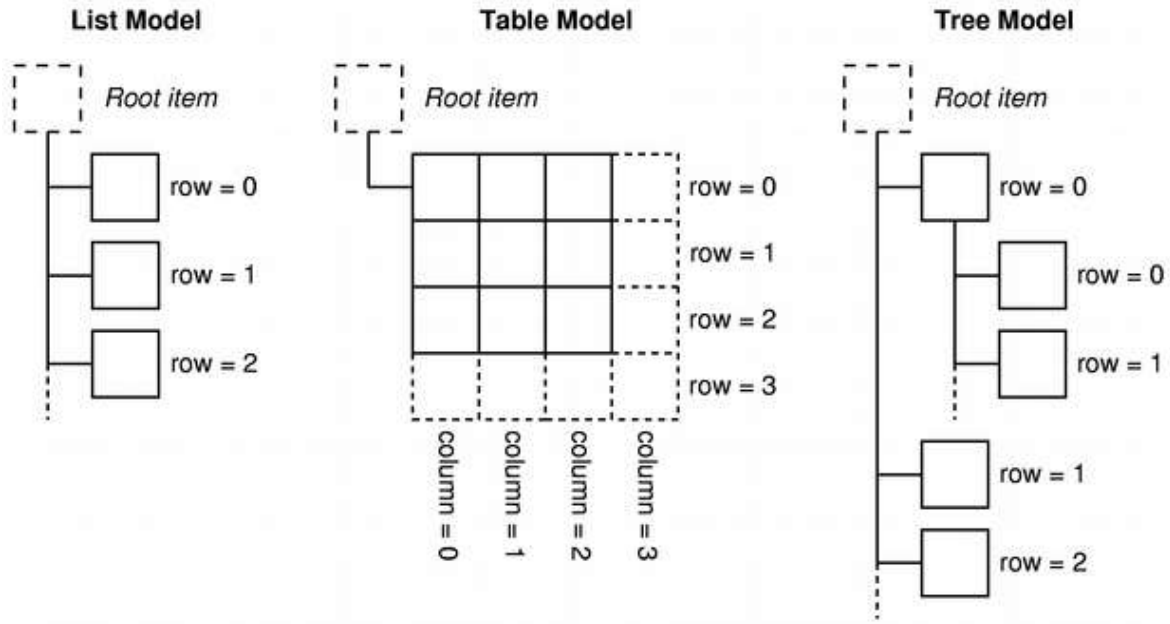
### 9.2.1. Osnovni koncept

U model/view arhitekturi, model omogućuje standardno sučelje koje pregledi i delegati koriste za pristup podacima. U Qt-u, standardno sučelje je definirano QAbstractItemModel klasom. Bez obzira kako su elementi podataka pohranjeni u strukturi podataka, sve podklase od QAbstractItemModel predstavljaju podatke kao hijerarhijsku strukturu koja sadrži tablice elemenata. Pregledi koriste ovu konvenciju za pristup elementima podataka u modelu, ali nisu ograničeni u načinu na koji te informacije prikazuju korisniku.




*Slika 9-3: Arhitektura modela*

Modeli, također, obavještavaju svaki pregled o promjenama podataka putem *signals and slots* mehanizma.




*Slika 9-4: Prikaz modela*

## 9.2.2. Indeksi modela

Kako bi se osigurala nezavisna reprezentacija podataka od načina na koji im se pristupa, uvodi se koncept indeksa modela. Svaki dio informacije koji se može dobiti putem modela predstavljen je indeksom modela. Pregledi i delegati koriste te indekse kako bi dohvatili potrebne elemente za prikaz.

Kao rezultat toga, samo model treba znati kako dobiti podatke, a vrsta podataka kojima upravlja model može se definirati prilično općenito. Indeksi modela sadrže pokazivač na model koji ih je stvorio kako bi se spriječili nesporazumi ako se radi s više od jednog modela.

```
QAbstractItemModel * model = index.model();
```

Indeksi modela pružaju privremene reference na dijelove informacije i kao takvi se mogu koristiti za popravljanje ili izmjenu podataka putem modela. Budući da modeli mogu reorganizirati svoju unutarnju strukturu, s vremena na vrijeme, indeksi modela mogu postati pogrešni tako da ne smiju biti spremljeni. Ako je potrebna dugoročna referenca na informaciju, treba stvoriti stalni indeks modela (*persistent model index*). To daje referencu na informaciju koju model drži ažuriranu. QModelIndex daje privremene indekse modela, a QPersistentModelIndex stalne.

Za dobivanje indeksa modela koji odgovara nekom elementu podataka modelu moraju biti navedena tri svojstva: broj reda, broj stupca i index modela od roditelja elementa.

### 9.2.3. Reci i stupci

U najosnovnijem obliku, modelu se može pristupiti pomoću broja retka i stupca, isto kao i u jednostavnoj tablici u kojoj se nalaze elementi. To ne znači da su osnovni dijelovi podataka pohranjeni u takvoj strukturi. Korištenje broja retka i stupca je samo konvencija kako bi se omogućilo komponentama komuniciranje jedne s drugom. Informacija o bilo kojem elementu može se dohvatiti zadavanjem modelu broja njegova retka i stupca i tada se vraća indeks koji predstavlja elementa:

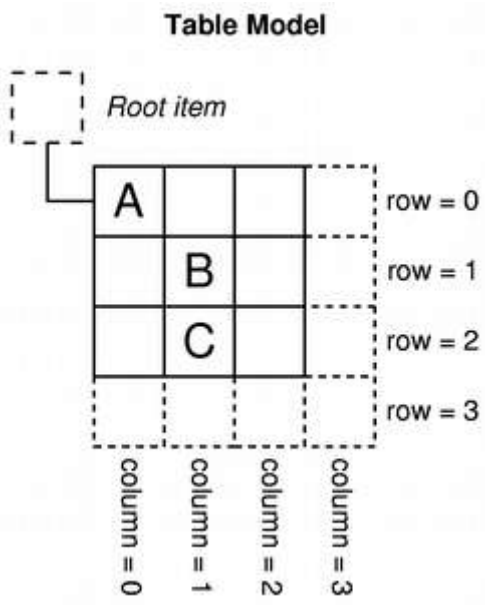
```
QModelIndex index = model->index(row, column, ...);
```

Modelima, koji pružaju sučelja jednostavnim strukturama podataka kao što su liste i tablice, ne trebaju nikakve druge informacije, ali je, kao što gornji kod to pokazuje, potrebno dostaviti više informacija prilikom dobivanja indeksa modela.

Dijagram 9-5 prikazuje prikaz osnovnog modela tablice u kojem je svaki element određen parom brojeva retka i stupca. Slanjem relevantnih brojeva retka i stupca modelu dobivamo indeks modela koji pokazuje na element podataka.

```
QModelIndex indexA = model->index(0, 0,
      QModelIndex());
QModelIndex indexB = model->index(1, 1,
      QModelIndex());
QModelIndex indexC = model->index(2, 1,
      QModelIndex());
```

Elementi na vrhu modela se uvijek određuju pozivom QModelIndex() kao njihovog roditelja.




*Slika 9-5: Reci i stupci*

### 9.2.4. Roditelji elemenata

Sučelje modela nalik na tablicu idealno je prilikom korištenja podataka u pregledima tablice ili liste. Brojevni sustav redaka i stupaca upravo je način na koji ti pregledi prikazuju elemente. Međutim, strukture kao što su pregledi stabla zahtijevaju fleksibilnije sučelje. Kao rezultat toga, svaki element može biti roditelj druge tablice elemenata.

Kada zatražimo indeks za element modela, moramo dati neke informacije o roditelju elementa. Izvan modela, jedini način da se odredi element je preko indeksa modela, tako da je indeks roditelja također potreban:

```
QModelIndex index = model->index(row, column, parent);
```

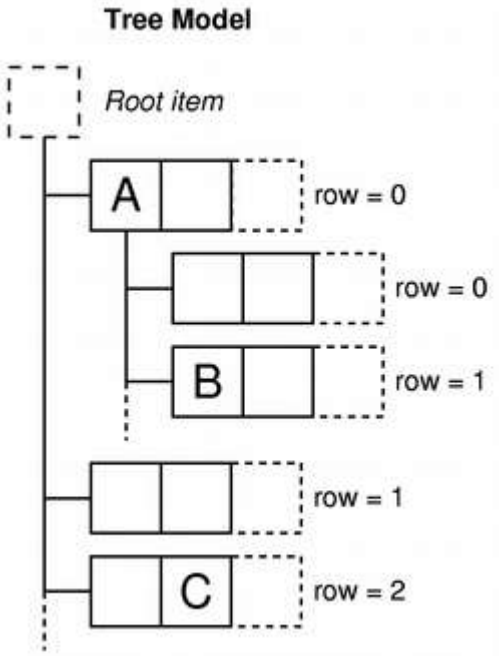
Dijagram 9-6 prikazuje pregled modela stabla u kojem je svaki element određen pomoću roditelja i broja retka i stupca.

Elementi "A" i "C" su vršni elementi i kao takvi su na jednakoj razini.

```
QModelIndex indexA = model->index(0, 0,
      QModelIndex());
QModelIndex indexC = model->index(2, 1,
      QModelIndex());
```

Element "A" ima broj djece. Indeks modela za element "B" dobiva se sljedećim kôdom:

```
QModelIndex indexB = model->index(1, 0,
indexA);
```




*Slika 9-6: Model stabla*

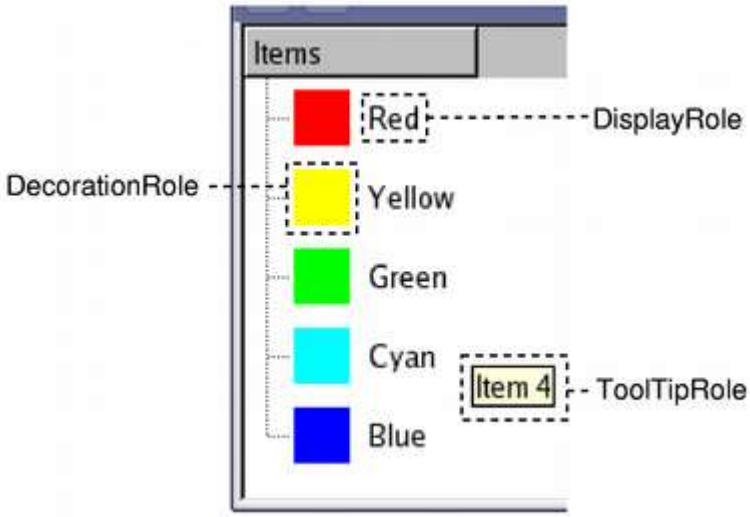
### 9.2.5. Uloge elemenata

Elementi u modelu mogu obavljati različite uloge za ostale komponente, čime se različite vrste podataka mogu dobiti u različitim situacijama. Na primjer, Qt::DisplayRole se koristi za pristup *stringu* koji se može prikazati kao tekst u pregledu. Obično, elementi sadrže podatke za niz različitih uloga, a standardne uloge su definirane sa Qt::ItemDataRole.

Od modela možemo tražiti podatke elementa slanjem indeksa modela odgovarajućeg elementa i zadavanjem uloge:

```
QVariant value = model->data(index, role);
```

Uloga elementa ukazuje modelu koji je tip podataka potreban. Pregled može prikazati uloge na različite načine, tako da je važno dostaviti odgovarajuće podatke za svaku ulogu.




*Slika 9-7: Uloge elemenata*

Najčešće uloge podataka elemenata pokrivene su standardnim ulogama definiranim u Qt::ItemDataRole. Dobivanjem odgovarajućeg podatka elementa za svaku ulogu, modeli mogu pružiti nagovještaj pregledima i delegatima o tome kako elementi trebaju biti predstavljeni korisniku. Različite vrste pregleda imaju slobodu

tumačenja, a mogu i zanemariti te podatke prema potrebi. Također je moguće definirati dodatne uloge za specifične namjene.

# 9.3. Pregledi

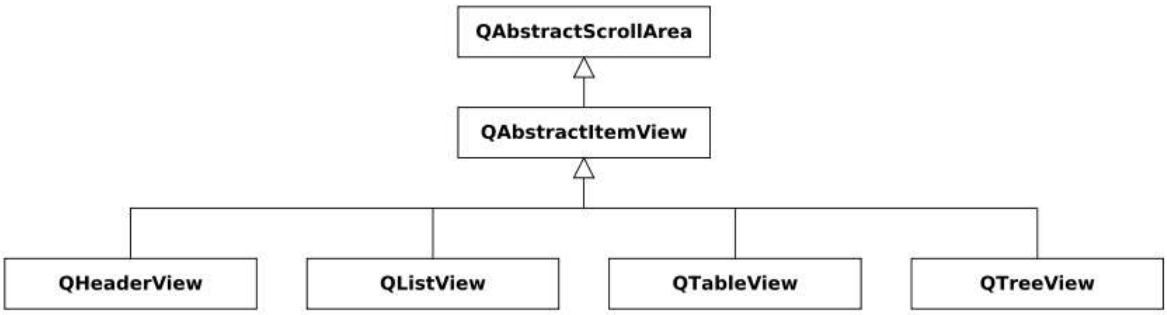
## 9.3.1. Osnovni koncept

U model/view arhitekturi, pregled (*view*) dobiva elemente podataka iz modela i prikazuje ih korisniku. Način na koji su podaci prikazani ne mora biti isti načinu na koji su podaci postavljeni u modelu i može biti potpuno različit od osnovne strukture podataka koja se koristi za pohranu podataka.

Razdvajanje sadržaja i prikaza postignuto je korištenjem standardnog sučelja modela iz AbstractItemModel klase, standardnog sučelja pregleda iz QAbstractItemView klase i korištenjem indeksa modela koji predstavljaju elemente podataka na općeniti način. Pregledi obično upravljaju ukupnim izgledom podataka dobivenih iz modela. Oni mogu sami prikazati pojedine elemente podataka ili iskoristiti delegate za prikaz i uređivanje.

Kao što prikazuju podatke, pregledi upravljaju i navigacijom između predmeta, te odabirom elemenata. Pregledi implementiraju i osnovne značajke korisničkog sučelja, kao što su kontekstualni izbornici te drag and drop funkcija. Pregled može pružiti osnovno uređivanje elemenata ili može zajedno s delegatom pružiti prilagođeno uređivanje.

Pregled se može konstruirati bez modela, ali model mora biti osiguran prije prikaza korisnih informacija. Pregledi prate odabir elemenata od strane korisnika pomoću selekcija koje se mogu držati odvojeno za svaki pregled ili zajednički dijeliti između više pregleda.

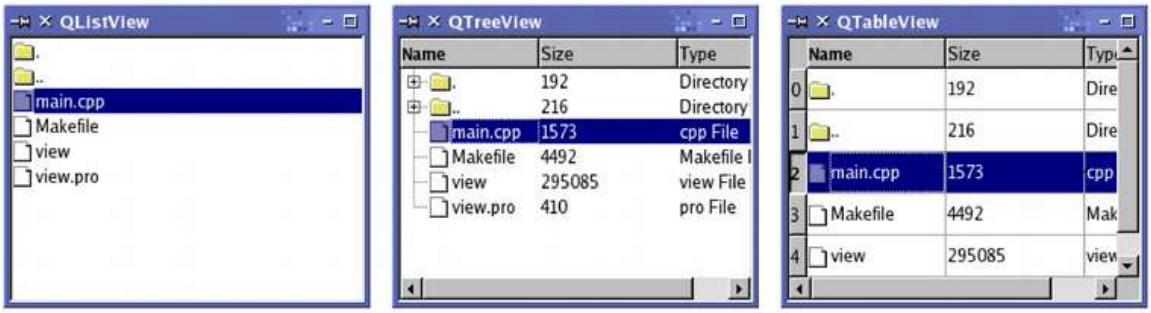



*Slika 9-8: Arhitektura pregleda*

Neki pregledi, kao što su QTableView i QTreeView, prikazuju zaglavlja. Ona su implementirana također od klase pregleda, QHeaderView. Zaglavlja obično koriste isti model kao i pregled koji ih sadrži. Oni dohvate podatke iz modela pomoću QAbstractItemModel::headerData() funkcije i obično prikazuju informacije zaglavlja u obliku labele. Nova zaglavlja se mogu konstruirati proširivanjem QHeaderView klase kako bi se dobilo više specijaliziranih labela za preglede.

### 9.3.2. Korištenje gotovih pregleda

Qt nudi tri gotove klase pregleda koje prikazuju podatke iz modela na načine koji su poznati većini korisnika. QListView može prikazati elemente iz modela kao jednostavnu listu ili u obliku klasičnog pregleda ikona. QTreeView prikazuje elemente iz modela kao hijerarhijske liste, omogućujući duboko ugniježđenim strukturama da budu prikazane na kompaktan način. QTableView prikazuje elemente iz modela u obliku tablice, kao i *layout* aplikacija za tablične kalkulacije.




*Slika 9-9: Standardni pregledi*

Zadano ponašanje standardnih pregleda trebalo bi biti dovoljno za većinu aplikacija. Oni pružaju osnovna svojstva uređivanja sadržaja, a mogu se i prilagoditi, tako da odgovaraju potrebama više specijaliziranih korisničkih sučelja.

### 9.3.3. Upravljanje odabirom elemenata

Mehanizma za rukovanjem odabirom elemenata unutar pregleda osigurava QItemSelectionModel klasa. Pretpostavljeno, svi standardni pregledi stvaraju svoje vlastite modele odabira (selection model) i komuniciraju s njima na uobičajeni način. Modeli odabira korišteni od strane pregleda mogu se dobiti putem selectionModel() funkcije, a zamijeniti pomoću setSelectionModel(). Sposobnost kontrole modela odabira korisna je kada želimo koristiti više pregleda sa istim modelom podataka.

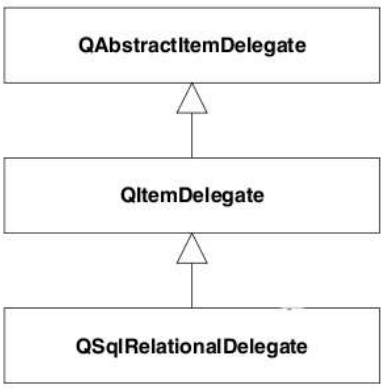
Iako je zgodno da klase pregleda pružaju svoj model odabira, kod korištenja više od jednog pregleda na istom modelu često je poželjno da se podaci modela i korisnikov odabir prikazuju dosljedno u svim pregledima. Budući da klase pregleda dopuštaju promjenu njihova unutarnjeg modela odabira, možemo postići jedinstven izbor između pregleda na sljedeći način:

```
secondTableView->setSelectionModel(firstTableView-
>selectionModel());
```

Drugom pregledu dan je model odabira prvog pregleda. Oba pregleda tada rade sa istim selekcijskim modelom, održavajući i podatke i odabrane elemente sinkroniziranim.

# 9.4. Delegati

## 9.4.1. Osnovni koncept




*Slika 9-10: Arhitektura delegata*

Za razliku od Model-View-Controller uzorka, model/view dizajn ne uključuje potpuno odvojenu komponentu za upravljanje interakcije s korisnikom. Općenito, pregled je odgovoran za prikaz modela podataka korisniku i za obradu korisničkog unosa. Kako bi se omogućila određena fleksibilnost u načinu na koji se korisnički unos dobiva, interakcija se izvodi preko delegata. Te komponente daju sposobnost ulaznih akcija i odgovorne su za prikazivanje pojedinih elemenata u nekim pregledima. Standardno sučelje za kontrolu delegata je definiran u klasi QAbstractItemDelegate.

Od delegata se očekuje mogućnost prikaza njihova sadržaja implementacijom paint() i sizeHint() funkcija. Međutim, jednostavan delegat temeljen na *widgetu* može proširiti QItemDelegate klasu umjesto QAbstractItemDelegate klase, i iskoristiti gotove implementacije ovih funkcija.

Uređivanje za delegate može se implementirati pomoću *widgeta* za upravljanje procesima uređivanja ili obradom događaja izravno.

## 9.4.2. Korištenje gotovih delegata

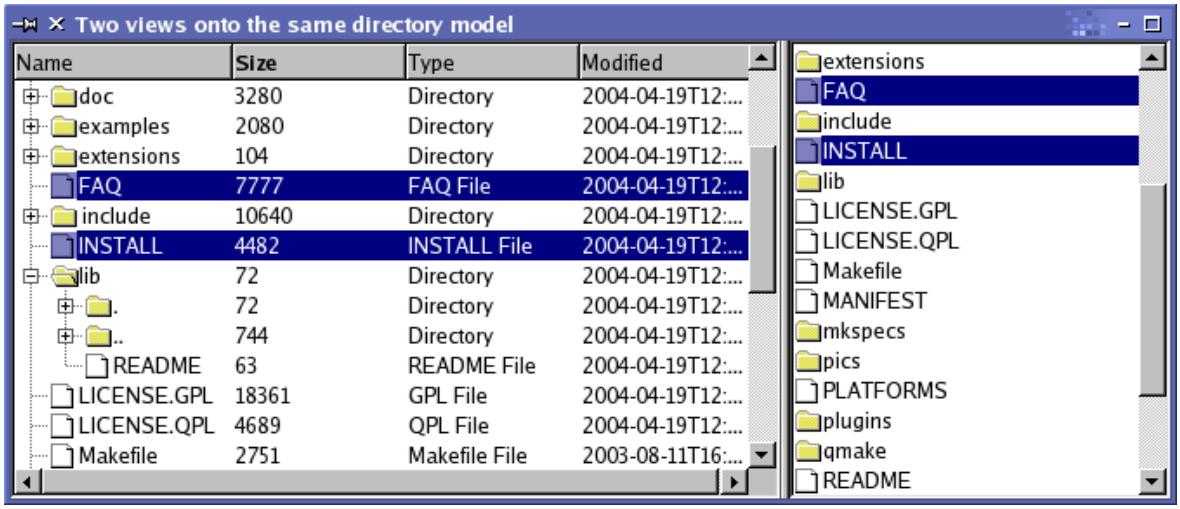
Standardni pregledi Qt-a koriste QItemDelegate kako bi pružili mogućnosti uređivanja sadržaja. Ta zadana implementacija sučelja delegata prikazuje elemente u uobičajenom stilu za svaki od standardnih pregleda: QListView, QTableView i QTreeView.

Sve standardne uloge mogu biti obrađene od zadanog delegata kojeg koriste standardni pregledi.

Prikaz može vratiti delegata kojeg koristi pomoću itemDelegate() funkcije. SetItemDelegate() funkcija omogućuje postavljanje prilagođenog delegata na standardni pregled te je potrebno koristiti tu funkciju prilikom postavljanja delegata za prilagođeni pregled.

## 9.5. Korištenje pregleda sa gotovim modelima

QListView i QTreeView klase su najprikladniji pregledi za korištenje s QDirModel. Primjer navedene u nastavku prikazuje sadržaj direktorija u stablo pored istih podatka u pregledu liste. Pregledi dijele korisnički odabir, tako da su odabrane stavke označene su oba pregleda.



*Slika 9-11: QDirModel sa QTreeView i QListView*

QDirModel se postavi tako da je spreman za uporabu i kreiraju se neki pregledi za prikaz sadržaja direktorija. Ovo je primjer najjednostavnijeg načina korištenja modela. Konstrukcija i korištenje modela vrši se unutar jedne main() funkcije:

```
int main(int argc, char *argv[]){
      QApplication app(argc, argv);
      QSplitter *splitter = new QSplitter;
      QDirModel *model = new QDirModel;
```

Model je postavljen tako da koristi podatke iz zadanog direktorija. Stvorena su dva pregleda, tako da se mogu ispitati elementi u modelu na dva različita načina:

```
      QTreeView *tree = new QTreeView(splitter);
      tree->setModel(model);
      tree->setRootIndex(model->index(QDir::currentPath()));

      QListView *list = new QListView(splitter);
      list->setModel(model);
      list->setRootIndex(model->index(QDir::currentPath()));
```

Pregledi su konstruirani na isti način kao i drugi *widgeti*. Postavljanje pregleda za prikaz elemenata u modelu vrši se jednostavanim pozivom setModel() funkcije s modelom direktorija kao argumentom. Pozivi setRootIndex() govore pregledu koji direktorij prikazati postavljajući indeks modela koji dobijemo od modela direktorija.

Funkcija indeks(), koja se koristi kako bi se dohvatio indeks direktorija u modelu, jedinstvena je za QDirModel.

Ostatak funkcija samo prikazuje preglede pomoću *splitter widgeta* i pokreće aplikacijsku petlju:

```
      splitter->setWindowTitle("Two views onto the same directory
model");
      splitter->show();
      return app.exec();
}
```

# 10. Zaključak

Qt 4 je jednostavniji za korištenje od prethodnih verzija, a također sadrži i mnoge moćnije funkcije. Njegove klase olakšavaju njihovo učenje, smanjuju vrijeme izrade aplikacija i povećavaju produktivnost samog programera. Qt je potpuno objektno orijentiran. Programi rađeni u Qt-u imaju prirodan izgled na svakoj podržanoj platformi. Samo prevođenje istog koda na podržanoj platformi omogućava rad aplikacije na toj platformi.

Sadrži grafičke elemente (*widgete*) koji omogućavaju standardnu funkcionalnost grafičkog sučelja. Predstavlja inovativnu alternativu u komuniciranju između objekata, nazvanu "s*ignal and slots*", koja zamjenjuje staru i nesigurnu *callback* tehniku. Također, pruža uobičajen model za baratanje događajima poput klika mišem, pritiska tipke i ostalih korisničkih ulaza. GUI aplikacije sadrže sve elemente koji se nalaze u modernim aplikacijama, kao npr. *menu*, *toolbar*. *drag and drop* i sl.

Ima odličnu podršku za multimediju i 3D grafiku, te standardni GUI framework za OpenGL programiranje. Njegov sustav za crtanje nudi visokokvalitetan prikaz na svim platformama.

Distribuiran je pod GNU[1] GPL[2] i LGPL[3], te komercijalnom licencom, što ga čini idealnim za učenje, besplatno programiranje manjih komercijalnih aplikacija, ali i za komercijalno programiranje velikih i složenih programa.

Kao *open source* ima veće mogućnosti razvoja i bolje razumijevanje samih programera. Qt 4 danas jednako podržava komercijalne platforme, kao i *open source* platforme tako da je pomoću Qt-a moguće kreirati *open source* aplikacije pod GPL licencom na svim podržanim platformama.

Qt *framework* sadrži mnoštvo korisnih alata, od kojih je najvažniji QDesigner, alat za vizualnu izradu grafičkih sučelja, koji može služiti i kod cjelokupne izrade aplikacija, pošto sadrži podršku za integraciju sa popularnim IDE[4] aplikacijama.

Kao takav, Qt je izvrstan *framework* koji zajedno sa C++ programskim jezikom postaje velika konkurencija na tržištu. Zato je Qt srce komercijalnih aplikacija od 1995. Koriste ga kompanije i organizacije kao sto su Adobe, Boeing, Google, IBM, Motorola, NASA, Skype te razne manje kompanije i organizacije.

1 GNU's Not Unix – besplatni *Unix like* operacijski sustavi
2 General Public Licence
3 Lesser General Public Licence
4 *Integrated Development Environment*

# Literatura

[1] J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4, Second Edition*, Courier Westford Inc., Westford, Massachusetts, 2008

[2] D. Molkentin, *The Book of Qt.4: The Art of Building Qt Application*s, No Starch Press Inc., San Francisco, 2007

[3] J. Thelin, *Foundations of Qt Development*, Appres, New York, 2007

[4] A. Ezust, P. Ezust, *Introduction to Design Patterns in C++ with Qt 4,* Prentice Hall PTR, New Yersey, 2006

[5] Qt Online Reference Documentation, http://doc.qt.nokia.com/