Avdelningen för datavetenskap

Almgren Mats & Hansen Erik

Coridendro

Ett verktyg för att grafiskt åskådliggöra incidensen av malignt melanom inom olika släkter

Coridendro

A tool to graphically represent the incidence of hereditable malign melanoma within different families

Examensarbete 10 poäng
Dataingenjörsprogrammet

Termin: VT-2006
Handledare: Robin Staxhammar
Examinator: Stefan Lindskog
Ev. löpnummer: 2006:11

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Coridendro - ett verktyg för att grafiskt åskådliggöra incidensen av malignt melanom inom olika släkter

Almgren Mats & Hansen Erik

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Almgren Mats

Hansen Erik

Godkänd, 2006-06-19

Handledare: Staxhammar Robin

Examinator: Lindskog Stefan

Sammanfattning

Landstinget i Värmland deltar i ett större projekt, vilket består i att undersöka incidensen av hereditärt malignt melanom, inom olika släkter. Projektet går i korthet ut på att varje landsting följer upp patienter med malignt melanom. För de fall där patientens sjukdom kan förmodas ha ärftliga orsaker, upprättas ett släktträd med den aktuelle patienten som proband (den person som har insjuknat). Därefter ritas patientens släktingar in i släktträdet.

Varje person i släktträdet representeras av en symbol. Symbolens utseende beror på huruvida personen den representerar är: sjuk/frisk, avliden/i livet och man/kvinna. De symboler som ritas ut, sammanbinds därefter med linjer.

I dagsläget ritas dessa släktträd med papper och penna, vilket tar alldeles för lång tid. Dessutom uppstår det problem då släktträdens utseende varierar beroende på vilken person som har ritat dem.

Resultatet av vår analys av problemet, blev ett program där användaren klickar ut symboler på givna punkter. Programmet kan dessutom rita upp de linjer som sammanbinder symbolerna. Användaren markerar ifrån vilken symbol som linjen skall ritas samt vart linjen skall sluta.

Coridendro

Abstract

The county council of Värmland is taking part in a larger project which in general consists of surveying malign melanoma within different families. Counties in Sweden are following-up on patients suffering from malign melanoma, which is likely to have inheritable causes. When a suspected heritable malign melanoma is found an investigation is started. A pedigree is drawn, describing the propositus' family.

Every individual in the pedigree is represented by a symbol. The appearance of this symbol is determined by: unhealthy/healthy, diseased/alive, man/woman. When the individuals have been identified and the according symbols drawn, the symbols are connected by lines representing kinship.

Today the pedigrees are drawn by hand using pen and paper. This is a time consuming task and the appearance differ depending on who is drawing the pedigree.

The result of our problem-analysis is a graphical software tool. On a drawing area the user may draw the symbols on predefined locations. The tool also allows for symbols to be connected with lines. Symbols are connected through selecting a start-symbol and an end-symbol.

Innehållsförteckning

Figurförteckning

1 Inledning

Detta kapitel ger en motivering till varför Landstinget har börjat följa upp hudcancer. Det beskriver dessutom översiktligt hur uppföljningen av hudcancer går till, samt på vilket sätt vår applikation kommer att underlätta landstingets arbete.

1.1 Bakgrund

1.1.1 Sjukdomsbeskrivning

Malignt melanom är en elakartad form av hudcancer [19]. Prognosen för att klara av en sådan sjukdom är starkt beroende på om sjukdomen har spridit sig till andra organ, d.v.s. att melanomet har bildat metastas (dottersvulst). I detta fall är prognosen att klara av sjukdomen försämrad. Det är med andra ord viktigt att i ett tidigt skede diagnostisera tumörer och behandla dessa för att förhindra att de sprids i kroppen.

Det finns ett flertal olika orsaker till varför individer utvecklar Malignt melanom i huden, såsom exempelvis hudtyp i kombination med solvanor, eller om andra personer i släkten har haft malignt melanom [19]. Det faktum att arvsanlagen har betydelse för att utveckla melanom upptäcktes av en läkare vid namn William Norris i början av 1800-talet [19].

1.1.2 Kartläggning av sjukdomen

Från 1950-talet började forskare att intressera sig för de ärftliga samband som verkade finnas vid incidens (insjuknade) av Malignt melanom.

Vid mitten av 1970-talet gjorde Clark[4] stora genombrott med en omfattande undersökning som innehöll sex familjer, där varje familj hade minst två individer som utvecklat malignt melanom. Undersökningen resulterade bl.a. i vetskapen att vissa familjemedlemmar löpte så stor risk som 100 % att utveckla Malignt melanom innan 72 års ålder, där melanom ännu inte utvecklats men närvaron av dysplastiska nevi kunnats fastställas. Dysplastiska nevi är ett födelsemärke som innebär en ökad risk för cancer. De familjemedlemmar där det inte var möjligt att fastställa närvaro av dysplatiska nevi erhöll ingen ökad risk. Syndromet fick senare namnet Dysplatiskt Nevus Syndrom, DNS.

Motsvarande undersökningar utfördes i Australien, Holland samt Sverige. Sedan dess har mycket forskning gjorts inom detta område och nu genomför sjukvården i Sverige kontinuerliga kontroller av familjemedlemmar som tillhör högriskfamiljer.

1.1.3 Generellt system

Karolinska Institutet, KI, har utvecklat ett system för att åskådliggöra samt arkivera sambandet mellan arvsanlag och malignt melanom. Det går ut på att sjukhus runt om i landet skickar information om de patienter som ingår i riskfamiljerna till det Karolinska Institutet för sammanställning och analys.

1.1.4 Släktträdsrepresentation

KI har valt ett släktträdssystem för att representera riskfamiljerna. Dessa släktträd består av linjer som representerar släktskap samt figurer som representerar information om individen i fråga.

I dagsläget ritas dessa släktträd för hand på papper. Beroende på vem som ritar så ritas träden utifrån den personens tolkning av mallen, vilket ger att exempelvis symbolernas storlek, linjernas tjocklek och hur linjerna sammanbinder symbolerna kan variera.

1.2 Mål

Arbetet går ut att på att skapa ett verktyg för att underlätta grafisk representation av släktträd, vilka visar sjukdomsutfall av Malignt melanom. Med verktyget ska man kunna skriva ut släktträden så att dessa kan arkiveras. Vi ska dessutom producera en mindre manual för att underlätta programmets framtida nyttjande.

2 Beskrivning av nuvarande system

Kapitlet beskriver i detalj hur det nuvarande systemet fungerar. Här presenteras också de problem som finns med det nuvarande systemet samt ett förslag på hur dessa problem kan avhjälpas.

2.1 Beskrivning av systemet i helhet

För att kartlägga hur genetiskt ärftliga sjukdomar ärvs inom olika släkter, har Karolinska Institutet tagit fram ett system. Detta system beskrivs i en manual vid namnet ”Manual för handläggning av familjer med hereditärt malignt melanom och Dysplatiskt nevussyndrom” [9]. All information i detta delkapitel kommer ifrån denna referens. När en patient insjuknar i kutant malignt melanom (malignt melanom inklusive ”melanoma in situ”, vilket är ”malignt melanom” begränsat till hudens yttre cellager), kontrolleras det huruvida sjukdomsutbrottet kan bero på ärftliga orsaker. Orsaken till sjukdomen kan vara ärftlig om patienten ifråga har minst en nära[1] släkting som har insjuknat i malignt melanom. Om så är fallet, upprättas ett släktträd med denna individ som proband (den person som har insjuknat). Avsikten är nu att undersöka hela familjen, för att kunna avgöra om några individer uppvisar DN (Dysplastiska Nevi: beskriver ett födelsemärke på huden, där risken för att få kutant malignt melanom, är förhöjd). Den aktuelle patientens familjemedlemmar börjar i regel undersökas från och med puberteten. När den aktuella familjemedlemmen ankommer till det första läkarbesöket kontrolleras och dokumenteras de Dysplatiska och Banala Nevi (ett ofarligt födelsemärke) som finns på patientens kropp. Eventuella hudförändringar som skulle kunna vara malignt melanom undersökes snabbt. Efter ovan nämnda undersökningar, kommer familjemedlemmen att utfrågas om namnet på eventuella andra släktingar. Den familjemedlemmen, vilken utgör ”stamfader” i släktträdet, informeras om möjligheten för sina släktingar att bli undersökt på

[1] Nära släkting: syskon till såväl föräldrar som föräldrarnas föräldrar. Dessutom ingår föräldrarna och föräldrarnas föräldrar. Förutom detta, tillkommer egna: syskon, kusiner och barn.

den hudmottagning som tillhör den aktuelle släktingen. Nu ritas ett släktträd upp med den aktuella familjemedlemmen som proband (dvs. den person som antavlan utgår ifrån).

För att kunna förklara uppföljningen av läkarbesöket, måste vi förklara vad den kliniska diagnosen DNS, är för något. DNS definieras som följer.

2.2 DNS

Den kliniska diagnosen DNS (Dysplastiska Nevus Syndrom) definieras på följande sätt.

Definition: Betrakta en mängd av individer. Denna mängd betecknas med namnet A. Mängden A innehåller alla nära[1] släktingar till individ a (dock inte a sig självt). Alla individer (inklusive a) kan förses med ett av följande två attribut, eller båda:

- MM: kutant malignt melanom (kutant malignt melanom innefattar också melanom in situ, vilket är malignt melanom begränsat till hudens yttre cellager).
- DN: dysplastiskt nevi.

Person a har DNS om minst tre individer påträffas i mängden A, där varje individ har något av ovanstående attribut. Person a måste dessutom inneha något av ovan nämnda attribut.

2.3 Uppföljning

Beroende på vad undersökningen kom fram till, bestäms tidsintervall för återbesök. I huvudsak finns följande tre riktlinjer som beskriver inom vilket tidsintervall patienten ska återkomma [9]:

1. Patienter som har behandlats för malignt melanom och har DNS, kontrolleras var 3:e månad. Då 5 år har förlöpt sedan sjukdomsutbrottet, sker fortsatta kontroller med mellan 3-6 månaders mellanrum.
2. Patienter med DN och DNS kontrolleras mellan 3 – 6 månader.
3. Patienter med ärftlighet för malignt melanom, men utan DN får återkomma vid behov.

2.4 Beskrivning av systemets släktträdsdel

I dagsläget ritas alla släktträd för hand av två sjuksköterskor, vid namnen Anita Skoogh och Sara Rosengren på Centralsjukhuset i karlstad. För att utföra denna studie har ett generellt system för dokumentering av genetiska sjukdomar tillämpats. De symboler som är

nödvändiga för den här uppgiften har valts ut och använts. Av de symboler som finns i det generella systemet används nedanstående symboler. Nedanstående ska endast betraktas som en skiss, där symbolernas storlek inte nödvändigtvis är densamma som de vi implementerat i ritprogrammet.

Fylld symbol: Person som är sjuk.

Rektangulär symbol: Person som är man.

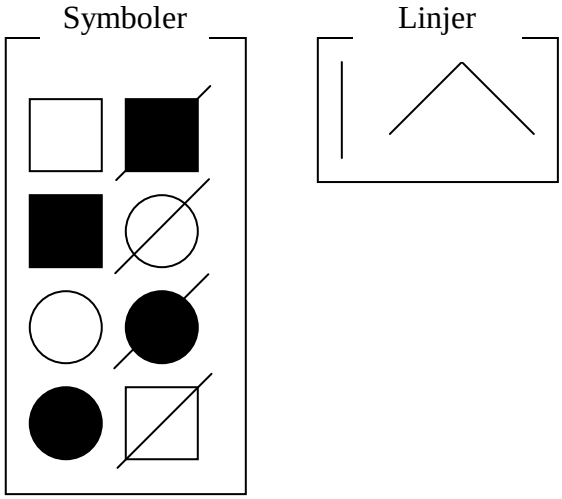
Cirkulär symbol: Person som är kvinna.

Symbol med streck över: Person som är avliden.

En symbol kan inneha flera av ovanstående attribut.

Enkel linje: Indikerar släktskap mellan två olika personer (linjen sammanbinder två symboler).

Dubbel linje: Indikerar tvillingskap (linjen sammanbinder tre symboler).



Figur 1: Beskrivning av de figurer som används i nuvarande system

Sjusköterskorna ritar ut symbolerna efter en mall som specificerar ungefär hur symbolerna ska se ut. Mallen specificerar vilken geometrisk form som de olika figurerna ska ha såsom rektangel, cirkel mm. Däremot beskriver inte mallen vilken storlek som figurerna ska ha. Mallen specificerar vidare inte fullt ut hur linjerna ska dras mellan symbolerna. Det är sjuksköterskorna själva som väljer en lämplig storlek på symbolerna, samt hur linjerna skall dras mellan symbolerna. När sjuksköterskorna har ritat ut symbolerna och dragit linjerna, lägger de till patienternas identitetsinformation såsom personnummer. När släktträden är färdiga arkiveras de i pärmar.

2.5 Problem med nuvarande system

Problemen med nuvarande system återfinns i uppritningen av släktträden. Sjuksköterskorna har själva definierat storlek på symbolerna, samt hur linjerna ska dras mellan symbolerna. Det är också tidskrävande att rita upp alla släktträd för hand samtidigt som släktträdens utseende varierar, beroende på vem som har ritat dem.

2.5.1 Lösningen

Vi skapar ett program som förmår användaren att rita upp släktträden på ett förutbestämt sätt. Användaren behöver inte tveka på hur släktträdet ska utformas. Linjerna ritas ut i 90-graders vinklar. När start och slutpunkt har markerats, ritas linjen upp. När användaren klickar på en symbol i programmet, för att rita upp den, får den en förutbestämd storlek som inte går att ändra. På detta sätt kan enhetliga släktträd ritas.

3 Förutsättningar och krav

Här beskriver vi de, från kunden såväl som utvecklarna, uttalade förväntningarna på programmets presentation, beteende och innehåll.

3.1 Förutsättningar

Utvecklarna kan inte förvänta sig någon form av ersättning för det arbete som de lägger ned. De datorer som krävs för utvecklingsarbetet kommer att tillhandahållas av utvecklarna själva, förutom versionshanteringssystemet som tillhandahålls av Sourceforge [14]. De personer som från Landstinget är inblandade på olika sätt är:

- Sara Rosengren i rollen som kund och sakkunnig i frågor rörande befintligt system
- Anita Skoogh i rollen som kund och sakkunnig i frågor rörande befintligt system
- Jan Olsson i rollen som sakkunnig i IT frågor
- Tomas Nilsson, chef IT enheten

Vid Karlstads universitet fanns Robin Staxhammar som dels agerade handledare i fråga om examensarbetet och främst då examensrapporten, men som även är kompetent inom objektorienterad programmering. Det är Robin som agerat bollplank under designen av de klasser som ingår i programmet.

Vi hade vid arbetets början inte gått någon kurs i programspråket C#.[2] Erik hade skapat ett mindre program åt en mattelärare vid universitetet men det var allt. Efter Donald Ross inspirerande programspråksföreläsningar var vi dock övertygade om att det inte skulle innebära något problem att skriva programmet i, ett för oss, nytt programspråk.

I projektarbetskursen erhöll vi en viss erfarenhet av att arbeta med versionshanteringssystemet CVS [6] och det skulle visa sig nyttigt vid övergången till Subversion [5]. Subversion är tänkt att ersätta CVS i framtiden och de två har mycket gemensamt.

[2] C# beskrivs i kapitel 5 i den här rapporten

3.2 Krav

3.2.1 Från kund

- För att programmet ska vara användbart för personalen på sjukhuset, är det nödvändigt att det går att köra på sjukhusets datorer, vilket innebär att programmet ska kunna användas under Microsoft Windows XP och Microsoft Windows 2000.
- För att programmet ska vara bakåtkompatibelt med det gamla papperssystemet krävs det att användaren kan skriva ut diagrammet och arkivera det.
- De symboler som fanns i det tidigare papperssystemet, se kapitel 2.4, ska också finnas i programmet. Användaren ska med hjälp av sin tidigare erfarenhet från papperssystemet kunna gå över och använda programmet istället. Det generella systemet[3] utvecklat av Karolinska Institutet fungerar väl och tanken med programmet är att bygga vidare på det, inte att skapa ett nytt.
- Information om de patienter som med symboler representerats i diagram får endast sparas på papper; detta med anledning av personuppgiftslagen. För att examensarbetet ska vara genomförbart krävs av sjukhusets IT-enhet att programmet inte sparar någon patientinformation.
- Det ska vara lätt att placera ut symboler på ritytan och sammanbinda dessa med linjer. Användaren ska exempelvis inte behöva lägga ned mycket tid för att få symboler att ligga på samma nivå i höjdled. Det skulle vara bra om det fanns fördefinierade punkter för linjer och symboler, då skulle det vara enkelt att få diagrammen att se bra ut.
- Programmet ska vara lätt att installera. För att administratörerna på sjukhuset ska kunna tillhandahålla programmet på ett tillfredsställande sätt, krävs det att programmet ska gå att installera med hjälp av ett installationsprogram. Ett alternativ vore att tillhandahålla programmet som en webservice (programmet körs över det lokala nätverket vid behov).
- En ovan datoranvändare ska kunna använda programmet med hjälp av manualen. Om inte programmet är lätt att använda spelar det inte någon roll hur smarta funktioner vi implementerat i det.

[3] som beskrivs i kapitel 2 i den här rapporten

- Om programmet ska vara användbart i praktiken i ett längre tidsperspektiv kan det bli aktuellt att utvecklarna utför support och underhåll mot kompensation.

3.2.2 Krav från utvecklare

- En dator där utvecklarna kan genomföra en pilotinstalltion för att säkerställa att programmet är lätt och smärtfritt att installera.
- Stöd av sakkunnig som använder papperssystemet.

4 Beskrivning av konstruktionslösningen

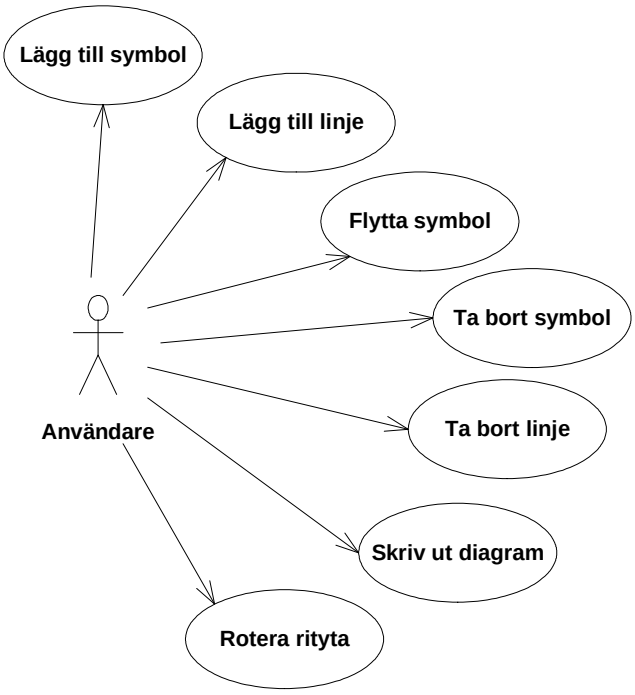
I det här kapitlet ger vi en beskrivning av den information som kunden har försett oss med samt hur denna information har resulterat i en specifikation och en design. Därefter går vi igenom de olika delar som designen innehåller d.v.s. användningsfall, användargränssnitt och klasshierarki.

4.1 Specifikation

Vi har fått en muntlig genomgång med kunden. Förutom detta har vi fått tillgång till dokumentationen av det gamla systemet. Kunden har dessutom försett oss med ett par släktträd som liknar de som finns i det nuvarande systemet. Vidare har kunden specificerat vilka symboler som behövs i systemet. Tillsammans har detta lagt grunden till specifikationen av den applikationen som vi har skapat.

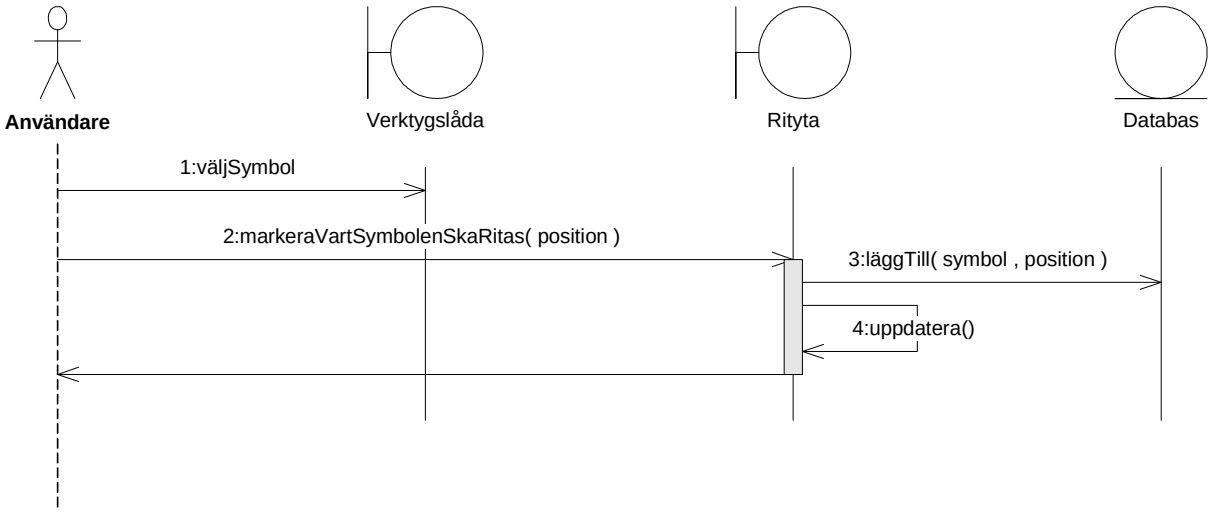
4.2 Design

4.2.1 Användningsfall



Figur 2: Visar alla tänkbara användningsfall

Vi har valt att dela in vårat program i ett antal användningsfall. Dessa har vi valt att åskådliggöra i ovanstående användningsfallsdiagram som baseras på kraven i kapitel 3.2. Vi har identifierat sex användningsfall. Det första är ”Lägg till symbol”. På grund av komplexiteten i användningsfallet ”lägg till en symbol”, har vi valt att åskådliggöra detta med sekvensdiagrammet i Figur 3. Användningsfallet ”lägg till linje”, beskriver fallet när en användare vill lägga till en linje. Fallet är snarlikt med att lägga till en symbol, därför har vi valt att inte ta med ett sekvensdiagram för detta fall. Nästkommande användningsfall, ”Flytta symbol”, beskriver scenariot att flytta en symbol på ritytan, ifrån en koordinat till en annan koordinat. Efter detta kommer användningsfallet att ta bort en figur ifrån ritytan, vilket utförs genom att användaren klickar på en figur i ritytan. Figuren kommer därefter att tas bort ifrån ritytan. Om användaren väljer att klicka på knappen ”skriv ut” i verktygslådan, uppkommer användningsfallet ”Skriv ut diagram”. En förhandsgranskningsdialog visas nu upp för användaren. Om användaren är nöjd med förhandsgranskningen, kan användaren välja att skriva ut diagrammet. Sist men inte minst finns användningsfallet ”Rotera rityta”. Detta användningsfall används för att rotera ritytan mellan stående och liggande. Detta kan dock endast utföras när ritytan är tom.



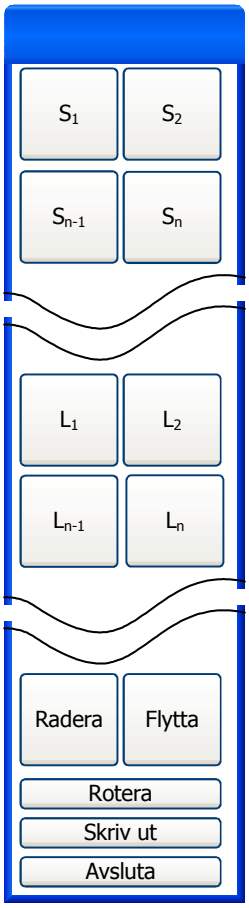
Figur 3: Beskrivning av användarfallet ”Lägg till symbol”

Ovanstående bild visar användningsfallet ”lägg till symbol”. Användaren startar applikationen och väljer därefter vilken symbol som ska ritas ut. Denna process markeras med anrop 1. När användaren har valt den symbol som ska ritas ut, måste användaren markera vart i ritytan som den aktuella symbolen skall ritas ut, vilket indikeras med anrop 2 i ovanstående sekvensdiagram. Det objekt som utgör ritytan lägger sedan till symbolen i en lista (anrop 3).

Efter detta kommer ritytan att uppdateras för att visa den nya symbolen, vilket indikeras med anrop 4.

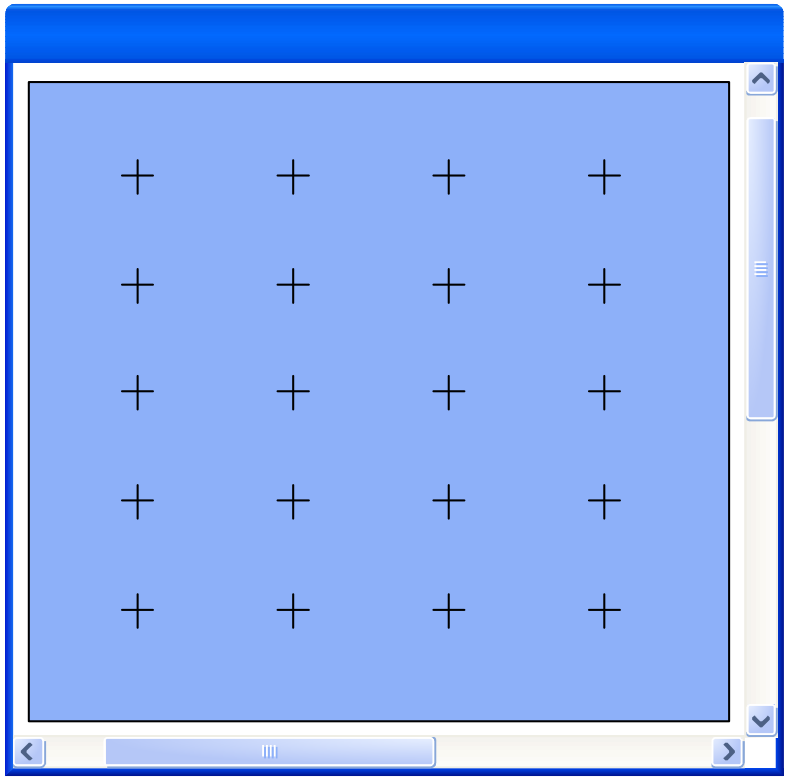
4.2.2 Användargränssnitt

Nedanstående bild visar hur verktygslådan i vår applikation skall se ut. Den första delen visar vilka symboler som är möjliga att rita ut, knapp S_1 till knapp S_n. Nästa del innehåller de olika linjer som användas för att sammanbinda symbolerna. Den sista delen innehåller funktioner för att kunna flytta och ta bort symboler. I denna del återfinns även de knappar som behövs för att rotera ritytan, skriva ut ritytan samt avsluta applikationen.



Figur 4: Programmets verktygslåda.

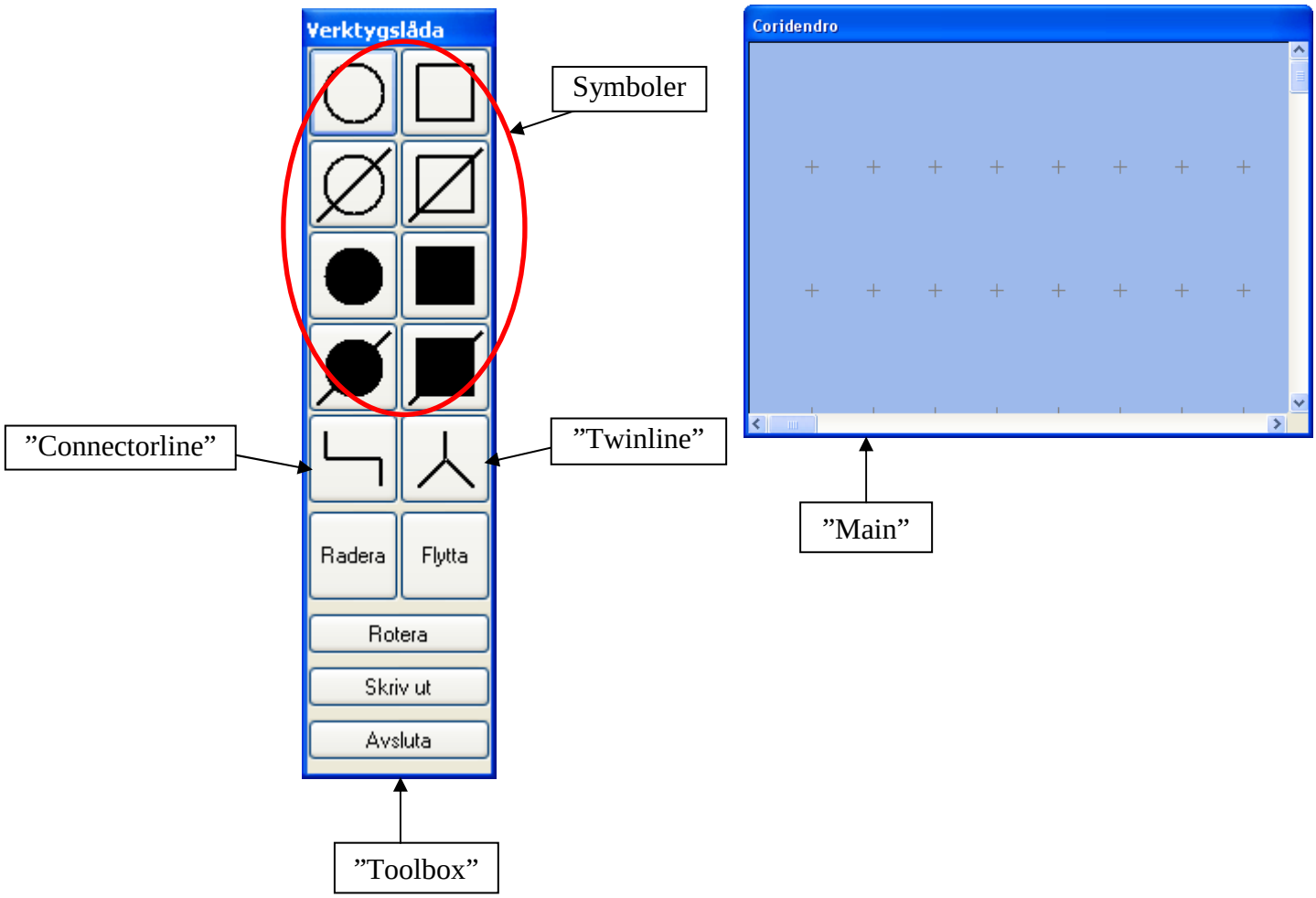
Nedanstående bild beskriver den rityta som visas för användaren. I denna rityta kan användaren lägga till olika figurer. I ritytan nedan ser vi olika fästpunkter. Symbolerna kan endast placeras ut på dessa fästpunkter, vilket försäkrar att symboler hamnar rakt i såväl horisontell som vertikal riktning.



Figur 5: Programmets rityta

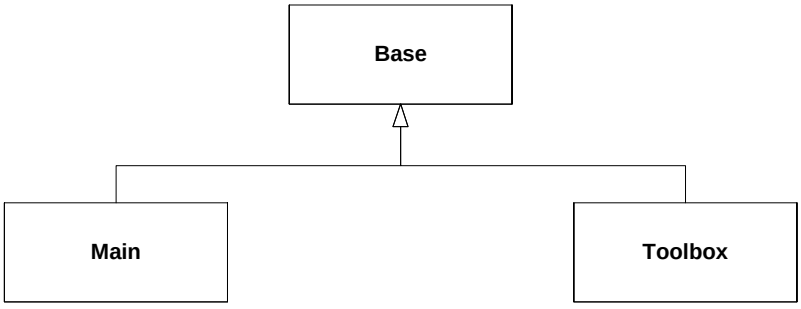
4.2.3 Klasshierarki

För att underlätta utvecklandet av applikationen har vi använt oss av en objektorienterad design. Programmet kan delas in i olika klasser. I nedanstående figur ser vi hur programmets komponenter är kopplade till olika klasser. Verktygslådan är en instans av klassen ”Toolbox”, vars huvudsakliga uppgift är att indikera vilken symbol som användaren har valt. Ritytan är en instans av klassen ”Main” och innehåller logik för att utföra olika aktiviteter på ritytan. Vilka aktiviteter som ritytan utför, beror på vilken knapp som har tryckts in i verktygslådan i kombination med eventuella markeringar i ritytan. Symbolerna som visas i figuren nedan, har en gemensam klass som heter ”Symbol”. ”Symbol” innehåller sedan underklasser för att hantera de olika varianterna av symboler som visas i figuren nedan (fylld rektangel, cirkel, cirkel med streck etc.). Klassen ”ConnectorLine” som visas i figuren nedan, innehåller logik för att rita upp en linje mellan två symboler. Klassen ”Twinline” innehåller den logik som är nödvändig för att beskriva släktskap mellan tvillingar, se avsnitt 2.4.



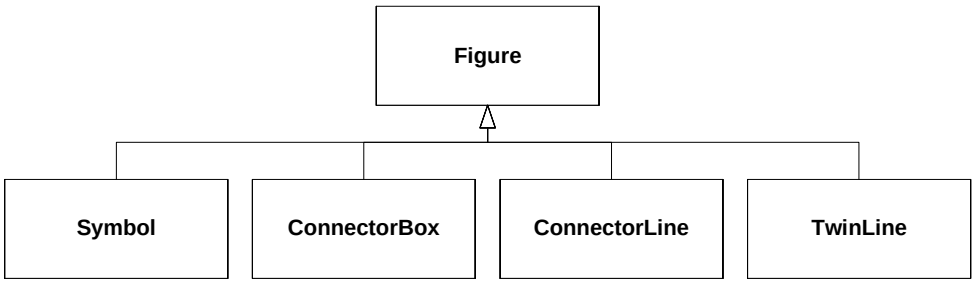
Figur 6: Beskriver kopplingen mellan klasshierarki och användargränssnitt.

De båda klasserna ”Main” och ”Toolbox” ärver av basklassen ”base”, vilken utökar den fördefinierade klassen ”Form”. Verktygslådan är en instans av klassen ”Toolbox”. Vi valde att skapa en gemensam basklass för ”Main” och ”Toolbox”, eftersom vi behövde en klass för gemensamma typer (enumeratorer). En översikt på den ovan beskrivna klasshierarkin, visas nedan:



Figur 7: Formulärens klasshierarki

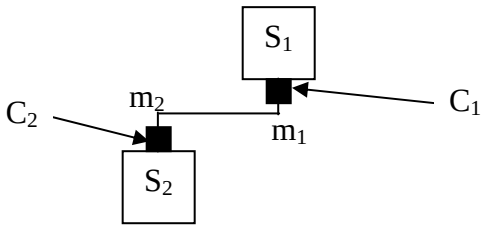
I klassdiagrammet nedan visas hur Figurhierarkin hänger samman.



Figur 8: Figurernas klasshierarki

”Figure”-klassen representerar allt som kan ritas ut på ritytan. En ”Symbol” innehåller två stycken ”Conectorbox”:ar. En ”Connectorbox” representerar en fästpunkt på en ”symbol”. När en användare vill rita ut en ”Symbol” och markerar att han/hon vill rita ut den, kommer programmet att visa de ”Connectorbox”:ar som hör till respektive symbol. ”Symbol” består i sin tur av ett antal subklasser, vilka beskrivs närmare i kommande kapitel. För att kunna binda samman de olika symbolerna, används ”ConnectorLine” och ”TwinLine”.

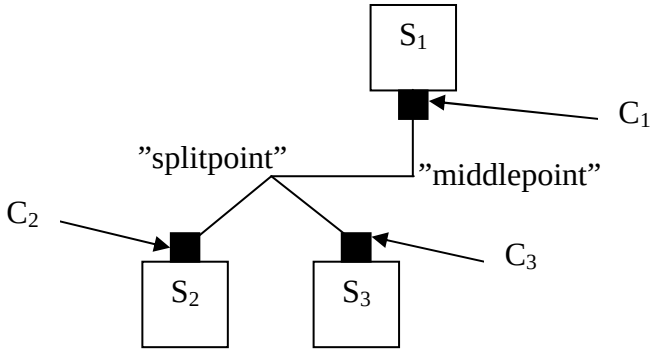
”ConnectorLine” är en linje mellan två symboler. Denna linje ritas ut mellan de olika symbolerna i räta vinklar. Linjen delas in i tre olika delar enligt nedanstående skiss.



Figur 9: Visuell beskrivning av ”ConnectorLine”

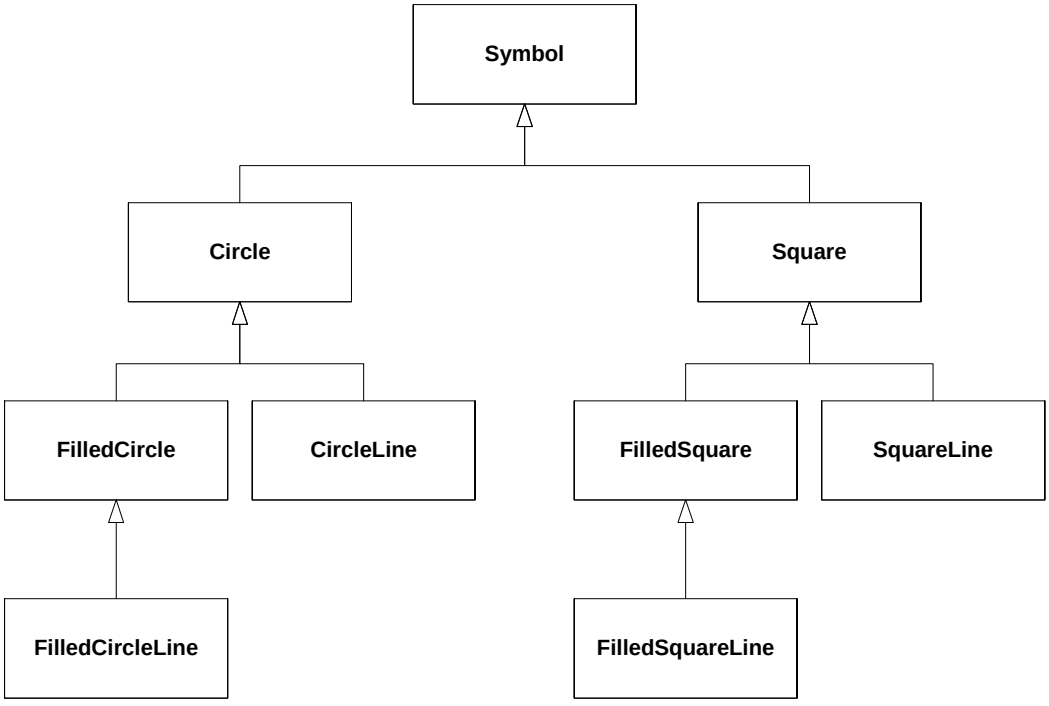
Den första delen av linjen börjar ifrån ”ConnectorBox” C_1 som finns vid symbolen S_1 och linjen slutar vid punkten m_1. Sedan ritas en linje upp mellan punkten m_1 och m_2. Efter detta ritas en linje up mellan punkten m_2 till ”Connectorbox” C_2.

”TwinLine” är en linje som ritas mellan tre figurer, enligt nedanstående figur. En ”TwinLine” består av fyra olika linjer. Den första linjen ritas upp ifrån ”Connectorbox” C_1 till ”middlepoint”. Den andra linjen ritas sedan upp mellan ”middlepoint” och ”splitpoint” och den tredje linjen ritas upp mellan ”splitpoint” och ”Connectorbox” C_2. Den fjärde och sista linjen ritas upp mellan ”splitpoint” och ”Connectorbox” C_3.



Figur 10: Visuell beskrivning av ”TwinLine”

”Symbol”-klassen utgör en basklass för alla de symboler som är möjliga att rita ut. I vårt program skall det vara möjligt att rita ut följande typer av symboler: rektangel, fylld rektangel, fylld rektangel med streck över, ihålig rektangel med streck över, cirkel, fylld cirkel, fylld cirkel med streck över, ihålig cirkel med streck över. Eftersom det är såpass många figurer och figurerna till stor del liknar varandra, är det nödvändigt att strukturera dem på ett bra sätt. Varje variant av de olika figurerna representeras som en klass. Klasshierarkin visas i nedanstående bild.



Figur 11: Symbolernas klasshierarki

Eftersom samtliga symboler består av antingen en cirkel eller en fyrkant, var det första steget i klassindelningen att skapa en klass ”Circle” samt en klass ”Square”. I figuren ovan ser vi att ”Circle” är en underklass till ”Symbol”, precis som ”Square”. Förutom detta kunde vi sedan identifiera att det behövdes fyllda cirklar, liksom fyllda fyrkanter. Då valde vi att först lägga

till en underklass ”FilledCircle” till klassen ”Circle”. Vi utökade sedan klassen ”Square” på samma sätt med en ”FilledSquare”. Vid det här laget har fyra stycken symboler identifierats, sedan valde vi att komplettera alla dessa med en underklass för symboler med ett streck över. Dessa klasser är ”FilledCircleLine”, ”CircleLine”, ”FilledSquareLine” samt ”SquareLine”.

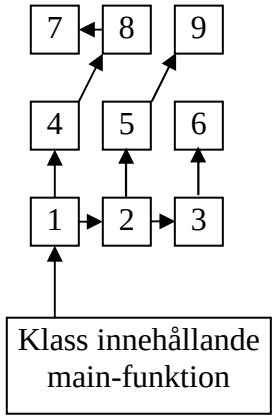
5 Programspråket C#

I det här kapitlet beskriver vi programspråket C#. Vi fokuserar mestadels på dess objektorienterade struktur. Förutom detta beskriver vi ”nyheter” som inte finns i exempelvis Java och C++. Exempel på dessa nyheter är ”properties”. Dessutom beskriver vi varför vi valde att skapa programmet i C#.

5.1 Översiktlig beskrivning

C# är ett objektorienterat programspråk som påminner om Java och C++, men bjuder på en del nya funktioner som exempelvis ”delegates” och ”properties” som kommer att beskrivas närmare i senare kapitel.

C# är mer renodlat ur ett objektorienterat perspektiv än C++. Det är exempelvis inte tillåtet att deklarera variabler och funktioner utanför en klass. Samtliga program skall bestå av minst en klass, varav en klass som innehåller en statisk main-funktion, vilket är samma princip som i Java. Det är inte helt ovanligt att ”main”-funktionen skapar en instans av en annan klass. Denna instans kan sedan skapa andra instanser. Detta förklaras närmare i nedanstående bild.



Figur 12: En applikations anropskedja

Ovanstående fyrkanter motsvarar klasser. Programmet utgår ifrån ”main”-klassen, i det här fallet numrerad till 1. ”Main-klassen” (den klass som innehåller main-funktionen) skapar instanser av andra klasser, vilket indikeras med pilar.

Det finns dock vissa undantag ifrån det objektorienterade paradigmet. Ett exempel på detta är möjligheten att skapa statiska klasser och funktioner (funktioner som inte behöver något objekt att operera på).

5.2 Datatyper

5.2.1 Definierade av Programspråket

5.2.1.1 Vektor: ett exempel på en statisk struktur i objektorienterad form

Den klassiska strukturen ”array” som återfinns i alla moderna programspråk [15], ser lite annorlunda ut i C#. På följande sätt deklareras en vektor i C++ `int x[10];` [15]. Detta skapar en statisk vektor som innehåller 10 heltal [15]. Denna möjlighet finns inte i C# och java, istället får programmerare skriva följande: `int [x] = new int [10];` [15]. Programkodsuttrycket skapar en vektor av tio element som kommer att allokeras på ”heapen”. Vektorer i C#, kan endast allokeras på ”heapen”. Vektorn i sig är ett objekt på samma sätt som i java, vilket innebär att den innehåller ett antal attribut som exempelvis storlek. Denna konstruktion underlättar för programmerare, eftersom det är omöjligt att ta reda på hur många element en heltalsvektor innehåller i C++.

5.2.2 Definierade av C#:s API

5.2.2.1 ”ArrayList”

”ArrayList” är en fördefinierad klass i C#:s API som vi använder flitigt i vår applikation. ”ArrayList” definierar en av de många samlingar som finns fördefinierade i C# [15]. Listan är dynamisk, vilket innebär att den växer i storlek, allteftersom objekt läggs till i listan. Den implementerar ett flertal olika funktioner såsom: ”add” (lägger till ett objekt sist i listan) och ”remove” (tar bort ett objekt). Dessutom finns vissa operatorer överlagrade; såsom ”`[]`” (där i står för en position i ”ArrayList”:en), vilket ytterligare bidrar till att underlätta programmering, då en ”ArrayList” i vissa fall kan användas som en vektor.

5.2.2.2 Hashtabell

En hashtabell liknar till stor del en vektor. I denna samling är varje värde associerat med en nyckel av någon typ, där hashtabellens innehåll måste vara av samma typ.

5.2.3 Egendefinierade typer

5.2.3.1 Klasser

C# består av klasser. En klass kan användas för att skapa en användardefinierad typ. Den består av variabler samt funktioner. Variablerna och funktionerna har någon av följande attribut: ”public”, ”internal” eller ”private”. En publik klassfunktion kan nås utifrån klassen, till skillnad ifrån en privat funktion [7]. En ”internal”-medlem kan nås inom arvshierarkin.

5.2.4 Arv

C# innehåller arv, vilket är naturligt för ett objektorienterat språk. C# implementerar såväl abstrakta funktioner som abstrakta klasser. En abstrakt klass kan innehålla såväl abstrakta som icke-abstrakta funktioner. Det omvända gäller däremot inte. En abstrakt funktion är en funktion som inte har någon funktionskropp. En sådan funktion deklareras med nyckelordet ”abstract” framför funktionen. I den klass där det är möjligt att specificera funktionen fullständigt, skrivs nyckelordet ”override” i funktionsdeklarationen på följande sätt: `public override void draw(Graphics g);.`

Konstruktorn fungerar som vanligt, d.v.s att en defaultkonstruktor skapas om inte användaren har definierat en själv, precis som i både java och C++. Destruktorn är däremot lite speciell. I Java finns det ingen anledning att använda sig av en destruktor, eftersom den virtuella maskinen tar hand om detta. C# ger programmeraren möjlighet till att skapa en destruktor [3], det är däremot inte självklart att destruktorn anropas när objekten tas bort. Systemet väljer själv när det är dags att anropa destruktorn [3]. Det är däremot fullt möjligt att specificera vad som skall destrueras, genom att specificera detta i destruktorn för objektet.

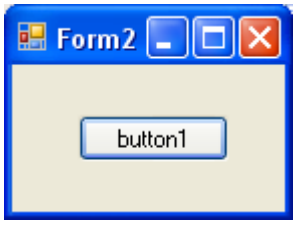
5.3 Events

Ett ”event” kan sägas bestå av två delar. Ett objekt som har gjort något och ett annat objekt som vill bli underrättat när det första objektet gjorde något [15]. När det första objektet har gjort något, kommer det att skicka ett ”event” till det andra objektet. Det andra objektet tar emot ”event”:et och utför en operation som en reaktion på detta ”event”.

Eventsystemet består av ”Event Sources” samt ”Event Handlers”. En ”event source” notifierar andra objekt om att något har hänt. En ”Event Handler” registrerar sig för vissa typer av ”events” [15]. Om en ”Event handler” har registrerat sig för ett visst ”event” och ett sådant ”event” skickas ut från en ”Event Source”, kommer denna ”Event handler” att utföra någon form av operation som reaktion på detta ”event” [15].

5.3.1 Exempel

Låt oss säga att vi skapar en knapp och när någon trycker på denna knapp, visas en meddelanderuta. Först måste vi definiera den funktion som vi anropar när någon klickar på knappen. Nedan visar vi ett exempelformulär som vi har skapat i Visual Studio 2005. Detta formulär innehåller en knapp.



Figur 13: Exempelformulär med knapp

Om användaren väljer att trycka på knappen, kommer en textruta upp med texten ”hej”. För att åstadkomma detta, skapas följande källkod.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class Form2 : Form
    {

        public Form2()
        {
            InitializeComponent();
            button1.Click += new System.EventHandler(visaTextRuta);
        }

        public void visaTextRuta(object a, EventArgs e)
        {
            MessageBox.Show("Hej");
        }
    }
}
```

När programmet startas, initieras formuläret Form2 av kod som Visual Studio 2005 själv har skapat. Denna initiering äger rum i funktionsanropet `InitializeComponent()`. Vi har valt att kalla knappen för ”button1”. Vidare kan vi konstatera att vi vill visa textrutan när användaren trycker på knappen, vid ett så kallat ”Click”-event som finns fördefinierat. Detta ger programkoden `button1.Click`. Nu måste en eventhanterare knytas till detta ”event”, vilket görs genom att använda den automatiskt överlagrade ”+=”-operatorn. Detta ger i sin tur

följande kod: `button1.Click += new System.EventHandler(visaTextRuta);`. Eventhanteraren används för att anropa en viss funktion när ett visst ”event” inträffar. I ovanstående fall är det funktionen med namnet visaTextRuta som anropas när ett ”Click”-”event” inträffar. Eventfunktioner (funktioner som anropas av eventhanteraren då ett ”event” uppstår) måste ta emot de formella parametrar som specificeras i funktionshuvudet för visaTextRuta.

5.4 ”Properties”

C# är ett av de första språken att implementera egenskapen ”Properties” [15]. ”Properties” används för att exportera informationen ifrån de privata datamedlemmarna på ett säkert sätt. På vilket sätt informationen exporteras beskrivs senare i detta delkapitel. En ”property” ser ut som en publik datamedlem och kan användas precis som en publik datamedlem. En ”property” definieras på följande sätt i C#.

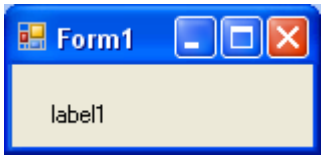
```
public class KlassNamn
{
    public typ PropertyNamn
    {
        get { return “lokal variabel att returnera”; }
        set { “namn på lokal variabel att sätta” = value; }
    }
}
```

PropertyNamn definierar namnet på ”property”:n av typen *typ*. När ”property” används för att sätta en variabel, kan den användas precis som en vanlig variabel (med exempelvis punktnotation). Detta ställer dock höga krav på variabelnamnen, eftersom det måste vara tydligt vad som avses för den som tittar på klassen ”utifrån”. Om en ”property” tillsammans med en privat datamedlem används, istället för ett publikt fält, finns det en möjlighet att skriva programkod för att testa den information som skall sparas till den privata variabeln [20].

En ”property” är mycket lätt att implementera, vilket illustreras i ovanstående kodexempel. ”value” är det värde som variabeln skall sättas till och är ett nyckelord. För att sätta en ”property” kan exempelvis följande anrop utföras (I det här fallet sätts variabeln till värdet 5, men det är inte nödvändigt med en konstant): `ObjektNamn.PropertyNamn=5;` För att referera till värdet, används följande anrop: `ObjektNamn.propertyNamn;` En ”property” kan antingen skapas för hand eller med hjälp av Visual Studio 2005.

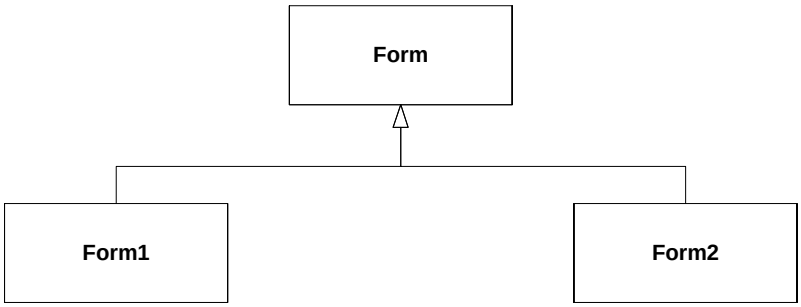
5.5 ”Delegates”

”Delegates” kan liknas vid funktionspekare. En ”delegate” kan användas för att referera till en viss typ av funktioner. En ”delegate” kan även referera till funktioner som ligger utom det egna objektet. Detta har vi utnyttjat i vår kod och nu tänker vi visa ett förenklat exempel. Antag att vi har en applikation som består av två formulär. Det första formuläret visas i Figur 13. Det andra formuläret visas här nedan.



Figur 14: Exempelformulär med ”label”

De bägge formulären är instanser av två olika klasser, vilka båda ärver direkt ifrån systemklassen form, enligt följande klassdiagram.



Figur 15: Exempelprogrammets klasshieraki

Här följer källkoden för Form1:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        private Form2 testform;

        public Form1()
        {
            testform = new Form2();
            testform.Show();
            InitializeComponent();
            testform.hejsan = new Form2.del(hejsan3);/*del1*/
            testform.hejsan += new Form2.del(hejsan2);/*del2*/
        }

        public void hejsan3(String s)
        {
            label1.Text = s;
        }

        public void hejsan2(String s)
        {
            MessageBox.Show("tusan");
        }
    }
}
```

Ovanstående kod för Form1, skapar också en instans av Form2. Denna instans av Form2 har vi valt att kallla för testform. Efter att ett objekt av Form2-klassen har skapats med new-operatorn, gör vi den synlig med anropet `testform.show()`. För att kunna förklara resterande rader är det nödvändigt att beskriva Form2-klassen först. Källkoden för Form2:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class Form2 : Form
    {
        public delegate void del(String s);
        public del hejsan;

        public Form2()
        {
            InitializeComponent();
            button1.Click += new System.EventHandler(setText);
        }

        public void setText(object a, EventArgs e)
        {
            hejsan("bla");
        }
    }
}
```

Vi börjar med att skapa en delegate med kommandot: `public delegate void del (String s);`. Detta kommando skapar en ”delegate” med namnet ”del”. Denna ”delegate” kan användas för att anropa alla funktioner som inte returnerar något och som tar en sträng som inparameter. Nu skapar vi en instans av det nyligen definierade ”delegate”:t ”del”. Detta görs med programkodsuttrycket `public del hejsan;`. Detta skapar en publik instans av delegate:t ”del” med namnet hejsan. Objektet hejsan kan nu användas för att kalla på alla funktioner som är associerade med ”delgate”:t ”del”. Ett exempel på ett sådant anrop är `hejsan("bla");` i ovanstående programkod. Alla funktioner associerade med ”delegate”:t hejsan anropas nu med den aktuella parametern ”bla”. I källkoden för Form1, associeras några medlemsfunktioner för denna klass till ”delegate”:t hejsan, i klassen Form2. Detta sker med följande programkodsrader.

```
testform.hejsan = new Form2.del(hejsan3);/*del1*/
testform.hejsan += new Form2.del(hejsan2);/*del2*/
```

Den översta programkodsraden associerar den lokala klassfunktionen hejsan3 till ”delegate”:t hejsan, vilket är definierat i klassen Form2. Nästkommande rad associerar ytterligare en funktion (hejsan2) till ”delegate”:t hejsan. För att associera ytterligare funktioner till ett ”delegate”, används den automatiskt överlagrade ”+=”-operatorn.

När användaren av den ovan beskrivna applikationen trycker på knappen ”button1”, kommer ”delegate”:t hejsan att anropa sina associerade funktioner. Detta skapar en möjlighet för ett objekt A att anropa objektfunktioner ifrån den klass B, där objektet A ursprungligen var skapat i. Det är denna mekanism som vi använder i vår exjobbsapplikation för att förflytta kontrollen mellan olika formulär.

5.6 Exception Handling

Precis som de flesta andra programspråk, implementerar C# undantag (”exceptions”). Det implementeras med hjälp av ”try”-block som vi känner igen ifrån exempelvis Java [13]. Ett ”try”-block måste efterföljas av en eller flera ”catch”-block samt avslutas med ett ”finally”-block [15]. Ett ”catch”-block kan antingen ta emot ett specifikt undantag eller ta emot vilket undantag som helst [15].

5.7 Varför vi valde C#

Vi valde C# eftersom vi ville använda ett så renodlat objektorienterat språk som möjligt. Vår uppdragsgivare ville inte installera javas virtuella maskin. Vidare valde vi att använda ”properties” som C# tillhandahöll. Vid en första anblick verkade detta vara ett bra alternativ till traditionella ”get” och ”set” – funktioner men skapade problem med förlorad abstraktion. För att kunna bibehålla abstraktionen krävs det väldigt utförliga variabelnamn. Den största fördelen blev slutligen ”delegates”. ”delegates” möjliggjorde anrop till funktioner som inte låg i klassen på ett enkelt sätt. Exempelvis från klassen ”Toolbox” till klassen ”Main”.

6 Implementation

Här beskriver vi hur valet av utvecklingsmiljö har underlättat utvecklingen samt implementationen av applikationen i huvudsak.

6.1 Utvecklingsmiljö

Valet av utvecklingsmiljö var inte helt självklart under uppstarten av projektet. Vi visste att vi behövde någon form av bibliotek för att rita ut grafik, skapa grafiska användargränssnitt och hantera samlingar av objekt. Dessutom behövde vi ett versionshanteringssystem för att dels hantera ändringar och dels ge möjligheten att gå tillbaka till en tidigare version (rollback). Förutom detta används versionshanteringssystemet för att utvecklarna ska kunna arbeta vid olika tider och platser och ändå ha tillgång till den senaste källkoden. Vi hade uppfattningen att det skulle underlätta om det i utvecklingsmiljön ingick någon form av verktyg för att rita upp programmets grafiska gränssnitt, likt Glade [18]. Eftersom vi då skulle kunna bygga upp applikationens grafiska gränssnitt med ”markera och dra” principen utan att behöva skriva i koordinater vart kontroller såsom knappar ska befinna sig.

Som versionshanteringssystem valde vi Subversion [5] eftersom det dels är gratis och dels finns tillgängligt hos Sourceforge [14]. Sourceforge låter utvecklare av öppen mjukvara använda deras tjänster (ex. Subversion [5], webbhotell och SSH -access) mot att källkoden tillhandahålls under en öppenkällkodslicens.

Efter det första mötet med Landstingets IT-enhet visade det sig att de tillhandahöll Visual Studio 2005 [17] med tillhörande licens som uppfyllde våra krav. För att kunna använda Subversion tillsammans med Visual Studio 2005 använde vi oss av AnkhSVN [2] som är ett plugin för Visual Studio 2005.

6.1.1 .NET framework

För att hantera samlingar och grafik blev det naturliga valet .NET framework [1] från Microsoft. Det fanns andra alternativ, exempelvis Mono [11], men då vi fått reda på att vi skulle ha tillgång till Visual Studio 2005 var steget inte särskilt långt till .NET framework då det följer med vid installationen av Visual Studio 2005. Visual Studio 2005 är framtagen med tanke på .Net framework och därav fungerar dessa bra tillsammans. En annan faktor som

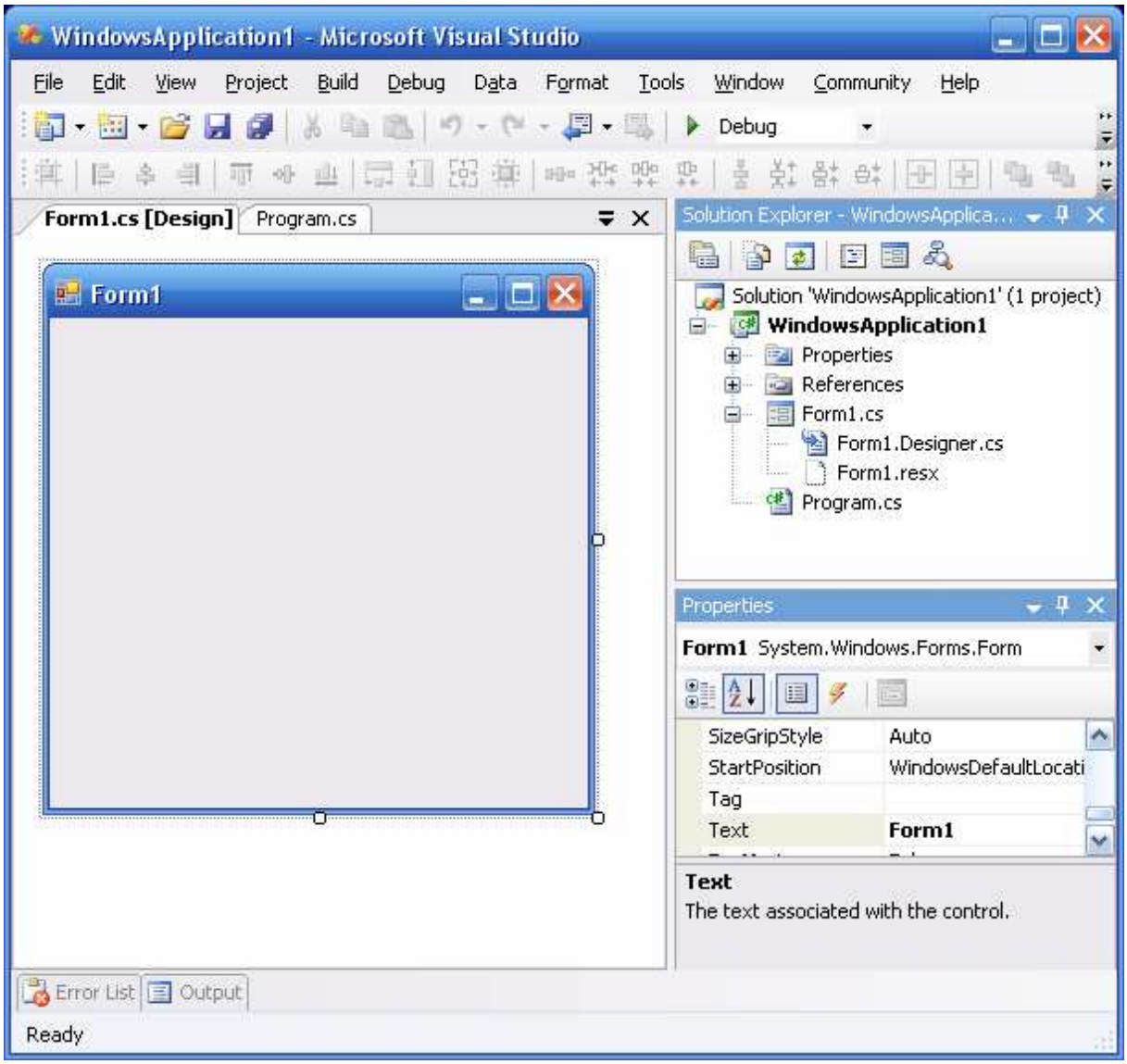
vägde tungt i valet var att Microsoft utvecklar både Visual Studio 2005 och .NET framework. För att undvika eventuella kompabilitetsproblem mellan editorn och .NET framework valde vi Visual Studio 2005 vid skapandet av det grafiska användargränssnittet.

.NET framework är en samling klassbibliotek som innehåller stöd (genom klasser) för att hantera grafik genom GDI+ [8], knappar och andra kontroller att använda vid skapandet av grafiska användargränssnitt, hantering av samlingar och mycket mer.

För den som vill använda .NET framework på en annan dator än utvecklingsdatorn, är det nödvändigt att säkerställa att .NET framework finns installerat. En fördel med .NET framework är att det är gratis att ta hem ifrån Microsofts webbplats. Det är till och med så att när man skapat installationsprogrammet för sin applikation i Visual Studio 2005 kan det själv undersöka om förvillkoren är tillfredsställda och annars fråga om användaren vill ladda hem och installera exempelvis .NET framework ifrån Microsofts webbplats.

6.1.2 Visual Studio 2005

Visual Studio 2005 [17] är en integrerad utvecklingsmiljö (IDE) med många användbara funktioner. Förutom de vanliga funktionerna, till exempel nyckelordsmarkering (syntax highlighting) och kodkomplettering så finns det även en klassdesigner som genererar kod utifrån klassdiagrammet man ritar. En av de viktigaste funktionerna är den som används vid skapandet av användargränssnitt. Detta är en enkel och smidig funktion som låter utvecklaren lägga till nya ”events”, genom ett par knapptryckningar, samtidigt som den erbjuder en bra överblick av kontrollens egenskaper.



Figur 16: Visual Studio 2005

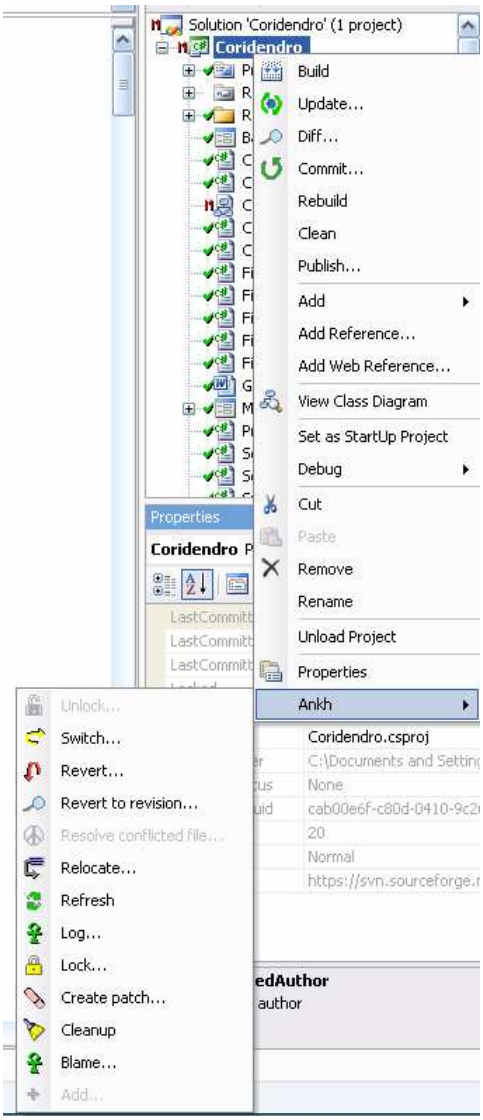
En nackdel är att mycket kod förmyndas av Visual Studio 2005 självt. För att exemplifiera ovanstående fenomen kan vi betrakta skapandet av formulär. Den förmyndade koden skrivs nämligen lätt över av Visual Studio 2005 när man genomför förändringar i användargränssnittet. Däremot kan man i språket C# utöka den förmyndade koden (klassen) genom att använda ”extend” och skriva den egna koden i en annan fil. Detta används automatiskt.

6.1.3 MSDN library

För att kunna använda .NET framework fullt ut har vi dragit nytta av den dokumentation som finns samlad i MSDN (Microsoft developer network) library. Det lättaste sättet att komma åt MSDN library [10] är via Internet.

6.1.4 AnkhSVN

I Visual Studio 2005 finns det som standard inte något stöd för versionshanteringssystemet Subversion. Vi löste det genom att använda oss av AnkhSVN [2] som är ett tredjepartsplugin för just Visual Studio 2005. Det finns alternativ (exempelvis Turtoisesvn [16]) som integreras i Windows Explorer (vid högerklick på en fil i utforskaren kan man välja att utföra Subversion kommandon). Vi ansåg att det skulle vara smidigare att ha kontroll över versionshanteringen i Visual Studio 2005, på samma arbetsyta som användes för utveckling, eftersom vi då slipper hoppa fram och tillbaka mellan olika fönster och har en översiktsbild över vilka filer som ändrats och behöver uppdateras i den pågående versionen. Här nedan finns en bild av hur AnkhSVN kan se ut i kombination med Visual Studio 2005.



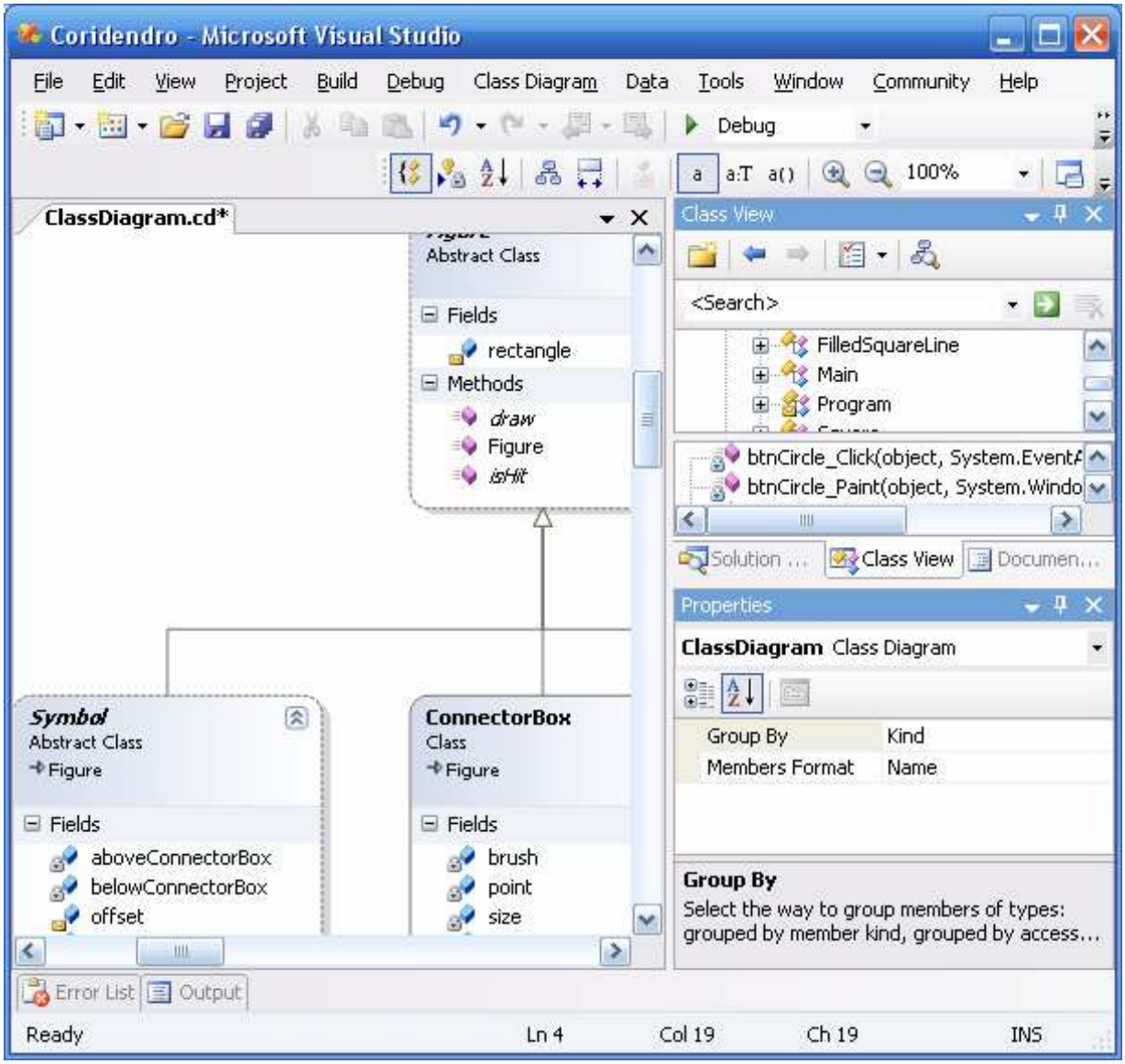
Figur 17: AnkhSVN (Subversionstöd för Visual Studio 2005)

Ett möjligt bekymmer med AnkhSVN kan vara då två utvecklare tar hem samma version (”checkout”) och genomför ändringar i samma fil under samma tidsperiod och sedan väljer att

skicka upp filerna på nytt (”commit”). Då får man se vilka ändringar som gjorts (”diff”) genom att Subversion lägger in kommentarer i filerna. Det finns inte något stöd för att snabbt och enkelt välja vilka ändringar som är aktuella/inaktuella i Visual Studio 2005, man får helt enkelt redigera filen för hand vilket kan vara kan vara tidsödande. När Subversion inför sådana markeringar i klassdiagramfiler uppstår ett annat problem, det finns nämligen inte något sätt att i Visual Studio 2005 eller AnkhSVN redigera den text som diagramfilen (en fil i XML format) består av. Detta leder till att filen uppfattas som ogiltig av Visual Studio 2005. Lösningen är att generera en ny diagramfil utifrån den källkod som finns i projektet.

6.1.5 Klassdiagram

Med klassdiagramfunktionen får man lätt översikt över vilka klasser som finns inom ett projekt samt arvshierarkin och klassernas medlemmar.



Figur 18: Klassdiagramfunktionen i Visual Studio 2005

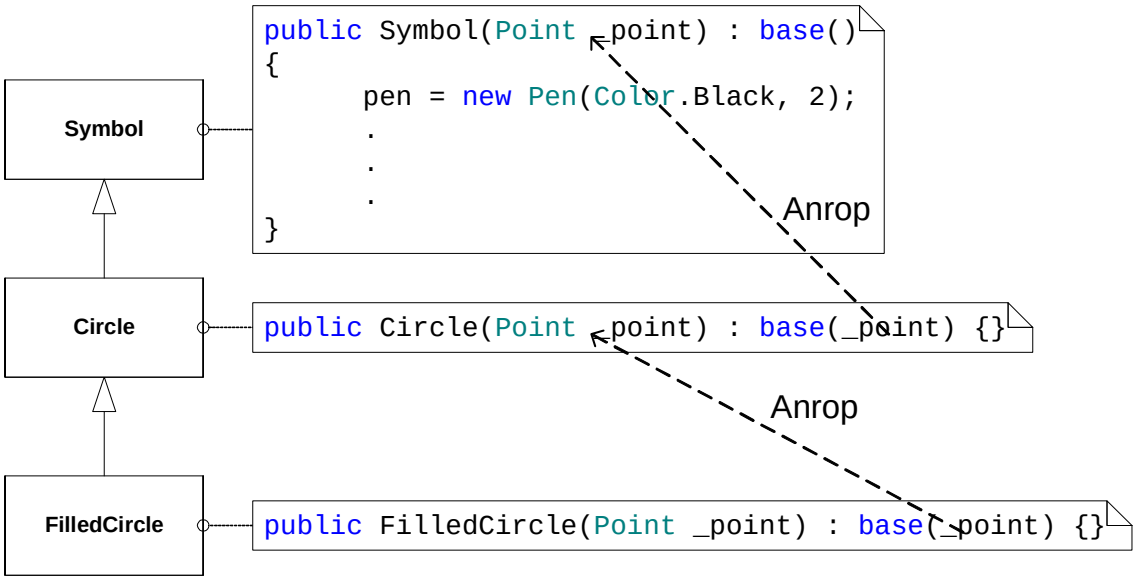
I figuren ovan kan man se en bråkdel av de funktioner som ingår. Vi har kunnat skapa det mesta av kodinkapslingen i klassdiagramsläget och sedan implementerat logiken i kodläget. Klassdiagrammet ändras allteftersom koden ändras och vise versa. Under utvecklingsarbetet har vi även använt klassdiagramsläget för att få en översikt över programmet och få en uppfattning om vart vi är på väg, likt en karta.

6.2 Klasser

6.2.1 ”Base” konstruktionen

För att möjliggöra den inkapsling och struktur av objekt som vi använt i designen har vi använt klasskonceptet i C#, se avsnitt 5.2.2.1. Vi har inte gjort objekt av allting, istället har vi försökt hålla det på en överskådlig nivå där de klasser som underlättat för översikten av strukturen implementerats.

I de underklasser där vi vill anropa konstruktorn i den ärvda klassen för att exempelvis instantiera objekt har vi använt oss av ”base”-konstruktionen. Den gör att den ärvda konstruktron anropas innan den aktuella, på så vis kan man traversera uppåt i klasshierarkin ända upp till toppen. Ett exempel på hur vi använder denna följer här nedan:



Figur 19: Exempel på användning av ": base()" konstruktionen

När vi skapar ett ”FilledCircle”-objekt med parametern ”Point”; skickas den parametern genom ”Circle”-konstruktorn och upp till ”Symbol” där den används för att bestämma vart symbolen befinner sig på ritytan i x och y led. På så vis har vi kunnat generalisera och

undvikit duplicering av kod med hjälp av abstraktion. Eftersom alla ritbara symboler har en position på ritytan kan vi använda en gemensam konstruktor för dessa.

6.2.2 Abstrakta medlemmar

Eftersom alla underklasser till ”Figure”-klassen ska implementera ”draw”-funktionen har vi valt att gör denna abstrakt i basklassen. Detta resulterar i att alla underklasser måste implementera en egen version av denna funktion. Konceptet att alla klasser implementerar en egen version på detta sätt kallas polymorfism. Polymorfism förenklar generell iterering av en samling, vilket visas här nedan:

```
foreach (Figure fig in Figures)
    fig.draw(Graphics);
```

Vi tvingar utvecklaren att tänka på att implementera en ”draw” funktion genom att placera ett abstrakt funktionshuvud i ”Figure” klassen med följande kod: `public abstract void draw(Graphics g);`. När utvecklaren sedan väljer att implementera en subklass till ”Figure”, exempelvis ”Triangle” kan han inte missa att den funktionen måste vara med eftersom det uppstår ett kompileringsfel utan den. Då vi inte vill att det ska finnas rena ”Figure”-objekt (endast de objekt som ärver från ”Figure” ska gå att instantiera) passade det utmärkt att göra ”Figure” abstrakt. Resultatet blir en klass som inte går att instantiera, det går endast att ärva från den. Med hjälp av den abstrakta klassen framtvingar vi ett polymorft beteende, som visades i iterationsexemplet ovan.

6.2.3 Utökade klasser

I programmet har vi använt oss av nyckelordet ”partial” för att utöka en befintlig klass med mer innehåll. Visual Studio, se kapitel 6.1, hanterar den kod som hör till användargränssnittet men den underliggande logiken måste vi skapa själva, dörren in är via ”partial” som låter oss utöka den klass som Visual Studio skapat med egen kod:

```
public partial class Main : Base
{
    .
    .
    .
}
```

Med ”partial” får vi också möjligheten att lagra kod i olika filer, vi har exempelvis valt att separera koden som hanterar användargränssnittet från den underliggande logiken genom att lagra logiken utanför i separata filer. Till logik räknas den funktionalitet som inte syns och till användargränssnitt räknas resten av funktionaliteten.

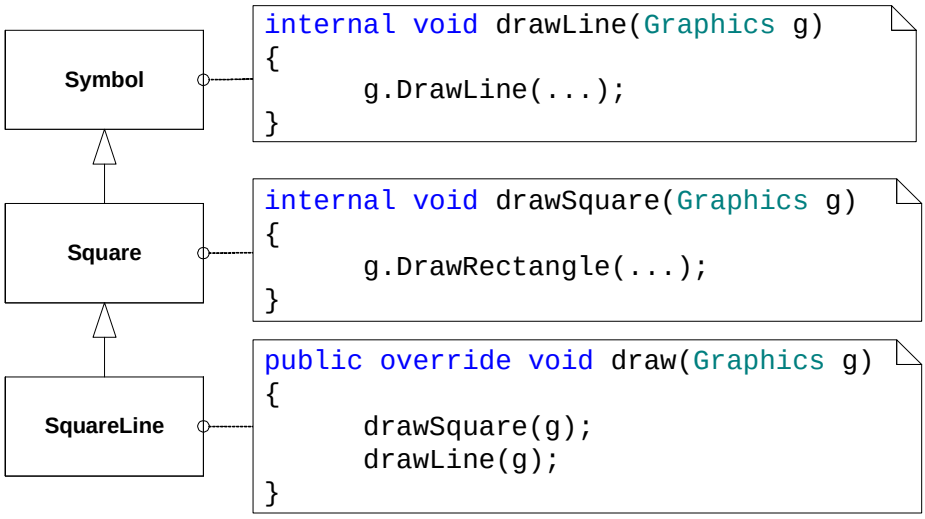
6.2.4 ”Draw” funktionerna

De klasser som ärver från ”Figure” måste enligt tidigare implementera en funktion ”draw” som tar emot ett ”Graphics”-objekt. Eftersom vi valt att gruppera objekten efter utseende kan vi återanvända det mesta av koden som används för att rita. Låt säga att utvecklaren vill implementera en funktion för att rita ut en kvadrat med en parameter, sida: `public void kvadrat(int sida)`.

Han har redan en funktion för att rita ut en rektangel som tar bredd och höjd som parametrar: `public void rektangel(int bredd, int höjd)`. Då skulle han kunna använda sig av funktionen för att rita kvadraten på följande sätt:

```
public void kvadrat(int sida)
{
    rectangel(sida, sida);
}
```

På liknande sätt har vi återanvänt kod i ”draw”-funktionen i ”Figure”-objekten, i de symboler (objekt som ärver från ”Symbol” klassen) som ser ut som en ofylld cirkel anropas funktionen ”drawCircle” som tillhandahålls genom arv från klassen ”Circle”. Här följer ett exempel för att illustrera detta:



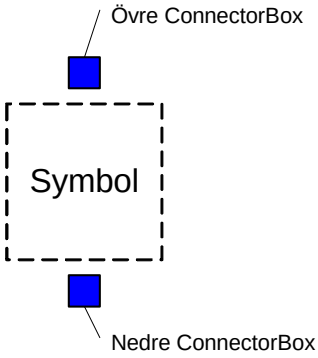
Figur 20: Exempel på återanvändning av kod inom draw funktionerna

Det är ”internal” som gör det möjligt för oss att använda en funktion såsom ”drawLine” i en ärvande klass utan att göra den publik. Vi har inte någon anledning att göra funktionen publik eftersom det **inte** ska gå att omvandla en symbol som inte har en linje till en som har en linje utan att skapa en ny instans av symbolen. Om vi gjort den publik hade det varit möjligt att skapa ett `Square` objekt **med** linje (inte ”SquareLine”) vilket skulle göra det lätt att blanda ihop olika typer av objekt. Endast ”draw”-funktionen är publik i respektive klass.

6.2.5 ”ConnectorBox”

När användaren vill rita ut en linje har vi valt att visa punkter på symbolerna där linjen kan fästa, en sådan punkt har vi valt att kalla ”ConnectorBox”. Eftersom det inte alltid är önskvärt att visa dessa skickas ett ”delegate”-anrop från verktygslådan när användaren väljer att rita en linje och sedan döljs dessa efter att linjen ritats.

Alla symboler har två ”ConnectorBox”-objekt associerade med sig, en ovanför och en undertill:



Figur 21: Hur ConnectorBox:arna är placerade i förhållande till symbolen

Eftersom ”ConnectorBox”-objekten är medlemmar i ”Symbol”-objekten är det lätt att komma åt dessa utifrån ett Symbol objekt. Om vi exempelvis vill att anslutningspunkterna ska vara synliga för en viss symbol använder vi dessa två rader kod:

```
symbol.AboveConnectorBox.Visible = true;
symbol.BelowConnectorBox.Visible = true;
```

Därefter behöver ritytan uppdateras för att ändringarna ska få effekt. För att visa vilka anslutningspunkter som användaren har valt (och inte kan välja igen) markerar vi den med gul färg. Den som håller på att bli vald markeras med röd färg. När linjen ritats ut nollställs färgerna och punkterna kan väljas igen för att rita ut en ny linje, med samma punkter om så önskas.

6.3 Ritfunktioner

När användaren valt vilken typ av symbol som han vill rita ut så markerar han vart på ritytan som han vill ha symbolen genom att trycka på vänster musknapp. Det som då inträffar är att en ny instans av den typen av objekt som valts skapas och att den läggs till i listan av symboler, som också innehåller de symboler som tidigare finns utritade. Inga förändringar

syns på skärmen förrän den delen av ritytan som innehåller symbolen uppdateras, ritas om, och den nya symbolen ritas ut.

6.3.1 GDI+

För att rita grafiken i programmet har vi använt oss av GDI+-biblioteket [8]. GDI+ låter programmeraren skapa kod som ritar direkt på kontroller (ex. formulär, paneler och knappar). Det finns vissa fördefinierade funktioner, exempelvis för att rita en linje eller en ofylld rektangel. Sedan bygger man med hjälp av dessa basfunktioner ihop den figur man vill ha ritad. Här nedan följer ett exempel på hur man med hjälp av GDI+ kan rita ut en linje: `e.Graphics.DrawLine(new Pen(Brushes.Blue), 10, 10, 50, 50);`. ”e” är ett ”event” som kastats av en kontroll där linjen ska ritas, ”Graphics” är grafikmedlemmen som tillhör den kontroll vi vill rita på och till sist ”DrawLine” är en medlemsfunktion hos grafikobjektet som skapar linjen på grafikmedlemmen. Det finns fyra olika medlemsfunktioner hos ”Graphics”-klassen som skapar linjer men som tar olika inparametrar (”overloading”). Den mest tilltalande är kanske från början den som vi visar exempel på här ovan, med x-koordinater och y-koordinater som ”integers”. Efter en tids utvecklande kan det mycket väl vara så att man kommer på att det skulle vara smidigare att kapsla in koordinaten i en ”Point”, en klass som representerar en punkt i två dimensioner och innehåller både x och y-koordinaten. Det finns med andra ord rum för valbarhet inom GDI+. I exemplet ovan finns fem parametrar, dessa är: `new Pen(Brushes.Blue)` vilket är den penna som ska användas för att rita linjen och fyra nummer som talar om vart linjen ska ritas (X_1, Y_1, X_2, Y_2); start och slutpunkt.

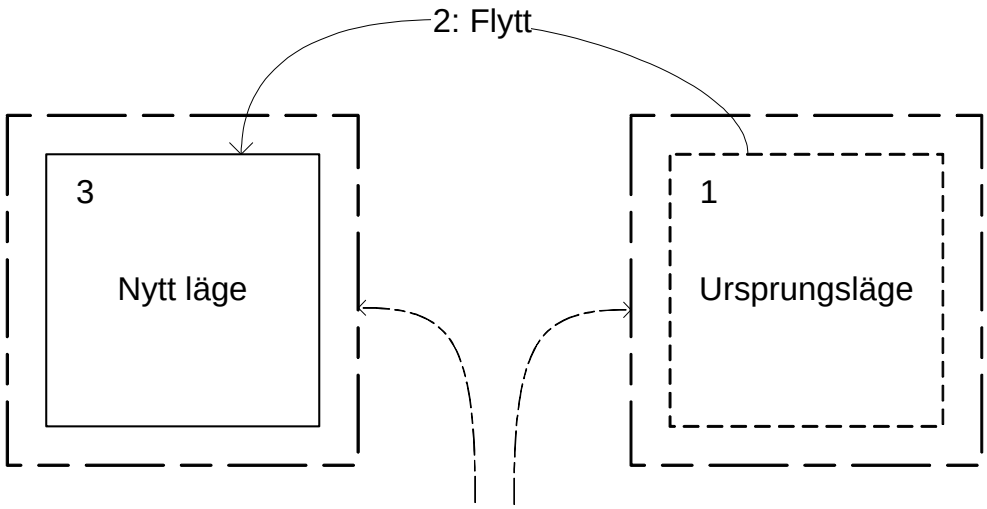
När man använder sig av GDI+ för att rita symboler behöver man ett ”Graphics”-objekt att använda som ritbord. ”Graphics”-objektet fungerar som en målarduk för de funktioner som finns inom GDI+-biblioteket.

6.3.2 Symboler och linjer

Efter att användaren klickat ut en symbol uppdateras den ytan på ritytan där symbolen finns så att symbolen blir synlig. Då användaren väljer att ta bort en viss symbol så uppdateras den ytan där symbolen tidigare fanns. Innan ytan där symbolen finns eller tidigare fanns uppdateras syns inte ändringarna. Man måste anropa ”draw”-funktionen hos klassen ”symbol” för att rita upp den och sedan tala om för ritytan vilken del som ändrats för att uppdatera den och visa ändringarna. Det finns en möjlighet att uppdatera hela ritytan på en gång men det går långsamt i jämförelse med att endast uppdatera den del som förändrats.

En annan del i optimeringen som vi gjort är att bara rita om något då en förändring skett. Ett exempel på detta är när användaren väljer att flytta en symbol, tar tag i symbolen och flyttar muspekaren men inte nog långt för att symbolen ska flyttas till nästa fästpunkt. Då ritas inte området som hör till symbolen om. Det här resulterar i att användaren slipper se det flimmer som annars uppstår då muspekaren flyttas hundratals punkter i sekunden och således leder till hundratals uppdateringar av ritytan där symbolen befinner sig.

För att veta vilken del av ritytan som ändrats så sparar vi vart och hur stort området är där symbolen finns i form av ett ”Rectangle”-objekt samtidigt som symbolen skapas. När användaren sedan väljer att flytta symbolen på ritytan uppdateras den här informationen. I figuren här nedanför visar vi hur det här fungerar.



Figur 22: Beskriver uppdateringen av det gamla och nya området

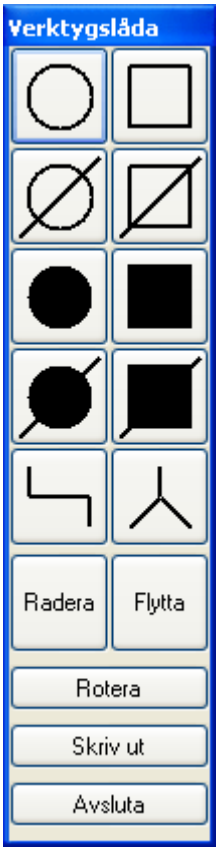
Då användaren väljer att skriva ut sitt träd används ett ”delegate”-anrop (eftersom anropet sker uppåt i hierarkin) från ”Toolbox”-objektet för att be ritytan skapa en avbild av hur ritytan ser ut (förutom bindningspunkter och eventuella bindningsboxar), därefter visas bilden i ett förhandsgranskningsfönster för att låta användaren få en god uppfattning om hur resultatet kommer att bli efter utskrift. Om användaren väljer att fortsätta skrivs trädet ut på den skrivare som är satt som standardskrivare i Windows.

Vid ritande av linjer får användaren välja vilka symboler linjen ska ritas mellan samt om linjen ska ansluta uppe eller nere på symbolerna. Därefter skapas linjeobjektet och sparas i en lista på samma sätt som symbolerna ovan.

6.3.3 Bilder på knappar

Bilderna på knapparna ritar vi på samma sätt som symbolerna på ritytan förutom suddgummit som är en ”bitmap”-bild och linjeknappen som ritas direkt med GDI+. Eftersom vi valt att använda symboler som ikoner på de flesta knapparna behöver vi inte skicka med extra bilder med programmet som annars skulle ta onödigt stor plats. Dessutom kan användaren nu vara försäkrad om att samma bild som finns på knappen ritas på ritytan.

Om utvecklaren skulle vilja ändra utseendet på en symbol behöver han inte bry sig om att ändra ikonen eftersom den ändras samtidigt.



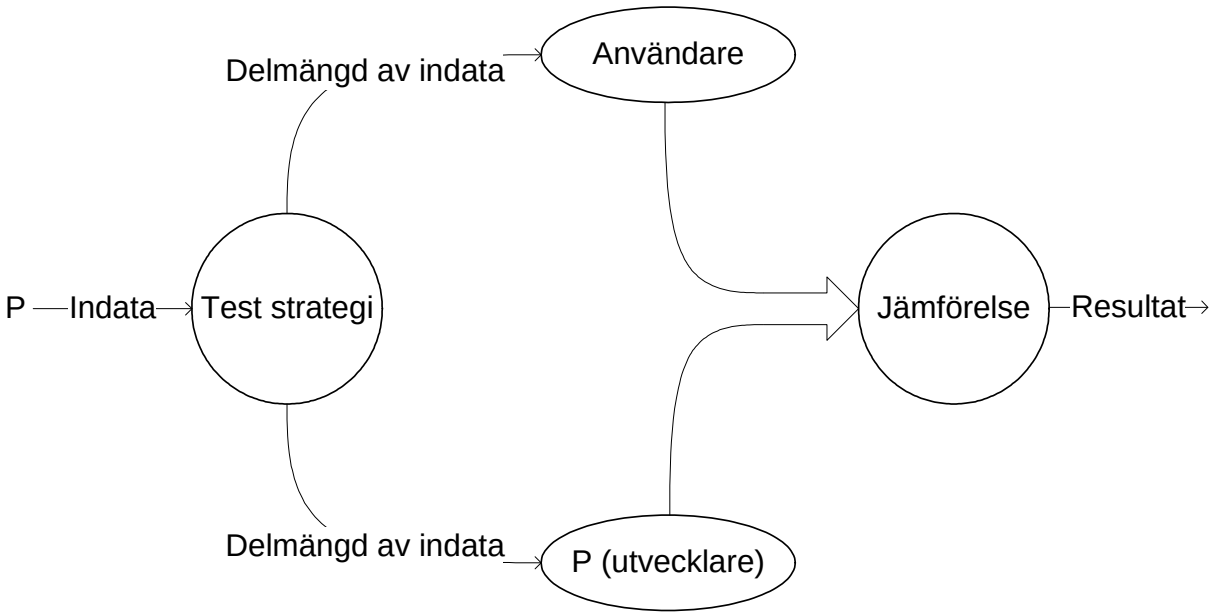
Figur 23: De olika knapparna

Här ovanför finns en bild av hur de färdiga symbolerna ser ut när de ritas ut på knapparna i applikationen.

7 Test

I detta kapitel beskrivs hur vi använt användartest och labbtest för att verifiera och förbättra applikationen mot de framtida användarna. Här finns också resultaten av det förenklade användbarhetstest som vi har genomfört.

7.1 Användbarhetstest



Figur 24: Konceptuell bild av användbarhetstest

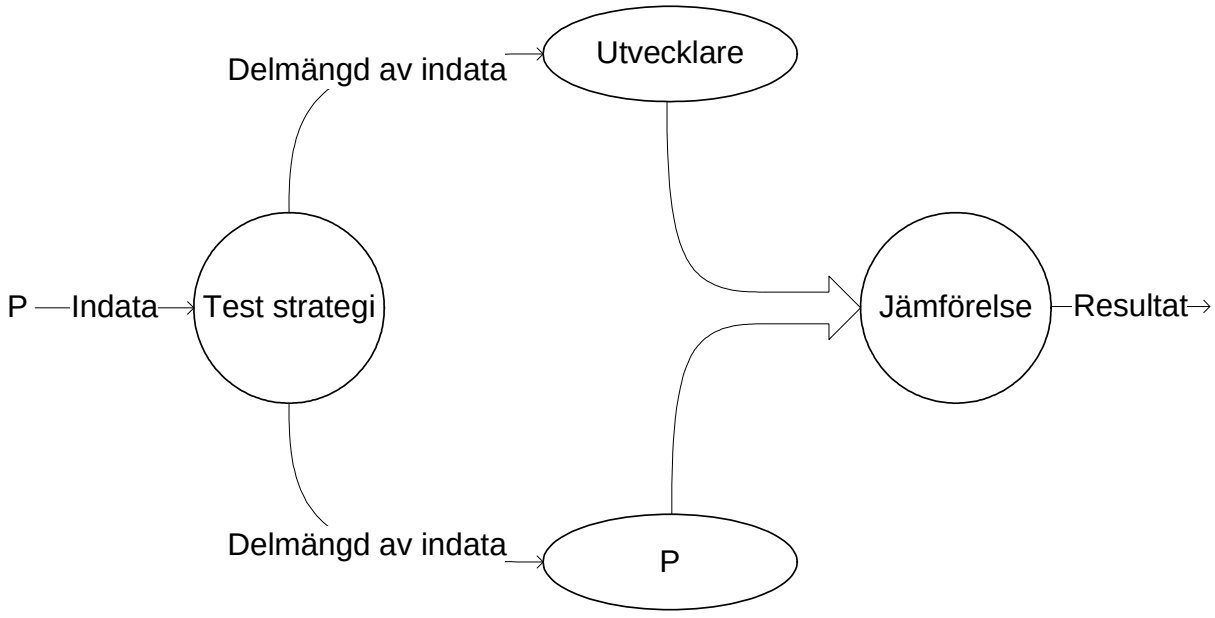
I bilden ovan visas det generella flödet vid ett användbarhetstest. Strukturen till användbarhetstestet lånade vi från boken ”Användbarhet i praktiken” [12] och instruktionerna finns bifogade i appendix A. Ett användbarhetstest hör till kategorin "black-box" där de inre delarna är dolda för den som konstruerar testfallen. Idén med ett användbarhetstest är främst att se hur väl applikationens grafiska gränssnitt tas emot av den framtida användaren och att i största möjliga mån avhjälpa felaktigheter och missförstånd mellan utvecklare och den tidigare fastslagna specifikationen. Resultatet är en lista av funktioner som kunden kanske inte specificerat ifrån början eller som behöver förändras för att möta kundens nu förfinade krav.

Om kundens specifikation var fulländad skulle inte användbarhetstest behövas så länge som utvecklarna följt den fastslagna specifikationen. Då skulle projektets utfall bero helt och

hållet av utvecklarnas förmåga. Det här är vad som kallas vattenfallsmodellen, specifikationen skrivs i ett tidigt skede och följs sedan till projektets slut när utvecklarna lämnar över den färdiga produkten och kunden blir inte sällan besviken på resultatet. Vi har dessutom kontinuerligt diskuterat applikationens utformande med kunden.

Vi hann genomföra ett användbarhetstest under implementationsfasen och detta resulterade i en helt ny funktion och två förändringar. Den nya funktionen var roteringsfunktionen som låter användaren välja mellan stående och liggande då diagrammet ritas upp och senare eventuellt skrivs ut. På kundens begäran har vi ändrat utskriftsfunktionen så att den skriver ut "Av:" och "Familjenummer:". Dessutom uppdaterade vi funktionen där användaren väljer verktyg i verktygslådan, tidigare ”tappade” användaren verktyget efter att det använts men efter ändringen kan användaren välja att fortsätta att använda samma verktyg utan att behöva välja samma verktyg upprepade gånger. I övrigt var kunden och användarna nöjda med produkten och såg fram emot att få komma att använda den i produktionsmiljön.

7.2 Labbtest



Figur 25: Konceptuell bild av labbtest

I bilden ovan visas det generella flödet vid ett labbtest. Den form av labbtest som vi använt faller under kategorin "white-box", alltså att de inre delarna av applikationen är kända och där man testar kritiska gränser; exempelvis första och sista elementen i en indexerad array. Vi har inte tillämpat "unit-testing" i dess fulla bemärkelse. Allt eftersom vi implementerat funktioner har dessa testats programmatiskt och med hjälp av användargränssnittet. När testen inte gett

tillfredsställande resultat har vi reviderat och testat igen. En fördel med att utveckla en grafisk applikation är att vi tidigt kunnat se med blotta ögat när ett fel uppstått och har kunnat rätta till dessa fortlöpande.

8 Resultat och rekommendationer

Nedan visas de mål som vi har specificerat samt huruvida dessa är uppfyllda.

- Verktyg för att underlätta grafisk representation av släktträd – uppfyllt!
- En mindre manual för att underlätta programmets framtida nyttjande – uppfyllt!

Det verktyg som vi åtagit oss att skapa har skapats utifrån de krav som ställts. Tester har visat att vi har uppfyllt kraven. De få önskemål om förbättringar som kunden har ansett vara nödvändiga att utföra efter användbarhetstestet, har vi åtgärdat.

Viss omstrukturering av koden kan vara nödvändig för att underlätta framtida underhåll men det handlar mest om att dela upp redan befintlig kod i fler klasser. Ett exempel på detta skulle kunna vara att strukturera om ”main”-klassen med hjälp av samlingar av objekt såsom symboler och linjer. På detta sätt skulle koden kunna flyttas ut så att ”main”-klassen blir mindre och mer flexibel att hantera.

Samarbetet med landstinget har fungerat väl. Vi har fått det stöd i form av information och mjukvara (Visual Studio 2005) som varit nödvändigt för att kunna färdigställa applikationen. Anita Skoogh och Sara Rosengren har mer än uppfyllt sina uppgifter som informationskällor, de har försett oss med den bakgrund som vi frågat om och även kommit med konstruktiva idéer.

De kontakter vi haft inom landstinget har alla tagit sig tid för att hjälpa oss med det som vi har frågat om.

Programmet kan exekvera i en Windows XP och Windows 2000 miljö som används på centralsjukhuset. Det finns en funktion "Skriv ut" som skriver ut släktträdet på papper, det gör att det nya systemet är bakåtkompatibelt med det gamla. Alla de symboler som fanns i det gamla systemet och som beskrivs i kapitel 2.4 finns med i det nya. Det gör att det är förhållandevis lätt för en användare av det gamla systemet att ta till sig det nya. Ingen information lagras i datorn, vilket annars skulle ha lett till problem med personuppgiftslagen. Med hjälp av s.k "fästpunkter" kan användaren på ett enkelt och intuitivt sätt placera ut symboler rakt i höjd och sidled. "ConnectorBoxes" används i programmet för att användaren på ett smidigt sätt ska kunna sammanbinda symboler med linjer. Med hjälp av Microsoft Visual Studio 2005 har vi skapat två olika versioner av programmet; en utan installationsprogram och en med ett enkelt installtionsprogram som också kontrollerar att förvilkoren är uppfyllda.

Den manual som vi skapat kan med fördel användas av nybörjare som handledning vid en introduktion.

9 Summering av projektet

Under examensarbetets gång har vi lärt oss väldigt mycket om det som sker i kulisserna vid utvecklingen av en applikation. Eftersom vi arbetat med människor som haft en annan huvudfunktion än att hjälpa oss, till skillnad från lärare, har vi fått öva på att organisera möten och boka dessa så att alla har möjlighet att närvara.

Vi har bland annat övat på arbete i projektform där vi själva varit ytterst ansvariga. Tidigare har någon lärare drivit på oss och haft det yttersta ansvaret. Om inte vi själva skulle lösa problem som uppstod under exempelvis modellering, kunde vi inte räkna med hjälp från handledaren eftersom han inte var insatt i kundmiljön och inte heller av kunden, eftersom de inte var insatta i de metoder och arbetssätt som vi använde vid modellering. Det gick bra och vi kom starkare ut på andra sidan.

Applikationens struktur formades efterhand. Vi hade ingen erfarenhet av vare sig designmönster eller grafisk programmering (programmering av ritytan) vilket ledde till en grundlösning som fungerade men som sedan byggts om till den nuvarande. Det hade varit en fördel att ha haft kunskap om åtminstone grundläggande designmönster.

Referenser

[1] *.NET Framework Developer Center*. http://msdn.microsoft.com/netframework, 2006-06-15.

[2] *AnkhSVN*. http://ankhsvn.tigris.org, 2006-06-15.

[3] H. Cabrera, J. Faircloth, S. Goldberg. "*C# for Java Programmers*". Syngress Publishing, 2002. ISBN: 1-931836-54-X

[4] W. H. Clark, R. R. Reimer, M. Greene, A. M. Ainsworth, M. J. Mastrangelo. *Origin of familial malignant melanomas from heritable melanocytic lesions. 'The B-K mole syndrome'*. Arch Dermatol, 1978.

[5] B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato. *Version Control with Subversion*. O'Reilly & Associates, 2004. ISBN: 0-596-00448-6.

[6] *Concurrent Versions System*. http://www.nongnu.org/cvs, 2006-06-15.

[7] P. Drayton, B. Albahari, T. Neward. *C# in a Nutshel*. O´Reily & Associates, 2002. ISBN: 0-596-00181-9.

[8] *GDI+*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdicpp/GDIPlus/GDIPlus.asp, 2006-06-15.

[9] *Manual för handläggning av familjer med hereditärt malignt melanom och dysplastiskt nevus syndrom (DNS)*. Svenska melanomstudiegruppen, 1999-04-19.

[10] *Microsoft Developer Network*. http://www.msdn.com, 2006-06-15.

[11] *Mono Project*. http://www.mono-project.com, 2006-06-15.

[12] I. Ottersten, J. Berndtsson. *Användbarheten i praktiken*. Narayana Press, 2002. ISBN: 91-44-04122-5.

[13] J. P. Russel. *Learn Java In a Weekend*. Premier Press, 2002. ISBN: 1-931841-60-8.

[14] *Sourceforge*. http://sourceforge.net/docs/about, 2006-06-15.

[15] M. Telles. *C# Black Book*. Paraglyph Press, 2001. ISBN: 1 932111-17-4

[16] *TortoiseSVN*. http://tortoisesvn.tigris.org, 2006-06-15.

[17] *Visual Studio 2005*. http://msdn.microsoft.com/vstudio, 2006-06-15.

[18] M. Warkus. *Official GNOME 2 Developer's Guide*. No Starch Press, 2004. ISBN: 1-59327-030-5.

[19] G. Westman. *Malignt melanom: en kunskapsöversikt*. Uppsala: Akademiska sjukhuset, 1992. ISBN: 91-87934-59-0.

[20] M. Williams. *Microsoft Visual C# (core reference)*. Microsoft Press, 2002. ISBN: 0-7356-1290-0.

A Instruktioner för användbarhetstest

Bestäm vad användaren ska göra:

- Vad som ska ritas -olika diagram
- Vad som ska skrivas ut -ett av dessa

Vem gör vad:

- Användare -kunskap om tillämpningsområdet
- Handledare -ansvarar för att användaren känner sig tillfreds och att ställa frågor som klargör användarens bild av produkten
- Observatör -har till uppgift att notera vad som händer.

Uppgifter för observatör:

- Anteckna hur användaren bär sig åt för att lösa uppgiften och anteckna vad som sägs men också hur användaren verkar reagera då denne använder programmet

Uppgifter för handledare:

- Bjud in användaren, varför/hur/vad
- Installera och förbered mjukvara
- Förklara för användaren vad som ska till att ske (uppmana till att tänka högt), presentation
- Lämna över manualen
- Lämna över uppgiften
- Led användaren om denne behöver om det, inte annars
- Avsluta genom att fråga användaren:
  – Fungerade programmet som du tänkt?
  – Vad kan göras bättre (alltid finns det något)?
  – Är det något som är jättebra?
- Prioritera ihop med kunden de saker som kan förbättras

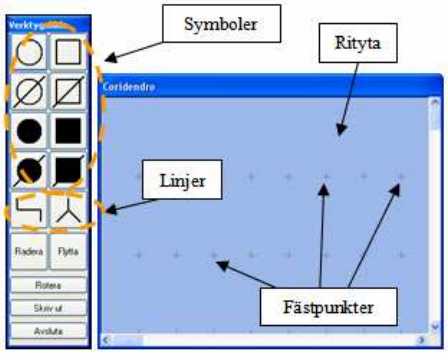
Efter (för observatör och handledare):

- Diskutera:
  – Vad har vi gjort, summera
  – Vad som gick bra/dåligt?
  – Hur verkade användaren ta emot GUI:t?
  – Hur bar användaren sig åt för att lösa uppgiften?
  – Vad kan göras bättre nästa gång, utvärdera.
- Skriv ihop en rapport, lämna över en kopia till kund.

Manual för Coridendro

Den här manualen är riktad mot användare som har lite eller ingen datorvana. Det här kan ses mer som en introduktion där du som användare, efter att ha läst och följt instruktionerna, ska kunna använda programmet på egen hand.

Alla val och markeringar görs med **vänster** musknapp.



1	**Starta** programmet genom att trycka på Start knappen och där navigera dig fram till programmet Coridendro som finns under **[Start]** -> **[Alla Program]**.
2	När programmet startats ser du två fönster, det ena är **verktygslådan** som innehåller knappar och det andra innehåller **rityta**n som är blå med svarta kryss jämt fördelade.
3	Ändra storlek på fönstret med rityta n och förflytta dig runt med hjälp av den horisontella samt vertikala rullist som finns vid kanten av fönstret. Eftersom inte hela rityta n är synlig krävs det att du förflyttar dig runt med **rullisterna** för att kunna rita på hela rityta n.
4	Beroende på om du vill använda dig av stående eller liggande rityta och utskriftsformat kan du **rotera** rityta n genom att trycka på "Rotera" knappen i verktygslådan.
5	Prova nu att rita ut ett par **symboler**. Välj vilken symbol du vill rita ut genom att klicka på den i verktygslådan, markera därefter vart på rityta n du vill ha den, den kommer att fästa på den **fästpunkt** som är närmast muspekaren. Det går att hålla inne vänster musknapp och flytta muspekaren för att flytta omkring symbolen, när du bestämt dig släpper du knappen och symbolen fäster. Rita ut några till i närheten av den du just ritat ut.
6	Nu är det dags att **sammanbinda** de symboler du tidigare ritat ut. Markera **linjeverktyget** i verktygslådan (ser ut som en krokig linje), därefter håller du inne vänster musknapp för att se att den närmaste anslutningspunkten på den närmaste symbolen blir **röd**, det är nu den anslutningspunkten som är vald. Om du släpper knappen märker du att den blir **gul**, nu är det den punkten som är startpunkt för linjen. Upprepa samma procedur en gång till för att välja slutpunkt för linjen. Det går också att sammanbinda symbolerna med en s.k. **tvillinglinje**, det gör du genom att markera tvillingverktyget i verktygslådan (ser ut som ett uppochnedvänt y). Därefter markerar du som med linjeverktyget förutom att du kommer att få markera en gång extra; en gång för föräldern och två för de två barnen.
7	Det går bra att **flytta** symboler som hamnat fel, även de som är kopplade med linjer. Välj "Flytta" knappen i verktygslådan och ta tag i en symbol genom att välja den med vänster musknapp. Håll därefter inne knappen och dra den dit du vill ha den, släpp sedan upp knappen.
8	För att **ta bort** symboler väljer du "Radera" verktyget i verktygslådan och trycker därefter på den symbol som du vill ta bort från rityta n. Om en symbol är sammankopplad med andra med linjer så kommer även linjerna att tas bort.
9	**Skriva ut** diagrammet gör du genom att trycka på "Skriv ut" knappen i verktygslådan. Efter att du tryckt på knappen får du se en förhandsgranskning som visar hur diagrammet kommer att se ut när det är utskrivet på papper. Om du anser att det ser bra ut trycker du på "ok" för att fortsätta.
10	Om verktygslådan skulle hamna bakom något annat fönster, exempelvis fönstret med rityta n, kan du få fram den igen genom att högerklicka någonstans på rityta n. Det här kan vara bra om du arbetar med rityta n i fullskärmsläge (som du får fram genom att dubbelklicka i ramen på fönstret med rityta n).
11	När du är klar med programmet trycker du på "Avsluta" för att **avsluta** programmet.