Miha Lesjak

RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE V OKOLJU Qt

Diplomsko delo

Maribor, junij 2010

Diplomsko delo univerzitetnega študijskega programa

RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE

V OKOLJU Qt

Študent: Miha Lesjak

Študijski program: UN ŠP – Računalništvo in informatika

Smer: Programska oprema

Mentor: red. prof. dr. Borut Žalik

Somentor: doc. dr. Gregor Klajnšek

Maribor, junij 2010

Številka: RI-554
Datum in kraj: 11. 06. 2010, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 1/2010)

SKLEP O DIPLOMSKEM DELU

1. **Mihi Lesjaku,** študentu univerzitetnega študijskega programa RAČUNALNIŠTVO IN INFORMATIKA, smer Programska oprema, se dovoljuje izdelati diplomsko delo pri predmetu Multimediji.

2. **MENTOR: red. prof. dr. Borut Žalik**
   SOMENTOR: doc. dr. Gregor Klajnšek

3. **Naslov diplomskega dela:**
   RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE V OKOLJU Qt

4. **Naslov diplomskega dela v angleškem jeziku:**
   CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT IN Qt

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (en vezan izvod in dva nevezana izvoda) ter en izvod elektronske verzije do 11. 06. 2011 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:

Obvestiti:

- kandidata,
- mentorja,
- somentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju dr. Borutu Žaliku za prijaznost, pomoč in nepogrešljive nasvete pri opravljanju diplomskega dela.

Posebna zahvala velja ženi Vesni za potrpežljivost in vzpodbude ter staršem, ki so mi omogočili študij in me vseskozi tudi podpirali.

RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE V OKOLJU Qt

Ključne besede: okolje Qt, Qt Creator, Qt Graphics View, razvoj mobilnih aplikacij, razvoj prenosljivih aplikacij, mobilni uporabniški vmesniki, Symbian, mobilne naprave, mobilni telefoni, kretnje

UDK: 004.777:621.395.721.5(043.2)

Povzetek

Diplomsko delo predstavi okolje za razvoj prenosljivih aplikacij Qt. Nalogo pričnemo s predstavitvijo okolja, njegove arhitekture, priloženih orodij ter s preučitvijo posebnosti, ki jih uvaja. Nadalje naloga razišče procese, potrebne za načrtovanje mobilnih aplikacij ter analizira principe modernih uporabniških vmesnikov za le – te. V zadnjem delu naloge povežemo obe tematiki ter razvijemo aplikacijo »bralnik novic«, ki združuje bogato grafiko, animacije ter upravljanje preko kretenj.

CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT IN Qt

Key words: Qt, Qt Creator, Qt Graphics View, mobile application development, cross-platform development, mobile user interfaces, Symbian, mobile devices, mobile phones, gestures

UDK: 004.777:621.395.721.5(043.2)

Abstract

This thesis describes the Qt framework for cross-platform application development. Through the introductory chapters an overview of the framework is provided and its architecture, enclosed tools and features are presented. Next, a general design process required for creation of mobile applications is described and an analysis of modern user interface design principles is made. In the last part of the thesis an RSS news reader application with an innovative user interface is described.

VSEBINA

1 UVOD ... 1

2 SPLOŠNO O OKOLJU QT ... 3

2.1 Okolje Qt v uporabi ... 3
2.2 Kaj je okolje Qt? ... 4
2.3 Orodja v okolju Qt ... 5
2.4 Pregled komponent razredne knjižnice okolja Qt ... 7
2.5 Zgodovina ter licenciranje ... 11

3 OKOLJE QT ZA RAZVIJALCE ... 13

3.1 Pozdravljen svet ... 13
3.2 qmake ... 15
3.3 Lastništvo nad objekti in hierarhija ... 15
3.4 Signali in reže ... 17
3.5 Metaobjektni sistem ogrodja Qt ... 20
3.6 Lastnosti ... 22
3.7 Orodje *moc* ... 23
3.8 Implicitna souporaba ... 24
3.9 Izjeme v ogrodju Qt ... 26
3.10 Dogodki in filtri dogodkov ... 26

4 RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILJE NAPRAVE ... 30

4.1 Branje novic na mobilnih napravah Symbian^1 ... 30
4.2 Načrtovanje in oblikovanje ... 32
4.3 Moderni bralnik novic za prenosne telefone ... 35
4.4 Uporabniški vmesniki in ogrodje Graphics View ... 36
4.5 Kretnje ... 40
4.6 Kretnje in ogrodje Qt ... 42
4.7 Animacije ... 49
4.8 Povezljivost s spletom, snemanje novic in slik ... 50
4.9 Izjeme na platformi Symbian ... 52

4.10 INTERNACIONALIZACIJA 52
4.11 OPTIMIZACIJE 54

5 OKOLJE QT NA MOBILNIH NAPRAVAH 62

5.1 PROGRAMSKI VMESNIK QT MOBILITY 63
5.2 OGRODJE SMART INSTALLER NA PLATFORMI SYMBIAN 64

6 PRIHODNOST 65

6.1 NOKIA QT SDK 65
6.2 QT QUICK 66

7 SKLEP 70

8 LITERATURA 72

UPORABLJENE KRATICE

3G	Third Generation (omrežje tretje generacije)
API	Application Programming Interface (aplikacijski programski vmesnik)
DOM	Document Object Model (objektni model dokumentov)
FM	Frequency Modulation (frekvenčna modulacija)
FTP	File Transfer Protocol (protokol za prenos datotek)
GUI	Graphical User Interface (grafični uporabniški vmesnik)
GPL	GNU General Public Licence (splošno dovoljenje GNU)
HTTP	Hypertext Transfer Protocol (protokol za prenos hiperteksta)
IDE	Integrated Development Environment (razvojno okolje)
IM	Instant Messaging (takojšnje sporočanje)
IP	Internet Protocol (internetni protokol)
LGPL	Lesser GNU Public Licence (manj splošno dovoljenje GNU)
MMS	Multimedia Messaging Service (storitev za pošiljanje multimedijskih sporočil)
RSS	Really Simple Syndication (protokol za objavo in distribucijo spletnih vsebin v zapisu XML)
SDK	Software Development Kit (okolje za razvoj programske opreme)
SMS	Short Messaging Service (storitev za pošiljanje kratkih sporočil)
UI	User Interface (uporabniški vmesnik)
URL	Uniform Resource Locator (enolični krajevnik vira)
WLAN	Wireless Local Area Network (brezžično lokalno omrežje)
XML	Extensible Markup Language (razširljiv označevalni jezik)
WRT	Web Runtime (izvajalno okolje spletnih aplikacij)

1 UVOD

Ko so se mobilni telefoni ob koncu preteklega desetletja dodobra uveljavili, se je vsem postavilo vprašanje, kaj bo nasledilo glasovne storitve. V tistem času je razvoj aplikacij podpiralo le malo število mobilnih telefonov in kar nekaj časa je preteklo, preden so ponudniki storitev in mobilni operaterji spoznali, da bodo stranke lahko zadovoljili le s širokim naborom različnih aplikacij in storitev. Začelo se je obdobje razvijalcev mobilnih aplikacij ter z njimi tudi številnih podpornih okolij, ki bi jim to delo olajšali.

Pojavljati so se začele različne platforme ter z njimi tudi razvojna okolja, ki pa med platformami niso bila prenosljiva. To pomanjkljivost so kmalu odpravila okolja, ki so podpirala platformno neodvisen razvoj, aplikacije pa so s pomočjo izvajalnih okolij pričele delovati na mobilnih telefonih različnih proizvajalcev.

V preteklosti so razvijalci morali sprejeti odločitev, ali naj za razvoj izberejo okolje, ki deluje na več platformah (npr. Adobe Flash Lite® ali J2ME™) ter s tem vzeti v zakup težave, kot so počasnejša hitrost izvajanja ter omejen dostop do funkcij platforme, ali pa izbrati okolje, razvito prav za izbrano platformo (npr. Symbian, Android ali pa Apple SDK) ter s tem tvegati počasnejši in na platformo vezan razvoj. Ob koncu leta 2008 pa se je ponudila še tretja možnost, ki združuje najboljše iz obeh svetov – enkraten razvoj aplikacije z možnostjo neposrednega izvajanja na različnih platformah.

To tretjo možnost naj bi dosegli z uporabo okolja Qt™, ki se mu posvečamo v diplomski nalogi. Kot cilj, ki ga bomo zasledovali skozi nalogo, smo si zastavili opis in predstavitev okolja Qt, ki omogoča razvoj prenosljivih aplikacij za različne platforme; od namiznih do mobilnih naprav, ter pregled gradnikov in konceptov modernih uporabniških vmesnikov za mobilne naprave. Ugotovitve bomo podprli z izdelavo aplikacije za pregled novic preko fotografij in jo razvili tako, da bo delovala na različnih platformah, interakcija z uporabniki pa bo potekala preko vmesnika, zgrajenega na konceptih modernih mobilnih uporabniških vmesnikov.

Diplomska naloga, ki jo sestavlja sedem poglavij, v drugem poglavju predstavi okolje Qt, opravi pregled spremljajočih orodij ter opiše komponente, ki sestavljajo njegovo razredno knjižnico.

V tretjem poglavju opišemo posebnosti okolja Qt z razvijalskega stališča, saj uvaja mnogo posebnosti, kot so signali in reže, metaobjektni sistem, sistem lastnosti in implicitna souporaba.

Faze razvoja aplikacije za mobilne naprave opredelimo v četrtem poglavju . Nadalje pojasnimo koncepte modernih uporabniških vmesnikov, in sicer preučimo interakcijo preko kretenj, kinetično pomikanje, animacije ter tehnologije v okolju Qt, ki omogočajo implementacijo grafično bogatih uporabniških vmesnikov. Ob koncu poglavja izpostavimo problematiko izvajanja aplikacij na mobilnih napravah ter nanizamo in razdelamo najpogostejše optimizacije pri aplikacijah razvitih v okolju Qt.

V petem poglavju opišemo specifiko mobilnih naprav ter pojasnimo, na kakšen način jim okolje Qt omogoča podporo.

Prihodnost razvoja okolja Qt predstavimo v šestem poglavju ter v zadnjem, sedmem poglavju, povzamemo delo.

2 SPLOŠNO O OKOLJU Qt

Qt je okolje, ki omogoča razvijalcem, da svoje projekte razvijejo z manj programske kode, ustvarijo več ter končne izdelke namestijo na različnih platformah (angl. »code less, create more and deploy everywhere«). Okolje Qt se je pričelo uporabljati leta 1996, v petnajstih letih pa ga je za svoje razvojno okolje izbralo že več kot 350.000 razvijalcev iz več kot sedemdeset različnih industrij, kot so npr. letalska, obrambna, avtomobilska, komunikacijska, naftna in medicinska industrija, ter s tem razvilo več deset tisoč aplikacij [1]. V letu 2008 je okolje Qt postalo temeljni kamen razvijalske ter tudi platformne ponudbe enega največjih proizvajalcev prenosnih telefonov na svetu – Nokie.

Cilj okolja Qt je omogočiti razvoj naprednih aplikacij z zahtevnimi uporabniškimi vmesniki v karseda kratkem času ter tako, da delujejo na večih platformah. Prednosti okolja Qt so:

- izdelava aplikacij za več platform z isto (eno) izvorno kodo,
- prenosljivost aplikacij med različnimi platformami samo s prevajanjem,
- prenosljivost razvijalskih virov preko različnih platform,
- odpornost na spremembe v prihodnosti ter najpomembnejše,
- razvijalcem omogoča, da se lahko osredotočijo na dodajanje vrednosti (angl. core value) programju, ki ga razvijajo, saj se jim pri tem ni treba obremenjevali s specifikami različnih platform.

V primerjavi z drugimi okolji bi naj bilo z ogrodjem Qt možno razviti aplikacije kar v polovičnem času, ter zmanjšati obseg programske kode za več kot polovico [2].

2.1 Okolje Qt v uporabi

Zanimivo je, da se v vsakdanjem življenju z okoljem Qt posredno srečuje nekaj milijonov ljudi, pri čemer pa se tega seveda večina ne zaveda. Za nazornejšo predstavo v nadaljevanju podajamo le nekaj primerov (slika 2.1) [3]:

- zjutraj nas prebudi prenosni telefon z aplikacijo budilka razvito v okolju Qt,
- pot do letališča najdemo s pomočjo navigacijske naprave, katere uporabniški vmesnik je napisan v okolju Qt,
- naš let varno pristane zaradi programske opreme za upravljanje letov, napisane v okolju Qt,
- koliko sladkorja želimo v kavi, izberemo na kavnem avtomatu, ki nam to izbiro omogoča preko vmesnika, napisanega v okolju Qt,
- izračuni, ki jih uporabimo pri delu, vizualiziramo s pomočjo programa, napisanega v okolju Qt,
- poročilo natisnemo preko uporabniškega vmesnika tiskalnika, ki je napisan v okolju Qt,
- pozno popoldne zaključimo delovni dan z video klicem s poslovnim partnerjem preko videotelefona, ki je bil razvit na okolju Qt,
- s prijatelji se pogovarjamo preko programa Skype,
- vtise iz predaha v tujini uredimo v Adobe Photoshop Album®, nato pa jih naložimo na digitalni slikovni okvir, ki jih prikazuje s pomočjo okolja Qt,
- svet raziskujemo preko Google Earth™,
- zvečer si ogledamo film v predvajalniku, napisanem v okolju Qt, čigar posebne efekte so oblikovali s programsko opremo, napisano v okolju Qt.

Slika 2.1: Programska oprema ter naprave, ki jih poganja okolje Qt.

2.2 Kaj je okolje Qt?

Qt (izgovori se kot angleška beseda »cute«) je okolje za razvoj aplikacij na objektno orientiran ter prenosljiv način. Prav prenosljivost je tisto, kar daje okolju Qt največjo

prednost, saj razvijalci aplikacijo napišejo le enkrat, prevedejo pa jo lahko za različne namizne kot tudi vgrajene operacijske sisteme brez da bi bilo v izvorni kodi potrebno karkoli spreminjati. To je mogoče zato, ker so programski vmesniki (angl. API) s stališča razredne knjižnice okolja Qt na vseh platformah enaki. Okolje Qt pa gre še korak dlje, saj ne ponuja prenosljivosti le preko razredne knjižnice, ampak tudi preko verige orodij (angl. toolchain).

Poleg enakega programskega vmesnika nam okolje Qt omogoča, da izgledajo na različnih platformah vse aplikacije pristno. Izgled (angl. look and feel) aplikacij Qt se namreč prilagodi izgledu platforme (operacijskega sistema), na kateri teče aplikacija (slika 2.2). To favorizira Qt pred drugimi okolji, npr. pred okoljem Java™ in tehnologijo Java Swing. Pri slednjem bi bil izgled neke aplikacije na vseh platformah enak, kar pa bi lahko uporabnike, nevajene takšnega izgleda, pričelo hitro motiti [4].

Slika 2.2: Izgled istega gradnika okolja Qt na različnih platformah [5].

2.3 Orodja v okolju Qt

Pomembna so tudi orodja, ki so del okolja Qt, in omogočajo preprost razvoj aplikacij preko različnih razvojnih sklopov od samega kodiranja, razhroščevanja, internacionalizacije do prevajanja (slika 2.3).

Slika 2.3: Celotno razvojno okolje Qt [6].

Qt Creator

Qt Creator je prenosljivo razvojno okolje, prilagojeno potrebam razvijalcev, saj ponuja podčrtovanje sintakse, dopolnjevanje, statično preverjanje in preurejanje kode. Qt Creator ponuja še kontekstno občutljivo pomoč, napredno urejanje kode in integracijo z orodji za nadzor verzij (kot so Git, Perforce in Subversion), razhroščevalnik ter brskanje in iskanje po izvorni kodi. Vsebuje tudi orodja za razvoj uporabniških vmesnikov po principu »primi in spusti« ter takojšnji predogled uporabniških vmesnikov v naravnem izgledu na izbrani platformi [7].

Qt Linguist

Pri razvoju aplikacij se vse bolj uveljavlja zahteva, da je aplikacija izdelana v »domačem« uporabnikovem jeziku. Za internacionalizacijo Qt ponuja orodje Qt Linguist, ki ponuja zbiranje nizov iz izvorne kode, podpira standard Unicode, omogoča preklapljanje med

jeziki, ki se berejo od leve proti desni in obratno ter se zaveda velikosti pisav v določenih jezikih. Ker je popolnoma ločen od samega razvojnega okolja in ker je razvit kot splošno namemsko orodje, ga lahko brez težav uporabljajo tudi uporabniki brez kakršnega koli tehničnega ali razvojnega predznanja [7].

qmake

Orodje *qmake* je prenosljivo orodje za gradnjo programja na različnih platformah. Z njegovo pomočjo razvijalci opišejo postopek prevajanja, *qmake* pa poskrbi tudi za samodejno ustvarjanje datotek, ki opisujejo prevajanje in gradnjo na specifični platformi (npr. Makefile na platformi Linux in projekt Microsoft Visual Studio na platformi Windows) [7].

2.4 Pregled komponent razredne knjižnice okolja Qt

Qt razredna knjižnica ponuja bogat nabor komponent, ki nam omogočajo razvoj modernih ter prenosljivih aplikacij (slika 2.4).

Slika 2.4: Komponente razredne knjižnice okolja Qt [8].

Jedro

Samo jedro (angl. core) okolja Qt opravlja funkcije upravljanja z objekti in dogodki, vhodno/izhodne (V/I) operacije, nitenje in nadzor nad sočasnim izvajanjem (angl. concurrency), upravljanje z vtičniki (angl. plugins) in nastavitvami ter implementacijo sistema signalov in rež [8].

Uporabniški vmesnik

Okolje Qt ponuja že izdelane gradnike uporabniških vmesnikov, s katerimi lahko preprosto izdelamo aplikacije, ki nudijo »bogato« uporabniško izkušnjo (angl. rich user experience). Qt uporablja za izrisovanje teh gradnikov programske vmesnike (angl. API) platforme, na kateri teče, ter s tem dosega najboljšo izkoriščenost sistemskih sredstev. Obenem pa zagotavlja, da imajo gradniki aplikacij, zgrajenih z okoljem Qt, enak izgled kot vse ostale aplikacije platforme [8].

3D grafika z OpenGL

Knjižnica OpenGL® je med najbolj razširjenimi prenosljivimi knjižnicami za tridimenzionalno upodabljanje [9]. Vendar pa sama po sebi ne ponuja nobenih gradnikov za gradnjo aplikacij, kot so npr. gumbi ali vnosna polja. Okolje Qt zapolni potrebo po manjkajočih gradnikih v knjižnici OpenGL ter obenem zagotavlja prenosljivost takšnih aplikacij [8].

Nitenje

Nitenje (angl. threading) je paradigma za reševanje problemov, ki so procesorsko zelo zahtevni in bi pri obdelovanju »zamrznili« uporabniški vmesnik. Okolje Qt ponuja preproste vmesnike za implementacijo nitenja, pa naj bo to z vidika obvladovanja niti in podatkov ali pa z vidika komunikacije med nitmi [8].

Medobjektna komunikacija

Pri razvoju uporabniških vmesnikov predstavlja pogosto težavo komunikacija med različnimi objekti oziroma gradniki (angl. inter-object communication). Okolje Qt ta problem rešuje s sistemom signalov in rež, ki je realizacija načrtovalskega vzorca opazovalca (angl. observer design pattern). Preprosto povedano, okolje Qt ob nekem dogodku odda signal, sproženje signala pa izvrši režo, s katero je signal povezan. Signale in reže si bomo podrobneje pogledali v kasnejših poglavjih [8].

2D grafika

Okolje Qt ponuja napredno implementacijo 2D grafike s platnom (angl. canvas), preko katerega lahko upravljamo z velikim številom grafičnih elementov. Le – ti so lahko razporejeni v drevo BSP (angl. Binary Space Partitioning), posledica tega pa je zelo hitro odkrivanje izbranega elementa iz množice, kar omogoča vizualizacijo velikega števila elementov v realnem času. Podpora za povečavo, rotacijo ter ostale transformacije je seveda vgrajena; omogočeni pa so razni učinki (angl. effects) ter animacije [8].

Multimedija

Predvajanje različnih multimedijskih objektov je v okolju Qt omogočeno preko integracije z ogrodjem Phonon. Pri tem je naloga ogrodja Phonon, da abstrahira specifiko različnih multimedijskih formatov ter jih predvajanja preko različnih platform, okolje Qt pa poskrbi za enostavno interakcijo ogrodja Phonon z drugimi komponentami Qt [8].

Povezljivost

V sklopu povezljivosti ponuja okolje Qt celotno abstrakcijo strežnikov in odjemalcev, vgrajeno podporo protokolom (npr. HTTP, FTP, DNS, asinhroni HTTP 1.1.) ter omogoča preprost dostop do različnih tipov podatkov (npr. HTML, XML, slikovni podatki) [8].

Integracija WebKit

V okolje Qt je popolnoma integrirano jedro (angl. engine) odprtokodnega spletnega brskalnika WebKit. WebKit je uveljavljeno jedro, saj ga za svoje brskalnike uporabljajo podjetja kot so Apple®, Google™ in Nokia (preko brskalnika, vgrajenega v naprave Symbian S60, ter za okolje WRT) [6]. Največja prednost te integracije je zagotovo »zlitje« spleta in namizja, pri čemer se meja med enim in drugim zabriše. Spletne aplikacije lahko med drugim opravljajo časovno zahtevne operacije, dostopajo do funkcij platforme, obenem pa imajo uporabniške vmesnike izdelane s HTML, CSS in JavaScript (kar pa omogoča zelo hiter razvoj)… S takšnim »mešanjem« platformne in spletne kode dobimo tako imenovane »hibridne« aplikacije [8].

XML

Okolje Qt ponuja branje in zapisovanje dokumentov XML preko tokov (angl. stream), implementacijo protokolov SAX ter DOM v jeziku C++, podporo XQuery 1.0 in XPath 2.0 za poizvedbe v dokumentih XML ter transformacije XSLT [6].

Skriptiranje

Okolje Qt vsebuje implementacijo jedra za skriptiranje (angl. scripting engine) po standardu ECMA, ki je osnova za skriptni jezik JavaScript verzije 1.5. Ta implementacija je popolnoma integrirana v okolje Qt, podpira sistem signalov in rež ter ima vgrajen razhroščevalnik. S tem ponuja veliko možnosti: npr. del aplikacije lahko napišemo v skriptnem jeziku ter si s tem prihranimo čas, ali pa damo uporabnikom možnost, da si sami prikrojijo uporabniško izkušnjo [6].

Podatkovne baze

Uporaba podatkovnih baz preko okolja Qt je omogočena zaradi vgrajenih gonilnikov za: ODBC, MySQL, PSQL, SQLite, Oracle... Ponuja tudi napredne možnosti vizualizacije podatkov iz baz preko različnih pogledov ter form [6].

Ogrodje za testiranje

Del okolja Qt je tudi knjižnica za testiranje enot (angl. unit testing), ki med drugim omogoča tudi testiranje grafičnih vmesnikov, in sicer podpira to preko simuliranja dogodkov miške ter tipkovnice [8].

Podpora programskim jezikom

Okolje Qt je razvito v programskem jeziku C++, vendar pa obstajajo premostitveni paketi (angl. bindings) za jezike Ada, Pascal, Perl, PHP, Ruby, Python in Java [10].

2.5 Zgodovina ter licenciranje

Okolje Qt je proizvod podjetja Qt Development Frameworks (prej Trolltech), ki je bilo ustanovljeno leta 1994. V letu 1996 pa je bilo okolje Qt prvič ponujeno na trg.

Želja podjetja Qt Development Frameworks je bila že od samega začetka, da bi bilo okolje na voljo čimveč razvijalcem, ki pa bi lahko poleg razvoja lastnih aplikacij prispevali tudi k testiranju, rasti in razvoju samega okolja. Ravno zaradi tega, je okolje Qt sedaj na voljo po dveh odprtokodnih licencah (Qt GNU LGPL v. 2.1 in Qt GNU GPL v. 3.0) ter po komercialni licenci (Qt Commercial Developer License). Obe odprtokodni licenci zagotavljata stekanje vseh popravkov s strani katerega koli razvijalca nazaj v samo okolje, medtem ko komercialna licenca nudi podporo ter omogoča, da popravke okolja Qt zadržimo kot lastno intelektualno lastnino (tabela 2.1).

Tabela 2.1: Primerjava licenc [11].

	Komercialna	LGPL	GPL
Cena	Plačilo licence	Zastonj	Zastonj
Ali je treba spremembe ogrodja Qt posredovati naprej?	Ne	Da	Da
Omogoča izdelavo zaprtokodnih aplikacij?	Da, izvorne kode ni treba posredovati.	Da, vendar pa v skladu s pogoji licence LGPL v. 2.1	Ne, saj licenca GNU ne omogoča tega
Posodobitve ogrodja Qt?	Da, sporočilo, da je na voljo nova različica, je poslano takoj	Da, so na voljo	Da, so na voljo
Podpora	Da	Ne, na voljo za plačilo	Ne, na voljo za plačilo
Omogoča prodajo izvajalnega okolja (angl. runtime)	Da, za nekatere uporabnike vgrajenih sistemov	Ne	Ne

Navedeno se je izkazalo za odlično odločitev. Okolje Qt je namreč postalo dostopno zelo širokemu krogu uporabnikov (na trgu je danes več kot petnajst miljonov naprav Linux, ki

temeljijo na okolju Qt), omogočilo pa je tudi hiter ter kvaliteten razvoj preko sprejemanja popravkov s strani razvijalcev, ki uporabljajo odprtokodne licence.

V letu 2008 je podjetje Qt Development Frameworks kupil finski proizvajalec prenosnih telefonov Nokia, in sicer kot del strategije prenosljive programske opreme med prenosnimi in namiznimi napravami [1]. Danes Nokia okolje Qt predstavlja kot privzeto okolje za razvoj aplikacij za platforme Symbian, Maemo in Meego.

3 OKOLJE Qt ZA RAZVIJALCE

Do sedaj smo pregledali orodja ter komponente, ki spremljajo okolje Qt. Sedaj pa je čas, da si »umažemo roke« in pogledamo, kaj nam okolje omogoča iz razvijalskega stališča. Okolje Qt prinaša razvijalcu veliko prednosti, vendar pa jih mora razvijalec dodobra spoznati, da jih lahko unčikovito uporablja pri razvoju aplikacij – in ravno temu je namenjeno to poglavje.

Okolje Qt je napisano v jeziku C++, vendar pa ga na področjih medobjektne (angl. inter-object) komunikacije ter fleksibilnosti pri razvoju grafičnih uporabniških vmesnikov celo razširja. Najpomembnejše izmed razširitev so [12]:

- zelo fleksibilen in močan način za medobjektno komunikacijo imenovan »signali in reže« (angl. signals and slots);
- sistem lastnosti (angl. property) objektov;
- dogodki in filtri dogodkov (angl. events and events filters);
- kontekstualno prevajanje nizov za internacionalizacijo;
- hierarhični sistem objektnih dreves, ki organizira lastništvo nad objekti na naraven način;
- varovani kazalci (angl. guarded pointers), ki si samodejno postavijo vrednost na nič, ko se objekt, ki ga referencirajo, uniči;
- dinamično pretvarjanje objektov.

Kljub navedenim posebnostim, bolje prednostim, pa je najpomembneje, da se okolje Qt še vedno prevede na večini standardnih prevajalnikov C++ preko različnih platform.

3.1 Pozdravljen svet

Za začetek ustvarimo preprosto aplikacijo, ki prikaže gumb z besedilom »Pozdravljen svet«. Potrebujemo le eno datoteko, ki je po ustaljeni praksi jezika C++ imenovana *main.cpp*, njena vsebina pa je zapisana v izpisu 3.1.

```
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton *gumb = new QPushButton("Pozdravljen svet");
    gumb->show();
    return a.exec();
}
```

Izpis 3.1: Izvorna koda za program »Pozdravljen svet«.

Na kratko opišimo izvorno kodo. Stavek v prvi vrstici ukaže prevajalniku, naj vključi vse razrede iz komponente, ki podpira grajenje grafičnih vmesnikov. Za zagon aplikacije moramo ustvariti objekt *QApplication*, ki med drugim vsebuje dogodkovno zanko (angl. event loop) in upravlja z globalnimi viri aplikacije. Zaradi tega lahko sklepamo, da je *QApplication* vedno osnovni gradnik aplikacij z grafičnim vmesnikom. *QPushButton* je gradnik[1], ki ponazarja gumb z besedilom, ki ga podamo v konstruktorju objekta. Gumb prikažemo (v okolju Qt so gradniki privzeto skriti) ter zaženemo aplikacijo s predzadnjo vrstico v izvorni kodi zgoraj. Le – ta požene dogodkovno zanko, preko katere operacijski sistem platforme sporoča okolju Qt o dogodkih, ki so se zgodili – najsi bo to pritisnjena tipka, premik miške ali pa zahteva po izrisu dela zaslona.

Sedaj moramo aplikacijo še zgraditi (angl. build). To storimo z orodjem `qmake`, ki je bilo omenjeno že v prejšnjem poglavju. Ustvarimo tekstovno datoteko *projekt.pro*, katere vsebina je zelo preprosta in vidna v izpisu 3.2, saj imenuje samo datoteko *main.cpp*. Za naš preprost primer ter za orodje *qmake* pa je to več kot dovolj.

```
SOURCES += main.cpp
```

Izpis 3.2: Vsebina datoteke *projekt.pro* pri programu »Pozdravljen svet«.

Isto izvorno kodo lahko prevedemo na različnih platformah, slika 3.1 pa ponazarja izgled te aplikacije na platformi Windows 7, Maemo in Symbian^1.

---

[1] Za gradnike kot je *QPushButton*, obstaja različna terminologija. V okolju Windows se v angleščini imenujejo »control«, v *nix in okolju Qt pa »widget«. Slovar računalništva in informatike (islovar [13]) za slovenski jezik nudi skupni izraz »gradnik«, ki ga bomo v nadaljevanju uporabljali.

Slika 3.1: Izgled aplikacije »Pozdravljen svet« na različnih platformah.

3.2 qmake

Orodje *qmake* je bilo razvito z namenom poenostaviti gradnjo aplikacij preko različnih platform. Proces gradnje je definiran z datoteko, ki opisuje gradnjo (angl. Makefile), le – to pa `qmake` samodejno ustvari glede na platformo in orodja, ki so na voljo. Razvijalec mora tako le enkrat opisati proces gradnje v formatu, ki ga podpira *qmake*, zatem pa ga lahko izvaja na različnih platformah. Orodje *qmake* ni vezano samo na projekte, napisane v okolju Qt, temveč ga lahko uporabljamo tudi pri projektih, ki okolja Qt ne uporabljajo [14].

3.3 Lastništvo nad objekti in hierarhija

V okolju Qt obstaja hierarhija, v kateri je večina razredov dedovana iz objekta *QObject*, čeprav to ni zahtevano. Razlog za navedeno je v tem, da morajo za delovanje sistema signalov in rež, udeleženi razredi dedovati iz objekta *QObject* – lahko bi rekli da dedovanemu objektu poleg podatkov dodamo še obnašanje [15].

Sedaj ko smo izpostavili bistven razred v okolju Qt - *QObject*, omenimo eno izmed zanimivejših lastnosti, ki jih le – ta omogoča: hierarhično lastništvo nad objekti (angl.

parent/child relationships). Ko ustvarimo primerek razreda *QObject* ali pa primerek razreda dedovanega iz slednjega, mu lahko kot argument konstruktorja podamo starša (angl. parent) - pri čemer je starš obveščen o svojih potomcih ter jih uvrsti v listo potomcev.

Ob zgoraj navedenem se nam postavi vprašanje, zakaj okolje Qt sploh ustvarja hierarhična drevesa lastništva nad objekti. Prvi odgovor na to vprašanje je, da se določene lastnosti prenašajo po lastniškem drevesu od staršev do potomcev – npr. pri prikazovanju in skrivanju ter omogočanju in onemogočanju velja, da če bomo skrili starša, bomo skrili tudi vse njegove potomce. Drugi razlog za uporabo hierarhičnega drevesa pa lahko poimenujemo »siromakovo upravljanje s spominom«. To pomeni, da bo starš v primeru, če bo zbrisan sam, poskrbel za brisanje vseh svojih potomcev. Torej velja, da moramo v primeru, če konsistentno določamo starše, voditi le kazalec na najvišjega starša in zbrisati samo slednjega. V tem primeru pa potomcev sami ne smemo brisati, saj nad njimi nimamo več lastništva. Ravno zaradi takšnega delovanja obstaja nenapisano pravilo, da razrede ogrodja Qt, ki dedujejo iz razreda *QObject* , ustvarimo na kopici (angl. heap); izjemi predstavljata razreda *QApplication* in *QFile*. Zaradi jasnosti zapišimo še, da dedovanje ni isto kot hierarhično lastništvo nad objekti.

Program v izpisu 3.3, sicer zapisan zelo nekonvencionalno, ne pušča pomnilnika, saj brisanje objekta *dialog* poskrbi za brisanje dveh potomcev razredov *QPushButton* in *QLabel*.

```
QDialog *dialog = new QDialog();
new QPushButton("Gumb", dialog);
new QLabel("Besedilo", dialog);
delete dialog;
```

Izpis 3.3: Hierarhično lastništvo nad objekti in brisanje.

Razrede, ki ne dedujejo iz razreda *QObject*, ustvarjamo na skladu (angl. stack). Primer za slednje sta razreda *QColor* in *QStringList*, ki ju ustvarimo na način, zapisan v izpisu 3.4.

```
QColor barva;
QStringList nizi;
```

Izpis 3.4: Izvorna koda za program »Pozdravljen svet«.

Ta razreda nista dedovana iz razreda *QObject*, ker ne vsebujeta (niti ne potrebujeta) obnašanja (npr. na barvo ne moremo pritisniti, nizi se ne morejo odzvati na pritisk tipke).

3.4 Signali in reže

Postavimo si vprašanja: kako lahko razredi med seboj komunicirajo? Kako lahko določen razred odreagira na pritisk gumba na miški ali na pritisnjeno tipko tipkovnice?

Rešitev tega problema je več:

- MFC uporablja tako imenovane mape sporočil (angl. message maps), ki so nabor slabo dokumentiranih makrojev, ki jih je zelo težko uporabljati izven razvojnega okolja Microsoft Visual Studio.
- Motif uporablja povratne klice (angl. callbacks), ki pa niso močno tipizirani (angl. type safe), zaradi česar lahko razvijalec hitro naredi napake, ki so vidne šele ob izvajanju.
- Java 1.0 je v preteklosti uporabljala virtualne metode, kar pa se je izkazalo kot zelo nepraktično, saj je bilo dedovanje potrebno za vsak gradnik, ki se je želel odzvati na dogodke.

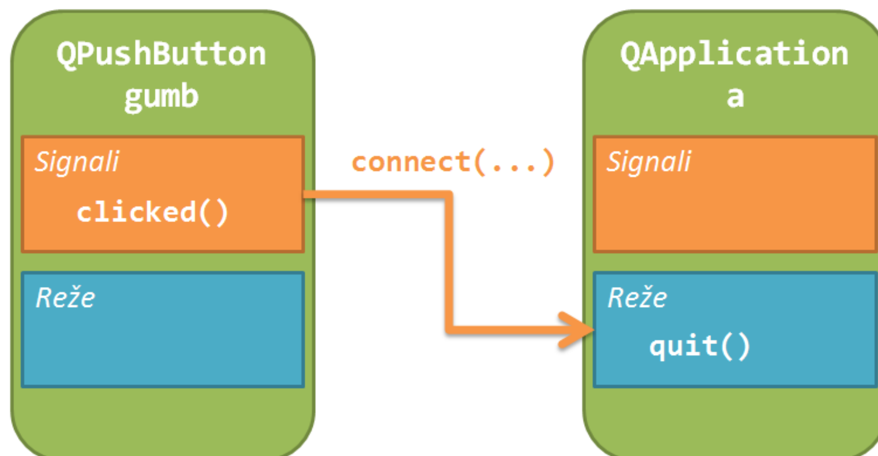
Okolje Qt uporablja sistem signalov in rež za višjenivojske dogodke (npr. uporabnik je kliknil na gradnik) ter virtualne metode preko sistema dogodkov za nižjenivojske dogodke (npr. del gradnika je potrebno ponovno izrisati). Ker je sistem dogodkov večini bralcev zagotovo že poznan iz okolja Java, bomo podrobnejši opis izpustili, z njim pa se bomo pri razvoju aplikacije zagotovo še srečali. Kot posebnost okolja Qt je zanimivejši sistem signalov in rež, zato bo deležen podrobnejše razlage [16]. Ponazorimo ga s primerom.

Razširimo izvorno kodo aplikacije »Pozdravljen svet«, tako da se bo ob pritisku na gumb zaprla. Ob dogodku v sistemu, v našem primeru pritisk na gumb, moramo primerno odreagirati in zapreti aplikacijo – povezati moramo signal, ki se sproži ob pritisku na gumb z režo, ki zapre aplikacijo. Na srečo gradniki in razredi okolja Qt ponujajo pestro število že vgrajenih signalov, po potrebi pa jih lahko definiramo tudi sami. V našem primeru Qt ponuja tako ustrezen signal kot tudi režo, povezavo med njima pa vzpostavimo z eno samo vrstico kode (izpis 3.5).

```
QObject::connect(gumb, SIGNAL(clicked()), &a, SLOT(quit()));
```

Izpis 3.5: Ustvarjanje povezave med signalom *clicked()* in režo *quit()* za objekt *gumb*.

Klic statične metode *QObject::connect()* ustvari povezavo (slika 3.2) signala z režo, ki je navadna metoda C++. Edini pogoj je, da ima enak podpis (signaturo oz. parametre) kot signal, ki jo kliče.



Slika 3.2: Povezava, ki smo jo ustvarili s klicem *connect()*.

Omenili smo že, da okolje Qt uporablja signale za višjenivojske dogodke, nismo pa še podrobneje razložili, kaj signali in reže sploh so.

Signali

Razredi okolja Qt oddajajo signale ob spremembah notranjega stanja objekta, ki bi lahko bile zanimive tako za objekt, ki signal definira, kot tudi za objekte v njegovi »okolici«. Signal lahko odda samo razred, ki ga definira, ter razredi, ki so iz njega dedovani. Ko je signal oddan, se redoma (po vrstnem redu, po katerem so bile ustvarjene povezave) izvšijo reže, ki so na ta signal povezane – enako bi se zgodilo, če bi te reže poklicali neposredno. Zaradi navedenega je sistem signalov in rež neodvisen od zanke dogodkov grafičnega vmesnika (angl. GUI event loop), izvajanje izvorne kode pa se za vrstico, ki odda signal, nadaljuje šele, ko se je končalo izvajanje vseh rež povezanih na ta signal[1].

[1] Izvajanje je drugačno v primeru, ko so signali in reže povezani preko čakalne vrste. Takrat se izvajanje nadaljuje takoj po oddaji signala, reže pa se izvršijo kasneje.

Reže

Reže so metode, ki se izvedejo, ko se sproži signal, na katerega so te povezane. Reže se obnašajo enako kot navadne metode C++ in jih lahko tako tudi kličemo; njihova posebnost pa je že prej omenjeno izvajanje ob sproženih signalih. Omeniti velja še eno posebnost – reže so lahko sprožene s strani komponent, ne glede na pravico dostopa do njih. Recimo, da imamo objekt A ter objekt B, pri čemer objekta med seboj nista povezana preko dedovanja. Signal objekta A lahko sproži režo objekta B, ki je bila deklarirana kot zaščitena (angl. private); to je sicer nekonvencionalno po standardu C++, pa vendar omogoča boljšo medsebojno povezljivost komponent v sistemu.

Proženje rež je približno desetkrat počasnejše kot če bi metode klicali neposredno pod predpostavko, da ne bi bile navidezne (angl. virtual). Na prvi pogled pomeni to velik vpliv na hitrost, vendar pa temu ni tako; razlika v času je veliko manjša kot pa npr. pri operacijah *new* in *delete* ali pa pri izvedbi sistemskega klica. Ravno zaradi slednjega končni uporabniki tega zamika pri izvajanju sploh ne bodo opazili, sistem signalov in rež pa bo vseeno močno poenostavil in pospešil razvoj aplikacij [16]. Prav to je razlog, zakaj se uporabi signalov in rež pri razvoju ne izogibamo.

Povezave

Podrobneje si poglejmo še klic, ki ustvari povezavo med objektoma. V klicu metode *connect()* sta oba, tako reža kot signal, zajeta z makrojema *SLOT()* ter *SIGNAL()*. Ta makroja sta potrebna, saj okolje Qt pričakuje nize kot parametre metode, le – ti pa morajo ustrezati internim standardom okolja ter dinamično referencirati ustrezajoče reže. Prednost tega pristopa je, da lahko okolje Qt med izvajanjem preveri, če so vse povezave veljavne in nas, v primeru da niso, o tem tudi obvesti. Poglejmo si primer (izpis 3.6), kjer bi povezali signal in režo na neveljaven način zaradi neujemanja tipa parametrov.

```
QObject::connect(vrtavka, SIGNAL(valueChanged(QString)), drsnik,
SLOT(setValue(int));
```

Izpis 3.6: Nepravilno povezanje signala in reže zaradi neujemanja parametrov.

Okolje Qt ne pretvarja tipov parametrov in tega ne bosta opazila niti prevajalnik niti povezovalnik. Vseeno pa bo ob zagonu na izhod aplikacije izpisano opozorilo, saj Qt ne bo mogel uspešno opraviti povezovanja – pravimo, da so signali in reže močno tipizirani

(angl. type safe). Omeniti velja še, da pri povezovanju parametrov ne poimenujemo, ampak le napišemo njihove tipe.

Pri sistemu signalov in rež oddajniku signala ni potrebno vedeti, ali je kdo v sistemu povezan na njegov oddan signal, po drugi strani pa tudi za reže ni pomembno, kdo je signal oddal. Celo več, če je to potrebno, lahko signal povežemo tudi na drug signal. Ravno zaradi teh lastnosti je sistem signalov in rež odličen mehanizem enkapsulacije in tako omogoča preprost razvoj medsebojno neodvisnih komponent, ki se lahko med izvajanjem dinamično povezujejo [4].

3.5 Metaobjektni sistem ogrodja Qt

Razširitev, kot je sistem signalov in rež, ne bi mogla delovati brez podpornega sistema. V okolju Qt slednjega predstavlja metaobjektni sistem (angl. meta-object system), ki služi kot osnova tudi za druge razširitve, ki jih Qt dodaja standardnemu objektnemu C++. Metaobjektni sistem temelji na sledečih zahtevah [17]:

1. Razredi, ki želijo uporabljati metaobjektni sistem morajo dedovati iz *QObject*.
2. Znotraj privatne sekcije deklaracije razreda moramo zapisati *Q_OBJECT* makro, ki omogoči metaobjektne lastnosti, kot so signali in reže ter dinamične lastnosti.
3. Metaobjektni prevajalnik ali moc (angl. Meta-Object Compiler, *moc*) [18], ki vsem razredom doda potrebno kodo, da je razred udeležen v metaobjektnem sistemu.

Izpis 3.7 prikazuje primer razreda, dedovanega iz razreda *QObject* ter razširjenega s sistemom signalov in rež.

```
#include <QObject>
class Stevec : public QObject {
    Q_OBJECT
public:
    Stevec() { _vrednost = 0; }
    int vrednost() const { return _vrednost; }
 public slots:
    void nastaviVrednost(int vrednost);
signals:
    void vrednostSpremenjena(int nova);
private:
   int _vrednost;
};
```

Izpis 3.7: Razred *Stevec* je dedovan iz razreda *QObject* in razširjen s sistemom signalov in rež.

Možno, semantično in sintaktilno pravilno je, da razred deduje iz objekta *QObject* in ne deklarira makroja *Q_OBJECT*. Prva posledica tega je, da metaobjektni sistem za ta razred ne bo na voljo in s tem tudi ne razširitev kot so signali, reže in lastnosti. To opazimo zlahka, kar pa ni nujno tudi za drugo pomanjkljivost, ki jo s tem ustvarimo. Za tak razred bo pri povpraševanju po metapodatkih odgovor podal prvi starš, ki ima metaobjektni sistem. To je seveda nepravilno in je napaka, ki jo je težko odkriti. Da bi se ji izognili, je s strani razvijalcev ogrodja Qt priporočeno, da vsak razred, ki deduje iz objekta *QObject*, deklarira tudi makro *Q_OBJECT*.

Predprevajalniku `moc` pa se ne moremo izogniti, saj brez njegovega posredovanja standardni prevajalnik C++ ne bo znal interpretirati rezerviranih besed kot so *signals*, *slots* ali pa *emit*.

Z metaobjektnim sistemom pa pridobimo veliko prednosti, med katerimi so [17]:

- povpraševanje po imenu razredov med izvajanjem brez vključene podpore za informacije o tipih (angl. RTTI) prevajalnika C++,
- povpraševanje iz katerega razreda je objekt dedovan,
- podpora internacionalizaciji,
- dinamične lasnosti med izvajanjem,
- ustvarjanje razredov med izvajanjem ter
- dinamične pretvorbe (angl. dynamic cast) brez vključene podpore RTTI.

Kot primer zapišimo poizvedbo v metaobjektni sistem okolja Qt. Okolje Qt za vse razrede, udeležene v metaobjektnem sistemu Qt, kreira primerek razreda *QMetaObject*, ki omogoča poizvedovanje po lastnostih razreda, kot so npr. ime razreda, kdo je starš razreda ter katere signale in reže ta razred ima.

Izpis 3.8 prikazuje, kako izpišemo vse lastnosti preko metapodatkov gumba, ki smo ga uporabili v aplikaciji »Pozdravljen svet«.

```
const QMetaObject *metaObjekt = gumb->metaObject();
for (int i = 0; i < metaObjekt->propertyCount(); ++i) {
    QMetaProperty metaLastnost = metaObjekt->property(i);
    qDebug() << "Ime: " << metaLastnost.name() << ", vrednost: " <<
gumb->property( metaLastnost.name() );
}

Izpis:
Ime: objectName , vrednost: QVariant(QString, "")
Ime: modal , vrednost: QVariant(bool, false)
Ime: windowModality , vrednost: QVariant(int, 0)
Ime: enabled , vrednost: QVariant(bool, true)
Ime: geometry , vrednost: QVariant(QRect, QRect(192,212 108x23) )
Ime: frameGeometry , vrednost: QVariant(QRect, QRect(184,184
124x59) )
Ime: normalGeometry , vrednost: QVariant(QRect, QRect(192,212
108x23) )
...
```

Izpis 3.8: Izvorna koda, ki izpiše vse lastnosti objekta *gumb*.

3.6 Lastnosti

Okolje Qt razvijalcem ponuja napreden sistem lastnosti (angl. properties), ki jih lahko dodamo razredom pred prevajanjem ali pa celo med izvajanjem, v vsakem primeru pa se obnašajo kot spremenljivke razreda (angl. member variables). Ta sistem spominja na sisteme, ki jih ponujajo nekateri prevajalniki z nestandardnimi lastnostmi (npr. *_property* ali *[property]*), vendar pa sistem lastnosti okolja Qt deluje tako, da ni odvisen od prevajalnika, zaradi česar je prenosljiv med prevajalniki in tudi med platformami [19].

Nekaj primerov lastnosti razreda *QWidget* je ponazorjeno v izpisu 3.9.

```
Q_PROPERTY(bool fokus READ imaFokus)
Q_PROPERTY(bool omogocen READ jeOmogocen WRITE nastaviOmogocen)
Q_PROPERTY(QCursor kazalec READ kazalec WRITE nastaviKazalec RESET
odnastaviKazalec)
```

Izpis 3.9: Nekaj lastnosti razreda *QWidget*.

Do same lastnosti objekta lahko dostopamo že na podlagi njenega imena (glej vrstico 4 v izpisu 3.10), torej ne da bi nam bilo potrebno imeti o objektu, kateremu ta lastnost pripada, kakršnekoli podatke.

```
QPushButton *gumb = new QPushButton;
QObject *objekt = gumb;
gumb->setDown(true);
objekt->setProperty("down", true);
```

Izpis 3.10: Nekaj lastnosti razreda *QWidget*.

Za razliko od zgoraj navedenega je dostopanje do lastnosti objekta preko direktnega klica metode (glej vrstico 3 v izpisu 3.10) hitrejše, pri prevajanju pa omogoča tudi odkrivanje napak. Pomanjkljivost takšnega dostopanja je, da moramo imeti pri njegovi uporabi v času prevajanja o razredu vse podatke. Lastnosti objekta pa lahko preprosto odkrijemo s pomočjo primera, ki smo ga zapisali v izpisu 3.8.

S klicem metode *QObject::setProperty()* lahko lastnosti dinamično dodajamo objektu tudi med samim izvajanjem. Če lastnost ob klicu metode že obstaja, se ji bo nastavila vrednost, ki jo podamo kot argument, v nasprotnem primeru pa bo okolje Qt to lastnost ustvarilo ter jo dodalo objektu, nad katerim smo predmetno metodo izvršili. Tako dodane lastnosti lahko tudi odstranimo. Pomembno je omeniti še dejstvo, da so lastnosti na voljo le za tisti objekt, preko katerega smo te metode izvršili in ne za vse primerke istega razreda

Dinamičnost ter posledično tudi fleksibilnost lastnosti v okolju Qt, je zelo uporabna pri razvoju razčlenjevalnikov kode (angl. parser), v sledečih poglavjih pa bomo videli da tudi drugje.

3.7 Orodje *moc*

Omenili smo že, da sistema signalov in rež ter lastnosti, pred samim prevajanjem za svoje delovanje potrebujeta dodaten korak (angl. precompiler). Le – tega ob izdelavi primera »Pozdravljen svet« zaradi orodja `qmake` sploh nismo opazili, saj ga ta v proces gradnje vključi samodejno. Ta korak predprevajanja izvede orodje *`moc`* (angl. Meta-Object Compiler), njegovo delovanje pa je predmet tega podpoglavja [18].

Omenili smo že, da dodatne zmogljivosti okolja Qt (npr. signali in reže, lastnosti ter celoten sistem metaobjektov) niso del standarda C++, kljub temu pa se aplikacije Qt ter samo okolje Qt prevajajo na standardnih prevajalnikih C++. Takšno delovanje omogoča korak predprevajanja, ki dopolni izvorno kodo razvijalca, preden je le – ta predana

prevajalniku. Korak predprevajanja pretvori signale in reže ali pa lastnosti v standardni jezik C++ na način, da izvorna koda, ki jo razvijalec napiše, ohrani prvotno obliko.

Predprevajalnik `moc` prebere vse glave datotek izvorne kode C++ in v primeru, da predprevajalnik v deklaraciji razreda naleti na makro *Q_OBJECT*, ustvari zanj izvorno kodo, ki za ta razred implementira metaobjektni sistem. Predmetna izvorna koda med drugim zajema implementacijo potrebno za delovanje signalov in rež, informacije o tipih med izvajanjem ter dinamičen sistem lastnosti. Ta izvorna koda mora biti prevedena in povezana skupaj z izvorno kodo razreda – če za proces gradnje uporabljamo orodje `qmake`, je ta korak dodan samodejno in ostane za razvijalca priročno skrit.

3.8 Implicitna souporaba

Z namenom maksimizirati uporabo virov ter minimizirati kopiranje, implementira veliko razredov okolja Qt mehanizem imenovan implicitna souporaba. Razrede v implicitni souporabi lahko brez težav podajamo v metode kot argumente, saj se podatki razredov pri tem ne bodo kopirali (podatki se bodo skopirali šele takrat, ko bo prišlo do operacije pisanja (angl. copy on write)).

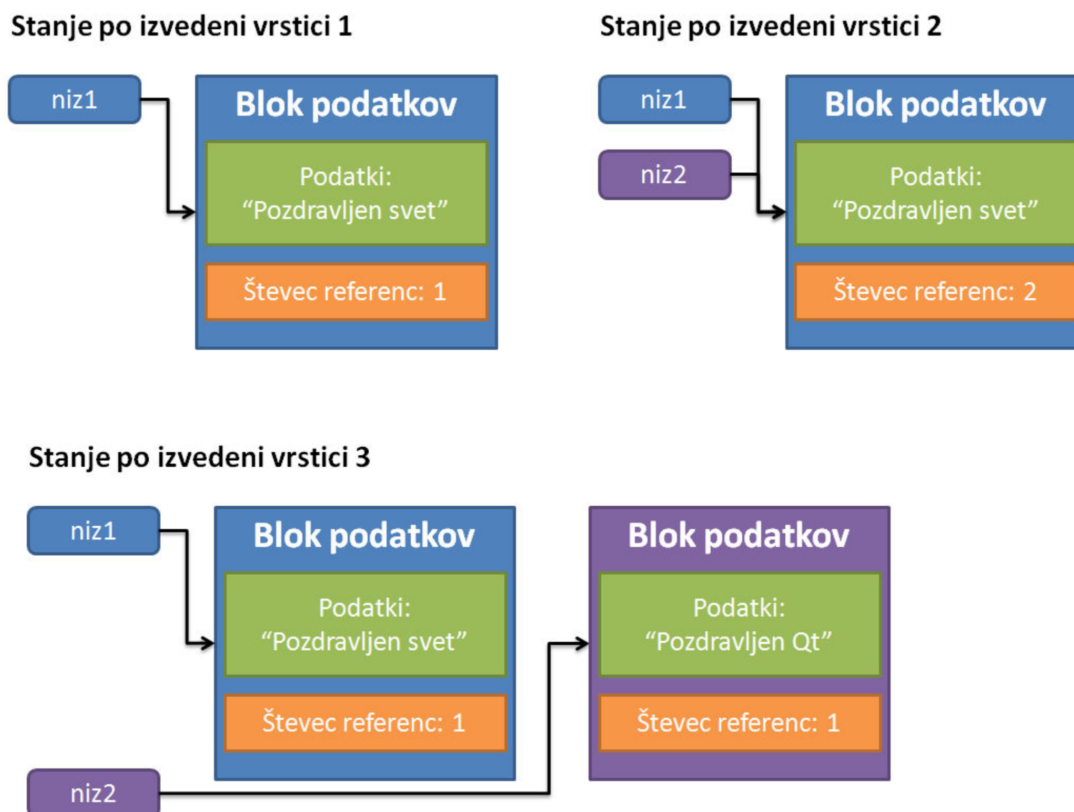
Razred v implicitni souporabi je sestavljen iz kazalca na blok podatkov v souporabi, le – ta pa je sestavljen is števca referenc in podatkov. Ko ustvarimo razred v souporabi, se števec postavi na vrednost 1, poveča pa se, kadarkoli se ustvari nov primerek razreda, ki referencira iste podatke. Kadar kateri od teh razredov podatkov ne referencira več, pa se ta zmanjša. Dejanski podatki v souporabi se zbrišejo, ko se števec referenc postavi na vrednost 0.

V nadaljevanju si oglejmo primer za razred *QString*, ki je eden izmed razredov, udeleženih v implicitni souporabi [4].

```
QString niz1 = "Pozdravljen svet";
QString niz2 = niz1;
niz2.replace( "svet", "Qt" );
```

Izpis 3.11: Nekaj lastnosti razreda *QWidget*.

Blok podatkov v souporabi glede na vrstico izvajanja izvorne kode zapisane v izpisu 3.11 rezultira v obliki, kot je prikazano na sliki 3.3.



Slika 3.3: Razred *QString* in spremembe v bloku podatkov glede na vrstico izvajanja primera.

Pojasnimo še dva pojma: plitko kopiranje (angl. shallow copy) in globoko kopiranje (angl. deep copy). Pri plitki kopiji se podatki ne kopirajo; slednje se je v našem primeru zgodilo po izvedbi druge vrstice. Ob izvedbi tretje vrstice, pa je zaradi spreminjanja podatkov v souporabi s strani objekta *niz2*, bila potrebna globoka kopija; cel blok podatkov se je skopiral, nato pa so se nad njimi izvedle spremembe. Iz navedenega sledi, da so plitke kopije glede spomina in hitrosti izvajanja zelo nezahtevne, globoke kopije pa so seveda počasnejše in zahtevajo več pomnilnika.

Za mnoge razrede ogrodja Qt je implicitna souporaba že implementirana in se tako od razvijalcev aplikacij v ogrodju Qt za njeno uporabo ne zahteva nikakršna aktivnost. Naštejmo le nekaj pomembnejših razredov ogrodja Qt, ki implementirajo implicitno souporabo: *QByteArray*, *QFont*, *QFontInfo*, *QFontMetrics*, *QHash*, *QImage*, *QList*, *QMap*, *QPixmap*, *QQueue*, *QRegExp*, *QString*, *QUrl*, *QVector* ter *QVariant*.

Pomembno je, posebej še za razvijalce mobilnih aplikacij, da vemo, kateri razredi so v implicitni souporabi, saj se lahko s tem izognemo nepotrebni porabi pomnilnika in procesorja.

3.9 Izjeme v ogrodju Qt

Omeniti velja še eno zanimivo odločitev, ki so jo sprejeli razvijalci okolja Qt. Nobeden izmed razredov okolja ne bo sprožil izjeme (ang. exception), temveč bo vračal napake prek rezultata izvajanja neke funkcije, ali pa preko argumentov, ki jih prenesemo po referenci. Te napake so v večini primerov vnaprej definirane vrednosti, veliko razredov okolja Qt pa ima pripravljene tudi metode za vračanje opisa napak, ki ga lahko posredujemo uporabnikom (npr. *QIODevice::errorString()*). Razlog za neuporabo izjem je delno posledica odločitev sprejetih v preteklosti, delno pa tudi zaradi priročnosti (npr. z vključenimi izjemami je velikost knjižnice Qt tudi do 20% večja) [20].

3.10 Dogodki in filtri dogodkov

Dogodki v okolju Qt so objekti, izpeljani iz razreda *QEvent*, ki opisujejo dogajanje znotraj same aplikacije ali v njeni okolici. Dogodke, ki jih lahko obravnavajo vsi razredi, izpeljani iz razreda *QObject*, delimo v tri kategorije [21]:

1. Spontani dogodki (angl. spontaneous events); ustvari jih okenski sistem, uvrščeni so v sistemsko vrsto dogodkov, dokler jih ne obdela dogodkovna zanka.
2. Prilepljeni dogodki (angl. posted events); ustvari jih okolje Qt ali aplikacija Qt, uvrščeni so v vrsto dogodkov okolja Qt, dokler jih ne obdela dogodkovna zanka.
3. Poslani dogodki (angl. sent events); tudi te ustvari okolje Qt ali aplikacija Qt, vendar pa so poslani neposredno prejemniku dogodka.

Opremljeni s tem znanjem lahko zapišemo psevdo kod za dogodkovno zanko orodja Qt. Spomnimo se, da jo poženemo s klicem *QApplication::exec()*, in sicer iz zadnje vrstice primera, ki smo ga zapisali v izpisu 3.1 na začetku tega poglavja.

```
while (!izhod_je_bil_klican) {
    while (!vrsta_prilepljenih_dogodkov_je_prazna) {
        obdelaj_naslednji_prilepljen_dogodek();
    }
    while (!vrsta_spontanih_dogodkov_je_prazna) {
        obdelaj_naslednji_spontan_dogodek();
    }
    while (!vrsta_prilepljenih_dogodkov_je_prazna) {
        obdelaj_naslednji_prilepljen_dogodek();
    }
}
```

Izpis 3.12: Psevdokod dogodkovne zanke orodja Qt.

Verjetno bi kdo pomislil, da je v psevdokodu izpisa 3.12 tiskarska napaka, saj se obdelovanje prilepljenih dogodkov podvaja, vendar pa temu ni tako. Druga zanka je potrebna zato, da okolje Qt obdela vse prilepljene dogodke, ki so se zgodili med obdelavo spontanih dogodkov. Poslani dogodki niso omenjeni, ker jih dogodkovna zanka ne obdeluje, kot povedano, pa so poslani neposredno prejemniku dogodka.

Kako deluje okolje Qt z dogodki, prikažimo na praktičnem primeru. Ko se gradnik prvič prikaže na zaslonu, okenski sistem (angl. window system) ustvari spontan dogodek za risanje (angl. paint event). Po zgornjem algoritmu delovanja dogodkovne zanke, ta vzame dogodek iz vrste in ga pošlje gradniku, kateremu je namenjen. Ni pa nujno, da vsi dogodki za risanje izvirajo iz sistema. Velikokrat jih ustvarimo sami, na primer zatem, ko pridobimo določene informacije in bi jih radi ponovno izrisali na zaslonu. To storimo s klicem metode *QWidget::update()*, ki ustvari dogodek za risanje kot prilepljen dogodek.

Na vprašanje, v čem je razlog, da bi se trudili z razpošiljanjem dogodkov, če pa lahko metodo za ponoven izris pokličemo neposredno, lahko odgovorimo, da zna okolje Qt prilepljene dogodke optimizirati, in sicer na način, da več dogodkov istega tipa združi v enega samega. Če v zanki desetkrat pokličemo prej omenjeno metodo *update()*, okolje Qt ne bo ustvarilo deset dogodkov, temveč le enega, ta pa bo vseboval deset regij za ponoven izris; podobno deluje okolje Qt tudi za druge dogodke, npr. za dogodke o miški.

Pri razvoju aplikacij lahko razvijalci dogodke obdelujejo na več načinov [21]:

- implementacija upravljalca dogodkov (angl. event handler). Razreda *QObject* in *QWidget* ponujata veliko število upravljalcev dogodkov, ki jih lahko preko virtualnih funkcij preobložimo s svojo implementacijo.

```
void MojRazred::mousePressEvent(QMouseEvent *dogodek)
{
    if (dogodek->button() == Qt::LeftButton) {
      // obdelaj levi gumb
    } else {
      // prepusti obdelavo staršu
      QCheckBox::mousePressEvent(dogodek);
    }
}
```

Izpis 3.13: Implementacija upravljalca dogodkov.

Če naša implementacija ne obdela vseh možnih scenarijev, je prav, da dogodek vrnemo obdelavi s strani starša, kot smo to storili v izpisu 3.13 za vse dogodke pritiska miške, ki niso pritisk na levi gumb.

- Implementacija filtra dogodkov na objekt *QObject*. Navedeno uporabimo, kadar želimo prestreči dogodke, ki so namenjeni tudi drugim objektom. Če ponazorimo s primerom, se lahko navedenega načina posluži implementacija gradnika dialog, ki mora (če je aktiven) ob pritisku na tipko »Enter«, izvesti določeno akcijo.

```
bool Filter::eventFilter(QObject *objekt, QEvent *dogodek)
{
    if (objekt == target && dogodek->type() == QEvent::KeyPress) {
        QKeyEvent *dogodekTipke = static_cast<QKeyEvent
*>(dogodek);
        if (dogodekTipke->key() == Qt::Key_Enter) {
            // Prestrežemo tipko »Enter« in storimo kar želimo
            return true;
        } else
            return false;
    }
    return false;
}
```

Izpis 3.14: Implementacija filtra dogodkov na objektu *QObject*.

Že iz izvorne kode v izpisu 3.14 je razvidno, da mora objekt *Filter* zaradi uporabe metode *eventFilter()* dedovati iz razreda *QObject*, poskrbeti pa moramo tudi za to, da primerek razreda *Filter* registriramo kot filter dogodkov, in sicer s klicem metode *QObject::installEventFilter()*.

- Implementacija metode *QObject::event()*, ki razpošilja dogodke različnim upravljalcem dogodkov za določen objekt.
- Implementacija filtra dogodkov na *qApp*, ki bdi nad vsemi dogodki v aplikaciji in ne le nad tistimi, ki so namenjeni enemu objektu. Te možnosti naj bi se poslužili le

izjemoma, saj z njo tvegamo povzročitev napak in nepričakovanega delovanja aplikacije.

- Implementacija metode *QApplication::notify()*, ki jo uporablja dogodkovna zanka okolja Qt za razpošiljanje dogodkov. Če jo preobložimo, bomo obveščeni o dogodkih preden se bodo prožile ostale možnosti implementacij obdelave dogodkov.

Z dogodki se bomo srečali še v kasnejših poglavjih, kjer nam bo do sedaj povedano znanje o obdelavi in prestrezanju dogodkov zelo koristilo. Napisali bomo preprost razpoznavalnik kretenj (angl. gesture recognizer), le – ta pa bo pretvarjal premike miške v kretnje, ki jih bomo lahko na višjem nivoju lažje obdelovali.

4 RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILJE NAPRAVE

Prejšnja poglavja so nas podučila, kaj okolje Qt je in kako deluje. S pomočjo osvojenega znanja bomo v tem poglavju razvili aplikacijo za prenosne telefone (angl. mobile phone), ki delujejo tudi na platformi Symbian^1 (Symbian S60 5th Edition), vendar pa bo aplikacija zaradi prenosljivosti okolja Qt delovala tudi na ostalih platformah (preverili smo Maemo, Windows Vista in Windows 7).

Izbrali smo si eno izmed obstoječih aplikacij platforme Symbian^1 in jo predelali tako, da jo izboljšamo ter ponovno razvijemo v okolju Qt. V tem poglavju bomo preučili trenutno stanje, preučili in načrtovali nov izgled aplikacije ter zapisali vse posebnosti, ki so bile potrebne za njen razvoj.

4.1 Branje novic na mobilnih napravah Symbian^1

Branje novic preko protokola RSS je na platformi Symbian^1 vgrajeno kar v spletni brskalnik. Bralnik novic iz virov RSS (angl. RSS news reader) ima sicer pristen izgled Symbian^1, ki pa med današnjimi uporabniškimi vmesniki daje precej suhoparen in neroden izgled. Delo z bralnikom novic je (kot tudi v drugih aplikacijah Symbian^1) oteženo zaradi zahteve po dvojnem pritisku za izvršitev akcije (angl. double tap); prvi pritisk element namreč označi, drugi pa na njem sproži akcijo.

Na sliki 4.1 si oglejmo delovanje bralnika, ki je prednaložen s platformo Symbian^1, na prenosnem telefonu Nokia N97 mini.

Slika 4.1: Da lahko preberemo prvo novico, moramo izvesti šest korakov.

Po pregledu s slike 4.1 opazimo, da so elementi na zaslonu majhne višine, uporaba vizualnih elementov, efektov in animacij pa je minimalna oziroma je sploh ni. Podpora kretnjam (angl. gestures) je sicer na voljo, vendar le za premikanje po seznamu, večino akcij pa mora uporabnik izvesti preko izbir iz menijev. Takšno delovanje pusti uporabniku slab vtis in željo po bogatejši uporabniški izkušnji. Glede na zmogljivosti modernih prenosnih telefonov lahko ustvarimo mnogo boljšo aplikacijo, ki bi izkoriščala podporo platforme Symbian^1 značilnostim, kot so zaslon, občutljiv na dotik, ločljivost nHD in 24 bitno barvno globino.

Preverimo, ali lahko aplikacijo v okolju Qt zasnujemo brez vseh prej navedenih slabosti.

4.2 Načrtovanje in oblikovanje

Delovanje in izgled aplikacije moramo izdelati pred začetkom samega razvoja. Ta proces obsega kar nekaj korakov in ga na tem mestu zaradi obsežnosti ne bomo podrobneje opisovali. Vseeno pa bomo navedli najpomembnejše korake in pripomočke, kot jih najdemo na spletni strani Forum Nokia [22], ki te procese opisuje s ciljem načrtovanja aplikacij za prenosne telefone.

Faza raziskovanja

V fazi raziskovanja je potrebno poiskati odgovore na vprašanja, ki si jih zastavimo, ko poskušamo opredeliti nov izdelek. Skozi to fazo ustvarjamo nove ideje, jih ocenjujemo in izločamo, preučimo konkurenčne izdelke, definiramo končne uporabnike ter poskusimo predvideti njihova ravnanja [23].

Pri koraku definiranja končnih uporabnikov obstaja zelo dobro orodje, ki ga lahko prenesemo na vse nivoje razvoja programske opreme, zato ga omenimo tudi tukaj: razvoj osebnostnih kartic - person (angl. personas). Osebnostne kartice nam omogočajo, da se razvijalci in načrtovalci poistovetijo s končnimi uporabniki, ki jih te kartice definirajo.

Faza raziskovanja je temelj za vse nadaljnje faze, saj lahko preko nje zelo hitro spoznamo nove in dobre ideje, slabe pa ovržemo. Poleg dobre osnove, ki je bistvena za dober končni idelek, pa faza raziskovanja prispeva tudi k oblikovanju širokega spektra informacij, ki jih uporabljamo skozi naslednje korake načrtovanja in oblikovanja.

Konceptualno načrtovanje

Bistveno pri konceptualnem načrtovanju (angl. conceptual design) je, da ustvarimo prve zametke izdelka, ki ga načrtujemo; pri tem pretresemo vse ideje in probleme ter se z omejitvami ne ukvarjamo.

Naslednji korak je skiciranje, ki je cenovno in časovno zelo ugodno orodje, predvsem v primerjavi s kodiranjem. Zbrane ideje preprosto prenesemo v skice na papirju, te skice pa umestimo v scenarije (slika 4.2). Ponovno imamo na voljo kar nekaj orodij, omenimo samo predloge, ki so na voljo na spletni strani Forum Nokia [24] in nam za aplikacije na prenosnih telefonih olajšajo skiciranje.

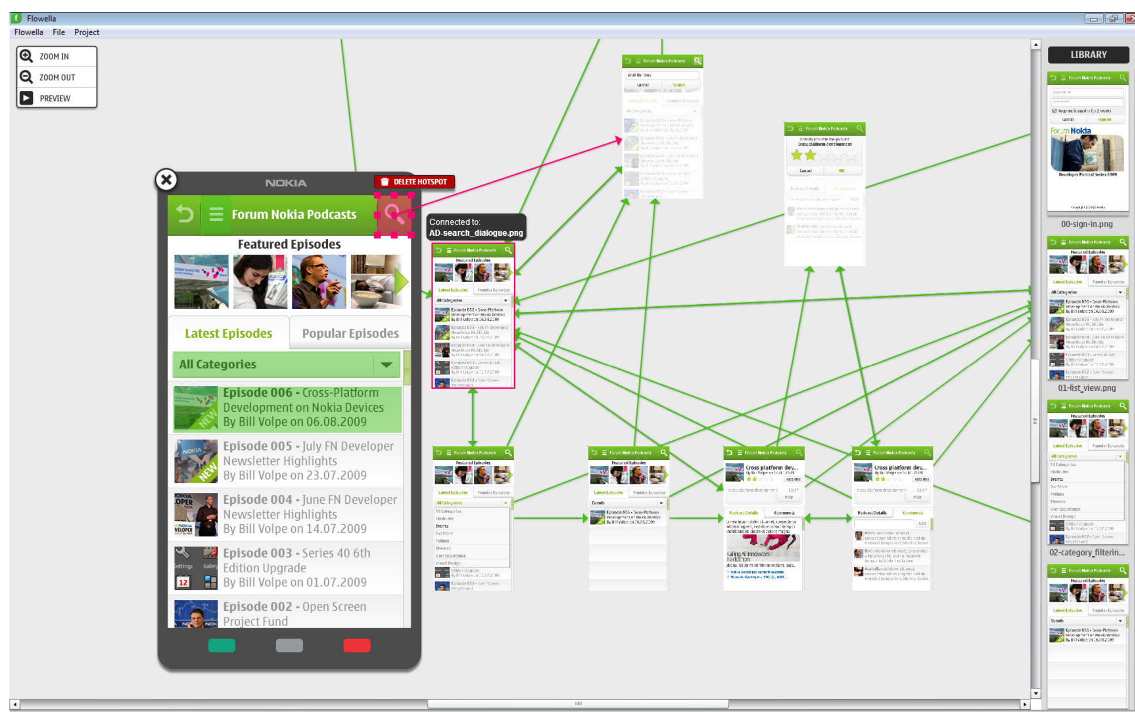
Slika 4.2: Izpolnjena predloga s skicami za aplikacijo, ki služi kot potovalni pomočnik.

Scenariji umestijo določeno funkcionalnost ali skico v določeno okolje in splet okoliščin. Navedene scenarije zapišemo na način, da uporabimo osebnostne kartice, jih umestimo v neko okolje in v niz dogodkov ter preko njih opišemo določeno funkcionalnost izdelka, ki ga razvijamo. Ti scenariji so seveda izmišljene zgodbe, vendar pa nam omogočijo, da v že zelo zgodnji fazi izdelave izdelka predvidimo situacije in načine uporabe našega izdelka s strani končnih uporabnikov. To na eni strani vodi v razvoj bolj kakovostne rešitve, na drugi strani pa prihranimo pri času razvoja in pri vzdrževanju samega izdelka.

Načrtovanje interakcije in prototipiranje

Načrtovanje interakcij (angl. interaction design) definira obnašanje izdelka, mesta pritiskov, ki izvedejo akcije (angl. touch points) ter ostale interakcije med uporabnikom in izdelkom. Paziti moramo, da interakcija sledi načelom konsistentnosti, zaupanja izdelku, iznajdljivosti, odzivnosti in užitku.

Ko z načrtovanjem interakcije končamo, sledi naslednji korak, in sicer prototipiranje (angl. prototyping). Forum Nokia ponuja za to nalogo odlično orodje imenovano »Flowella« (slika 4.3) [25].



Slika 4.3: Orodje Flowella omogoča hitro izdelovanje funkcionalnih prototipov.

Flowella nam omogoča, da ustvarimo prototipe iz obstoječih slik, na katerih označimo mesta pritiskov in prehode med zasloni; vse to brez da bi napisali vrstico kode. Tako ustvarjene prototipe nam Flowella prevede v aplikacijo, ki jo lahko uporabimo na prenosnih telefonih proizvajalca Nokia ter tudi na namiznih računalnikih.

Načrtovanje izgleda

Do sedaj smo uporabniški vmesnik opisovali preko preprostih skic in opisov. V tem koraku pa izgled uporabniškega vmesnika določimo dokončno: izberemo barvno shemo in pisave, izdelamo ikone ter izgled komponent na zaslonu, hkrati pa komponentam in informacijam na zaslonu določimo še razporeditev (angl. layout).

V zvezi z dejstvom, kako pomemben je sam izgled izdelka, opozorimo na zanimivo ugotovitev oblikovalca ter kognitivnega psihologa Dona Normana[1], kjer ta ugotavlja, da

[1] Don Norman, Emotional Design: Why We Love (or Hate) Everyday Things: »*Findings suggest the role of aesthetics in product design: attractive things make people feel good, which in turn makes them think more creatively. How does that make something easier to use? Simple, by making it easier for people to find solutions to the problems they encounter.*«

uporabniki napake lažje odpustijo v izdelkih, ki so zanje vizualno in estetsko prijetnejši. Seveda to ne predstavlja enega izmed načinov izogibanja napak v izdelku ampak le ugotovitev, ki poudarja pomembnost izgleda izdelka pri uporabnikih [26].

Testiranje, zagotavljanje kakovosti in vrednotenje

Bistvo testiranja predstavlja odkrivanje napak kar se da zgodaj v razvoju. Za samo izvedbo testiranja moramo pridobiti čim več mnenj o že izdelanih specifikacijah izdelka, težave v delovanju moramo odkriti in rešiti čim prej ter se na vse odkrite težave in napake odzvati, prilagoditi in jih odpraviti. Prej ko težave in napake odkrijemo, lažje jih bomo odpravili; ravno zato je zelo pomembno, pa čeprav je korak testiranja opisan kot zadnji, da ga poskušamo izvajati čim pogosteje in že med prej omenjenimi fazami razvoja in načrtovanja.

4.3 Moderni bralnik novic za prenosne telefone

Za cilj si postavimo načrtovati bralnik novic iz virov RSS na takšen način, da bo zanimiv čim širši množici. To storimo tako, da postavimo vsebino v samo središče aplikacije, celoten sistem nastavitev in navigacije pa omejimo le na nekaj gumbov, ki postopoma odkrivajo možnosti, preko katerih lahko z aplikacijo upravljamo. Navigiranje aplikacije omogočimo s kretnjami, prav tako pa poskrbimo za vizualni odziv elementov, s čimer uporabnik takoj vidi, da je pritisnil na nek gradnik, ki akcijo sproži.

Sedaj ponazorimo, kako novice najlažje postaviti v središče aplikacije. Skoraj vsaka novica je opremljena s fotografijo, ki še pred branjem novice ustvari prvi vtis. Ker pa fotografije na listi vseh novic v obstoječem bralniku niso prikazane, uporabnik fotografijo vidi šele takrat, ko začne novico brati.

Slika 4.4: Skica izpisa novice, kjer je fotografija postavljena v središče.

Navedeno popravimo tako, da fotografije novic enačimo z elementom v listi novic, čez fotografije pa na polprozornem ozadju izpišemo še naslov novice (slika 4.4).

Primarni zaslon aplikacije je lista vseh novic iz izbranega vira RSS. Ko uporabnik katero izmed novic izbere, se prikaže zaslon za branje novic. Na tem zaslonu pa je vidno celotno besedilo novice ter fotografija z ohranjenim razmerjem višine in širine. Oba zaslona prikazuje slika 4.5.

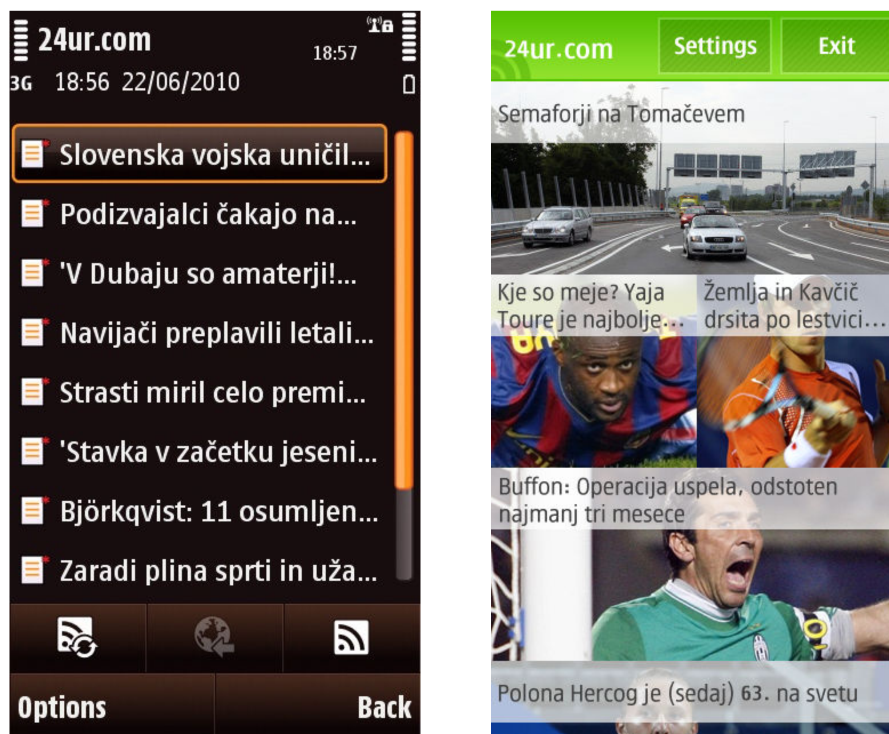
Slika 4.5: Skice zaslonov, ki se bodo v bralniku novic največ uporabljali.

4.4 Uporabniški vmesniki in ogrodje Graphics View

Dandanes vse več aplikacij na prenosnih telefonih vsebuje uporabniške vmesnike, ki črpajo svojo zasnovo iz realnega sveta. To se posledično odraža v animacijah, v uvajanju fizikalnih zakonov, transformacijah v tretji dimenziji in v transparentnosti.

Da bi sledili navedenemu razvoju morajo razvijalci velikokrat poseči izven nabora že obstoječih gradnikov, ki jim jih ponuja platforma ali okolje, v katerem razvijajo aplikacije. Ker so razvijalci zaradi slednjega primorani razviti nove gradnike, so s tem nagrajeni prav uporabniki, saj so moderne aplikacije vizualno atraktivne, prijetne za uporabo, nenazadnje

pa dajejo občutek pristnosti in originalnosti kot dela same naprave, na kateri se uporabljajo [27].



Slika 4.6: Primerjava uporabniškega vmesnika, zgrajenega s standardnimi komponentami (levo) ter z modernimi komponentami (desno).

Splošno o ogrodju Graphics View

Razvoj modernih aplikacij (kot vidno na sliki 4.6) ponuja okolje Qt, in sicer na osnovi ogrodja Graphics View [27]. Slednji nad elementi[1], ki jih vsebuje, omogoča transformacije, animacije in učinke – tako z elementi, optimiziranimi za uporabo v tem ogrodju, kot tudi s »standardnimi« gradniki okolja Qt.

Ogrodje Graphics View deluje preko prizorišča (angl. scene), ki je zmožno upravljati z velikim številom poljubnih 2D grafičnih elementov ter preko pogleda (angl. view), ki te elemente prikazuje, obenem pa omogoča tudi povečave in rotacije. Ogrodje implementira distribucijo dogodkov, ki omogoča elementom na prizorišču odzivanje na dogodke, kot so pritisk in premik miške, pritisk tipke itd.

[1] Gradnike v ogrodju Graphics View razvijemo iz elementov.

Ker ogrodje Graphics View uporablja drevo BSP (angl. Binary Space Partitioning), omogoča hitro odkrivanje elementov na prizorišču, s tem pa tudi upodabljanje velikih prizorišč (tudi do milijon elementov v realnem času).

Prizorišče

Prizorišče je v ogrodju Graphics View predstavljeno z razredom *QGraphicsScene*, ki skrbi za:

- implementacijo vmesnika za hitro upravljanje z velikim številom elementov,
- dostavo dogodkov do vsakega elementa na prizorišču,
- upravljanja s stanjem elementov (npr. z izbiro in fokusom) ter
- implementacijo netransformiranega risanja, primarno mišljenega za tiskanje.

Na prizorišče dodajamo elemente, ki so predstavljeni z razredom *QGraphicsItem*, kasneje pa jih lahko poiščemo preko raznovrstnih iskalnih metod, in sicer glede na to, ali element seka, ali vsebuje podano točko, pravokotnik, vektorsko pot ali pa mnogokotnik.

Pogled

Prizorišče ni gradnik uporabniškega vmesnika in ga kot takega ne moremo dodati na uporabniški vmesnik. Za to potrebujemo pogled, predstavljen z razredom *QGraphicsView*. Pogled izrisuje vsebino prizorišča na zaslon preko predstavitvenega polja (angl. »viewport«). Eno prizorišče lahko prikazuje več pogledov, ki podpirajo pomike, če je prizorišče preveliko, da bi se lahko celo izrisalo na zaslon. Z uporabo OpenGL pa je izrisovanje prizorišča lahko tudi strojno pospešeno.

Pogled je gradnik, ki prejema dogodke iz okolja Qt ter jih transformira v dogodke prizorišča na način, da pretvarja koordinate zaslona v koordinate prizorišča. Pogled lahko opravlja tudi transformacije nad koordinatnim sistemom prizorišča.

Element

Element, predstavljen z razredom *QGraphicsItem*, je osnovni gradnik za predstavljanje grafičnih gradnikov na prizorišču. Ogrodje Graphics View ima za osnovne geometrijske elemente (npr. pravokotnike, elipse), besedila in slike, že implementiranih veliko

gradnikov, ki jih moramo, da dosežemo zastavljen rezultat, v večini primerov razširiti z dodatno implementacijo ali pa med seboj združevati.

Vsi elementi so postavljeni v lokalni koordinatni sistem ter omogočajo pretvorbo koordinat med elementom in prizoriščem kot tudi med različnimi elementi. Elementi lahko svoj koordinatni sistem tudi transformirajo preko matrik, kar omogoča rotacijo in skaliranje posameznih elementov.

Omeniti velja še, da zaradi hitrosti, razred *QGraphicsItem* ne deduje iz objekta *QObject* [28].

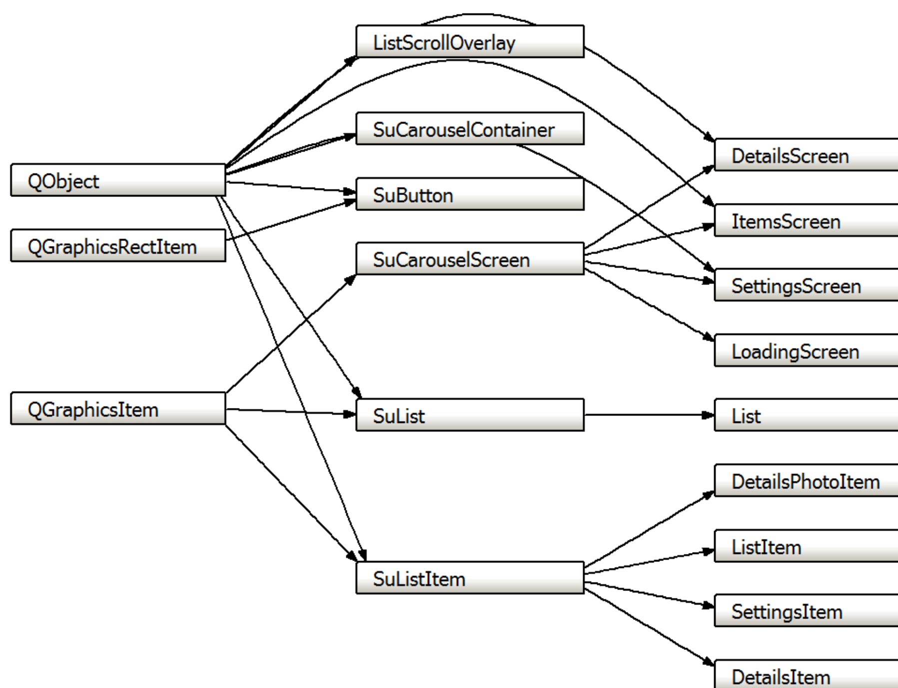
Uporaba ogrodja Graphics View v bralniku novic

Zaradi izdelave uporabniškega vmesnika bralnika novic, je potrebno implementacijo aplikacije opraviti z ogrodjem Graphics View, in sicer moramo razviti naslednje komponente:

- *zaslon* je območje, ki vsebuje gradnike, ki omogočajo uporabo aplikacije. Bralnik novic vsebuje tri zaslone: seznam novic, bralnik novic ter nastavitve, pri čemer pa je lahko zaslonov s seznamom novic več, po en za vsak vir RSS.
- *Upravljalnik zaslonov*, ki bo skrbel za upravljanje z zasloni. Bralnik novic lahko vsebuje več vodoravno razporejenih zaslonov, med katerimi se lahko uporabnik premika. Upravljalnik zaslonov skrbi za izris, pravilen vrstni red in dodajanje ter odstranjevanje le – teh.
- *Gumb* je naenostavnejši gradnik, saj uporabnik nanj le pritisne, ta pa ob tem odda dodeljen signal ter izvede učinek pritiska.
- Da lahko *element* uporabimo večkrat, ga zastavimo čim bolj splošno. V primeru, da element predstavlja novico, bo ta izrisal podrobnosti novice (fotografija in naslov), ob pritisku pa bo oddal dodeljen signal in izvedel učinek pritiska.
- *Seznam* novic upravlja z izrisom ter dodajanjem in odstranjevanjem elementov (novic). Ker pa je lahko novic za hkratni izris na zaslonu preveč, mora seznam poskrbeti tudi za pomikanje (ang. scroll).

Seznam in element si zastavimo tako, da ju lahko uporabimo na vseh zaslonih, ki jih aplikacija uporablja. To storimo na način, da osnovni razred ne implementira risanja ter da seznam podpira elemente različnih velikosti. Vsi zgoraj omenjeni gradniki so dedovani iz

razreda *QGraphicsItem*, v primeru da oddajajo signale, pa so dedovani tudi iz razreda *QObject* (slika 4.7).



Slika 4.7: Okrnjen razredni diagram gradnikov uporabniškega vmesnika.

Sam postopek izrisovanja novic, ki spada med zahtevnejše dele razvoja aplikacije, bomo podrobneje opisali v poglavju o optimizacijah.

4.5 Kretnje

Kretnje[1] (angl. gestures) postajajo pri napravah, ki omogočajo interakcijo preko dotika (angl. touch), vedno bolj uporabljana metoda interakcije med uporabnikom in aplikacijo. Interakcijo preko dotika omogočajo različne naprave, med katerimi je zagotovo najpogostejši prikazovalnik, obstajajo pa tudi miške [29] in sledilne ploščice (angl. touchpad). Trenutno najbolj znane naprave, ki omogočajo interakcijo preko dotika, proizvaja podjetje Apple [29, 30, 31, 32].

[1] V tem delu se bomo omejili le na kretnje, ki jih lahko opravimo pri dotiku z napravo. Zaradi navedenega, bomo kretnjo ob dotiku v tem delu imenovali s splošnim izrazom »kretnja«.

Kretnja je sestavljena iz giba z enim ali več prsti, ki drsijo po površju, občutljivem na dotik. Najenostavnejšo izmed kretenj predstavlja pritisk na gumb, izrisan na zaslonu, občutljivem na dotik. Oglejmo si nekaj najpogostejših kretenj ter njihovo uporabo pri interakciji uporabnika z aplikacijo.

Pritisk

Pritisk (angl. tap) je enkratni pritisk na zaslon v omejenem območju [33]. Čas, ko je prst na zaslonu, je kratek (slika 4.8).

Slika 4.8: Pritisk na zaslon.

Ko je prst pritisnjen na zaslon, aplikacija odreagira na pritisk (npr. označi pritisnjen element), ko pa se prst od zaslona dvigne, se izvede akcija, ki je povezana s pritiskom na ta element.

Poteg

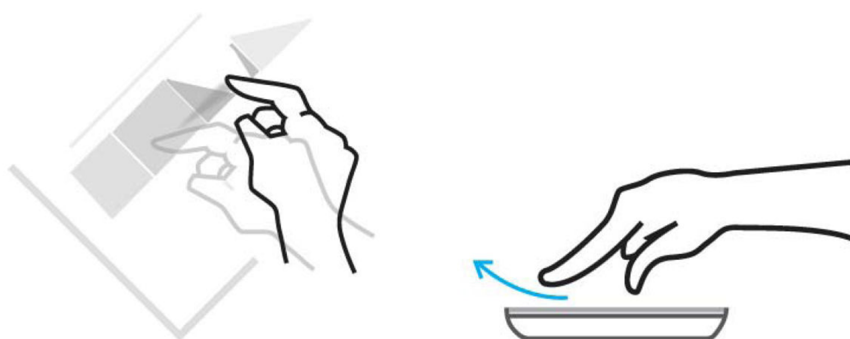
Poteg (angl. pan) je pritisk prsta, ki mu sledi premik v eno ali več smeri [33]. Poteg se konča, ko prst dvignemo, ali ko naredimo drugo kretnjo (slika 4.9).

Slika 4.9: Poteg po zaslonu.

V aplikacijah se poteg uporablja za premikanje vsebine, pri čemer element, ki smo se ga dotaknili, sledi premiku prsta. Poteg se lahko izvede tako v navpični kot tudi vodoravni smeri, pogosto pa se uporablja tudi za preurejanje elementov na zaslonu.

Sunek

Sunek (angl. swipe, flick) je pritisk prsta, ki mu sledi hiter premik v isti smeri ter odmik prsta [33]. Sunek je kot kretnja zelo podoben potegu (le ta se seveda lahko kadarkoli razvije v sunek).



Slika 4.10: Sunek po zaslonu.

Sunek (slika 4.10) se pogosto uporablja za premikanje večje količine vsebin znotraj aplikacije. Pri razpoznavi je najpogosteje omejen na navpično ali vodoravno smer, v aplikacijah pa se uporablja za indikacijo začetka ter karakteristike, potrebne za kinetično pomikanje (angl. kinetic scrolling).

4.6 Kretnje in ogrodje Qt

Okolje Qt že vsebuje implementacijo razpoznavanja kretenj preko razredov *QGestureRecognizer* in *QGesture*, prav tako pa podpira razpoznavo tipičnih kretenj, kot sta poteg in potisk. Kot takšnega bi ga lahko uporabili pri implementaciji aplikacije, vendar pa pri tem naletimo na dve pomembni omejitvi:

1. Ogrodje za razpoznavanje kretenj deluje le na razredih, izpeljanih iz *QWidget* in *QGraphicsObject*, kar pa pomeni dodatne stroške režije, ki jih pri trenutno zastavljenem prikazovanju elementov z *QGraphicsItem* ne potrebujemo [34].

2. Vgrajeni razpoznavalniki za kretnje delujejo samo z razredom *QTouchEvent*, ki pa na trenutnih platformah Symbian^1 ali Symbian^2 še ni uporabljen. Okolje Qt na platformi Symbian^1 in Symbian^2 oddajaja ob interakciji izpeljanke iz razreda *QMouseEvent* [35]. Zaradi navedenega, bi morali sami implementirati razpoznavalnike za kretnje, ki jih drugače okolje Qt že podpira.

Da bi se izognili omenjenima ovirama, smo razvili preprost razpoznavalnik kretenj, ki deluje na elementih ogrodja Graphics View.

Algoritem razpoznavalnika kretenj

Algoritem razpoznave kretenj lahko poenostavimo v štiri ne prezahtevne korake [36], ki se še dodatno poenostavijo z omejitvijo razpoznavalnika le na razpoznavanje kretenj za poteg in sunek.

1. **Filtriranje**

   Filtriranje prepreči beleženje majhnih premikov prsta (slika 4.11). Ker razpoznavamo le poteg in sunek, velike natančnosti ne potrebujemo. Zaradi slednjega se lahko izognemo večji kompleksnosti problema in potrebi po obdelavi večje količine podatkov.

Slika 4.11: Filtriranje kretnje.

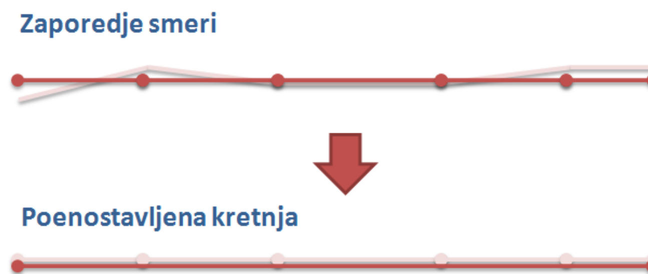
2. **Omejevanje smeri**

   Omenili smo že, da sta kretnji poteg in sunek, omejeni na navpično in vodoravno smer. Ko razpoznavamo kretnjo po vodoravni smeri, nas majhna odstopanja ne zanimajo in jih ne obdelujemo (slika 4.12).

Slika 4.12: Omejevanje smeri kretnje.

3. **Poenostavljanje zaporedja smeri**

   Pri poenostavljanju zaporedja smeri, izločamo zaporedne premike v isti smeri ter jih nadomestimo z enim večjim (slika 4.13).



Slika 4.13: Poenostavljanje zaporedja smeri.

4. **Ujemanje**

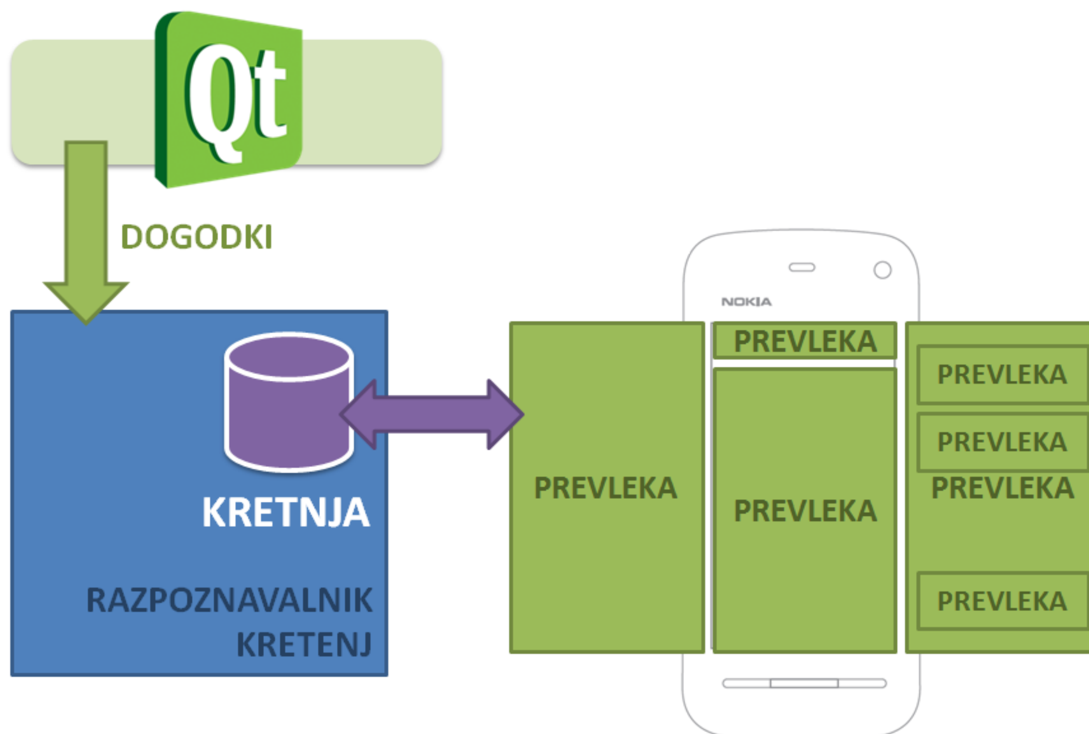
   Pri koraku ujemanja ugotovimo, za katero kretnjo gre. Z naborom kretenj, ki jih bo naš algoritem prepoznaval, bo ta korak preprost, pri sestavljenih kretnjah pa zahteva največ truda.

Implementacija

Za postavitev lastnega ogrodja (slika 4.14) za razpoznave kretenj, potrebujemo tri razrede, in sicer:

- *poslušalca dogodkov*, ki deluje kot filter relevantnih dogodkov iz sistema. Le – ta obdeluje dogodke tipov *QEvent::MouseButtonPress*, *QEvent::MouseButtonRelease* in *QEvent::MouseMove*, ostale pa posreduje nazaj v sistem.
- *Kretnjo*, ki vsebuje podatke o sami kretnji, ki jo uporabnik izvaja.

- *Prevleko* (angl. overlay) za vsak prostor na zaslonu, ki sprejema kretnje. Prevleka je gradnik na zaslonu, ki je popolnoma transparenten in ga uporabnik ne vidi. Uporabljamo ga za obdelavo podatkov za določeno mesto na prizorišču, ki se odziva na kretnje.

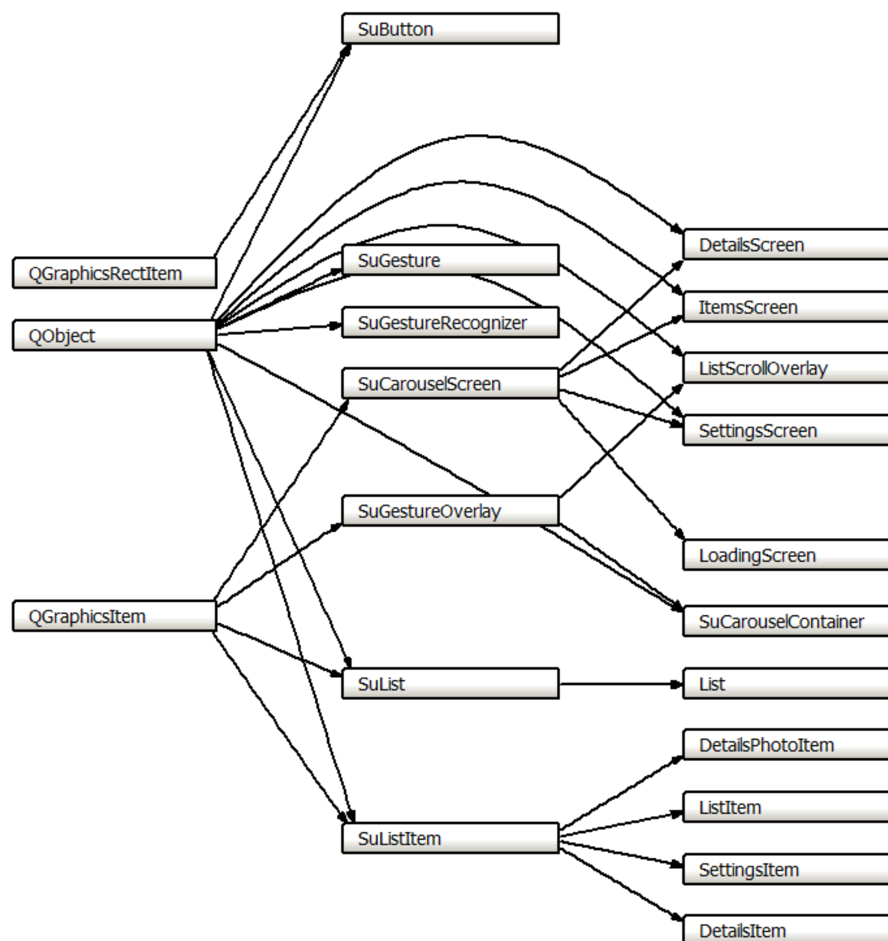


Slika 4.14: Shematski prikaz delovanja razpoznavalnika kretenj.

Prevleke so dedovane iz razreda *QGraphicsItem* in vsebujejo štiri čiste virtualne metode, ki jih morajo razredi, ki kretnje obdelujejo, implementirati. Ker imajo svoj položaj na prizorišču, lahko preprosto uporabimo že implementirano mapiranje koordinat razreda *QGraphicsItem*.

Razpoznavalnik kretenj implementira dogodkovni filter, ki posreduje vse relevantne dogodke razredu *Kretnja*, ta pa jih posreduje v obdelovanje vsem razredom *Prevleka*, ki so aktivni v njihovih koordinatah na prizorišču.

Delovanje sistema bomo podrobneje spoznali v nadaljnjem besedilu, saj nam bo navedeno služilo za implementacijo kinetičnega pomikanja elementov po zaslonu. Razredni diagram sistema za razpoznavo kretenj je prikazan na sliki 4.15.



Slika 4.15: Razredni diagram gradnikov uporabniškega vmesnika z ogrodjem za razpoznavanje kretenj.

Kinetično pomikanje

Kinetično pomikanje je pomikanje, ki ga sproži kretnja sunka in je najprimernejše za pomikanje po daljših seznamih. Elementi seznama se pomikajo samodejno tudi za tem, ko je uporabnik že dvignil prst, hitrost samega pomikanja pa se izračuna iz karakteristik sunka. Močnejši kot je sunek, dalj časa se bodo elementi pomikali, njihova hitrost pa bo ob tem pojemala, kot da so elementi seznama »obteženi«. Primer takšnega pomikanja je sunek kovanca po mizi. Najprej se bo kovanec premikal z veliko hitrostjo, ki pa bo zaradi trenja med kovancem in mizo pojemala, vse dokler se kovanec ne bo popolnoma ustavil.

Takšno pomikanje se je najbolj uveljavilo pri mobilnih napravah z zaslonom, občutljivim na dotik, še posebej pa je njegovo uporabo utrdil proizvajalec Apple, in sicer s prenosnim

telefonom iPhone [30]. Okolje Qt trenutno te podpore še nima vgrajene (z izjemo platforme Maemo), zato bomo kinetično pomikanje implementirali sami.

Slika 4.16: Visokonivojski diagram stanj pri izvajanju kinetičnega pomika ob sunku.

Ko se kretnja prične izvajati, si moramo zapomniti točko pritiska na zaslonu, saj jo bomo potrebovali za poznejše izračune. Če se med začetkom in koncem kretnje dogajajo premiki, izvajamo poteg in temu primerno premikamo elemente po zaslonu. Ob koncu kretnje preverimo, če je uporabnik izvedel sunek in pričnemo izvajati kinetični pomik s pomočjo časovnika. Vsak pričetek nove kretnje mora kinetični pomik ustaviti ter, če se pojavijo nadaljnji premiki, izvajati poteg, v nasprotnem primeru pa kretnjo prekinemo. Opisano prikazuje diagram stanj iz slike 4.16.

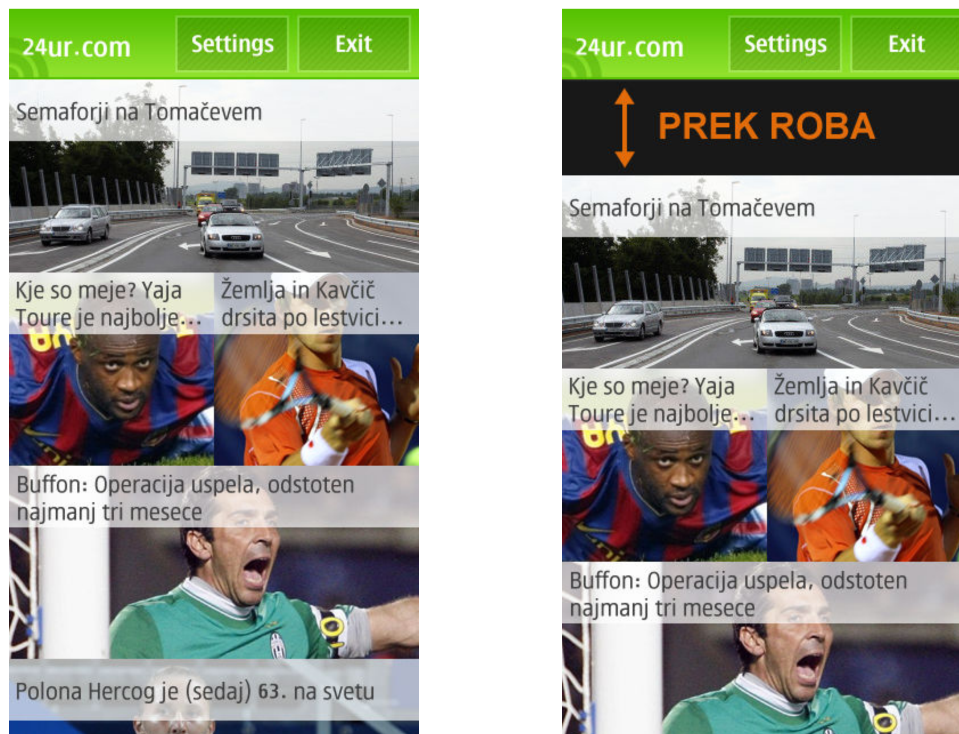
```
void Prevleka::začetekKretnje(QPointF pos) {
    _začetnaTočka = pos;
}

void Prevleka::premikMedKretnjo(QPointF pos, QPointF premik,
QPointF hitrost) {
    normalno_premakni(premik); // Poteg.
}

void Prevleka::konecKretnje(QPointF pos, QPointF hitrost){
    qreal x = pos.x() - _začetnaTočka.x();
    qreal y = pos.y() - _začetnaTočka.y();
    if (!kretnja_je_dovolj_dolga_in_ravna()) {
        return;
    }
```

```
        _hitrost = omejena_hitrost(minHitrost, hitrost.y(),
                                   maxHitrost);
        if (_hitrost != 0) {
            _čas = trenutniČas();
            začniTočkovnik(); // Pazimo, da je dovolj hiter za FPS.
        }
    }

    void Prevleka::prožiMeObIztekuČasovnika {
        bool končaj = true;
        if (_hitrost != 0) {
            zdaj = trenutniČas();
            qreal premik = _hitrost * razlika_med_časoma_v_s(_čas,
                                                             zdaj);
            _čas = zdaj;
            končaj = normalno_premakni(QPointF(0, premik));
            if (!stopTimer) {
                _hitrost = _hitrost > 0.0 ? qMax(qreal(0.0),
                               _hitrost - _pospešek) : qMin(qreal(0.0),
                               _hitrost + _pospešek);
                končaj = (_hitrost == 0.0);
            }
        }
        if (končaj) {
            ustavi_časovnik();
        }
    }
    bool Prevleka::move(QPointF premik){
        if (vsebine_je_dovolj_za_pomik) {
            qreal razdalja = premik.y();
            if (razdalja > 0 && vsebina->položaj().y() < PREK_ROBA) {
               qreal vrh = vsebina->položaj().y() + razdalja;
               vsebina->nastaviPoložajY(vrh>PREK_ROBA ? PREK_ROBA:vrh);
                return true;
            }
            if (razdalja < 0) {
               qreal vrh = m_vsebina->položaj().y() + razdalja;
               if (vrh+vsebina->višina()+PREK_ROBA >= moja_višina()) {
                    vsebina->nastaviPoložajY(top);
                    return true;
                }
            }
        }
        return false;
    }
```

Izpis 4.1: Implementacija kinetičnega pomikanja.

Psevdokod v izpisu 4.1 natančno opisuje implementacijo kinetičnega pomikanja. Omenimo še, da nam hitrost računa razred *Kretnja*, *PREK_ROBA* pa je konstanta, ki pove, koliko se lahko vsebina premakne preko (angl. overshooting) omejujočega pravokotnika (angl. bounding rectangle). Ker se vsebina premakne preko robov omejujočega pravokotnika, je potrebno zgornjemu psevdokodu dodati še implementacijo vračanja vsebine v omejujoči

pravokotnik. To storimo s pomočjo podpore animaciji, ki jih ponuja okolje Qt. Slika 4.17 prikazuje vsebino v omejujočem pravokotniku ter vsebino, ki sega preko robov.



Slika 4.17: Vsebina v omejujočem pravokotniku – levo in preko robov – desno.

4.7 Animacije

Okolje Qt ima že implementirano ogrodje za animacije. Ker deluje prek sistema lastnosti, ki smo ga podrobno spoznali v prejšnjih poglavjih, ponuja Qt pri animiranju gradnikov in katerih koli razredov, dedovanih iz objekta *QObject*, veliko prilagodljivosti.

Enostavne animacije preko sistema lastnosti v okolju Qt lahko opravi razred *QPropertyAnimation*. Te animacije lahko sestavljamo v kompleksnejša zaporedja, ki se izvajajo bodisi sočasno (*QParallelAnimationGroup*), bodisi v zaporedju (*QSequentialAnimationGroup*). V nadaljevanju si oglejmo, kako ustvarimo animacijo za vračanje vsebine v omejujoč pravokotnik.

```
qreal vrh = izračunaj_vrh();

QPropertyAnimation* anim = new QPropertyAnimation( vsebina,
Gradnik::IME_LASTNOSTI_ZA_VRH );
anim->setEasingCurve( QEasingCurve::OutExpo );
anim->setStartValue( vsebina->vrniVrh() );
anim->setEndValue( vrh );
anim->setDuration( 300 );
anim->start( QAbstractAnimation::DeleteWhenStopped );
```

Izpis 4.2: Primer animacije za vračanje vsebine v omejujoč pravokotnik.

Koda iz ispisa 4.2 bo lastnosti z imenom *Gradnik::IME_LASTNOSTI_ZA_VRH* objekta `vsebina` v času tristotih milisekund po krivulji *QEasingCurve::OutExpo* nastavljala vrednosti od *vsebina->vrniVrh()* do *vrh*. Ko pričnemo to animacijo izvajati, ji lahko nastavimo še možnost, da se po končanju izbriše ter tako samodejno poskrbi za čiščenje pomnilnika. Enako bi lahko animirali tudi velikost, barvo, položaj itd. [37]

Pri implementaciji bralnika novic smo animacije uporabili še ob pritisku gumba, pri prehodu med zasloni, za prikazovanje in skrivanje elementov ter za dodajanje in odstranjevanje le – teh v seznam ter iz njega.

4.8 Povezljivost s spletom, snemanje novic in slik

Dostop do mreže, ter s tem povezljivost z zunanjimi viri, je dosežena preko razreda *QNetworkAccessManager*, ki upravlja z zahtevami in odgovori [38]. Kot centralni točki mu zelo enostavno določimo splošne nastavitve (npr. o posredovalnem strežniku (angl. proxy) ali o predpomnilniku), ki jih nato upošteva pri generiranju zahtev. Zaradi navedenega razvijalci okolja Qt svetujejo, da uporabljamo en primerek razreda *QNetworkAccessManager* na aplikacijo.

QNetworkAccessManager ima asinhron programski vmesnik, ki temelji na sistemu signalov in rež. To nam olajša spremljanje napredka mrežne zahteve ter močno poenostavi celotno arhitekturo sistema. Niti, ki bi skrbela za mrežne zahteve, namreč ni potrebno ustvariti ter z njo tudi ne upravljati.

Pri bralniku novic smo ustvarili primerek razreda *QNetworkAccessManager* preko implementiranega razreda *Sistem*, ki vsebuje podatke ter upravlja z globalnimi viri znotraj aplikacije ter je implementiran po vzorcu edinca (angl. singleton). Le–ta poskrbi še za

obravnavanje napak, ki jih v našem primeru ne poskušamo reševati, ampak uporabnika o napaki le obvestimo.

```
QNetworkAccessManager *Sistem::vrniQNAM() {
    Sistem *inst = instanca();
    if (!inst->nam) {
        inst->nam = new QNetworkAccessManager();
//        inst->nam->setProxy(
QNetworkProxy(QNetworkProxy::HttpProxy, "127.0.0.1", 8888 ) ); //
Uporabi ob testiranju
        connect(nam, SIGNAL(authenticationRequired( QNetworkReply
*, QAuthenticator *)), inst, SLOT(onAuthenticationRequired(
QNetworkReply *, QAuthenticator *)));
        connect(nam, SIGNAL(proxyAuthenticationRequired( const
QNetworkProxy &, QAuthenticator *)), inst,
SLOT(onProxyAuthenticationRequired( const QNetworkProxy &,
QAuthenticator *)));
        connect(nam, SIGNAL(sslErrors( QNetworkReply *, const
QList<QSslError> &)), inst, SLOT(onSslErrors( QNetworkReply *,
const QList<QSslError> &)));
    }
    return inst->nam;
}

// Uporaba v razredih
RssBralnik:: RssBralnik (QObject *starš) : QObject(starš),
trenutniUrl(QString::null)
{
    ...
    nam = System::getQNAM();
    connect(nam, SIGNAL(finished( QNetworkReply *)), this,
SLOT(onFinished( QNetworkReply *)));
}

void RssBralnik::preberiRss(const QString &url)
{
    trenutniUrl = QUrl(url);
    nam->get( QNetworkRequest(trenutniUrl) );
}

void RssBralnik::onFinished( QNetworkReply *odgovor )
{
    // Preveri ali je odgovor namenjen nam.
    if (odgovor->url() != trenutniUrl) {
        return;
    }
    // Obdelaj odgovor.
    if (odgovor ->error() != QNetworkReply::NoError) {
        ...
        emit napaka( odgovor->errorString() );
    } else {
        ...
    }
    reply->deleteLater();
}
```

Izpis 4.3: Primer uporabe razreda *QNetworkAccessManager* v razredu *RssBralnik*.

Izpis 4.3 prikazuje, kako razred *RssBralnik* uporablja razred *QNetworkAccessManager* za dostop do novic iz virov RSS. Vidimo, da moramo ob vsakem odgovoru preveriti, ali je ta namenjen »nam«. Zgornji primer prikazuje osnoven primer implementacije. Bolj elegantna oblika implementacije pa bi bila, da bi tudi signal *finished* obravnavali v edincu; razredi, kot je *RssBralnik*, pa bi ob podanem naslovu za zahtevo, podali še kazalec na funkcijo, ki naj se izvede, ko je zahteva obdelana. Edinec bi ta kazalec zapisal v metapodatke zahteve in ga ob končanem odgovoru od tam prebral ter tudi izvršil funkcijo.

Objekt *QNetworkReply*, ki predstavlja odgovor, mora uporabnik zbrisati ob primernem času. Zaradi tega ne smemo uporabiti rezervirane besede `delete`, ampak uporabimo klic funkcije *deleteLater()*, ki objekt uvrsti v vrsto za uničenje. Ko razvijamo za platformo Symbian, pa se moramo, če hočemo dostopati do mreže, zavedati še ene zahteve. Programju, ki ga razvijamo, moramo dati varnostno zmožnost (angl. security capability) *NetworkServices*. To storimo preprosto prek orodja *qmake* ter spremenljivke *TARGET.CAPABILITY*. Dodajmo še, da moramo za povezavo vzpostaviti vsaj en mrežni vmesnik. To storimo preko programskih vmesnikov »Qt Mobility« ali preko platformne kode.

4.9 Izjeme na platformi Symbian

Platforma Symbian implementira lasten sistem izjem, ki pa je drugačen od tistega, ki smo ga vajeni pri jeziku C++. Pri razvoju aplikacij za platformo Symbian moramo biti na izjeme pozorni, sploh če dostopamo do same platforme Symbian. Ti mehanizmi ter njihova vpetost v ogrodje Qt je podrobneje opisana v [39].

4.10 Internacionalizacija

Razvijalci okolja Qt priporočajo, da za vse nize, ki bodo predmet prikaza na zaslonu, uporabimo *QString*. Slednji namreč uporablja kot znakovni nabor (angl. character set) standard Unicode, kar zagotavlja pravilno izrisovanje znakov na različnih platformah. *QString* podpira implicitno pretvarjanje iz *const char* *, kar omogoča spodaj (izpis 4.4) zapisano uporabo z metodami, ki sprejemajo kot argumente *const QString &*.

```
gumb->setText("Besedilo");
```

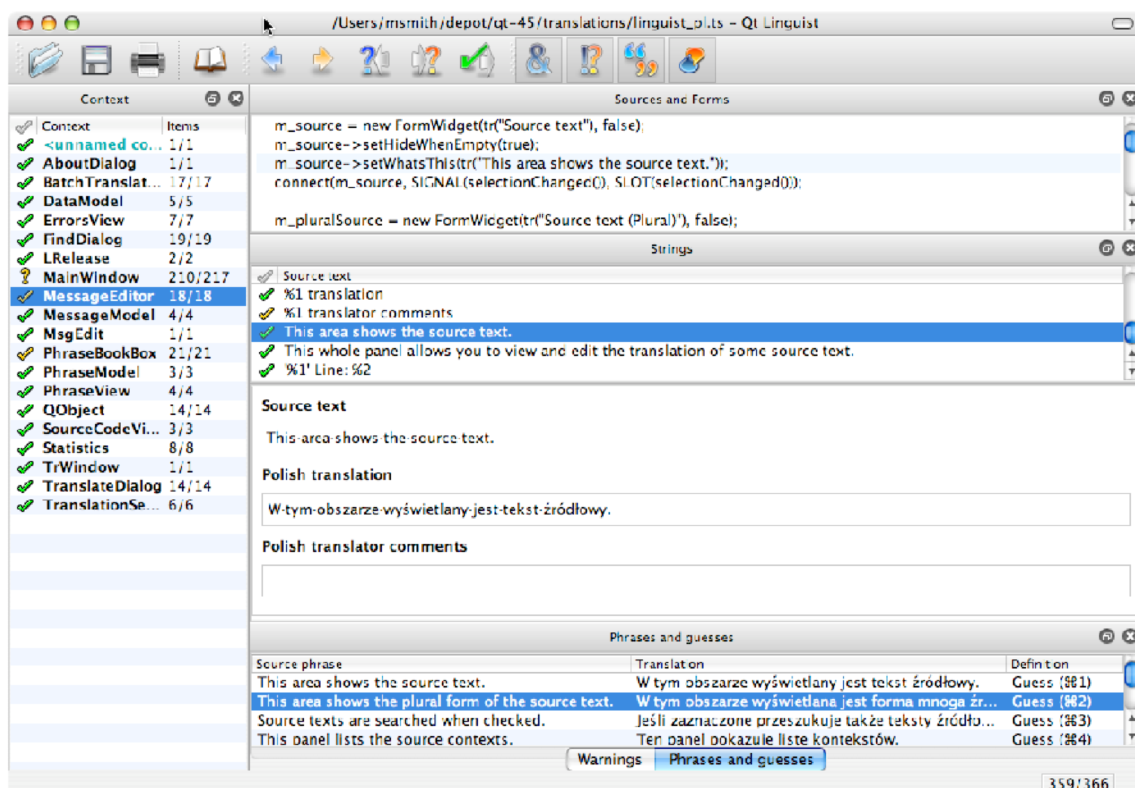
Izpis 4.4: Implicitno pretvarjanje v *QString*.

Da bi izkoristili podporo prevajanju, se spomnimo še ene prednosti objekta *QObject* – metode *tr()*. Ta metoda bo opravila mapiranje med *const char ** v niz *QString* po standardu Unicode preko objekta *QTranslator* [40].

```
gumb->setText(tr("Besedilo"));
```

Izpis 4.5: Zapis uporabniku vidnih nizov.

Zaradi te funkcionalnost poskrbimo, da nize vidne uporabniku vedno zapišemo preko klica metode *tr()*, saj s tem omogočimo podporo internacionalizaciji (izpis 4.5). Te nize lahko izluščimo z orodji, ki so priloženi okolju Qt ter jih prevedemo z aplikacijo Qt Linguist.



Slika 4.18: Qt Linguist na platformi Mac OS [40].

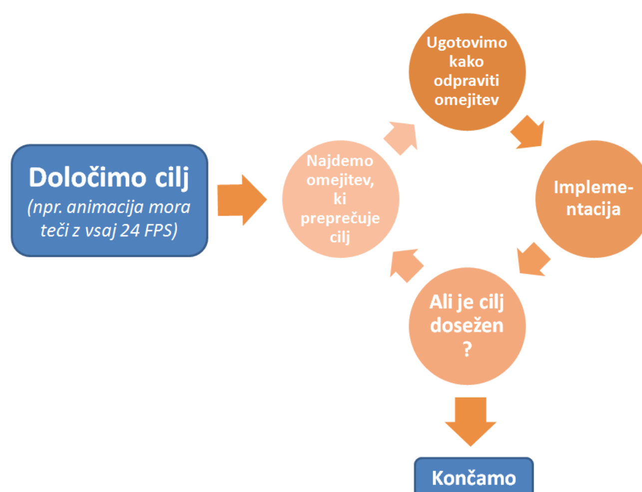
Qt Linguist je orodje za dodajanje prevodov aplikaciji, ki jo razvijamo (slika 4.18). Ker je namenjeno prevajalcem, ga na tem mestu le omenimo, naštejmo pa le nekaj razlogov, zakaj bi razvito aplikacijo ponudili v več jezikih:

- zaradi razširjenosti prenosnih telefonov, moramo v smislu distribucije misliti globalno.
    - Latinska Amerika ima skoraj več kot 520 milijonov uporabnikov prenosnih telefonov, 75% teh je predplačniških, preboj uporabe interneta pa je več kot 50%. Vendar pa, na primer, v Braziliji 95% vseh prebivalcev govori le portugalsko – in tega vabljivega števila možnih uporabnikov ne moremo doseči, če naše aplikacije ne bi prevedli v portugalščino.
- V nekaterih državah je uporaba domačega jezika, tudi na mobilnih telefonih, predpisana.
- Aplikacija v domačem jeziku izboljša uporabnikovo izkušnjo z aplikacijo.

4.11 Optimizacije

Optimizacija je zahtevna naloga, saj od razvijalca najprej zahteva, da poišče dele programja, ki se izvaja počasi, nato pa ga poskuša pohitriti.

Pomembno je, da optimizacije ne opravljamo na naključnih delih kode oziroma »po občutku«, saj se bo to v večini primerov rezultiralo v kompleksnejši kodi ter v pridobitvi na delih, kjer optimizacije niso potrebne. Okolje Qt ponuja makro *QBENCHMARK*, ki nam pomaga oceniti časovno trajanje delov kode in s tem določiti tiste dele, ki so optimizacije res potrebni. Format poročila, ki ga makro *QBENCHMARK* generira, je nastavljiv; omenimo pa, da sta med podprtimi formati orodji Valgrind ter Callgrind na voljo le na platformi Linux.



Slika 4.19: Proces izvajanja optimizacij.

Optimizirati (slika 4.19) začnimo z mislijo, da je naša aplikacija tako hitra, kot je hitro njeno najpočasnejše ozko grlo. Pomagamo si še s pripomočki kot so Nokia Energy Profiler [41] in Y-Tasks [42].

Splošno

V nadaljevanju je podanih nekaj napotkov, na katere moramo biti pozorni tako pri izvajanju optimizacij, kot tudi pri samem razvoju v okolju Qt [43].

- Razumevanje implicitne souporabe. Razrede, ki implicitne souporabe ne implementirajo, vedno prenašamo po konstantnih referencah.
- Uporaba pravih gradnikov (vsebnikov) za prave naloge.
- Uporaba zastavic, ki nam omogočajo, da prikrojimo delovanje gradnikov naši aplikaciji ter s tem odstranimo nepotrebno procesiranje za specifični primer.
- Ogrodje Graphics View je zaradi natančnih in hitrih transformacij ter učinkov nad elementi zgrajeno s predpostavko, da ciljna platforma podpira strojno procesiranje operacij s plavajočo vejico. V primeru ko operacije s plavajočo vejico na ciljni platformi niso strojno podprte, moramo biti pazljivi, da se takšnim operacijam izognemo, ali pa ogrodja sploh ne uporabimo.
- Razred *QImage* je bil načrtovan in optimiziran za V/I operacije, za neposreden dostop ter nadzor pikslov. Razred *QPixmap* pa je namenjen za risanje. Pretvorbe med *QImage* in *QPixmap* so časovno zahtevna operacija in se jim zato poskušamo izogniti, zato priporočamo uporabo razreda *QPixmap*.
- Glede na specifiko aplikacije, ki jo razvijamo, nam ogrodje Graphics View za optimiziranje ponuja kar nekaj možnosti. Izberemo lahko najbolj optimalen način posodabljanja predstavitvenega polja, vklopimo ali izklopimo indeksiranje glede na število elementov na prizorišču, se izogibamo signalu *changed()*, ki ga lahko oddaja prizorišče, če imamo možnost – nastavimo omejujoč pravokotnik prizorišča s *setSceneRect()* ter primerno nastavljamo načine predpomnjenja elementov.
- Pri razčlenjevanju dokumentov XML se izognemo tako grajenju kot tudi uporabi drevesa DOM, saj zavzame v pomnilniku veliko prostora. Za mobilne naprave je zaradi tega priporočena uporaba razreda *QXmlSimpleReader*.
- Uporabljamo tip *qreal*, ki ga na vgrajenih sistemih okolje Qt definira kot tip *float* in ne *double*.

V nadaljevanju podajamo nekaj optimizacij za platformo Symbian, ki so bile potrebne za korektno delovanje bralnika novic.

Nastavitve pogleda in prizorišča

Pri optimizaciji načina, kako *QGraphicsView* posodablja svoje predstavitveno polje ob spreminjanju delov prizorišča, imamo na razpolago več možnosti, od tistega, ki izračuna najmanjši možni pravokotnik, ki je potreben za posodobitve, pa vse do posodobitve celotnega predstavitvenega polja.

Za prizorišča z mnogo majhnimi elementi je lahko procesorsko bolj zahtevna možnost tista, ki računa najmanjše posodobitvene pravokotnike in obratno; najmanj zahtevna je možnost, ki vedno posodobi celo posodobitveno polje. Zgodi se namreč lahko, da računanje posodobitvenih pravokotnikov terja več procesorskega časa kot pa ponovni izris celotnega zaslona. Ugotovili smo, da v našem primeru različni načini posodabljanja predstavitvenega polja nimajo velikega vpliva na število izrisanih slik na sekundo, saj število elementov na prizorišču ni tako veliko in se po njem tudi ne premikajo.

V primeru, ko kakšen element preseže pravokotnik, ki omejuje pogled, raste prizorišče samodejno. Ker navedenega obnašanja prizorišča ne potrebujemo, se obdelavi podatkov pri takšnem širjenju izognemo tako, da velikost prizorišča nastavimo s klicem metode *QGraphicsScene::setSceneRect()*.

Uporabimo tudi zastavico, ki okolju Qt sporoči, da bralnik novic vedno zapolni vse piksle s klicem *QWidget::setAttribute(Qt::WA_OpaquePaintEvent)*.

Upravljanje s pomnilnikom

V pomnilniku zasedejo slike veliko prostora in imajo lahko pri nalaganju negativen vpliv na zmogljivosti. Pri mobilnih napravah se ta problem še posebej opazi, saj imajo te naprave v primerjavi z namiznimi računalniki občutno manjši pomnilnik in slabši procesor.

Slika v pomnilniku zasede približno *(širina * višina * globina) / 8* zlogov. Kaj hitro spoznamo, da v pomnilniku originalnih slik ni možno shranjevati, saj je v virih RSS večina slik prilagojena za ogledovanje na namiznih računalnikih in pri višjih resolucijah ter bi tako zasedle preveliko pomnilnika. Zato takoj, ko si pridobimo podatke slike, le–te

obdelamo za prikazovanje na mobilni napravi ter si takšne tudi hranimo. Algoritem za prilagajanje slik je predstavljen v nadaljevanju.

Vprašajmo se, kaj se bi zgodilo v primeru, če bi prikazovali ali predpomnili toliko slik, da nam bi zmanjkalo prostora v pomnilniku. Nerealno bi bilo pričakovati, da se pri dodeljeni uporabi do desetih mega zlogov pomnilnika za aplikacijo, ki prikazuje toliko fotografij, to ne bi tudi zgodilo. Zapišimo izsek kode upravljalca pomnilnika za pomnenje slik (izpis 4.6).

```
void OmaraSlik::slikaJePripravljena(const QString &ime, const
QPixmap& pix)
{
  // Omara je tipa QMap<QPixmap>
  // duplikati so izločeni že pri dodajanju
  omara[name] = pix;
  zasedeno += oceniVelikostSlike(pixmap);
  // Preveri če smo čez mejo, in če ja, odstrani predpomnjene slike
  while ((MAX_ZASEDENO < zasedeno) && !vrstni_red.isEmpty()) {
    // Odstrani najprej dodano sliko kot prvo, to si vodimo z vrsto
    // z imenom vrstni_red tipa QQueue
    QString prva = vrstni_red.dequeue();
    zasedeno -= odstrani_iz_pomnilnika(prva);
}
int OmaraSlik:: oceniVelikostSlike( const QPixmap &pix )
{
    return ((pix.width() * pix.height() * pix.depth()) >> 3);
}
```

Izpis 4.6: Izsek kode upravljalca pomnilnika za pomnenje slik.

Prikazano implementacijo pa moramo dopolniti tako, da omogoča ponovno pridobivanje slik, ki smo jih predhodno iz pomnilnika že odstranili, ter z ustvarjanjem zahtev, naj se slike pojavijo, ko bodo zares potrebne.

Ponazorimo primer delovanja celotnega sistema. Ko se novice pridobijo iz vira RSS, se podajo zahteve le za slike, ki so trenutno vidne na zaslonu. Zahteve po novo prikazanih slikah se ustvarijo, ko uporabnik prične pomikati seznam novic proti dnu. V primeru, da bo uporabnik seznam proti dnu pomikal dolgo, se bo pomnilnik za pomnjenje slik zapolnil, upravljalec pomnilnika pa bo začel odstranjevati slike, ki so bile prikazane najprej ter tako ustvaril prostor za tiste, ki jih trenutno prikazujemo.

Prilagajanje slik za prikaz na mobilnih napravah

Slike prilagodimo prikazovanju na mobilnih napravah s pomočjo razreda *QImage* (izpis 4.7).

```
QImage slika = nastavi_sliko();
// Pomanjševanje.
slika=slika.scaled(velikost,
                   Qt::KeepAspectRatioByExpanding,
                   Qt::FastTransformation);
// Centriranje.
slika=slika.copy(slika.rect().center().x()-(velikost.width()>>1),
                 slika.rect().center().y()-(velikost.height()>>1),
                 velikost.width(),
                 velikost.height() );
```

Izpis 4.7: Obdelava slik za prikaz na mobilnih napravah.

Klic *QImage::scaled()* s parametrom *Qt::FastTransformation* zagotavlja, da se slike pomanjšujejo zelo hitro, vendar pa s tem tvega kvaliteto pomanjšanih slik. Uporabimo lahko tudi drugi način pomanjševanja, in sicer preko parametra *Qt::SmoothTransformation*, ki pa je ob dobri kvaliteti pomanjšanih slik mnogo počasnejši. Proces pomanjševanja *Qt::SmoothTransformation* namreč opravi tako, da vzame 2x2 piksla, izračuna povprečno vrednost barve ter jo uporabi kot rezultat v novi sliki, medtem ko *Qt::FastTransformation* enostavno vzame enega od teh štirih pikslov.

Poslužimo se »trika« (izpis 4.8), ki združuje obe transformaciji, pri čemer hitrejša zmanjša števil0v pikslov, ki jih mora počasnejša obdelati.

```
slika = slika.scaled(velikost_večja_za_polovico_željene,
                     Qt::KeepAspectRatioByExpanding,
                     Qt::FastTransformation);
Slika = slika.scaled(velikost,
                     Qt::KeepAspectRatioByExpanding,
                     Qt::SmoothTransformation);
```

Izpis 4.8: Izboljšanje kvalitete pomanjševanja.

Takšno skaliranje nam v približno istem času opravi kvalitetnejše pomanjševanje slike [44].

Izrisovanje velikega števila slik

Element, ki predstavlja novico, lahko preprosto izrišemo kot kompozicijo objektov (izpis 4.9).

```
QGraphicsTextItem *naslov;
QGraphicsRectItem *ozadjeNaslova;
QGraphicsPixmapItem *slika;
```

Izpis 4.9: Primer kompozicije objektov, potrebnih za izris novice.

Namesto takšnega pristopa izberemo raje način, kjer element svojo celotno površje riše s pomočjo objekta *QPainter*. To nam omogoči nove načine za optimiziranje. Ker rišemo čez sliko polprozorno ozadje za naslov ter tudi naslov sam, bi bilo to, ob vsakem klicu elementa, potratno. Takšno delovanje je sicer že podprto s strani okolja Qt, vendar pa bi z njegovo uporabo izgubili možnost uporabe algoritma predpomnjenja slik, ki smo ga razvili zgoraj.

Tako opravimo izrisovanje le prvič – in sicer v razred *QPixmap*, ki ga shranimo v predpomnilnik. Ob naslednjem risanju naredimo vpogled v predpomnilnik ter v primeru, da je slika tam že izrisana, uporabimo slednjo. V nasprotnem primeru, torej če slika še ni izrisana, opravimo risanje in shranjevanje v predpomnilnik (izpis 4.10).

```
QPixmap slikaElementa( boundingRect().size().toSize() );
if ( Predpomnilnik::find( kljuc, &slikaElementa ) ) {
    // Element je v predpomnilniku!
    risar->drawPixmap( QPoint(0, 0), slikaElementa );
} else {
    // Prvič izrišimo element in ga shranimo v predpomnilnik.
    QPainter risarPoSliki;
    risarPoSliki.begin( & slikaElementa);
    ...
    risarPoSliki.end();

    risar->drawPixmap( QPoint(0, 0), slikaElementa);
    key = Predpomnilnik::insert(slikaElementa);
}
```

Izpis 4.10: Izrisovanje novice.

Razložimo še odločitev, zakaj implementirati predpomnilnik, namesto da bi uporabljali razred *QPixmapCache*, ki je del okolja Qt. Hrošč [45] v samem ogrodju onemogoča pravilno delovanje razreda *QPixmapCache*, ki predpomnjene slike predčasno odstrani iz pomnilnika, kljub temu da so še v uporabi.

Naslednjo optimizacijo pa lahko opravimo le, če imamo možnost spremeniti zahteve aplikacije - namesto da v en element rišemo eno novico, ta element razdelimo na dva stolpca ter vanj rišemo dve novici. Tako lahko z enako porabo pomnilnika, ki je sicer potrebna za pomnjenje slike ene novice, dosežemo pomnjenje dveh novic. Takšna razporeditev pa hkrati tudi zmanjša monotonost izgleda aplikacije (slika 4.20).

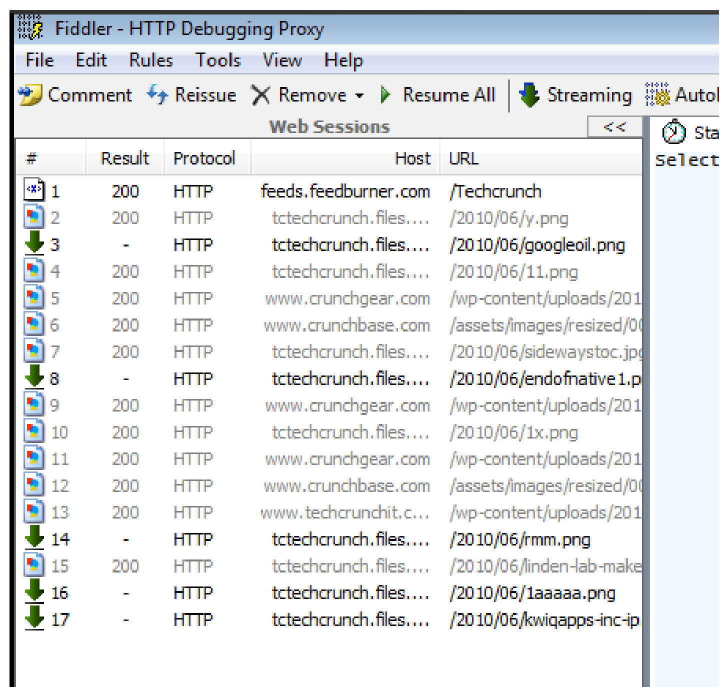
Slika 4.20: Skica risanja elementov, kjer v vsak drug element izrišemo dve novici namesto ene.

Snemanje slik iz spleta

Sekvenčno snemanje (angl. download) iz spleta, kjer naslednjo zahtevo izstavimo šele, ko se trenutna zaključi, je prepočasno. Paziti pa moramo tudi nasprotno: če hkrati podamo preveč zahtev, lahko to preveč obremeni delovanje sistema, saj ga lahko preobremeni z množico odgovorov v primeru, da se vsi vrnejo ob skoraj istem času.

Branje po dokumentaciji razreda *QNetworkAccessManager* razkrije, da zahteve uravnoveša že okolje Qt, zaradi česar nam ni potrebno implementirati vrste zahtev. Privzeto obnašanje smo preverili tudi z aplikacijo Fiddler. Da lahko naša aplikacija pošilja

zahteve preko aplikacije Fiddler, moramo nastaviti razredu *QNetworkAccessManager* posrednika ("proxy") s klicem metode *setProxy()*.



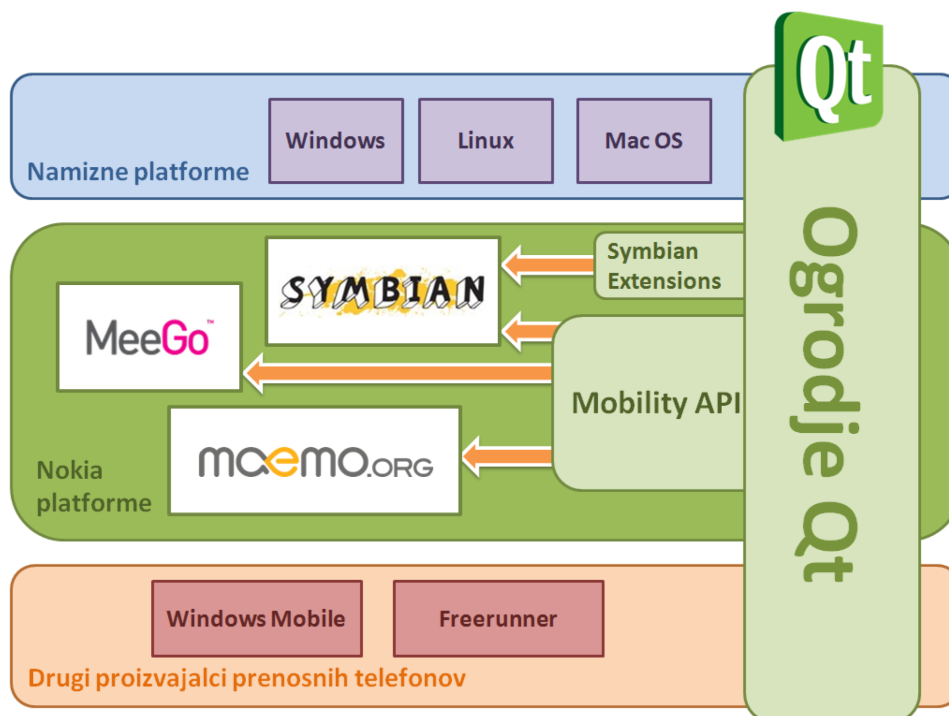
Slika 4.21: Izsek iz zaslonskega posnetka aplikacije Fiddler.

Slika 4.21 prikazuje, da je izmed vseh, hkrati aktivnih največ le določeno število zahtev, ostale pa so uvrščene v vrsto, kjer čakajo na izvajanje.

5 OKOLJE QT NA MOBILNIH NAPRAVAH

V samem začetku je okolje Qt podpiralo le platforme, kot so Windows, Mac OS, Linux, Windows Mobile in vgrajene sisteme. Po prevzemu s strani Nokie pa se je okolje Qt razširilo še na platforme Symbian, Maemo in Meego (sicer obe izpeljanki platforme Linux), kar je predstavljalo korak k cilju, da bi bilo okolje delujoče na vseh platformah (angl. »Qt Everywhere«).

Z navedeno širitvijo pa so razvijalci okolja Qt naleteli na težavo, ki bi lahko ogrozila prenosljivost programja, napisanega v okolju Qt. Težavo je predstavljal dostop do programskih vmesnikov, ki so značilni za prenosne telefone, ne pa tudi za namizne računalnike. Mednje med drugimi spadajo programski vmesniki za sporočanje (SMS, MMS), lokacijo, kamero ali pa pester nabor tipal (angl. sensors). Seveda je možno aplikacije razvijati tudi popolnoma brez omenjenih vmesnikov, vendar pa bi se s tem odrekli velikemu naboru zanimivih rešitev, ki jih le – ti ponujajo.



Slika 5.1: Dostop do programskih vmesnikov preko različnih platform za okolje Qt.

Iz slike 5.1 izhaja, da je aplikacija med platformami, ki jih Qt podpira, prenosljiva, vse dokler pri njenem razvoju uporabljamo samo okolje Qt [4]. Z uporabo Symbian Extensions programskega vmesnika, se omejimo na platformo Symbian; z uporabo programskih vmesnikov Mobility API pa na platforme, ki jih podpira Nokia: MeeGo, Maemo in Symbian [46].

5.1 Programski vmesnik Qt Mobility

Omenili smo že, da imajo mobilne naprave veliko specifičnih lastnosti, zaradi česar jih ne bi bilo smiselno vljučevati v samo okolje Qt. Razvijalci Qt so zato razvili programske vmesnike – imenovane Qt Mobility, ki ponujajo razvijalcem dostop do najbolj pogostih opravil, specifičnih za mobilne naprave. Qt Mobility je na voljo na vseh mobilnih napravah, ki jih izdeluje proizvajalec Nokia [47].

Omenimo nekaj najpomembnejših:

- **upravljanje nosilcev podatkov** (angl. bearer management), ki razvijalcem olajšuje delo pri izbiri najustreznejše povezave do različnih nosilcev IP (angl. Internet Protocol) ter povezav 3G. Omogoča:
  - samodejno izbiro povezave preko logike v programju, ali pa po želeni povezavi povpraša uporabnika;
  - požene ali ustavi različne komunikacijske vmesnike;
  - prenaša povezavo (angl. roaming) med različnimi komunikacijskimi vmesniki.

  Eden izmed primerov uporabe upravljanja nosilcev podatkov, je aplikacija za takojšnje sporočanje (angl. instant messaging), ki samodejno preide na povezavo WLAN, če je le ta na voljo.

- **Dostop do kontaktov** je zagotovo ena najpomembnejših storitev, ki jih nudi Qt Mobility, saj omogočajo mobilne naprave med ljudmi različne vrste komunikacij; navadno pa so ti shranjeni ravno med kontakti na mobilni napravi.
- **Lokacija** omogoča izdelavo kontekstno občutjivih (angl. context sensitive) aplikacij; lastnosti le – teh pa postajajo pri razvoju aplikacij za mobilne naprave že skoraj zahtevana komponenta.

- **Sporočanje** ponuja vmesnike za ustvarjanje in pošiljanje sporočil SMS, MMS ter elektronske pošte. Omogoča še brskanje, urejanje, iskanje ter obveščanje o spremembah v zbirkah sporočil.
- **Multimedija** je postala standardna komponenta mobilnih naprav. Preko programskih vmesnikov Qt Mobility lahko predvajamo in snemamo zvočne ali video posnetke, sprožimo slikovne predstavitve (angl. slide show) ter uporabljamo FM radio, če je na voljo v napravi.
- **Informacije o sistemu** omogočajo dostop do podatkov o napravi. Sem spadajo seznami senzorjev, možnih mrežnih povezav, verzije platforme, informacije o pomnilniku itd.

5.2 Ogrodje Smart Installer na platformi Symbian

Ogrodje Smart Installer olajša namestitev programja, ki temelji na okolju Qt za platformo Symbian [48]. Trenutno okolje Qt pri napravah, ki so dostopne na trgu, še ni del platforme Symbian . Zaradi navedenega mora programje, ki uporablja okolje Qt, le – to tudi namestiti. Ker pa je tipična namestitev okolja Qt za platformo Symbian velika kar dobrih deset megazlogov, medtem ko je velikost aplikacije za platformo Symbian manjša od 1M zloga, možnost, da bi okolje Qt vedno »pripeli« k namestitvi programja, seveda ne pride v upoštev.

Rešitev navedenega problema je Smart Installer, ki ga povežemo z namestitvijo naše aplikacije. Med samim procesom namestitve aplikacije Smart Installer:

- preveri, ali je okolje Qt že nameščeno na napravi,
- v primeru, da je okolje Qt že nameščeno na napravi ter da je ustrezne verzije, se bo namestitev nadaljevala z našo aplikacijo,
- v primeru, da okolje Qt na napravi še ni nameščeno, ali pa je neustrezne verzije, bo Smart Installer zadnjo verzijo okolja Qt prenesel iz spleta ter ga namestil, zatem pa se bo namestitev nadaljevala z našo aplikacijo.

Symbian je edina platforma, ki jo ogrodje Smart Installer trenutno podpira. S tem ogrodjem postane nameščanje programja preprosto tako za razvijalce kot tudi za končne uporabnike.

6 PRIHODNOST

Med razvojem bralnika novic smo naleteli na nekaj pomanjklivosti okolja Qt. Želeli bi, da bi bile določene lastnosti na voljo že v samem okolju Qt, kot npr. gradniki za ogrodje Graphics View, razpoznava kretenj ali pa kinetični pomik. Na vprašanje, ali se bo okolje Qt razvilo v tej smeri ter zapolnilo te pomanjklivosti, poskusimo odgovoriti v tem poglavju.

6.1 Nokia Qt SDK

Nokia Qt SDK je zbirka orodij ter ogrodij za razvoj programja v okolju Qt, in sicer za platforme, ki jih uradno podpira Nokia [49]. Glavne prednosti Nokia Qt SDK so:

- preprosta namestitev, ki nam brez dodatne konfiguracije omogoča razvoj aplikacij za mobilne naprave proizvajalca Nokia;
- Qt Simulator, ki omogoča hitro in preprosto testiranje in razhroščevanje aplikacij za mobilne naprave kar na namiznem računalniku. Vedeti moramo, da Qt Simulator mobilno napravo simulira in ne emulira, kar je dejstvo, zaradi katerega moramo v nekaterih primerih razhroščevanje ponoviti tudi na sami napravi – enako velja tudi za testiranje.
- Na razpolago so vsi prevajalniki, povezovalniki, razhroščevalniki ter druga orodja, ki omogočajo razvoj in grajenje aplikacij za platformi Symbian in Maemo.

Nokia Qt SDK za razvijalce na platformi Nokia zagotovo predstavlja dobro izbiro, saj vsebuje vsa potrebna orodja in ogrodja, v eni sami namestitvi. Edina slabost je, da ne vsebuje ogrodja Qt za druge platforme ter tako posledično omeji razvoj samo na mobilne platforme proizvajalca Nokia.

6.2 Qt Quick

Qt Quick (kratko za »Qt User Interface Kit«) bo[1] ogrodje za razvoj uporabniških vmesnikov na visokem nivoju in bo eden izmed najpomembnejših dodatkov k okolju Qt [50]. Qt Quick predstavlja deklarativno ogrodje za grajenje dinamičnih in popolnoma prilagojenih (angl. custom) uporabniških vmesnikov, ki so zgrajeni na osnovi množice elementov QML. Qt Quick je zgrajen v okolju Qt ter obsega spodaj opisane tehnologije:

- QML (Qt Meta-Object Language) je deklarativen jezik izpeljan iz jezika JavaScript in je zato preprost tako za učenje kot tudi za uporabo. QML temelji na lastnostih metaobjektnega sistema okolja Qt.
- Nova orodja v razvojnem okolju Qt Creator, ki vsebuje okolje za urejanje ter predogled izdelanih uporabniških vmesnikov v Qt Quick;
- QtDeclarative je nova komponenta okolja Qt z nalogo, da prevede deklarativen opis uporabniškega vmesnika iz jezika QML v elemente na prizorišču (`QGraphicsScene`) ogrodja GraphicsView. Skrbi pa tudi za povezljivost med jezikom QML in C++ ter tako povezuje uporabniške vmesnike s preostalo implementacijo.

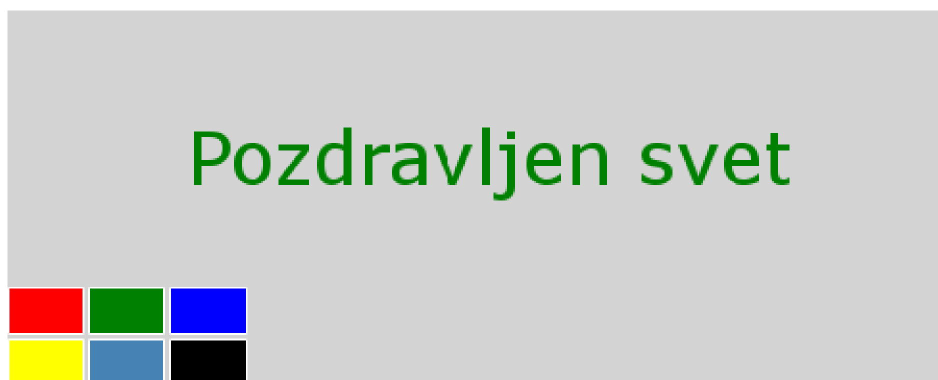
Uporabniške vmesnike v Qt Quick gradimo iz gradnikov, ki jim deklarativno definiramo tako izgled kot tudi obnašanje, opisujemo pa jih v dokumentih QML (angl. QML documents). Dokumenti QML [51] so dejansko izvorna koda uporabniškega vmesnika v jeziku QML. Poglejmo si preprost primer dokumenta QML ter uporabniški vmesnik, ki ga ustvari.

[1] V času pisanja je ogrodje Qt Quick razvijalcem na voljo kot predogled. Prva uradna izdaja je pričakovana v drugi polovici 2010.

```
import Qt 4.7

Rectangle {
  id: stran
  width: 500; height: 200
  color: "lightgray"

  Text {
    id: besedilo
    text: "Pozdravljen svet"
    y: 30
    anchors.horizontalCenter: page.horizontalCenter
    font.pointSize: 24; font.bold: true
  }

  Grid {
    id: barvnaMreza
    x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4
    rows: 2; columns: 3; spacing: 3

    Rectangle {
      color: "red";
      MouseArea {
        Anchors.fill: parent
        onClicked: besedilo.color = color }
      }
    }
    ...
    Rectangle {
      color: " black";
      MouseArea {
        Anchors.fill: parent
        onClicked: besedilo.color = color }
      }
    }
}
```

Izpis 6.1: Izpis dokumenta QML.

Izvorna koda v izpisu 6.1 temelji na dokumentaciji okolja Qt, ki je še v razvoju in se lahko spremeni.



Slika 6.1: Uporabniški vmesnik ustvarjen iz preprostega dokumenta QML.

Iz kode je razvidno, da ob klikih na barvno mrežo v levem spodnjem delu zaslona, spreminjamo barvo besedila »Pozdravljen svet« (slika 6.1).

Za razumevanje kode ni potrebno razvijalsko znanje in je berljiva tudi ostalim. Ravno zaradi navedenega in zaradi jasno začrtane meje med uporabniškim vmesnikom in implementacijo procesov, Qt Quick med razvijalci in oblikovalci omogoča tako sodelovanje, kot tudi vzporedno istočasno delo na isti aplikaciji. Oblikovalci lahko tako prevzamejo večjo vlogo v samem projektu, saj za izdelavo uporabniškega vmesnika ne potrebujejo znanja jezika C++. Enako velja za ustvarjanje prototipov, ki jih s Qt Quick lahko ustvarimo hitro, elegantno ter že zelo zgodaj v procesu razvoja aplikacije.

Integracija QML v obstoječe uporabniške vmesnike, ki temeljijo na okolju Qt

Obstajata dva načina integracije uporabniških vmesnikov QML v aplikacijo, zgrajeno v okolju Qt [52]:

- integracijo z uporabniškim vmesnikom, ki temelji na razredu *QWidget*, lahko opravimo prek razreda *QDeclarativeView*, ki je del komponente QtDeclarative.

```
QDeclarativeView *pogledQml = new QDeclarativeView;
pogledQml ->setSource(QUrl::fromLocalFile("mojqml.qml"));
QWidget *gradnik = mojObstoječGradnik();
QVBoxLayout *razporeditev = new QVBoxLayout(gradnik);
gradnik->addWidget(pogledQml);
```

Izpis 6.2: Integracija uporabniškega vmesnika QML, ki temelji na razredu *QWidget*.

Vedeti moramo, da se *QDeclarativeView* inicializira počasneje ter porabi več pomnilnika kot *QWidget*, zaradi česa lahko vidimo upad zmogljivosti.

- V primeru, da imamo uporabniški vmesnik zgrajen z ogrodjem Graphics View, imamo možnost, da gradnike QML dodamo na že obstoječe prizorišče (*QGraphicsScene*).

```
QGraphicsScene* prizorišče = mojePrizorišče();
QDeclarativeEngine *motor = new QDeclarativeEngine;
QDeclarativeComponent komponenta(motor,
  QUrl::fromLocalFile("mojqml.qml"));
QGraphicsObject *element =
  qobject_cast<QGraphicsObject *>(komponenta.create());
prizorišče->addItem(element);
```

Izpis 6.3: Integracija uporabniškega vmesnika QML, ki temelji na ogrodju Graphics View.

Kateri način integracije je pravi, pa se mora seveda odločiti razvijalec sam. Pri odločitvi bo upošteval različne karakteristike obstoječe izvorne kode uporabniškega vmesnika ter težnjo po obsežnosti zamenjave uporabniškega vmesnika, napisanega v ogrodju Qt, z uporabniškim vmesnikom napisanim v Qt Quick.

7 SKLEP

Okolje Qt razvijalcem mobilnih aplikacij omogoča, da ti programje v jeziku C++ razvijejo le enkrat, nato pa ga namestijo na različne platforme, od namiznih, vgrajenih do mobilnih, brez da bi jim bilo pri tem potrebno spremeniti izvorno kodo.

Podobno velja tudi za razvojno okolje in vsa spremljajoča orodja okolja Qt. Okolje Qt se prevede na skoraj vseh prevajalnikih C++ in je med platformami prenosljiv, enako pa velja tudi za večino priloženih orodij, saj so le - ta razvita v okolju Qt. Poleg izbire ciljne platforme za programje ima razvijalec tako na voljo še podobno izbiro razvojne platforme. Elegantno je rešena tudi problematika zmogljivosti, ki so na voljo le na mobilnih platformah. Le – te niso vključene v razredno knjižnico okolja Qt, ampak so razvijalcem ponujene preko ločenega programskega vmesnika Qt Mobility.

V diplomski nalogi smo preverili, ali okolje Qt omogoča hiter razvoj intuitivnih uporabniških vmesnikov. Izbrali smo eno izmed obstoječih aplikacij platforme Symbian^1 in jo predelali na način, da temelji na okolju Qt, pri čemer smo sledili procesu razvoja grafičnih vmesnikov, kot je predlagan s strani proizvajalca Nokia. Na podlagi navedene izvedbe lahko sklenemo, da je odgovor na zgoraj postavljeno vprašanje pritrdilen.

Spoznali smo, da okolje Qt res omogoča preprost razvoj prenosljivih aplikacij za mobilne naprave, ki pa lahko zaradi prenosljivosti brez večjih sprememb postanejo tudi aplikacije za namizne platforme. Enostavnost razvoja velikokrat presega celo privzeto razvojno okolje določene platforme. Velja pa omeniti spoznanje, da moramo v primeru, ko želimo programje učinkovito uporabljati na mobilnih napravah, kljub prenosljivosti okolja Qt, opraviti še številne spremembe. Težava pravzaprav ne leži v okolju Qt, ampak v razliki strojne opreme, ki je, za razliko od namiznih naprav, pri mobilnih napravah precej manj zmogljiva. Ugotovili smo tudi, da moramo pri razvoju modernih uporabniških vmesnikov veliko funkcionalnosti, ki se od ogrodja sicer pričakujejo, razviti sami, saj nekaterih še ni na voljo, druge pa na določenih platformah niso podprte.

V predzadnjem poglavju smo zapisali nekaj besed o prihajajočih spremembah okolja Qt, še posebej pa smo izpostavili Qt Quick. Pričakujemo, da bo ta zapolnil vrzel, ki smo jo v okolju Qt opazili z manjkajočimi gradniki uporabniškega vmesnika ter v podpori kretenj, seveda ob predpostavki, da se bodo uporabniški vmesniki, napisani v jeziku QML, z zadovoljivo hitrostjo izvajali tudi na mobilnih napravah.

Z okoljem Qt smo razvijalci pridobili lahko dostopno, fleksibilno ter prenosljivo okolje, ki omogoča učinkovit razvoj tako mobilnih kot namiznih aplikacij. Omenimo še, da so se razvijalci mnogo let trudili z razvojem programja na platformi Symbian, kar pa je nenadoma postalo enostavno in preprosto – in sicer s pomočjo okolja Qt. Na podlagi slednjega verjamemo, da lahko Qt, kot dobro razvijalsko orodje, enako prispeva tudi k uporabi ostalih platform.

Na podlagi ugotovitev celotne diplomske naloge lahko zaključimo, da trditev[1] »Čudovito oblikovanje ustvarja strastne uporabnike« za razvijanje aplikacij s pomočjo okolja Qt nedvomno drži.

[1] Zapisana na spletni strani Forum Nokia: »Delightful design creates passionate users« [53].

8 LITERATURA

[1] http://qt.nokia.com/about/, zadnji obisk v maju 2010

[2] http://qt.nokia.com/qt-in-use, zadnji obisk v maju 2010

[3] http://qt.nokia.com/qt-in-use/story/device, zadnji obisk v maju 2010

[4] F. H. P. Fitzek, T. Mikkonen, T. Torp, *Qt for Symbian*, John Wiley & Sons, 2010

[5] J. Blanchette, The QStyle API in Qt 4, *Qt Quarterly*, 13, (2005), http://doc.qt.nokia.com/qq/qq13-styles.html

[6] http://qt.nokia.com/products, zadnji obisk v maju 2010

[7] http://qt.nokia.com/products/appdev/developer-tools/developer-tools, zadnji obisk v juniju 2010

[8] http://qt.nokia.com/products/library/modular-class-library, zadnji obisk v juniju 2010

[9] http://www.opengl.org/about/overview/, zadnji obisk v juniju 2010

[10] http://qt.nokia.com/products/programming-language-support, zadnji obisk v maju 2010

[11] http://qt.nokia.com/products/licensing, zadnji obisk v maju 2010

[12] http://doc.qt.nokia.com/4.6/object.html, zadnji obisk v juniju 2010

[13] http://www.islovar.org/, zadnji obisk v juniju 2010

[14] http://doc.trolltech.com/4.6/qmake-manual.html, zadnji obisk v maju 2010

[15] http://doc.trolltech.com/4.6/objecttrees.html

[16] http://doc.trolltech.com/4.6/signalsandslots.html, zadnji obisk v juniju 2010

[17] http://doc.qt.nokia.com/4.6/metaobjects.html, zadnji obisk v juniju 2010

[18] http://doc.trolltech.com/4.6/moc.html, zadnji obisk v maju 2010

[19] http://doc.trolltech.com/4.6/properties.html, zadnji obisk v maju 2010

[20] http://doc.qt.nokia.com/4.6/exceptionsafety.html, zadnji obisk v juniju 2010

[21] J. Blanchette, Another Look at Events, *Qt Quarterly*, 11, (2004), http://doc.qt.nokia.com/qq/qq11-events.html

[22] http://www.forum.nokia.com/Design/Design_process/, zadnji obisk v maju 2010

[23] http://www.forum.nokia.com/Design/Design_process/Getting_started/Design_research.xhtml, zadnji obisk v maju 2010

[24] http://www.forum.nokia.com/info/sw.nokia.com/id/0ed3c247-a0f6-4f66-bc03-18a1f7825fca/Design_and_Paper_Prototyping_Templates.html, zadnji obisk v maju 2010

[25] http://www.forum.nokia.com/Library/Tools_and_downloads/Other/Flowella/, zadnji obisk v maju 2010

[26] http://www.forum.nokia.com/Design/Design_process/Getting_started/Visual_and_information_design.xhtml, zadnji obisk v maju 2010

[27] J. Thelin, Building Next Generation Uis Across Platforms with Qt, *Qt Quarterly*, 32, (2009), http://doc.qt.nokia.com/qq/32/qq32-next-gen-uis.html

[28] http://doc.qt.nokia.com/4.6/graphicsview.html, zadnji obisk v maju 2010

[29] http://www.apple.com/magicmouse/, zadnji obisk v juniju 2010

[30] http://www.apple.com/iphone/, zadnji obisk v juniju 2010

[31] http://www.apple.com/ipad/, zadnji obisk v juniju 2010

[32] http://www.apple.com/ipod/, zadnji obisk v juniju 2010

[33] *UI Design and Interaction Guide for Windows Phone 7 Series*, predizdaja, Microsoft Corporation, Seattle, 2010

[34] http://doc.qt.nokia.com/4.6/gestures-overview.html, zadnji obisk v juniju 2010

[35] http://developer.symbian.org/forum/showthread.php?p=17340, zadnji obisk v juniju 2010

[36] J. Thelin, Recognizing Mouse Gestures, *Qt Quarterly*, 18, (2006), http://doc.qt.nokia.com/qq/qq18-mousegestures.html

[37] http://doc.qt.nokia.com/4.6/animation-overview.html, zadnji obisk v maju 2010

[38] http://doc.trolltech.com/4.6/qnetworkaccessmanager.html, zadnji obisk v maju 2010

[39] http://doc.qt.nokia.com/4.6/symbianexceptionsafety.html, zadnji obisk v maju 2010

[40] http://doc.qt.nokia.com/4.6/linguist-programmers.html, zadnji obisk v maju 2010

[41] http://www.forum.nokia.com/info/sw.nokia.com/id/324866e9-0460-4fa4-ac53-01f0c392d40f/Nokia_Energy_Profiler.html, zadnji obisk v maju 2010

[42] http://www.drjukka.com/YTasks.html, zadnji obisk v maju 2010

[43] http://qt.nokia.com/developer/learning/archive/online/talks/developerdays2009/tech-talks/optimizing-performance-in-qt-based-applications, zadnji obisk v maju 2010

[44] http://labs.trolltech.com/blogs/2009/01/26/creating-thumbnail-preview/, zadnji obisk v maju 2010

[45] http://bugreports.qt.nokia.com/browse/QTBUG-10576, zadnji obisk v maju 2010

[46] http://qt.nokia.com/products/qt-for-mobile-platforms, zadnji obisk v maju 2010

[47] http://qt.nokia.com/products/appdev/add-on-products/catalog/4/new-qt-apis/mobility, zadnji obisk v maju 2010

[48] http://qt.nokia.com/developer/nokia-smart-installer-for-symbian, zadnji obisk v maju 2010

[49] http://www.forum.nokia.com/Develop/Qt/Tools/, zadnji obisk v maju 2010

[50] http://doc.qt.nokia.com/4.7-snapshot/declarativeui.html, zadnji obisk v juniju 2010

[51] http://doc.qt.nokia.com/4.7-snapshot/qdeclarativedocuments.html, zadnji obisk v maju 2010

[52] http://doc.qt.nokia.com/4.7-snapshot/qml-integration.html, zadnji obisk v maju 2010

[53] http://www.forum.nokia.com/Design/, zadnji obisk v maju 2010

Fakulteta za elektrotehniko, računalništvo in informatiko

Smetanova ulica 17
2000 Maribor

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor red. prof. dr. Borut Žalik izjavljam, da je študent Miha Lesjak izdelal diplomsko delo z naslovom:

RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE V OKOLJU Qt

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj: 8.7.2010

Podpis mentorja: B. Žalik

Fakulteta za elektrotehniko, računalništvo in informatiko

Smetanova ulica 17
2000 Maribor

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE DIPLOMSKEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek diplomanta:	Miha Lesjak
Vpisna številka:	93843974
Študijski program:	UN ŠP – Računalništvo in informatika
Naslov diplomskega dela:	RAZVOJ PRENOSLJIVIH APLIKACIJ ZA MOBILNE NAPRAVE V OKOLJU Qt
Mentor:	red. prof. dr. Borut Žalik
Somentor:	doc. dr. Gregor Klajnšek

Podpisani Miha Lesjak izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru.
Diplomsko delo sem izdelal sam ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah (Ur. l. RS, št. 16/2007) dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum diplomiranja, naslov diplomskega dela) na spletnih straneh in v publikacijah UM.

Datum in kraj: 7.7.2010, MARIBOR

Podpis diplomanta: