Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

Visualisering av datastrukturer
Utveckling av ett tolkningsverktyg

Mats Adborn

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MITTUNIVERSITETET
Institutionen för informationsteknologi och medier (ITM)

Examinator: Ulf Jennehag, ulf.jennehag@miun.se
Handledare: Örjan Sterner, orjan.sterner@miun.se
Författare: Mats Adborn, malu0900@student.miun.se
Utbildningsprogram: Programvaruteknik, 180 hp
Huvudområde: Datateknik
Termin, år: 6, 2013

Sammanfattning

Tolking och tillgodogörande av datastrukturer, organiserad information och programkodsfiler förekommer frekvent i arbete med mjukvaruutveckling. Denna information är lagrad i textbaserad form och dess förståelse kräver stor noggrannhet och tidsinvestering från utvecklarens sida. I syfte att försöka förenkla processen beskriver detta examensarbete utvecklingen av en prototyp till ett verktygsprogram, vilket automatiserar tolkning av XML-data och källkodsfiler för programmeringsspråken C och C++. Programmet skapar och presenterar sedan en visuell graf av den undersökta strukturen. Algoritmen klarar av att presentera godtyckligt stora XML-filer samt ett begränsat antal samtidigt inlästa källkodsfiler. Effekterna på tolkningens tidsåtgång och dess tillförlitlighet har utvärderats i en undersökning bland studenter inom mjukvaruutveckling. Resultatet visade på en viss mätbar ökning i antalet korrekta slutsatser som användaren drog efter att ha studerat datasammanhanget grafiskt jämfört med dess ursprungliga textform. Tidsåtgången mättes inte mer noggrant än subjektivt hos användarna, av vilka en övervägande andel ansåg att tiden förkortades med den grafiska representationen till deras hjälp. Examensarbetet visar att användandet av detta eller motsvarande verktyg kan öka tillgodogörandet av datastrukturer genom att både höja graden av tillförlitligheten hos denna information och samtidigt minska tidsåtgången. Däremot är den kvantifierbara vinsten av dessa resultat inte statistiskt säkerställd till en högre grad.

Nyckelord: Grafisk programmering, datastrukturer, XML, C/C++, Qt

Abstract

Interpretation and assimilation of data structures, organized information and source code files are frequently occuring during software development. This kind of information is stored in text-based form and its understanding requires great thoroughness and investment in time from the developer's part. This thesis describes the development of a utility program prototype, which automates the parsing of XML data and source code files from the programming languages C and C++, in purpose of trying to simplify the interpretation process. The program creates and presents a visual graph of the structure found, using an algorithm which can present arbitrary large XML files as well as a limited number of concurrent source code files. The effects on the interpretation time and its reliablity has been evaluated in a survey among software development students. The result showed a certain increase in the number of correct conclusions from the participants' side after studying the visual representation compared to its orignial text-based form. The amount of time used was not measured other than subjectively by the users themselves, of which a predominant proportion considered a reduction in needed time when using the graphical representation. The thesis shows that the use of this or an equivalent utility can enhance the assimilation of data structures by increasing the rate of reliabilty while simultaneously decreasing the needed amount of time. Still, the quantifyable gains of these results remains statistically largely uncertain.

Keywords: Graphical programming, data structures, XML, C/C++, Qt

Innehållsförteckning

Terminologi

CVA	Commonality and Variability Analysis. Metod för att skapa effektivare objektorienterad kod.
DOT	Ett grafbeskrivningsspråk.
ECMAScript	En scriptspråksstandard vilken inkluderar dialekter så som JavaScript, JScript och ActionScript.
FOSS	Free and Open-Source Software. Beskriver att programvaran och dess källkod är fri, tillgänglig och gratis att använda, modifiera och sprida.
moc	Meta Object Compiler. En preprocessor för kompilering av Qt-kod.
MVC	Model-View-Controller. Ett mjukvaruarkitektoniskt mönster för strukturerande av grafik och tillhörande data.
NMI	Nyans-Mättnad-Intensitet. Metod för att beskriva färg i datormiljö. På engelska *HSV*, *Hue-Saturation-Value*.
OpenGL	Open Graphics Library. Ett gränssnitt för 2D- och 3D-grafik.
QML	Qt Modelling Language. Ett deklarativt språk för konstruktion av grafiska gränssnitt i Qt.
Qt	Ett applikationsramverk för plattformsoberoende (grafisk) mjukvara. Qt är inte en förkortning.
STL	Standard Template Library. Ett kodbibliotek till C++.
UML	Unified Modelling Language. En visuell beskrivningsmetod för förklaring av strukturer i diagramform.
VSTM	Visual Short-Term Memory. En typ av korttidsminne för visuella intryck med koppling till färg, form och formation.
WebKit	En webbläsarmotor använd av bland annat Chrome och Safari.
wxWidgets	Ett programbibliotek för plattformsoberoende grafisk mjukvara.
XML	Extensible Markup Language. Ett märkspråk för data, läsbart av både människa och maskin.

1 Inledning

Den här rapporten beskriver arbetet med att utveckla ett verktygsprogram som visualiserar datastrukturer i syfte att förenkla förståelsen, förkorta tolkningstiden och samtidigt bibehålla eller höja tillförlitligheten hos de uppgifter som fås ur den grafiska representationen. Om sådana vinster existerar utvärderas slutligen i en användarundersökning.

Detta arbete avslutar det treåriga utbildningsprogrammet Programvaruteknik 180 hp vid Mittuniversitetet.

1.1 Bakgrund och problemmotivering

Vi människor baserar sitt huvudsakliga intag av information på synintryck. Vår förmåga att – snabbt *och* korrekt – kunna ta till oss egenskaper och förändringar i vår omgivning är fundamental för vår överlevnad, på samma sätt som för många andra organismer. I vårt nutida samhälle är dock kraven på denna förmåga högre än någon gång tidigare i historien, och inte minst inom tekniska yrken. På samma gång som behoven av att kunna ta in informationen har ökat, ligger vår biologiska förmåga till detta kvar på samma nivå som när vi levde betydligt mer primitivt.

Vi lever dessutom i en tid då vi producerar och lagrar massiva mängder information; information som i många fall måste tolkas av andra än upphovsmannen. För att underlätta denna tolkningsprocess har vi insett att gemensamma referensramar och formalia förenklar förståelsen. Samtidigt är detta inte tillräckligt, då basen för informationen fortfarande är det skrivna ordet. Med hjälp av färg, form och symbolik kan vi ta den perceptiva processen för förståelse av ett komplext sammanhang ett steg längre.

På samma sätt som vi ofta tar pennan till hjälp för att bit för bit rita upp vår tolkning av en situation eller beskrivning, med eller utan formella modellsystem, kan informationen (om den är lagrad i strukturerad form) bearbetas av ett datorprogram och presenteras som en grafisk modell. Med denna angreppsvinkel vinner vi precision och sparar tid: automatiserad strukturtolkning av data är någonting som maskiner definitivt gör snabbare och förhoppningsvis också mer korrekt än oss människor.

1.2 Övergripande syfte

Det övergripande syftet med detta projekt är att utveckla ett program som kan läsa av en datastruktur och sedan presentera den för användaren med hjälp av sammanhängande figurer. På så sätt kan förhoppningsvis både bearbetningstiden för användaren förkortas och korrektheten hos tolkningen förbättras. Med en generell intern struktur av programmet, torde man kunna skapa tolkningsalgoritmer för många olika typer av datastrukturer, vare sig det gäller märkspråk, databaser eller interna referenser filer emellan. I vissa fall kan en viss

grafisk representationsform vara mer fördelaktig än en annan, vilket programmet också bör ha stöd för.

I det långa loppet är förhoppningen att programmet ska kunna utgöra ett användbart verktyg för personer (vid mjukvaruutveckling) som ofta stöter på och snabbt behöver skaffa sig en överblick av data- och filstrukturer som skapats av andra, exempelvis programvarugränssnitt eller data lagrat i märkspråksform.

Användningsområdet för verktyget kan visa sig vara snävt och kanske till och med inte tillföra så mycket vid tolkningsprocessen av informationen. Efter att programmets funktion är skapad, kommer därmed vikten av användarvänlighet in i bilden. Inom projektets ramar kan det dock visa sig att denna blir bristande och får lämnas åt framtida vidareutveckling. Men det är givetvis viktigt att i projektets slutskede utvärdera om verktyget bidrar till (eller med vidareutveckling skulle bidra) i en betydande grad att förenkla tillgodogörandet av informationen för användaren.

1.3 Avgränsningar

Projektet kommer att fokusera på att utveckla en prototyp för programmet, där aspekter som stöd för olika datalagringsformer och användargränssnitt inte kan utvecklas till fullo. Även om den grundläggande strukturen för programmet är viktig för dess fortsatta utvecklingspotential, kan vissa programspråksmässiga genvägar tas för att nå en användbar version av programmet, i strävan mot att kunna bedöma dess nyttoaspekt i verkliga situationer.

Gällande valet av de datastrukturer som verktyget skall klara av har det begränsats till XML- samt källkodsfiler för C/C++.

1.4 Konkreta och verifierbara mål

Detta arbete ska konstruera en prototyp till ett grafiskt verktyg med vilket datastrukturer ska kunna läsas in och presenteras visuellt. I konkret form betyder det att projektet ska:

- utveckla en tolk som läser XML- och källkodsfiler
- ta fram en modell för tillfällig lagring av det inlästa resultatet
- skapa ett grafiskt gränssnitt för att presentera modellen
- utvärdera det utvecklade verktyget via en användarundersökning

De mål som rör själva programmet specificeras ytterligare i dess kravspecifikation (se Avsnitt 4.1).

Frågeställningen efter färdigställd prototyp blir således: kan detta verktyg, i nuvarande eller vidareutvecklad form, bidra till att i en betydande grad förenkla tillgodogörandet av lagrad information utifrån att presentera dess struktur?

1.5 Etiska aspekter

Detta arbete handlar om utvecklingen av ett verktygsprogram, som körs lokalt på användarens dator utan att lagra eller skicka vidare information, varken under eller mellan programkörningar. Det är således inte relevant att utvärdera några etiska aspekter som själva verktygsprogrammet berör.

Gällande undersökningen har Vetenskapsrådets fyra forskningsetiska principer – informerande, samtycke, konfidentialitet och nyttjande – följts. [1]

1.6 Översikt

Detta kapitel innehåller en introduktion till arbetet, dess frågeställning och mål. Kapitel 2 beskriver kortfattat delar av det teoretiska bakgrundsmaterialet som detta arbete vilar på. Kapitel 3 presenterar och motiverar den metod som valts för att ta fram verktyget och utvärdera dess effekter på tolkningen. Sedan följer det fjärde kapitlet som tar upp konstruktion och programmeringsmässiga val som gjorts under utvecklingsmomentet. Kapitel 5 tar upp de resultat som nåtts under arbetets gång och den efterföljande utvärderingen, medan kapitel 6 analyserar och diskuterar kring dessa resultat. Avslutningsvis hittas en referensförteckning och rapportens bilagor.

2 Teori

Detta kapitel beskriver den teoretiska grund som arbetet bygger på för att ge läsaren en bättre förståelse för de utförda momenten som beskrivs i de kommande kapitlen. Här introduceras tekniska aspekter så som det ramverk programmet är byggt på, tillsammans med övergripande teorier kring god programkonstruktion. Kortfattat beskrivs även några liknande verktyg och det matematiska område som motsvarar projektets praktiska inriktning.

2.1 Applikationsramverket Qt

Qt är ett applikationsramverk, alltså en samling förenklande mjukvara (klasser, datatyper, funktioner och byggverktyg) som erbjuder utvecklare ett standardiserad sätt att konstruera applikationer. Dessutom är det i de flesta avseenden plattformsoberoende: Qt-kod skriven för ett visst operativsystem eller processorarkitektur kan mer eller mindre alltid kompileras och köras på en annan plattform framgångsrikt. I detta ligger en av de stora anledningarna till att använda Qt, då det har en komplett uppsättning av klasser för att bygga grafiska applikationer, samt att abstrahera systemberoende funktionalitet så som trådar i generella Qt-klasser. Det slutgiltiga utseendet för en sådan applikation baseras på den plattform det är kompilerat för varvid man får det förväntade systemutseendet. I dagsläget finns Qt för grafiska plattformar så som Windows, Mac OS X, X11, Wayland, Embedded Linux, BlackBerry 10, Android, iOS och VxWorks. [2]

Qt uppnår denna typ av plattformsoberoende genom dess preprocessor *Meta Object Compiler* (moc). moc söker igenom projektets källkodsfiler efter vissa Qt-specifika makron och nyckelord, och omvandlar dem till vanlig kompilerbar kod i C++. Så som betydelsen av just moc antyder så handlar det om att skapa och hantera *metaobjekt*, kort förklarat objekt som innehåller information om ett annat objekt. Med hjälp av detta system möjliggörs skickande av meddelanden objekt emellan (*signals*/*slots*), introspektion, reflektiv programmering och unionstrukturer, egenskaper som C++ inte har naturligt (eller som åtminstone expanderats i Qt). [3] Just signals och slots utgör ett attraktivt alternativ till att använda så kallade *callback*-funktioner och händelsebaserad programmering, tack vare en högre abstraktionsnivå och enklare generering i varierande sammanhang.

Qt har en stor uppsättning klasser (nära 900 i femte versionen) för en rad olika mjukvarukonstruktioner. Bortsett från ovan nämnd funktionalitet hittar man även 2D-grafik, OpenGL, XML, nätverkskommunikation, WebKit-integrering, databashantering, ECMAScript-stöd med mera. Qt har dessutom alternativ (plattformsspecifika typdefinitioner eller jämfört med standardklasserna utökad funktionalitet) till många datatyper, standardklasser eller STL-containrar.

C++ är det primära programmeringsspråket som används i Qt, men inofficiella portningar till en rad andra existerar. Med de senaste versionerna har ett nytt, deklarativt alternativ till den procedurella gränssnittsprogrammeringen introducerats. Detta kompletteras med ECMAScript för viss programlogik, samtidigt som bindningar till C++-källkod kan anslutas där detta behov uppstår. Detta deklarativa språk går under benämningen *Qt Modelling Language* (QML). [4]

Qt har utvecklats sedan mitten av 90-talet, inledningsvis i egen regi inom företaget Trolltech. Detta köptes upp av Nokia 2008 som tre år senare sålde det vidare till finska Digia. [5] Som kuriosa kan nämnas att Qt ska uttalas som engelskans *cute* [kju:t].

2.2 Designmönster

Designmönster (eng. *design pattern*) är ett begrepp som innefattar välstuderade och effektiva lösningar på situationer som inte sällan dyker upp under konstruktionsprocessen. Dessa lösningar är generellt beskrivna på designnivå och kan därför implementeras i många olika programmeringsspråk, ofta bara med små skillnader utifrån det aktuella språkets egenskaper. Designmönster delas allmänt in i tre kategorier efter deras funktion: skapande, struktur och beteende. Bland de mer namnkunniga mönstren hittar man *singleton*, *factory method* och *adapter*.

Ett angränsande koncept till designmönster är *Commonality and Variability Analysis* (CVA). Detta handlar om att i en struktur identifiera dels de gemensamma dragen, dels de som varierar. Det gemensamma låter man sedan på implementationsnivå representeras av en gemensam, ofta abstrakt basklass; det varierande kapslas in i separata, konkreta underklasser. Detta kan med andra ord omformuleras i *open-closed principle*: låt metoder, klasser och moduler vara öppna för utvidgning men stängda för ändringar. [6]

Två ytterligare generella koncept som många designmönster använder sig av är att sträva efter är ”svag koppling” (*loose coupling*) och ”hög sammanhållning” (*high cohesion*). Det förstnämnda begreppet innebär att en programmeringsmodul (klass, metod) ska veta så lite om sin omgivning som möjligt och därför vara lätt att byta ut mot en motsvarande; med det senare begreppet menar man att en modul utför *en* specifik uppgift, vilket förenklar förståelse och återanvändning.

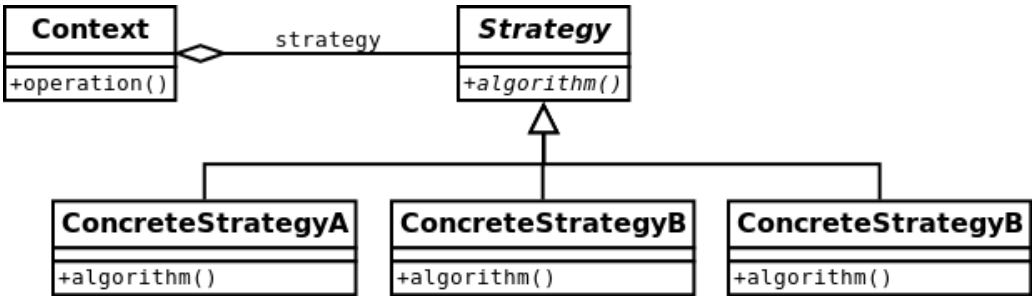
Det verk som lyft fram och haft störst påverkan inom detta område för mjukvaruutveckling är boken *Design Patterns: Elements of Reusable Object-Oriented Software* av fyra författare som ofta kallas för ”Gang of Four” (Gamma, Helm, Johnson, Vlissides). [7]

2.2.1 Strategy

Strategy eller *policy* är ett beteendeinriktat designmönster som går ut på att man kapslar in ett antal olika algoritmer i var sin klass och med hjälp av en gemensam basklass till dessa, låta den specifika algoritmklassen väljas fritt beroende på

klientens sammanhang. Detta hellre än att ha alla algoritmer i en och samma klass och låta exempelvis en switch-sats eller liknande välja rätt.

Figur 1 nedan visar ett generellt klassdiagram över strategy:



Figur 1: Strategy-mönstrets generella struktur

Man kan till exempel tänka sig att använda ett stategy-mönster om vi har tre algoritmer för hur en fil ska läsas av, beroende av filens typ. Varje algoritm implementeras då i varsin ConcreteStrategyX ovan, medan Strategy-klassen utgör det gemensamma gränssnittet för att interagera med de ärvda underklasserna på ett för klienten (Context) transparent sätt. Eftersom Context inte i förväg känner till vilken filtyp som kommer att dyka upp, har vi en referens till ett Strategy-objekt (datamedlemmen strategy) som sedan kan peka till den konkreta Strategy-klass vår klient bedömt vara rätt för den aktuella filen; valet av konkret Strategy-klass görs utifrån någon logik i klienten och ingår inte i mönstret.

2.2.2 Observer

Observer är ett designmönster som även är känt som *publish-subscribe*. I detta ingår ett objekt (*subject)* som innehåller en lista med andra objekt (*observers*), vilka är beroende av det förstnämndas tillstånd. Ändringar i subject kommer automatiskt att vidarebefordras till dess observers, som då kan uppdatera sitt eget tillstånd.

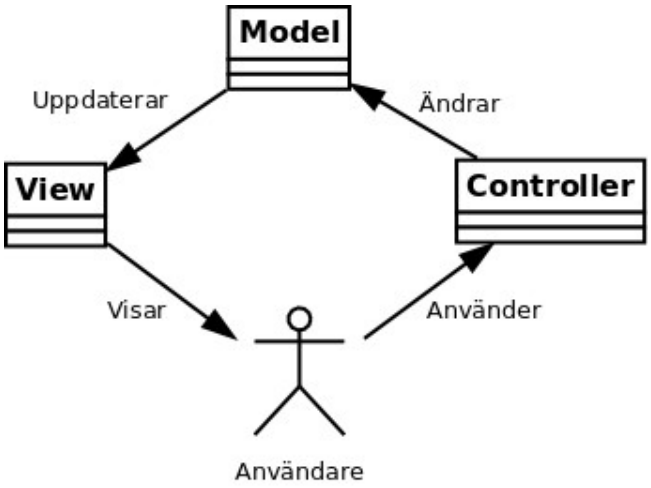
För att möjliggöra för andra objekt att lägga till och ta bort sig själva som observers, behöver subject ett sådant gränssnitt. Den interna datamedlem i subject som lagrar observers måste givetvis ha en fastslagen klass till vilken alla observers tillhör, men man önskar inte tvinga alla observers att vara av samma klass. Därmed behöver de ärva från en gemensam basklass någonstans i deras respektive klasshierarki. På så sätt behåller *loose coupling*: subject kan leverera till många olika typer av objekt utan att känna till någonting om deras specifika utseende bortsett från det gränssnitt de erhållit från observer-basklassen.

Ett exempel på detta designmönstret är Qt:s signals/slots, där alla observers ärver från klassen QObject. Observer-mönstret används ofta inom grafiska applikationer för att uppdatera det visuella när den bakomliggande informationen förändras, vilket beskrivs mer noggrant i följande avsnitt.

2.2.3 Model-View-Controller

Model-View-Controller, MVC, är trots likheter egentligen inte ett designmönster utan beskriver en arkitektonisk klasstruktur på en högre nivå, vilken internt med fördel kan bestå av ett antal designmönster. MVC används vid grafisk presentation av data med motiveringen att separera data (*model*, modell) från det visuella (*view*, vy) och använda ett medlande objekt (*controller*, kontroller) för att vidarebefordra användarens manipulering av vyn till modellen. [8] Genom denna uppdelning kan man enklare hantera de enskilda, ur logisk synvinkel knappast sammankopplade delarna, oberoende av varandra. Det blir således möjligt att byta ut modellen men bevara vyn, precis som att en och samma modell kan ha olika vyer (en eller flera samtidigt); här ser man designmönstret observer användas, där modellen utgör subject och vyerna observers.

I Figur 2 nedan visas hur de olika komponenterna interagerar inbördes och med användare.



Figur 2: Interaktion i Model-View-Controller

MVC har sina rötter från programmeringsspråket Smalltalk och framträdde som ett generellt koncept i artikeln *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.* [9]

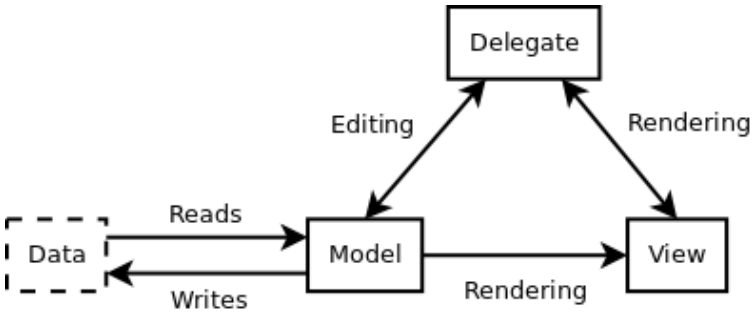
2.2.4 MVC i Qt

Qt använder sig av en variant av MVC för att hantera data i dess grafiska applikationer. Skillnaden mot den ursprungliga strukturen som beskrivs ovan består i att de ersatt kontrollern med en komponent kallad *delegate* (delegat, ombud). Funktionen för delegaten är liknande kontrollerns men inte identisk. Förutom att uppdatera modellen har den till uppgift att rendera varje enskilt objekt i modellen i vyn, vilket medför att båda uppgifterna ligger sammanpackade i en modifier- och utbytbar komponent.

Qt-dokumentationen brukar i dess version av MVC även se data som något som separat från modellen: modellen utgör ett gränssnitt för hur dess data ska

manipuleras utan att delegaten behöver känna till utseendet hos datastrukturen. [10] Qt erbjuder ett antal färdiga modell- och vyklasser för att användas direkt, men även abstrakta klasser tänka att ärva ifrån i egna konstruktioner.

Figur 3 visar hur Qt:s Model-View-arkitektur kan beskrivas grafiskt.



Figur 3: Qt:s variant av MVC

2.3 Liknande presentationsverktyg

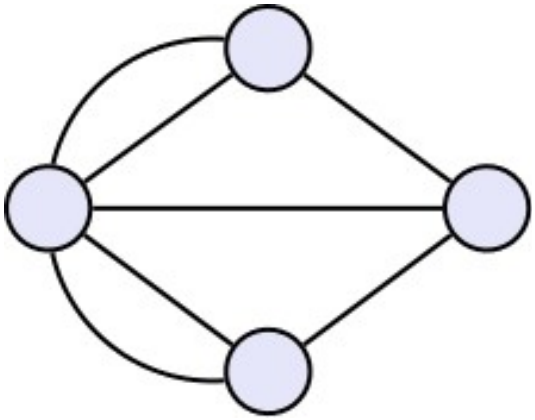
Det utvecklade verktyget är inte unikt i sig utan liknande hjälpmedel har existerat sedan många år tillbaka. Perl-scriptet cinclude2dot [11] läser av källkodsfiler i C/C++ och producerar en fil i dot-format; detta format ingår i grafbeskrivningsspråket DOT som utvecklats för verktyget Graphviz [12]. Med hjälp av programsviten för Graphviz (innehållandes ett antal olika verktyg för olika typer av strukturer med skiljande krav på algoritmens positionsgenerering) och dess utbredda stöd för bindningar till olika språk och sammanhang, tillåts användare att skapa grafiska representationer av tolkade strukturer.

Gource är ett verktyg som läser av loggar från versionshanteringssystem så som Git, Mercurial, Bazaar och Subversion, och producerar en grafisk presentation av dess filstruktur och hur den förändrats över tid. [13]

För Java finns ett tillägg till utvecklingsmiljön Eclipse kallat AgileJ som läser av projektstrukturer och dess ingående filer för att sedan grafiskt presentera kontinuerliga ögonblicksbilder över samtliga klasser och deras inbördes förhållanden. [14]

2.4 Grafteori

Den matematiska inriktning som studerar strukturer av punkter (även kallade noder eller hörn) och deras sammanbindande linjer (kanter, bågar) brukar kallas *grafteori*; dessa strukturer går under namnet *grafer*. Detta område har sitt ursprung i den schweiziske matematikern Leonard Eulers verk om Königsbergs sju broar, [15] vilken kan representeras genom grafen i Figur 4. Denna graf har fyra noder och sju kanter.



Figur 4: Graf över Königsbergs broar

En graf vars kanter inte korsar varandra kallas *planär* graf. Ovanstående figur är av sådan art. För tydlighet i visualiseringen av en graf vill man helst att den ska vara planär, men det kan i bästa fall kräva att vi använder böjda kanter (bågar) för att ta oss runt andra potentiellt korsande kanter. I värsta fall kan en planär graf inte vara möjlig att representera i två dimensioner.

Grafteorin är nära besläktad med topologin, läran om geometriska föremåls form utan hänsyn till avstånd. Tar man hänsyn till kanternas representativa avstånd, kan grafteorin beskriva exempelvis datornätverk, satellitpositionering eller neurala nätverk; inom dessa områden hittar man många algoritmer kring traverserande och positionerande av sådana strukturer för att exempelvis finna kortaste vägen mellan två punkter (Dijkstras algoritm [16]) eller strukturera upp en graf genom att minimera den inneboende ”energin” i en grafs kanter. [17]

3 Metod

Detta kapitel beskriver den metodik som valts för arbetets genomförande och resultatinhämtning, och inleds med en kort beskrivning av arbetsordningen följt av den programkodsmässiga val och till sist resultatutvärdering.

3.1 Arbetsupplägg

I korthet kommer projektet att följa detta upplägg för att nå målen:

1. Inledande studieperiod där Qt, MVC-modellen och grafisk programmering undersöks närmare, i syfte att för att lägga grunden för klokare beslut av programstruktur och konstruktion.
2. Konstruera verktyget från grund och uppåt (*bottom-up*). Till en början i en test- och verifieringsfas med en XML-tolk hela vägen till grafisk presentationsnivå, för att sedan lägga till en källkodstolk. Upplägget av programmets interna design bör i detta skede vara väl mottagligt för ett sådant tillägg och bör fungera direkt. Utvecklingsordningen blir som följande:
    1. Central programklass
    2. XML-tolk
    3. Nodklass, nodmodell
    4. Nodvy, noddelegat
    5. Källkodstolk
3. Genomföra en undersökning där data i textform (”rådata”) jämförs med från verktyget genererad utdata i syfte att utvärdera vilka effekter det har på tolkningstid och tillförlitlighet.

3.2 Programmeringsmiljö

För utveckling av verktyget kommer programmeringsspråket C++ användas tillsammans med applikationsramverket Qt (version 5.0.1). C++ erbjuder god prestanda och effektiv resurshantering. Tillsammans med Qt kan mängden egenkonstruerad kod reduceras i avseende på exempelvis XML och reguljära uttryck för symbolmatchning, men särskilt när det gäller den grafiska biten. Den integrerade utvecklingsmiljö som kommer att användas är Qt Creator, som utvecklas i nära samarbete med själva ramverket. All utveckling och testning görs i en 64-bitars miljö i operativsystemet Ubuntu 13.04.

Ett alternativ till C++ skulle kunna vara Java, och till Qt även wxWidgets [18], men valet föll till sist på punkten gällande personliga preferenser och erfarenhet.

3.2.1 Kodstil

Den stil på programspråkskoden som valts bygger på gängse regler för god läsbarhet och konsekvent användande. Detta inkluderar:

- klass-, funktions- och variabelnamn med inledande gemen och kamelpucklar för ordseparering
- konstanter i versaler med understreck för ordseparering
- namn på klasser, funktioner, variabler och konstanter har valts utifrån att beskriva deras uppgift, trots att längden ibland blivit lidande
- getter-funktioner (vilka hämtar en privat datamedlems värde ur en klass) anges enbart med datamedlemmens namn, medan setter-funktioner (vilka skriver ett nytt värde till en privat datamedlem) döps med prefixet ”set” före datamedlemmens namn. För till exempel datamedlemmen *position* skulle detta bli position() för getter-funktionen och setPosition(*värde*) för setter-funktionen.
- klasser kommenterade inledningsvis
- funktioner beskrivna vid deras implementationer
- koden i övrigt välkommenterad

Programmets kodstil följer den som Qt-dokumentationen presenterar, men skiljer sig på punkten för hur konstanter skrivs ut: i Qt används kamelpucklar med inledande versal, men då den valda stilen är mer generellt utbredd inom C/C++ har den fått företräde.

3.3 Utvärdering av resultat

Det som i slutänden är intressant att få reda på är, enligt frågeställningen:

- Bidrar verktyget till att hjälpa användare att tolka data?
- Om ja, bidrar verktyget *tillräckligt mycket* för att vara användbart? Ersätter det att man tittar på källan över huvud taget, eller kompletterar det bara? Förkortas i så fall den totala tiden?
- Om nej, vad kan man ändra på för att förhoppningsvis göra det bättre?

För att kunna besvara frågeställningen kommer en undersökning att genomföras mot en panel av begynnande utvecklare. Denna undersökning kommer att bygga på jämförelser av snarlik eller identisk data som presenteras dels i sin ursprungsform, dels så som verktyget ritar ut den. Kvantifieringen av vinsterna med att använda verktyget kommer således att bli något lidande då undersökningsformen vid nätbaserade formulär inte riktigt tillåter tidsavläsning per genomförd fråga under vetenskapligt kontrollerade former. Därför kommer tidskvantifieringen bli subjektiv från deltagarens sida och således blir varje enskilt svar inte hundraprocentigt tillförlitligt. Dock bör den generella bilden av för- eller nackdelar i användandet av verktyget träda fram genom det samlade intrycket.

Även om osäkerhet i hur mycket tid (om någon) som kan tjänas in vid verktygets användning, bör man inte underskatta de subjektivt upplevda vinsterna (svårmätbara, tangerande kvalitativ metod). Genom att utnyttja färg och form som pedagogiska redskap kan den sammanlagda bilden av en datastruktur bli mer framträdande än om enbart text presenterade samma information, vilket bygger på välutforskat arbete inom perception och minne, så som *visual short-term memory* (VSTM) (se exempelvis [19] och [20]).

Förutom tid, rymmer frågeställningens ”tillgodogörande av information” även tillförlitlighetsaspekten, alltså hur säker användaren kan vara att en viss dragen slutsats angående en datastruktur faktiskt är sann. Data presenterad i textform kan ”dölja” information vid en icke tillräckligt noggrann analys från användarens sida (ett antagande som torde öka i sannolikhet med växande datamängd), medan maskinellt avläsande av samma struktur inte (bortsett från buggar i mjukvaran) begår samma misstag. Tolkningsmomentet kan i vissa fall minskas avsevärt i komplexitet när det genomförs maskinellt jämfört med manuellt, vilket också borde leda till färre misstag i den slutgiltiga analysen hos användaren.

Skillnader i tillförlitligheten hos de båda typerna av analysmetoder borde vara möjlig att undersöka genom att presentera snarlika eller helst identiska strukturer för deltagaren. Identiska strukturer med identisk data och symbolnamn kan inte presenteras vid undersökning av både rådata och grafiskt visualiserat utan att kontaminera deltagarens svar vid det andra momentet med slutsatser dragna från det första. Dock återstår möjligheten att presentera en identisk struktur men använda andra symbolnamn. De skillnader som detta medför torde vara försumbara i sammanhanget. Att presentera helt olika datastrukturer leder till att några slutsatser kring de olika presentationsmetodernas tillförlitlighet i stort sätt är omöjliga att dra, och således inte ett giltigt alternativ.

Att använda undersökningar som metod för utvärdering innebär alltid att introducera ett riskmoment då deltagaren kan välja att inte svara sanningsenligt eller utifrån bästa förmåga. Jämför man med pappersbaserade undersökningar, riskerar nätbaserade dito vara ännu mer lidande av detta antagande. Utifrån detta är det svårare att dra några definitiva slutsatser baserat på det samlade underlaget, men praktiska problem så som att distansbaserad undervisning medger inte med enkelhet andra kvantifierbara metoder.

3.4 Undersökningens utformning

Undersökningen genomfördes bland studenter som läser andra och tredje året på programmet Programvaruteknik vid Mittuniversitetet vårterminen 2013. Dessa studenter har kunskaper om programmering i C++ efter tre eller fler kurser i språket, men endast studenterna på det tredje året har studerat XML; märkspråket introduceras därför kort i inledningen av underökningen för att jämna ut eventuella skillnader i kunskapsnivå.

Undersökningen delades in i två huvuddelar: tolkning av XML-data och tolkning av källkod i C/C++. Båda dessa delar följde upplägget att de inleddes med ett

tolkningsmoment av ren textinformation med tillhörande frågor, följt ett tolkningsmoment av en identisk grafisk datastruktur, med motsvarande frågor, för att avslutas med några jämförande frågor kring hur deltagaren upplevde skillnaderna. De frågor som undersökningen innehöll återfinns i Bilaga A. I XML-delen presenterades samma struktur men med olika elementnamn för båda momenten, medan källkodsdelen innehöll samma data i text- och grafform. Alla bilder som presenterades hade genererats direkt med verktygets hjälp utifrån den textbaserade strukturen.

3.5 Alternativ metod

Under andra omständigheter, så som fysisk närvaro bland deltagarna, skulle ett program som mäter tiden för varje fråga kunnat ha konstruerats och körts på en utsedd dator tillsammans med verktygsprogrammet. Alternativt skulle en egen webbplats med ett frågeformulär kunnat ha skapats med tillhörande logik och databas, för att på så sätt kunnat mäta tidsåtgången. Båda dessa alternativ skulle dock inneburit en hel del extra arbete. Att distribuera exekverbara programfiler av verktyget till deltagarna ses inte som ett gångbart alternativ, då det skulle medföra både en högre tröskel för deltagande i undersökningen, samt separat kompilerade filer för alla olika plattformar som deltagarna kan tänkas använda.

Gällande programkonstruktionen kunde en mer agil angreppsvinkel ha valts. Kortare iterationer kunde snabbare ha lett fram till ett körbart program, dock med risken att mer kod tvingats skrivas om eftersom kunskaperna kring denna typ av programmering växte ju längre arbetet fortskred.

4 Konstruktion

I det här kapitlet får läsaren följa den tekniska aspekten av projektet som programkonstruktionen inneburit. Efter en inledande genomgång av kravspecifikationen, fortsätter texten med att beskriva klasstruktur och de viktigare algoritmerna som programmet använder sig av för att nå upp till kraven.

4.1 Kravspecifikation

Kraven på programmet delades upp i två kategorier: nödvändiga och önskvärda. De nödvändiga kraven är liknande dem som ställts upp som mål med projektets verktyg, medan de önskvärda inte behövs men ändå bör strävas efter av utvecklingsmässiga och användarvänliga skäl.

4.1.1 Nödvändiga egenskaper

De nödvändiga egenskaperna hos programmet inkluderar:

- läsa XML-filer
- läsa källkodsfiler för C/C++
- ta fram en modell för tillfällig lagring av det inlästa resultatet
- skapa ett grafiskt gränssnitt för användaren
- presentera modellen på ett logiskt sätt

4.1.2 Önskvärda egenskaper

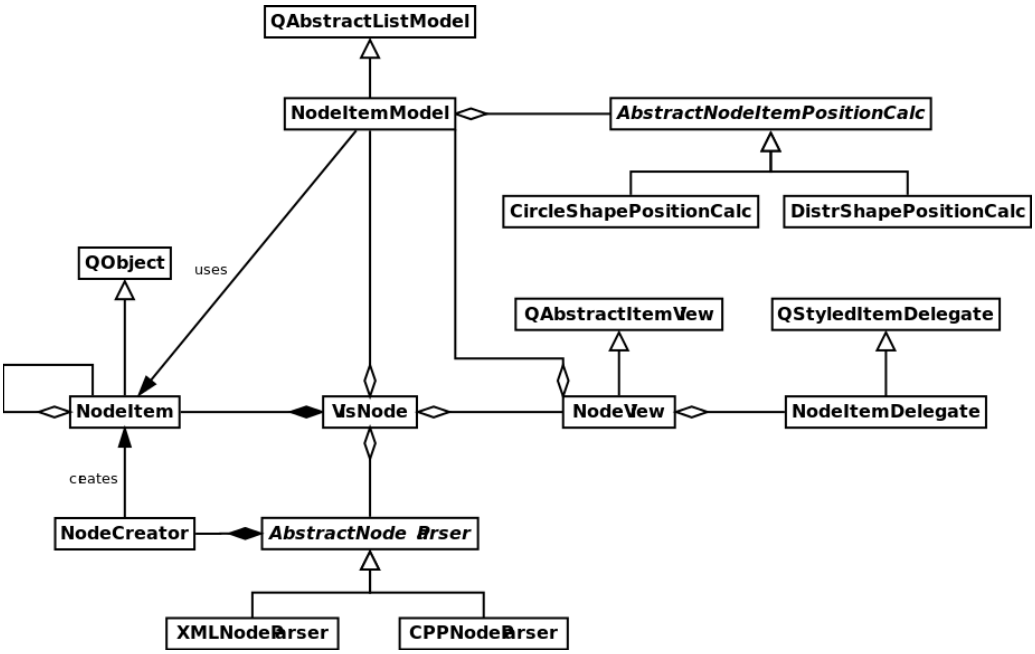
De önskvärda egenskaperna hos programmet är:

- plattformsoberoende
- öppen källkod
- starta programmet från terminalemulator eller motsvarande kommandoprompt såväl som genom systemets grafiska gränssnitt
- presentera modellen på ett användar- och tolkningsförenklande sätt
- tillåta modifiering av den grafiska modellen i efterhand
- loose coupling och high cohesion i klasstrukturen
- använda designmönster där lämpligt

4.2 Klasstruktur

Det utvecklade verktyget har egna 12 klasser fördelade på områdena inläsning av data (4 klasser), lagring av data (2 klasser), förberedande för presentation (3 klasser) och grafisk presentation (2 klasser), och till sist en klass för att styra de

övrigas instansiering och anropsordning. Figur 5 visar ett reducerat klassdiagram över den struktur programmet har, där endast kopplingarna mellan klasserna visas utan att avslöja något deras medlemmar. För ett fullständigt klassdiagram, se Bilaga B: Komplett klassdiagram.



Figur 5: Klassdiagram för VisNode

4.3 Centralklassens funktion

VisNode är den centrala klassen i programmet. Genom att instansiera ett sådant objekt och anropa dess enda publika funktion, run(), sätter man igång en kedja av instansieringar av programmets övriga objekt för användarinteraktivitet, filval, fastställande av inmatad datastrukturstyp, modellskapande, positionsberäkningar och till sist presentation genom grafisk utritning. Denna klass kan antingen ta en lista med filer som ska undersökas om man startat genom ett terminalkommando, eller om en sådan lista saknas så presenteras användaren en dialogruta för att välja filer den vägen. VisNode lagrar även den lista med noder som utgör datamodellen, vilken skickas med som en referens till många av de övriga klasserna i programmet för modifikation eller beräkningar. Skapandet av ett VisNode-objekt görs typiskt från main().

4.4 Inläsning av data

Utifrån vilken typ av filer som användaren har valt att undersöka, skapas en av två konkreta underklasser till den abstrakta basklassen AbstractNodeParser, alltså XMLNodeParser (för filtypen .xml) eller CPPNodeParser (för .c, .cc, .cpp, .cxx, .h, och .hpp). Undersökningen av de olika filtyperna har separerats i två klasser för att kapsla in skillnaderna i deras interna funktioner, men genom det

publika gränssnittet från basklassen kunna behandla dem lika utifrån sett. Detta följer designmönstret strategy.

De interna skillnaderna består av separata algoritmer för att gå igenom var och en av de medskickade filerna. Dessa algoritmer har i uppgift att hitta de symboler som sedan ska omvandlas till noder enligt den datamodell som programmet använder. För att sedan skapa själva noderna (av klassen NodeItem) används ett NodeCreator-objekt från basklassen. Genom att lyfta ut denna funktionalitet till en egen klass kan *high cohesion* bibehållas.

4.4.1 Skapandet av noder

När ett NodeCreator-objekt används för att lägga till en ny nod skickas både namnet på den nya noden och namnet på den nod som ska fungera som dess förälder med. Internt i NodeCreator kontrolleras båda dessa namn i den existerande datamodellen, och om något av dem inte redan existerar kommer det att skapas. Sedan läggs noden till i förälderns lista över dess barnnoder.

Eftersom vi undersöker *strukturer* av data, är det högst sannolikt att samma typ av data (element eller klassfil) dyker upp flera gånger i de undersökta filerna, men det är endast intressant att lagra varje typ en gång i modellen. Här har ett singleton-liknande designmönster använts.

4.4.2 XML-data

XMLNodeParser använder sig av en av Qt:s inbyggda XML-tolkklasser, QXmlStreamReader, för att hitta symboler, vilket i denna nomenklatur går under begreppet *tokens* (symboler). Tokens kan enligt XML vara starttagg (exempelvis <*font*>), stopptagg (</*font*>), attribut (*size="10"*), textdata (teckensträngar mellan start- och stopptagg) och så vidare. Den data vi främst är intresserad av är tokens av typen starttagg: varje nytt sådant utgör en potentiell ny nod i vår kommande datamodell.[1]

QXmlStreamReader-medlemen utnyttjas i en rekursiv algoritm som ser ut som följande:

1 Börja vid dokumentets början (ingen förälder)

2 Gör följande tills slutet på dokumentet nåtts eller den funna taggens namn är densamma som förälderns (stopptaggen hittad):

  2.1 Läs nästa token tills en start- eller stopptagg hittats

  2.2 Om det är en starttagg:

    2.2.1 Skapa en ny nod för detta element med elementets förälder som nodförälder

    2.2.2 Kör steg 2 användandes detta element som förälder

1 Potentiell används här för att understryka att vi endast är intresserade av att lagra varje element en gång i datamodellen. Däremot kan elementet dyka upp på andra platser i XML-dokumentet, varpå det läggs till som en barnnod till omgivande element (föräldranod).

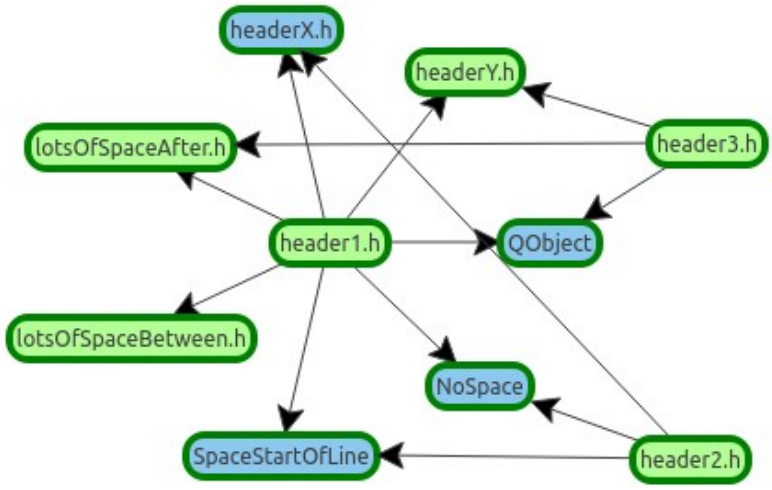
Den XML-fil som främst användes för att testa algoritmen och programkoden hittas i Bilaga C, vars grafiska representation återfinns som Figur 9 i resultatkapitlet på sidan 23.

4.4.3 Källkod i C/C++

CPPNodeParser undersöker källkodsfiler rad för rad och letar efter *include*-direktiv med hjälp av reguljära uttryck (användandes Qt-klassen QRegularExpression) och en textström. Algoritmen för detta kan beskrivas som:

1 Plocka ut filnamnet och använd som förälder
2 Skapa en nod med detta namn
3 Gör följande tills filens slut nåtts:
  3.1 Läs av en rad
  3.2 Om raden börjar med include-direktivet:
    3.2.1 Hitta positionerna för var klassnamnet börjar och slutar
    3.2.2 Skapa en ny nod för det hittade klassnamnet med filnamnet som förälder

För att verifiera att include-direktiven läses in korrekt användes tre enkla definitionsfiler (se Bilaga D), vilka resulterar i grafen som visas i Figur 6.



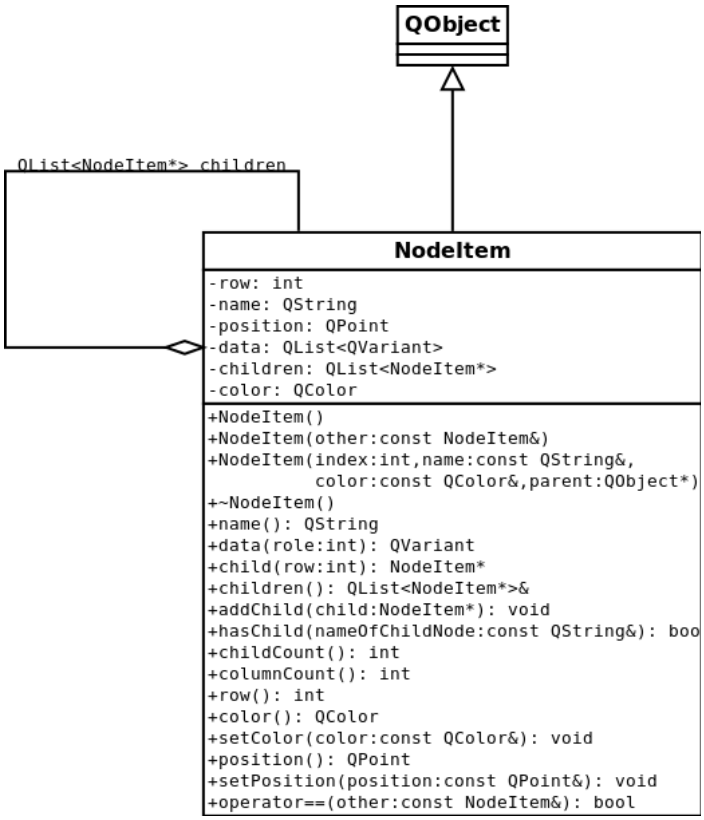
Figur 6: Exempel på graf med källkodsfiler

4.5 Lagring av inläst data

Varje unik, inläst symbol representeras av en nod (samma betydelse som i grafteorin), mer specifikt ett objekt av klassen NodeItem. Figur 7 nedan visar hur denna klass är uppbyggd. Att klassnamnet innehåller ordet ”item” är för att

förtydliga att klassen ingår i den MVC-modell som hanterar den grafiska presentationen.

Varje NodeItem har en lista med pekare till andra NodeItem som representerar dess barnnoder. Med hjälp av denna modell kan man representera både rent hierarkiska strukturer (XML) och mer ”spindelnätsliknande” strukturer (så som källkodsfiler uppdelade i deklarations- och definitionsfiler).



Figur 7: Klassbeskrivning av NodeItem

Alla NodeItem-objekt lagras sedan i en lista. Rent implementationsmässigt har QList använts, vilken är den rekommenderade containern i de flesta fall för Qt och vars egenskaper gör att den duger väl även i detta fall. Många av dess medlemsfunktioner är sådana både till namn och funktionalitet att de uppvisar en likhet i det sätt som modellklassen NodeItemModel arbetar, vilket förenklar användandet.

Tilläggas bör även att varje nod får en färg tillskriven sig när de skapas under filtolkningsprocessen. För noder skapade från källkod gäller att egenkonstruerade filer ges en blå bakgrund medan biblioteksfiler får en grön; för XML-filer används en färgskala som börjar med ljusgrön för att sedan förändra nyansen med ett konstant värde för varje ytterligare nivå nedåt i hierarkin från rotnoden som den aktuella noder ligger. Här har Nyans-Mättnad-Intensitet-skalan (NMI) använts.

4.6 Grafisk presentation

Den grafiska modell som valts för presentationen av nodstrukturen följer Qt:s variant av MVC-mönstret, som ersatt *Controller* med *Delegate*. I många fall när man arbetar med Qt:s Model-View-arkitektur kan man använda någon av de redan existerande, konkreta klasserna för var och en av de tre komponenterna. Dock har det inte varit möjligt i detta projekt eftersom den underliggande nodklassen har flera egna datamedlemmar (färg, position, barnnoder) som automatiskt inte är tillgängliga genom standardklasserna, samt att den utritade grafiska presentationsformen ligger utanför standardvyklassernas förmåga.

4.6.1 Modell

NodeItemModel är den konkreta underklass till QAbstractListModel som skapats i syfte att hantera den lista av NodeItems som konstruerats tidigare under programkörningen. Basklassen är inte den mest fundamentala abstrakta klass som går att välja här, utan är något specialiserad för just listliknande datamodeller och därmed reducerad i avseende på vilka rent virtuella funktioner som måste implementeras. Eftersom den datamodell som används är just en lista, fungerar denna konstruktion mycket väl.

Förutom att agera som ett förenande gränssnitt mot själva datan, har modellen i detta program även uppgiften att beräkna de inbördes positionerna hos noderna i den grafiska representationen.

4.6.2 Positionsberäkning

Då modellklassen är skräddarsydd för just NodeItem, har även en positionsberäkningsklass lagts in i densamma. Egentligen är det tre positionsberäkningsklasser: en abstrakt basklass och två konkreta underklasser till denna. Strukturen här är liknande det strategy-mönster som valts för tolkklasserna ovan, men valet av vilken underklass som ska användas har hårdkodats i den slutliga versionen av programmet. Den klass som exkluderats här (CircleShapePositionCalc) är enklare både i sin uppbyggnad och den struktur som noderna placeras i. Eftersom denna klass i praktiken inte används i verktyget ägnas inget ytterligare utrymme för att beskriva den algoritm som utnyttjas, mer än att säga att den nod med flest barnnoder placeras i centrum med övriga noder i en cirkel omkring.

Klassen DistrShapePositionCalc (”Distr” för *distributed*, fördelad) är i majoriteten av alla avseenden ett bättre val för beräkning av nodernas positioner jämfört med den enklare syskonklassen. DistrShapePositionCalc räknar det totala antalet kopplingar mellan NodeItems, alltså både från förälder till barn och barn till förälder. På så sätt får man en mer rättvis bild av vilka noder som bör placeras på en plats som tillåter att många linjer (grafteoretiska kanter) kan ritas ut mellan noderna. Dessutom, som namnet antyder, är positionsplaceringen utspridd: från en central nod omgiven av en cirkel av noder med koppling till denna, växer strukturen utåt från var och en av dessa omgivande noder i ett expanderande cirkelsektormönster. Se Figur 9 på sidan 23 för ett exempel på hur detta kan se ut.

Den algoritm DistrShapePositionCalc använder ser ut så här:

1 Skapa en hashtabell av alla noder för enkel åtkomst via textsträng
2 Beräkna alla kopplingar (både från och till) för varje nod
3 Skapa en lista fallande sorterad på antalet kopplingar för noderna
4 Välj ut den nod med flest antal kopplingar:
  4.1 Placera denna nod i origo
5 Hitta alla noder med en koppling till den centrala
6 Placera dessa i en cirkel runt om den centrala
7 För varje nyss utplacerad nod:
  7.1 Hitta alla noder med en koppling till denna nod (lokal centralpunkt)
  7.2 Exkludera alla noder som redan placerats ut
  7.3 Placera de återstående noderna i en bågformation riktat bort från den lokala centralpunkten
  7.4 Upprepa steg 7 rekursivt tills alla noder har placerats ut

Positions- och vinkelberäkningarna av noderna, både i cirkel- (steg 6) och bågform (steg 7.3), använder trigonometriska funktioner och tar antalet kopplade noder per lokal centralpunkt i beaktande för hur stor radien ska vara för att därmed minska antalet överlappande nodutritningar. Notera att dessa ”placeringar” inte är detsamma som ”utritning på skärmen” utan att det i detta skede endast handlar om att ge noderna ursprungspositioner för den kommande utritningen i vyn. Eftersom koordinatsystemen hos denna rent hypotetiska utplacering och de logiska koordinaterna i vyn skiljer sig åt, måste de slutligen översättas till den senare formen genom en förflyttning. Denna skillnad ligger i att positionsberäkningen gjorts utifrån origo (0,0) som centrum medan vykoordinater alltid har origo i övre vänstra hörnet.

4.6.3 Vy

Precis som tidigare har nämnts är vyklassen NodeView även den särskilt konstruerad för att ta hand om NodeItems och samarbeta med NodeItemModel. Dess uppgifter är således att på en tom bakgrund rita ut alla NodeItem-objekt utifrån de (nu översatta) positionerna som tidigare beräknats, samt de pilar (kanter) som representerar riktningen på kopplingen mellan två noder.

NodeView är antagligen den mest komplexa klassen i programmet, dels då den som konkret underklass till QAbstractItemView kräver återimplementering av minst 10 rent virtuella funktioner, dels då den ritar ut noder och pilar med trigonometriska operationer. Funktionen paintEvent() är här central då den enligt underliggande grafikhantering anropas varje gång fönstret ska ritas ut eller uppdateras. Varje sådan händelse (*event*) kräver att positionen för nod och pil översätts i *viewport*-koordinater, alltså den del av den underliggande vyn som för

tillfället visas utifrån positionen på reglagen i rullningslisterna. Detaljnivån på dessa operationer är relativt hög och kommer därför inte beskrivas i rapporten mer utförligt än att presentera pilritningsalgoritmen översiktligt:

1 För varje nod (föräldranod):
  1.1 För var och en av dess barnnoder:
    1.1.1 Rita ut en pil från nod till barnnod:
      1.1.1.1 Beräkna vinkeln som pillinjen lämnar föräldranoden
      1.1.1.2 Beräkna vinkeln som pillinjen når barnnoden med
      1.1.1.3 Använd barnnodens storlek och linjens vinkel för att beräkna var pilens spets träffar barnnodens kant
      1.1.1.4 Rita ut pillinje
      1.1.1.5 Rotera och översätt det generellt utritade pilhuvudet i korrekt riktning på korrekt position
      1.1.1.6 Rita ut pilhuvud
  1.2 Rita ut nod (med hjälp av delegatklassen)

Den som är intresserad av detaljerna kring detta hänvisas till den kommenterade källkoden (se avsnitt 5.1). Utritningen av pilar och noder fungerar för ändamålet, men kan säkerligen kan effektiviseras och förenklas.

Vyklassen svarar även på händelser som rör storleksförändringar av fönstret och de musklick som görs där. Med hjälp av det senare har navigering med hjälp av musen införts på så vis att en drag-och-släpprörelse på en nod flyttar den till önskad position, medan samma rörelse utanför noder tillåter förflyttning av den vyns visade del.

4.6.4 Delegat

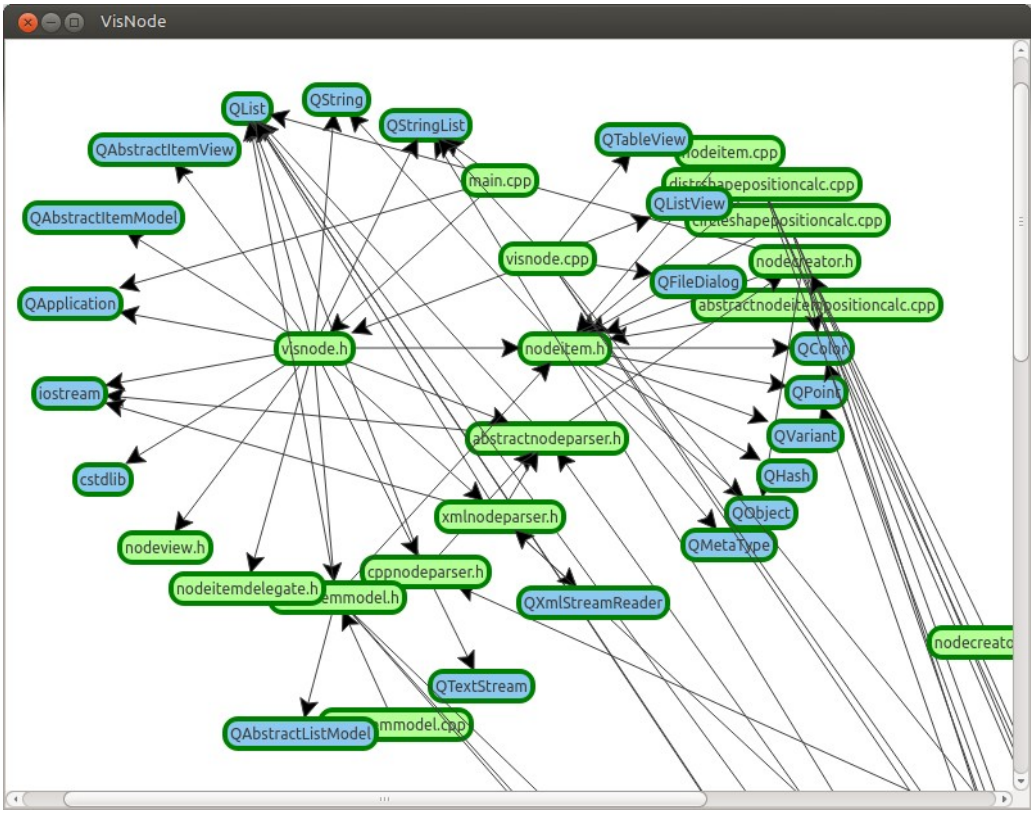
Delegatklassen NodeItemDelegate har som uppgift att rita ut själva noden. Eftersom modellen, bortsett från positionsförflyttningar med musens drag-och-släpprörelse, endast är läsbar, står inte delegatklassen i sig för några uppdateringar av existerande NodeItem-objekt. Både översiktligt och implementationsmässigt är NodeItemDelegate relativt enkel att förstå. I korthet använder den det aktuella NodeItem-objektets data för att skriva ut namn och bakgrundsfärg, för att sedan själv räkna ut storleken som behövs för ramen och till sist utföra dessa operationer på den position noden har.

5 Resultat

I detta kapitel presenteras de resultat som nåtts under arbetets gång. Inledningsvis presenteras verktygsprototypen i sin färdiga form och därefter sammanfattas undersökningens utfall.

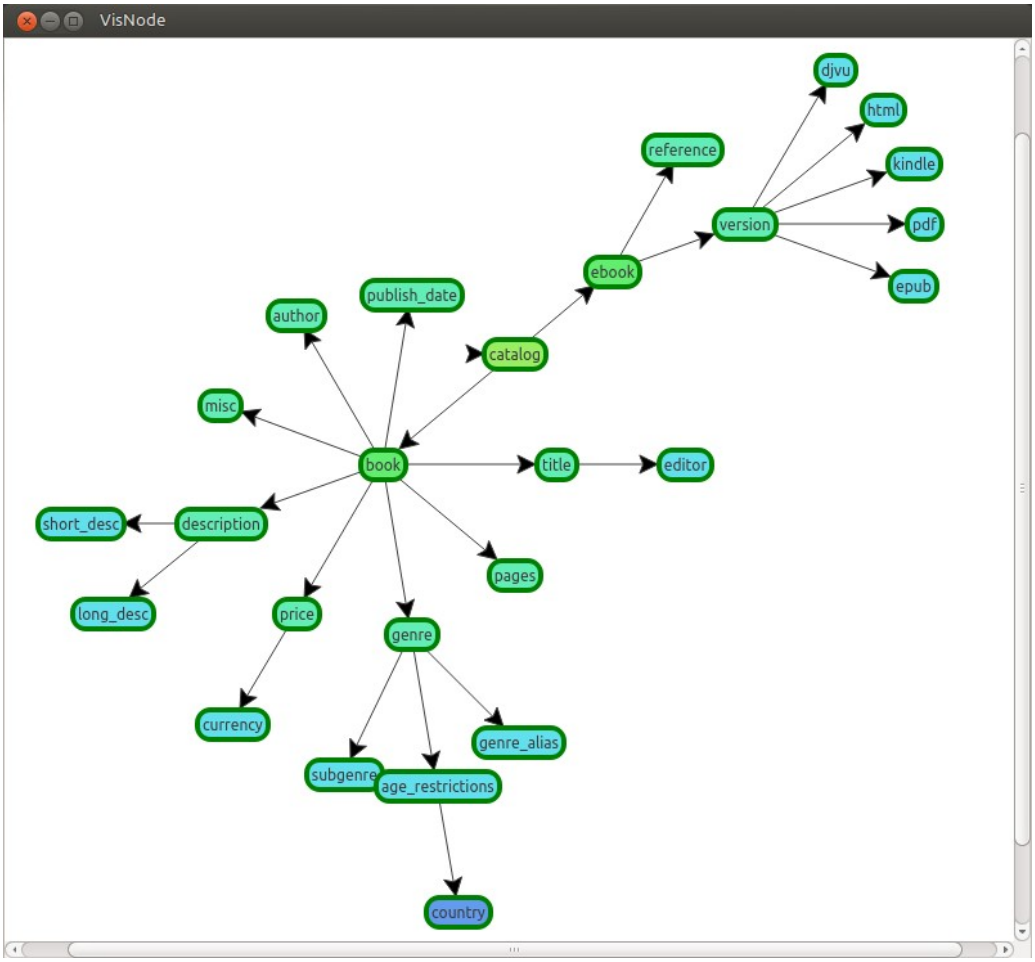
5.1 Verktygets slutliga form

Återgår man till att undersöka kravspecifikationen med det slutgiltiga programmet kan det konstateras att samtliga nödvändiga egenskaper existerar. Verktyget läser in och tolkar filer med både XML-data och källkod i C/C++ felfritt så långt som testning av olika filer visat. De inlästa noderna lagras tillfälligt i en lista, men någon möjlighet att spara strukturen mellan programkörningar finns inte (ingick inte heller i den uppställda kravspecifikationen). Två algoritmer för beräkning av nodernas positioner utvecklades och den mer avancerade av dem klarar av att skapa bra strukturer för alla XML-filer som testats. För samlingar av källkodsfiler som överstiger omkring 15 stycken tenderar strukturen att dels börja tappa form, dels bli svårtolkad på grund av det ökande antalet korsande linjer. Figur 8 visar hur detta kan se ut.



Figur 8: Verktygets presentation av 21 källkodsfiler

För en tydligare bild av hur positionsberäkningsalgoritmen ser ut när den fungerar som tänkt, se Figur 9, vilket är en representation av XML-filen i Bilaga C.



Figur 9: Verktygets grafiska presentation av en XML-fil

Sett till de önskvärda egenskaperna i kravspecifikationen som objektivt kan analyseras finner man att även dessa är implementerade i verktyget i någon grad. Startas programmet med filnamn som argument från terminalen kommer dessa filer att tolkas och presenteras direkt; startas det utan argument eller genom ett grafiskt gränssnitt visas ett dialogfönster där de önskade filerna väljs. När strukturen presenteras kan användaren använda muspekaren för att flytta på noderna och på så sätt modifiera strukturen; genom att dra i bakgrunden kan den visade delen av strukturen flyttas, vilket är lättare än att använda rullningslisterna.

Konstruktionsmässigt har klasserna och i viss mån medlemsfunktionerna strukturerats med CVA samt med svag koppling och hög sammanhållning i åtanke. Rent konkret har designmönstret *strategy* använts och även MVC-arkitekturen. Koden har främst skrivits för att fungera och är på vissa håll varken tydlig eller optimerad. Dock har kommentarer utnyttjats flitigt för att förenkla förståelsen för både utomstående och konstruktören själv i efterhand.

I och med att Qt har använts kan källkoden kompileras om på olika plattformar utan problem. Den som är intresserad av källkoden kan ladda hem den från:

https://github.com/maadborn/VisNode

Med versionskontrollsystemet Git installerat kan man enkelt kopiera ner all källkod till önskad plats med kommandot:

```
git clone https://github.com/maadborn/VisNode
```

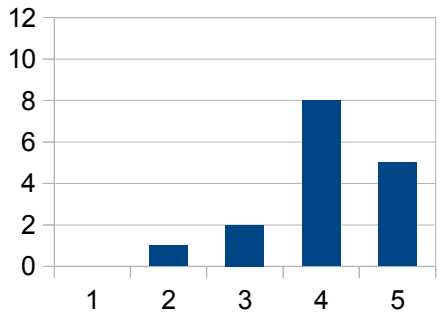
För att kompilera källkoden krävs Qt version 5. Källkoden är öppen för vem som helst att använda, modifiera och sprida fritt och utan kostnad, med andra ord *free and open-source software* (FOSS). Spridandet av exekverbara filer med Qt-innehåll är dock bara fritt så länge man inte länkar dem statiskt.

Verktyget gavs arbetsnamnet *VisNode*, en sammandragning av ”visualization of nodes”, vilket torde ge en antydning om vad det är för program.

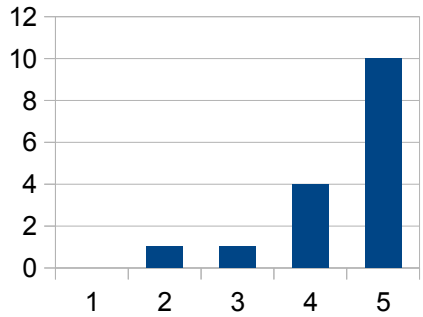
5.2 Undersökning av verktygets nyttoaspekt

Sett till helt korrekta svar i undersökningen kan ett visst mönster i tillförlitligheten skönjas när identiska strukturer och snarlika frågor har använts med skiljande presentationsform. Med enbart textdata gavs 69 % helt korrekta svar, medan motsvarande siffra var 88 % för den grafiska formen. Tidsaspekten har ej bedömts kvantitativt utifrån undersökningsformens begränsningar.

De subjektivt upplevda skillnaderna mellan att tolka datastrukturerna grafiskt gentemot enbart deras textform, visar en klar fördel i de grafiska modellernas riktning. Figur 10 och Figur 11 visar resultatfördelningen för dessa frågor, där 1 motsvarar ”Instämmer inte alls”, 3 ”Varken eller” och 5 ”Instämmer helt”.



Figur 10: Instämmande av vinster i tillförlitlighet med grafisk modell



Figur 11: Instämmande av vinster i tid med grafisk modell

6 Diskussion

Det här kapitlet innehåller de slutsatser som dragits under projektet, utvecklingen av verktyget och genomförandet av undersökningen, för att avslutas med några idéer om vidare arbete inom samma område. Det övergripande målet med detta examensarbete var att skapa ett verktyg för att grafiskt kunna visualisera datastrukturer och sedan utvärdera eventuella vinster i dess användande. Projektet resulterade i ett fungerande sådant program, vilket har utvärderats till att bidra positivt i en mätbar utsträckning.

6.1 Slutsatser

Denna rapport har visat hur man kan gå tillväga för att använda C++ tillsammans med Qt i syfte att konstruera en grafisk applikation. Det skapade verktyget kan läsa in XML-filer och källkodsfiler för C/C++, och sedan konstruera en grafisk modell för strukturen. Verktyget kan köras stabilt utan upptäckta och avgörande brister, medan den positionsberäknande algoritmen tillsammans med den grafiska modellen gör sig bäst för färre än 15 samtidigt inlästa filer.

Verktyget har utvärderats genom en undersökning bland studenter inom mjukvaruutveckling, där resultaten pekar mot objektivt uppmätta vinster i korrekt dragna slutsatser av datastrukturens innehåll (88 % helt korrekta svar med grafisk modell jämfört med 69 % för textbaserad form). Detta stämmer överens med deltagarnas egna åsikter om huruvida de upplevde att deras svar var mer tillförlitliga när de använde den grafiska modellen. Detta på samma gång som en övervägande majoritet av deltagarna svarade att de upplevde vinster i tid med en grafisk modell till hjälp.

Utifrån frågeställningen blir således slutsatsen att det finns vinster att hämta när det kommer till att ha ett grafiskt verktyg till hjälp för att tolka en datastruktur, vars ursprungsform är textbaserad. Tillförlitligheten var då omkring 25 % högre och med upplevd kortare tid ägnad åt tolkning och informationssökning.

Vidare kan det vara intressant att utvärdera hur mycket strukturanalys ger för förståelsen av den information som finns lagrad inuti, dels för XML-data, dels för källkod i C/C++. I det första fallet anser jag att den form som verktyget presenterar når långt i att bidra till användarens förståelse; söker man särskild information lagrad i strukturen kan strukturanalysen vara ett viktigt första steg för att sedan använda exempelvis frågespråket XPath för att hitta den. Vid källkodsanalys tror jag att vinsterna kan vara mindre. Här skulle en från källkodsfilerna automatiskt genererad klasstruktur i exempelvis (reducerad) UML vara till större nytta och verktygets representation mer ett komplement på en högre nivå av undersökning av okänd kod eller specifikt översikten av filernas sammanlänkning.

6.2 Diskussion kring konstruktionsval

Den grafiska modell som verktyget genererar påvisar vissa brister som härstammar ur den positionsberäkningsalgoritm som konstruerats för ändamålet. Främst handlar det om att grafen tenderar att förlora form vid mer komplicerade strukturer, i vilka kanterna blir många, långa och korsande. Algoritmen tar inte hänsyn till kanter som korsar varandra, vilket borde ha implementerats, för att på så sätt höja tolkningsbarheten. Samtidigt kan en planär graf vara omöjlig att uppnå med dessa strukturer oavsett algoritm, även om mer raffinerade algoritmer existerar och således utgör ett intressant spår för vidareutveckling av verktyget. Tredimensionella strukturer skulle kunna lösa problemet, men är samtidigt avsevärt svårare att skapa och positionsberäkna i. Sammantaget har just valet, konstruktionen och implementationen av positionsalgoritmen visat sig vara ett av de viktigaste områdena för att åskådliggöra grafiska representationer av textdata.

MVC har visat sig vara användbart under utvecklingen och har sannolikt förenklat arbetet i sin uppdelning av mer överskådlig och utbytbar kod. Förutom den klassuppsättning som använts, har Qt en parallell struktur baserad på vyklassen QGraphicsView tillsammans med modellklassen QGraphicsScene. Denna klassuppsättning hade i efterhand antagligen varit ett bättre val då den specifikt är utformad för godtycklig utritning av ett stort antal föremål, med inbyggd dra-och-släpp-funktionalitet, skalning, kollisionsdetektering, med mera.

En viktig observation gällande grafiska applikationer och koordinatsystem är origos placering i övre vänstra hörnet och att y-axeln är uppochnedvänd. Detta kan sätta griller i huvudet på programmeraren när det kommer till (förväntade jämförda med faktiska) positioneringar och hur trigonometriska funktioner beter sig.

6.3 Diskussion kring undersökningen

Undersökningen hade en relativt hög andel deltagare av de tillfrågade (runt 30 %), men det totala antalet var fortfarande lågt (8 personer), varvid de slutsatser som dragits i denna rapport inte kan fastställas som helt allmängiltiga ur statistisk synpunkt. Kunskaperna i främst XML kan ha varit på en betydligt lägre nivå bland deltagarna än vad man borde se i ett verkligt sammanhang. Med bättre kunskaper inom detta område skulle frekvensen av korrekta svar vid tolkningen av XML-strukturen i textform kunna förbättras och därmed minska skillnaderna mellan de olika presentationsformer.

6.4 Vidareutveckling och ytterligare forskning

Verktyget har flera områden som skulle kunna utvecklas betydligt, däribland:

- utökat stöd för fler datastrukturer;
- menyer och knappar i det grafiska gränssnittet för skalning, lagring, export till bildfil med mera;
- nytillägg och redigering av noder (vilket eventuellt skulle kunna propageras till rådatastrukturen).

Vissa delar av programkoden skulle dessutom må bra av refaktorisering och revidering av de val som gjorts under vägen, nu med större erfarenhet och en bättre helhetsbild.

På samma gång har arbetet handlat om prototyputveckling, *proof of concept* och utvärdering av eventuella vinster med sådana verktyg. Med de dragna slutsatserna kan en alternativ väg vara att bygga vidare på redan existerande format (så som DOT) och/eller verktyg (exempelvis Graphviz).

Vidare skulle det vara intressant att med mer välutvecklade verktyg utföra djupare och mer omfattande undersökningar av de kvantifierbara vinsterna med den grafiska representationen. Hur utspridd vetskapen är om och användningen av sådana verktyg, tillsammans med mängden konkreta användningsområden inom den professionella mjukvaruindustrin, skulle också vara intressant att ta reda på.

Källförteckning

[1] Vetenskapsrådet, ”Forskningsetiska principer inom humanistisk-samhällsvetenskaplig forskning”, http://www.codex.vr.se/texts/HSFR.pdf
Hämtad 2013-06-20

[2] Digia Plc, ”Supported Platforms”,
http://qt.digia.com/Product/Supported-Platforms/
Hämtad 2013-05-23

[3] A. Ezust, P. Ezust, *An Introduction to Design Patterns in C++ with Qt*. 2 uppl. Upper Saddle River: Prentice Hall, 2011

[4] Qt Blog, ”Qt Declarative API Changes”,
http://blog.qt.digia.com/blog/2009/08/21/qt-declarative-api-changes/#comment-4727
Hämtad 2013-05-23

[5] Digia Plc, ”About us”, http://qt.digia.com/About-us/
Hämtad 2013-05-24

[6] A. Shalloway, J. R. Trott, Design Patterns Explained: A New Perspective on Object-Oriented Design. 2 uppl. Boston: Addison-Wesley, 2005

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995

[8] J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4*. 2 uppl. Upper Saddle River: Prentice Hall, 2008

[9] G. E. Krasner, S. T. Pope, ”A cookbook for using the model-view controller user interface paradigm in Smalltalk-80”. *Journal of Object-Oriented Programming*, vol. 1, nr. 3, 1988, s. 26-49

[10] Qt Project, ”Model/View Programming”,
http://qt-project.org/doc/qt-5.0/qtwidgets/model-view-programming.html
Hämtad 2013-05-30

[11] Francis Irving, ”C/C++ Dependency Graphing”,
http://flourish.org/cinclude2dot/
Hämtad 2013-05-27

[12] Graphviz, ”Graphviz - Graph Visualization Software”,
http://www.graphviz.org/
Hämtad 2013-05-27

[13] Andrew Caudwell, ”Gource – software version control visualization”, http://code.google.com/p/gource/
Hämtad 2013-05-28

[14] AgileJ, ”UML Reverse Engineering for Java in Eclipse”, http://www.agilej.com/
Hämtad 2013-05-28

[15] Dartmouth College, ”E53 -- Solutio problematis ad geometriam situs pertinentis”, http://www.math.dartmouth.edu/~euler/pages/E053.html
Hämtad 2013-05-30

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs". *Numerische Mathematik,* nr 1, 1959, s. 269-271

[17] T. Kamada, S. Kawai, ”An algorithm for drawing general undirected graphs". *Information Processing Letters*, vol. 31, nr. 1, 1989, s. 7-15

[18] wxWidgets, ”About – wxWidgets”, http://www.wxwidgets.org/about/
Hämtad 2013-06-19

[19] H. Olsson, L. Poom, ”Visual memory needs categories”. *PNAS*, vol. 102, nr. 25, 2005, s. 8776-8780

[20] Y. Jiang, I.R. Olson, M. M. Chun, ”Organization of Visual Short-Term Memory”. *Journal of Experimental Psychology*, vol. 26, nr. 3, 2000, s. 683-702

Bilaga A: Undersökningens frågor

Denna bilaga presenterar de frågor som ställdes till studenterna i år 2 och 3 av programmet Programvaruteknik vid Mittuniversitetet läsåret 12/13.

På majoriteten av frågorna i del 1, 2, 4 och 5 hade deltagaren möjlighet att kryssa i ett godtyckligt antal rutor, varav endast en kombination var korrekt. Frågorna 11-14 tillät dock bara ett svar. På frågorna i del 3 och 6 fick deltagarna svara i grad av instämmande från 1 – Instämmer inte alls, 3 – Varken eller/Osäkert, och 5 – Instämmer helt.

Del 1: Frågor utifrån en XML-struktur i textform

1. Vilken/vilka noder hittar man _direkt_ under rotnoden (<farm>)?
2. Var kan du hitta <race_id>?
3. Finns dessa noder?

Del 2: Frågor utifrån en XML-struktur i bildform

4. Vilken/vilka noder hittar man _direkt_ under rotnoden (<catalog>)?
5. Finns dessa noder?
6. Var kan du hitta <item_number>?

Del 3: Jämförelse av tolkningen i del 1 och 2

7. Det gick snabbare att hitta ett svar med den grafiska modellen till hjälp
8. Det kändes säkrare att svaret blev rätt med den grafiska modellen till hjälp
9. Om XML-filen skulle innehålla många fler element men inte fler elementtyper (längre dokument, samma struktur), skulle den grafiska modellen vara till större hjälp för förståelsen

Del 4: Frågor utifrån källkod i C++ i textform

10. Du läser om en obskyr bugg i <sys/times.h> och vill rensa ditt projekt på denna header-fil. I vilken/vilka filer använder du dess innehåll?
11. Hur många filer i ditt projekt använder <sys/errno.h>?
12. Du måste porta programmet till en plattform med extremt litet lagringsutrymme. Du inser att Restart-klassen är "gigantisk". Används den i ditt projekt eller kan du ta bort den utan problem?

Del 5: Frågor utifrån källkod i C++ i bildform

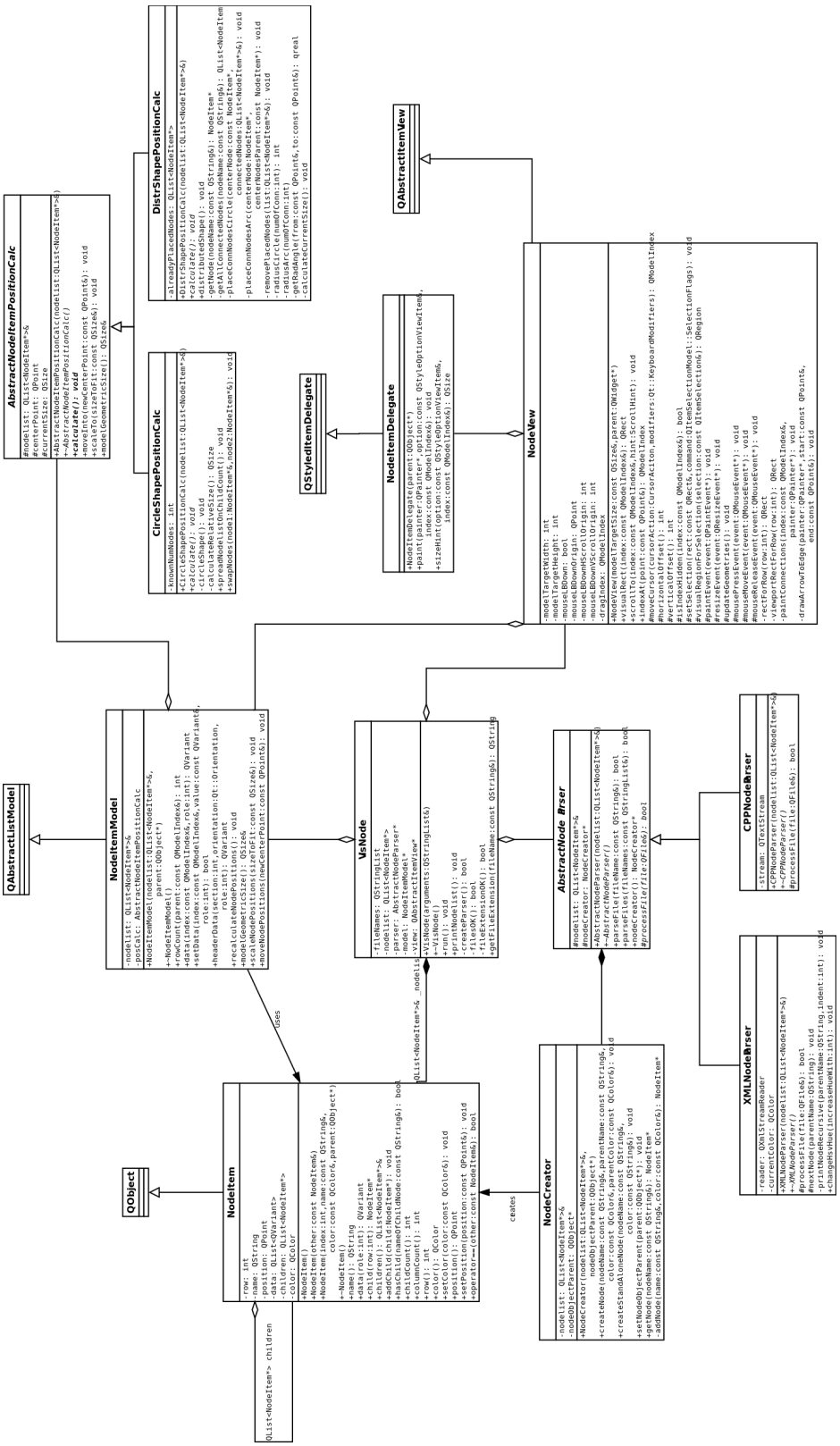
13. Hur många filer i ditt projekt använder <iostream>?

14. Du skulle på föregående sida besvara om Restart-klassen används i projektet. Om du får samma fråga igen med bilden till hjälp, hur svarar du då?
15. I vilken/vilka filer används fcntl.h?

Del 6: Jämförelse av tolkningen i del 4 och 5

16. Det gick snabbare att hitta ett svar med den grafiska modellen till hjälp
17. Det kändes säkrare att svaret blev rätt med den grafiska modellen till hjälp
18. Ju fler filer och inkluderingar projektet innehåller, till desto större hjälp borde den grafiska modellen vara för förståelsen av sammanhanget
19. Den grafiska modellen ersätter helt att man undersöker alla källkodsfilerna

Bilaga B: Komplett klassdiagram



Bilaga C: Testfil för XML-data

Denna fil generar bilden i Figur 9 när den körs genom verktyget.

```
<?xml version="1.0"?>
<catalog>
 <ebook>
      <version>
           <epub />
           <pdf />
           <kindle />
           <html />
           <djvu />
      </version>
      <reference />
 </ebook>
 <book>
      <title>
           <editor></editor>
      </title>
      <pages></pages>
      <genre>
           <subgenre />
           <age_restrictions>
                <country></country>
           </age_restrictions>
           <genre_alias />
      </genre>
      <price>
           <currency />
      </price>
      <description>
           <short_desc></short_desc>
           <long_desc></long_desc>
      </description>
      <misc></misc>
 </book>
 <book>
      <author></author>
      <title></title>
      <genre></genre>
      <price></price>
      <publish_date></publish_date>
      <description></description>
 </book>
</catalog>
```

Bilaga D: Testfiler för källkod

Dessa filer generar bilden i Figur 6 när de körs genom verktyget.

header1.h:

```
#ifndef HEADER1_H
#define HEADER1_H

#include <QObject>
#include<NoSpace>
 #include <SpaceStartOfLine>
#include      "lotsOfSpaceBetween.h"    // Test1
#include "lotsOfSpaceAfter.h"            //Test2
#include <Path/To/headerX.h> /* Test3 */
#include "another/headerY.h"

#endif
```

header2.h:

```
#ifndef HEADER2_H
#define HEADER2_H

#include<NoSpace>
 #include <SpaceStartOfLine>
#include <Path/To/headerX.h>

#endif
```

header3.h:

```
#ifndef HEADER3_H
#define HEADER3_H

#include <QObject>
#include "lotsOfSpaceAfter.h"              //Test2
#include "another/headerY.h"

#endif
```