|  |
| --- |
| Aarhus Universitet |
| ABC Tools |
| Undersøgelse af model understøttet udvikling af applikationer til Activity Based Computing |
| Rasmus Oudal Edberg (edberg@daimi.au.dk) |

# *Abstrakt*

# *Abstract (English)*

*Her beskrives projektet i meget korte træk (laves til sidst)*

Indhold

[*Abstrakt* 2](#_Toc150946640)

[*Abstract (English)* 2](#_Toc150946641)

[Introduktion 6](#_Toc150946642)

[Motivation 8](#_Toc150946643)

[Applikationer 8](#_Toc150946644)

[Problemet 8](#_Toc150946645)

[Relateret Arbejde 10](#_Toc150946646)

[Introduktion 10](#_Toc150946647)

[Whitehorse 10](#_Toc150946648)

[Application Connection Designer 10](#_Toc150946649)

[Logical Datacenter Designer 11](#_Toc150946650)

[System Designer 12](#_Toc150946651)

[Deployment Designer 13](#_Toc150946652)

[Metadata 14](#_Toc150946653)

[Pegboard 14](#_Toc150946654)

[Organisering 14](#_Toc150946655)

[Visualisering 15](#_Toc150946656)

[Simplifisering og vejledning 17](#_Toc150946657)

[Arkitekturen 17](#_Toc150946658)

[Metadata 18](#_Toc150946659)

[Gravity 18](#_Toc150946660)

[Modellen 18](#_Toc150946661)

[Frameworket 18](#_Toc150946662)

[Eksempel 19](#_Toc150946663)

[Together 21](#_Toc150946664)

[Diagrammer 21](#_Toc150946665)

[Patterns 22](#_Toc150946666)

[Audits 23](#_Toc150946667)

[Metrics 23](#_Toc150946668)

[Sammenligning 24](#_Toc150946669)

[Analyse 25](#_Toc150946670)

[ABC Frameworket 25](#_Toc150946671)

[Model understøttet udvikling 26](#_Toc150946672)

[Visual Studio Integration 27](#_Toc150946673)

[Visual Studio UI Elementer 27](#_Toc150946674)

[Visual Studio bruger tilpasning 28](#_Toc150946675)

[Makroer 29](#_Toc150946676)

[Add-Ins 29](#_Toc150946677)

[Objekt Model 30](#_Toc150946678)

[Pakker 31](#_Toc150946679)

[Objekt Model 31](#_Toc150946680)

[Class Designer 38](#_Toc150946681)

[Elementer 38](#_Toc150946682)

[Udvidelse 39](#_Toc150946683)

[DSL Tools 39](#_Toc150946684)

[System Definition Model 40](#_Toc150946685)

[Arkitektur 41](#_Toc150946686)

[ABC Tools 43](#_Toc150946687)

[Introduktion 43](#_Toc150946688)

[ABC Class Designer 44](#_Toc150946689)

[ABC Class Diagram Explorer 44](#_Toc150946690)

[ABC Class Diagram 44](#_Toc150946691)

[Toolbox 44](#_Toc150946692)

[Objekt Model 45](#_Toc150946693)

[Elementer 45](#_Toc150946694)

[Udvidelse 46](#_Toc150946695)

[Evaluering 51](#_Toc150946696)

[Diskussion 52](#_Toc150946697)

[Konklusion 53](#_Toc150946698)

[Bibliografi 54](#_Toc150946699)

# Introduktion

Dette speciale undersøger model understøttet udvikling indenfor Activity-Based Computing med udgangspunkt i følgende tese:

*”Ved at tilbyde model understøttet udvikling af applikationer til Activity-Based Computing vil udviklere være i stand til at overskue applikationer af væsentlig størrelse, samt udvikle mere målrettet i mod det paradigme, som Activity-Based Computing repræsenterer”.*

Strukturen i specialet er som følger. Først vil motivation for at tilbyde model understøttet udvikling til ABC frameworket blive beskrevet, i afsnittet .

Herefter vil der følge en survey af udviklingsmiljøer der tilbyder model understøttet udvikling, idet det er ønskeligt at hente inspiration til udviklingen af model understøttet udvikling, dette er beskrevet i afsnittet .

Med udgangspunkt i de erfaringer der er opnået i forbindelse med ovenstående survey foretages der en analyse af de elementer der skal til for at understøtte ABC specifikke elementer i en model understøttet udviklingsplatform, dette er beskrevet i afsnittet .

Da applikationer til ABC frameworket typisk bliver udviklet i Visual Studio vil mulighederne for integration i Visual Studio blive beskrevet i afsnittet .

Med udgangspunkt i den viden der opnået i forbindelse med analysen af ABC frameworket, samt analysen af integrations muligheder i Visual Studio er der udviklet en prototype, kaldet ABC Tools, der understøtter model understøttet udvikling til ABC frameworket. Denne prototype er beskrevet i afsnittet .

For at undersøge hvorvidt prototypen overholder de forventningerne der er beskrvet i tildels tesen og analysen af ABC Frameworket er der foretaget en evaluering af prototypen. Fremgangsmåden og resultaterne af evalueringen er beskrevet i afsnittet

Evaluering.

På baggrund af analysen af ABC frameworket diskuteres resultaterne der er opnået i forbindelse med evalueringen, dette er beskrevet i afsnittet .

Slutteligt vil der i konklusionen reflekteres over resultaterne og processen i forbindelse hermed blive gennemgået.

# Motivation

Motivationen for arbejdet i dette speciale findes i Activity-Based Computing(ABC) projektet. Activity-Based Computing er er nyt paradigme for håndtering af applikationer således at disse følger en naturlig aktivitets baseret arbejdsprocess.

For at applikationer kan indgå i denne process er det nødvendigt for disse applikationer at gøre deres tilstand til rådighed for ABC således at det er muligt for brugere at aktivere og suspendere aktiviteter på forskellige maskiner og samarbejde om aktivititeter på flere maskiner.

Der er tre forskellige metoder til at gøre det muligt for applikationer at understøtte ABC .

1. Eksisterende applikationer uden API.
2. Eksisterende applikationer med API.
3. Applikationer med adgang til kildekode

Den første slags applikationer kan ikke tilpasses ABC særligt meget i det det ikke er muligt at opdatere tilstanden af disse applikationer ude fra. Det er derfor kun muligt at aktivere sådanne applikationer vha. de informationer operativ systemet stiller til rådighed, vinduestørrelse, placering etc.

Den anden slags applikation gør det ofte muligt at få adgang til en del af applikationens tilstand og det er derfor muligt at udvikle usynlige applikationer der indkapsler sådanne applikationer og derved gør det muligt at styre applikationens tilstand i forbindelse med suspendering/aktivering og samarbejde.

De sidste form for applikationen for der er fuld adgang til kildekoden, fx hos applikationer der er på udviklingsstadiet er det muligt at organisere applikationens tilstand således at den kan arbejde perfekt sammen med ABC frameworket.

## Applikationer

For at gøre en applikations tilstand tilgængelig for ABC tilbyder ABC Frameworket en række annotationer, som kan annotere de variable/klasser som man ønsker at gøre tilgængelige for ABC frameworket.

|  |
| --- |
| [StatefulComponent("talkyou")]  public System.Windows.Forms.TextBox tbTalkYou;  …  [StatefulClass("chat")]  public class ChatApplication; |

Figur – ABC Annotationer.

Ved at bruge disse annotationer er det temmelig simpelt at få en applikationen til at spille sammen med ABC Frameworket.

## Problemet

Selv om det er temmelig simpelt at gøre en applikations tilstand tilgængelig for ABC, så er der en række problemstillinger, som dette speciale vil undersøge.

For det første er de fleste eksempler på applikationer der er udviklet til ABC frameworket mindre applikationer. Selve ABC projektet har dog en tæt tilknytning til Healthcare området. Dette område er normalt kendt for komplekse applikationer i Enterprise størrelsen. Applikationer i Enterprise størrelsen kendetegnes ofte ved flere hundrede (evt. tusinde) klasse og endnu flere variable.

Når applikationer når denne størrelse vil sandsynligvis være besværligt at holde styr på alle de forskellige variable og klasser, hvis tilstand er gjort tilgængelig for ABC.

En metode til at overskue applikationer i Enterprise størrelse er at benytte model understøttet udvikling, som fx UML og lign. Det er derfor interessant at kigge på eksisterende teknologier inden for model understøttet udvikling for at finde en løsning på problemet.

# Relateret Arbejde

## Introduktion

Idet det ønskes at yde tools support til udvikling af applikationer til ABC frameworket præsenterer dette afsnit en survey af tool support for udvikling af applikationer til andre frameworks.

Det er ønskeligt at finde de egenskaber de forskellige eksisterende tools support i eksisterende platforme tilbyder for at have et udgangspunkt i udvælgelsen af de egenskaber tool support til ABC frameworket skal tilbyde.

I de kommende afsnit vil forskellige tool support for forskellige platforme blive gennemgået og kapitlet vil slutte med en opsummering af de egenskaber der er fundet i disse platformes tools support.

## Whitehorse

Whitehorse er Microsofts “Distributed Systems Designers”, en udvidelse til deres Visual Studio 2005 udviklingsmiljø.

Motivationen for Whitehorse er at hjælpe udviklere med at visualisere og validere komplekse Enterprise størrelse distribuerede systemer.

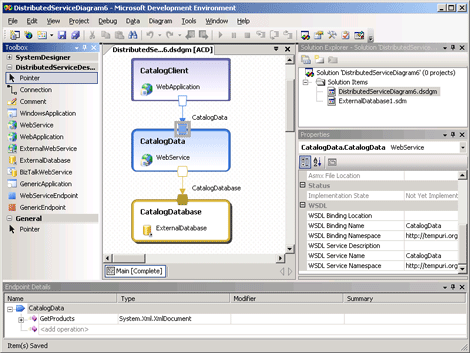
Whitehorse bygger på fire grundprincipper:

* Visualisering
* Synkronisering
* Deployment
* Sikkerhed

Visualisering understøttes igennem 4 visuelle designere i udviklingsmiljøet. Synkronisering sker ved at de fire diagrammer altid synkroniseres i takt med ændringer i koden. Deployment understøttes i kraft af en visuel designer, der giver mulighed for at beskrive forskellige deployment scenarier. Sikkerhed understøttes ved at visualisere de forskellige kommunikations protokoller og grænseflader i det samlede system. I de kommende afsnit vil de forskellige visuelle designere blive gennemgået.

### Application Connection Designer

Den første visuelle designer kaldes *Application Connection Designer (ACD).* ACD giver brugeren mulighed for at definere de forskellige applikationer der indgår i et distribueret system, samt kommunikationsgrænsefladerne imellem disse applikationer. ACD kan ses på Figur 2.



Figur 2 - Application Connection Designer(Gibson, et al., 2004)

Som det fremgår af Figur 2 understøtter ACD alle de forskellige applikationstyper, der kan udvikles i Visual Studio 2005. Desuden er der også understøttelse for en generisk applikation, idet man skelner imellem interne og eksterne applikationer i ACD. Interne applikationer er de applikationer der udvikles i Visual Studio og eksterne applikationer er applikationer der enten skal defineres på et senere tidspunk i projektforløbet eller applikationer der ikke understøttes af Visual Studio.

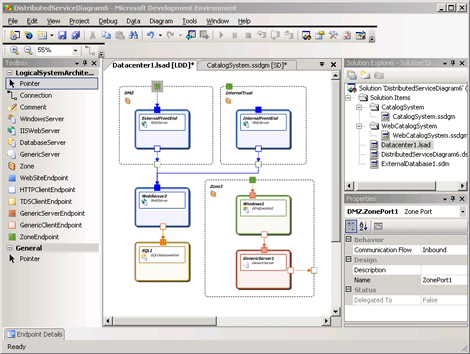
Som det fremgår af Figur 2 er hver eneste applikation visualiseret som en kasse og der er mulighed for trække de forskellige applikationstyper fra toolboxen over i diagrammet. Hver kasse har desuden en nogle mindre kasser på deres kanter, disse er såkaldte *endpoints*. Endpoints svarer til kommunikationskanaler imellem applikationer. ACD arbejder med to forskellige typer endpoints, provider og consumer. Provider endpoints svarer til en service der tilbydes af en applikation og en consumer er det endpoint for en applikationer der benytter providerens service.

ACD tillader at man benytter såkaldt ”*deferred implementation”*  hvilket betyder at de applikationer man har defineret i ACD diagrammet ikke nødvendigvis skal implementeres[[1]](#footnote-2) med det samme. Dette giver mulighed for at designere kan ændre designet uden at det har for store konsekvenser.

ACD tillader desuden at man kan opsætte hosting constraints for de applikationer man designer. Det er således muligt fx. at specificere hvilken version af .NET frameworket der skal være på en host o.lign.

### Logical Datacenter Designer

Det næste vindue i Whitehorse er *Logical Datacenter Designer (LDD)*. Dette vindue giver mulighed for at definere de forskellige serviere, kommunikationstyper, kommunikationsveje og services i en virksomheds datacenter. LDD er vist på Figur 3.



Figur - Logical Datacenter Designer (Gibson, et al., 2004)

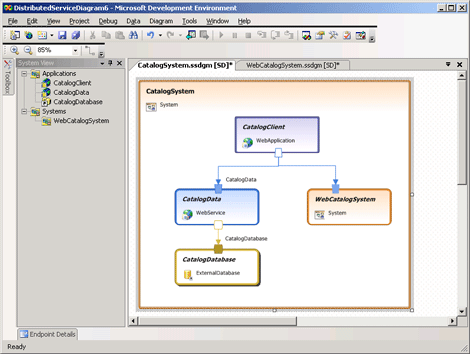
LDD giver driftsafdelingen mulighed for at visualisere designet af datacentret for udviklerne i en virksomhed. Det er igen muligt at benytte toolboxen til at vælge imellem forskellige prædefinerede servertyper, som kan trækkes ind på diagrammet. Det er desuden også muligt at importere diagrammet fra virksomhedens datacenter.

LDD giver også mulighed for at opsætte hosting constraints, dog således at det her er muligt fx. at specificere hvilken version af .NET frameworket den enkelte server kan tilbyde.

LDD benytter *zones* og *endpoints* til at definere fx. forskellige netværk, samt hvilken kommunikation der er tilladt imellem disse netværk. Dette kunne fx. være to forskellige netværk der er forbundet via en VPN[[2]](#footnote-3) forbindelse.

### System Designer

Det tredje vindue i Whitehorse er *System Designer (SD)*. SD giver mulighed for at sammesætte applikationer til et system, dvs. definere hvilke applikationer fra ACD der skal afvikles på en given maskine. Det er med andre ord muligt at definere en deployment enhed. SD kan ses på Figur 4.

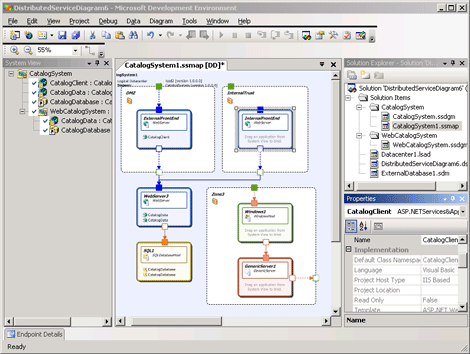


Figur - System Designer(Gibson, et al., 2004)

Som det fremgår af Figur 4 er det muligt at vælge de enkelte applikationer i toolboxen og herfra trække dem ind på diagrammet. Det er desuden også muligt at vælge sammensætte systemer udfra eksisterende systemer. SD viser også kommunikationsgrænseflader imellem de applikationer der tilføjes til diagrammet.

### Deployment Designer

Det sidste vindue i Whitehorse er *Deployment Designer (DD)*. Dette vindue giver mulighed for at definere hvordan enkelte systemer skal deployes i et givent datacenter. DD er vist på Figur 5.



Figur - Deployment Designer(Gibson, et al., 2004)

Ved at vælge de enkelte systemer eller applikationer fra toolboxen er det muligt at trække disse ind på serverene i forskellige servere i datacentret.

Ud fra de begrænsninger der er defineret for både applikationer (i ACD) og servere (i LDD) er det nu muligt at validere at det deployment scenarie man har designet. Valideringsfejl vil blive vist i på en fejlliste i Visual Studio og man har så mulighed for at omkonfigure applikationerne eller datacentret for at løse disse fejl.

### Metadata

Det sidste der skal fremhæves ved Whitehorse er den måde metadata håndteres på. Whitehorse bygger på en meta-model der hedder *System Definition Model (SDM)*. SDM gør det muligt at beskrive forbindelser, konfigurationer for både applikationer, hosting miljøer, netværks topologier, operativ systemer og servere. Det er SDM der gør det muligt at validere deployment fordi det både beskriver applikationer og de systemer hvor applikationerne skal køre under.

De enkelte diagrammer bliver gemt som filer, således at de kan gemmes under source control og genbruges i andre projekter.

Det er også muligt at udvide SDM med nye applikationtyper, kommunikationsprotokoller og servertyper igennem en udvidelses model.

## Pegboard

Pegboard er et framework til udvikling af mobile applikationer . Frameworket bygger på Eclipse platformen og forsøger at give udvikleren et større overblik over den komplekse opgave at bygge distribuerede applikationer til heterogene platforme igennem understøttelse via udviklingsværktøjet.

Pegboard arbejder efter 4 grundprincipper:

* Organisation
* Visualization
* Simplification
* Guidance

I de følgende afsnit det blive forklaret hvorledes Pegboard understøtter disse principper, endvidere vil arktekturen bag Pegboard blive forklaret.

### Organisering

Organisations understøttelse opnås ved at Pegboard tilbyder udviklere en Pegboard projekt type i Eclipse. Strukturen af et Pegboard projektet er vist på Figur 6. Projektet kan indeholde en række *Subapplications*, som er *Composit Projects*, dvs. de kan indeholde et vilkårligt antal eclipse projekter også kaldet *Functional Component* i Pegboard frameworket. Hver enkelt eclipse indeholder så en række *resources*, hvilket er et Eclipse term for kodeelementer, dvs. klasser, konfigurationsfiler osv.

Kodeelemter kan deles imellem de enkelte projekter, således at der er mulighed for kodegenbrug imellem de enkelte applikationer. Dette er implementeret som symbolske lænker, som det bl.a. er kendt fra unix filsysterm.



Figur 6 - Pegboard projekt opbygning

Kommunikationen imellem de enkelte subapplications foregår ved hjælp af en *Connector*. En *Connecter* kan repræsentere forskellige kommunikations teknologier, såsom HTTP[[3]](#footnote-4) og eller SOAP[[4]](#footnote-5).

### Visualisering

Visualisering opnås ved at vise et overblik over det distribuerede system i form af to nye vinduer i eclipse, ”Composite Explorer” og ”Composite View”. Disse vinduer er vist på hhv. Figur 7 og Figur 8.



Figur 7 - Pegboards Composite Explorer(Soroker, et al., 2006)

”Composite Explorer” giver udvikleren overblik over den distribuerede applikation i en træ struktur, en struktur de fleste udviklere i forvejen er bekendte med idet de fleste udviklingsværktøjer bruger sådanne strukturer til at give et overblik over kode artifakter.

Vi kan desuden se at strukturen som vi tidligere nævnte er præcist gengivet i dette billede. Roden af træstrukturen er således Pegboard projektet og under disse ligger *subapplications*. Det ses desuden også at ”Order Entry Device” *subapplication* indeholder flere Eclipse projekter, ”OE Common”, ”OE hybrid” og ”OE service” .

Det fremgår også af Figur 7 hvorledes Pegboard illustrerer delt kode kode for udvikleren, idet ”sharedCode” resourcen optræder med en genvejspil alle steder undtagen i ”Order Entry Shared Area”, hvor koden rent faktisk er linket til.

Den eneste struktur element der ikke vises i ”Composite Explorer” er *Connection* elementet. For at se forbindelserne i mellem de enkelte *subapplications* i et Pegboard projekt er man nødt til benytte sig af ”Composite Viewer” vinduet, som ses på Figur 8.



Figur - Pegboards Composite Viewer

”Composite Viewer” vinduet giver en grafisk repræsentation af alle *subapplications* i Pegboard projektet. For hver *subapplication* viser den hvilke *functional components* applikation består af. Desiden viser den også *connectors* imellem disse *subapplications*, samt hvilken type *connector* der er tale om, i dette tilfælde en *Message Queue Everywhere* forbindelse.

### Simplifisering og vejledning

Simplification understøttes i kraft af mulighederne for kørsel og debug i Pegboard. Hvor Eclipse normalt kun er i stand til at afvikle én applikation af gangen så giver Pegboard mulighed for at afvikle og debugge samtlige applikationer i et distribueret system på én gang.

Guidance understøttes i form af arkitektur mønstre. I den applikation der vises på Figur 8 fremgår det at vi det er en klient-server arkitektur der er benyttet. Ved at lade brugeren vælge fra en liste af mulige arkitektur mønstre ved oprettelsen af et projekt er guider Pegboard brugeren ved at oprette grundstrukturen til den valgte arkitektur.

### Arkitekturen

Arkitekturen bag Pegboard er opdelt i tre lag, som det fremgår af Figur 9. Det nederste lag i Arkitekturen er Eclipse platformen, herfra får Pegboard sit udviklingsmiljø, adgang til projekter, plugins osv.



Figur - Pegboard Arkitektur

Det næste lag er composite laget. I dette lag har implementeret muligheden for *composite projects*, herunder kørsel af flere samtidige projekter. Dette lag alene er en bedrift i sig selv idet andre projekter der ønsker en friere strukturering af projekter i Eclipse fint kunne benytte dette lag alene.

For at ”Composite” laget kan benytte sig af et vilkårligt antal projekt typer er der lavet en extension arkitektur, hvor det er muligt at tilføje *tool bridges* for nye projekt typer. På den måde er nye værktøjer ikke afhængige af Pegboard og Pegboard er ikke afhængig af de nye projekt typer.

Det sidste lag er pegboard laget. Det er i dette lag at ”Composite Explorer”, ”Composite Viewer” vinduerne er implementeret. Det er desuden i dette lag at understøttelsen for arkitektur mønstre er implementeret.

### Metadata

Det sidste der er skal bemærkes ved Pegboard systemet er den måde hvorpå Pegboard systemet håndterer metadata. Pegboard benytter flere forskellige konfigurationsfiler til at holde styr på dens projekter. Der bliver lavet konfigurations filer på Pegboard projekt niveau, som holder styr på den enkelte projekter indenfor Pegboard projektet. Desuden er der konfigurationsfiler inden for de enkelte projekter, således at projekterne kan genbruges i andre Pegboard projekter.

## Gravity

Med udganspunkt i kravene fra Pervasive Computing og Ubiquitous Computing bygger Gravity på den ide at alle fremtidige clientside applikationer bygges ud fra genbrugelige byggesten, såsom webservice og kodekomponenter.

I modsætning til andre udviklingsmiljøer er komponterne i Gravity dynamisk tilgængelige, hvilket betyder at de komponenter kan fremkomme og forsvinde på ethvert tidspunkt.

### Modellen

Grundideen er en samling af koncepter fra både komponent orienteret og service orienteret udvikling. Herved opnåes en service orienteret komponent model. I modsætning til SOA er en *Consumer* ikke bundet til en specifik *Provider*.

Dette betyder at det er muligt at skifte i mellem forskellige instanser af samme *provider* . Det kan faktisk være helt forskellige implementeringer der overholder samme interface. For at det kan lade sig gøre er at også nødvendigt at en service komponent ikke indeholder dens service beskrivelse i det der kan være flere komponenter, der overholder samme beskrivelse.

For at holde styr på de forskellige service beskrivelser og komponent instanser benytter Gravity sig af et *Service Registry*, hvor komponenter kan registrere sig for at tilbyde en service og hvor komponenter kan søge efter tilgængelige services.

### Frameworket

For at understøtte denne model følger der et framework med i Gravity. Frameworket understøtter en lang række elementer der er nødvendige for at understøtte den dynamiske model. Disse elementer er som følger:

* Deployment
* Design af Applikationer
* Service afhængigheder

Gravity arbejder efter princippet ”*deploy af any time*”. Frameworket understøtter direkte installation, opdatering, aktivering og fjernelse af komponenter.

Design af applikationer understøttes ligesom deployment efter princippet *”design at any time”*, hvilket betyder at en applikation altid er i en tilstand hvor den kan designes. Frameworket understøtter både automatisk og brugerstyret layout af komponenter.

For at styre service afhængigheder for en applikation benytter Gravity en *Service Binder*. En *Service Binder* giver udvikleren mulighed for udelukkende at styre afhængingheder til komponenter vha. metadata istedet for selv at styre kompleks inter komponent kommunikation.

Hvert enkelt komponent er beskrevet i en XML fil kaldet *instance descriptor*. For hver enkel komponent der er beskrevet i *instance descriptor* filen opretter *Service Binderen* en *Instance Manager*.

En *Instance Manager*  svarer til intentionen om at oprette en komponent. Den er ansvarlig for at overvåge en komponents service afhængigheder, oprette/nedlægge komponenter, bind/unbind services til komponenten og håndtere registrering af bundne services.

En *Instance Descriptor* indeholder nok information omkring hver service til at en *Instance Manager* kan oprette forbindelse til komponenten. Den indeholder informationer såsom klassenavn, properties, service afhængigheder. Service afhængigheder er bl.a. beskrevet i form af interfacenavn, kardinalitet og *binding policy.*

En binding policy beskriver om en afhængighed til service er statisk eller dynamisk. En statisk binding kan ikke ændres i runtime uden at den pågældende komponent instans invalideres. En dynamisk binding tillader at service bindinger kan ændres uden at invalidere komponent instansen.

### Eksempel

I det følgende afsnit vil de visuelle dele af Gravity blive forklaret. Med udgangspunkt i en teksbehandlings applikation, der er udviklet i Gravity.

|  |  |
| --- | --- |
| Figur - Gravity i Design tilstand (Hall, et al., 2003) | Figur - Gravity i runtime tilstand (Hall, et al., 2003) |
| **Figur - Gravity i runtime tilstand med manglende komponent** | Figur - Gravity i runtime tilstand med alternativ component (Hall, et al., 2003) |

På Figur 10 vises en Gravity applikation i Design tilstand. Som det fremgår af figuren har Gravity i denne tilstand en liste over komponenter i venstre side og applikationen designes visuelt i højre side. Listen med komponenter i venstre side opdateres automatisk efter hvilke komponenter der er tilgængelige.

Det er muligt for udvikleren at trække komponenter fra komponent listen over på den grafiske repræsentation af applikationen, som det er kendt fra de fleste visuelle udviklingsmiljøer.

På Figur 11 vises applikationen i runtime tilstand. I applikationen er der åbnet to filer (buffer0 og buffer1), den ene fil vises i redigeringsområdet af applikationen. Den nederste del af applikationen viser en fil browser komponent, der viser stien til den aktuelle fil.

På **Figur** 12 vises applikationen i en tilstand hvor fil browser komponenten er forsvundet og der ikke er alternative komponenter til rådighed.

Hvis der havde været alternative komponent instanser der havde tilbudt fil browser servicen til rådighed havde applikationen automatisk valgt denne, som vist på Figur 13. Det kan her ses at fil browser servicen er implementeret som en folder liste frem for en træstruktur, som på Figur 11.

## Together

Together er modelleringsværktøj fra Borland. Det fungerer bl.a. som en udvidelse til både Visual Studio og Eclipse. Together bygger på princippet *Instant UML Visualization*. Bagved dette princip ligger et ønske om at være kompatibel med UML(TODO) standarderne og samtidig sørge for at kode og model til enhver tid er synkroniseret (kaldet *LiveSource*).

Foruden at understøtte modellering efter UML standarderne understøtter Together også mønstre fx GoF(TODO) og statiske kode analyse igennem såkaldte *Audits*.

### Diagrammer

Da Together forsøger at efterleve UML standarden understøtter også de primære diagramtyper der er defineret i UML standarden:

* Class diagram
* Use Case diagram
* Component diagram
* Deployment diagram
* Sequence diagram
* Collaboration diagram
* Activity diagram
* Statechart diagram

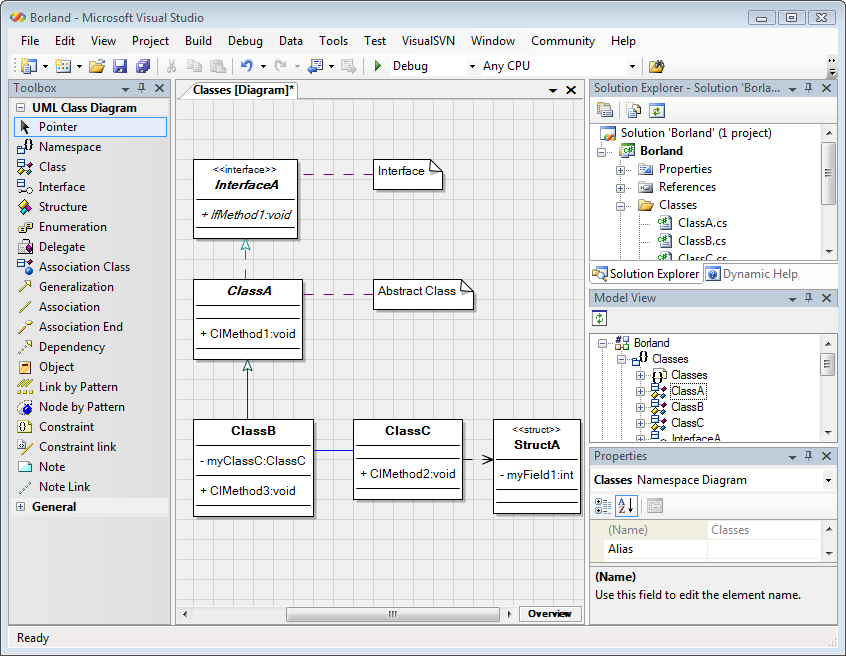
*LiveSource* er kun understøttet af *Class* diagrammet. De andre diagrammer kan benytte elementer fra kildekoden, men ændrer ikke på kildekoden automatisk.

På Figur 14 er Together integrationen i Visual Studio vist. Der er tre integrationspunkter med Visual Studio. Det første er et nyt view kaldet *Model View*, der hele giver opdateret visning af diagrammer og objekter (klasser etc.) i systemet.

Det næste integrationspunkt er selve diagrammet. En speciel egenskab ved Together er at der automatiske bliver tilføjet et *Class* diagram til alle namespace, både namspaces i systemet og de indbyggede namspaces, (fx System).

Det næste integrationspunkt er toolboxen, afhængigt af hvilken diagram type der arbejdes med er toolboxen udfyldt med de passende grafiske notationer for diagram typen. Disse grafiske notationer kan så trækkes/slippes ind på diagrammet.

Når der trækkes et klasse, interface eller structure ind på diagrammet bliver der automatisk oprettet en tilsvarende kodefil. Hvis koden ændres bliver diagrammet automatisk opdateret så det stemmer overens med koden.

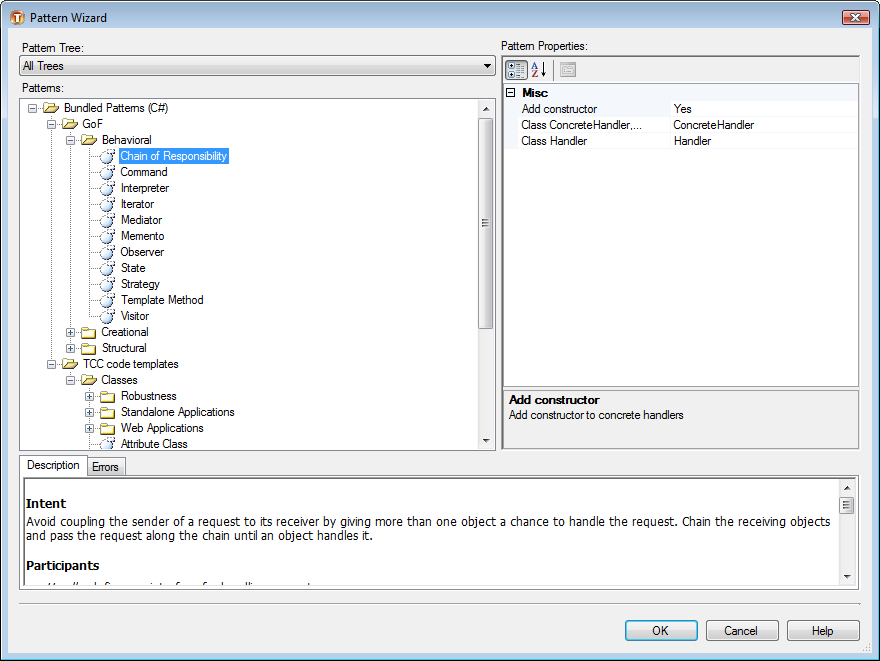


Figur - Together *Class* diagram.

De andre diagram typer sørger ikke for automatisk kodegenerering og kan derfor udelukkende bruges som retningslinier.

### Patterns

Together understøtter at tilføje hele blokke af kode til en diagram i form af design mønstre. På Figur 15 er *Pattern Wizard* vist. *Pattern Wizard* er det værktøj der bruges til at vælge de mønstre man ønsker at indsætte op et *Class* diagram.

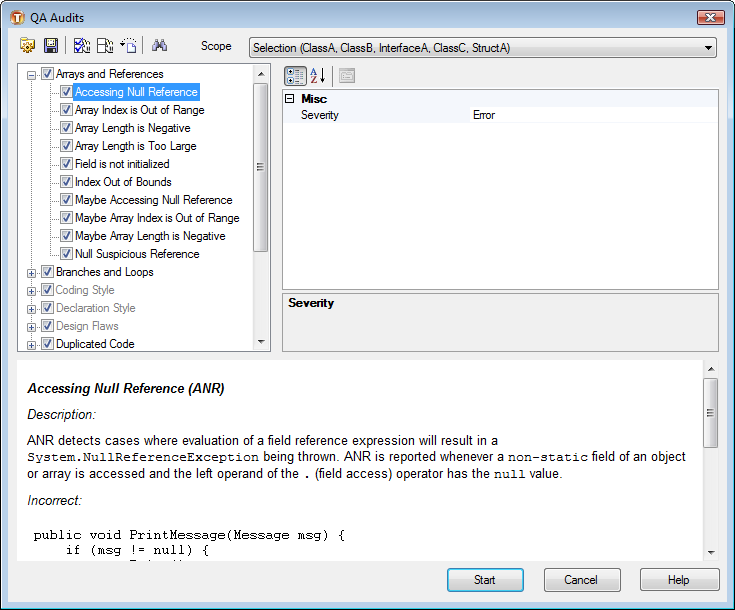


Figur - Together pattern wizard.

Mønstrene er opdelt i grupper og der en forklaring af mønstrets formål og de klasser der indgår i mønstret. Det er desuden muligt at specificere diverse egenskaber for mønstret, såsom klasse navne etc.

### Audits

*Audits* giver mulighed for at køre en lang række forskellige statiske tjek på koden i et projekt. Det er muligt at udvælge de specifikke klasser man ønsker at tjekke.

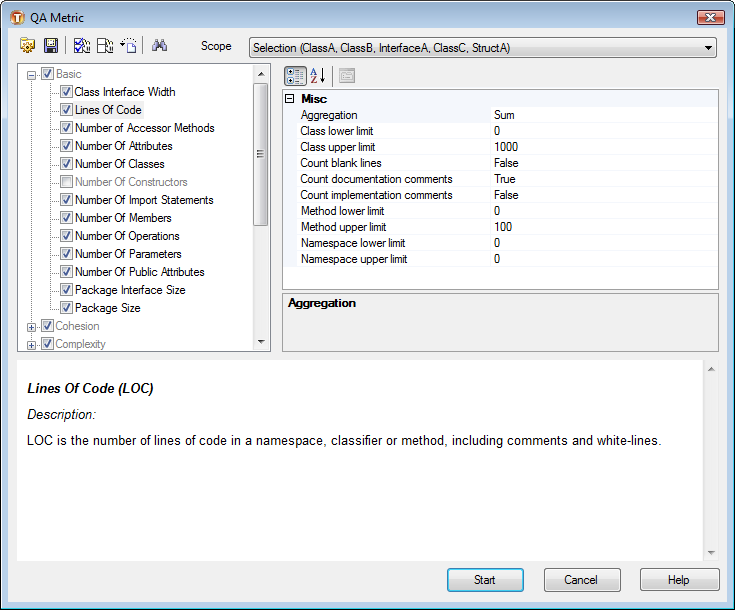


Figur - Together *Audits* værktøj.

For hver tjek er der information omkring tjekket og mulighed for at indstille alvorlighedsgraden af den fejl der tjekkes for.

### Metrics

*Metrics* giver mulighed for rapportering af en lang række egenskaber for projektet fx antal kodelinier.



Figur - Together *Metrics* værktøj.

På Figur 17 er *Metrics* værktøjet vist. Med udgangspunkt i de modelelementer der er valgt er det muligt at til og fra vælge metrics som der ønskes rapporteret.

Til hver *Metric* er der tilknyttet en række justerbare egenskaber og der er desuden en forklaring på den valgte *Metric.*

## Sammenligning

Der skelnes imellem tre forskellige slags udviklingsmiljøer, Generelle, Generelle med domæne udvidelse, Domæne specifikke.

# Analyse

I dette beskrives den analyse der er lavet af behov for værktøjsunderstøttelse til programmering imod ABC frameworket. Analysen tager udgangspunkt i afsnittene og . Analysen består af to afsnit.

Det første afsnit forklarer en række antagelser og foreslåede udvidelser til ABC Frameworket, som er fundet i forbindelse med analysen af ABC Frameworket med henblik på udvikling af applikationer i Enterprise størrelsen.

Det næste afsnit forklarer hvilke elementer af model understøttet udvikling det er fundet ønskeligt at anvende/udvikle i forbindelse med at understøtte modellering af applikationer i Enterprise størrelsen.

### ABC Frameworket

Som beskrevet i og er der på nuværende tidspunkt en mulighed for at benytte *StatefulComponent* attributten til at annotere medlemsvariable i klasser. Når en medlemsvariabel er annoteret med denne attribut vil medlemsvariablen blive en *komponent* i den *application* klassen indgår i.

Foruden *StatefulComponent* attributten er der implementeret et skelet til *StatefulClass*, der skal gøre det muligt at serialisere en hel klasse som en komponent i ABC protokollen. Muligheden for at gøre tilstanden af en komplet klasse tilgængelig vil fx gøre det unødvendigt at tilføje *StatefulComponent* annotationer til alle medlemsvariable i databærende klasse.

På nuværende tidspunkt har man altså inddraget mulighed for at ABC frameworket kan arbejde med applikationer på attribut og klasse niveau.

Hvis man følger den tankegang der ligger til grund for ovenstående kunne man ønske at udvide denne understøttelse endnu mere. Derfor foreslås det her understøttelse på endnu to niveauer:

* Pakke niveau (namespaces)
* Arkitektur niveau (kommunikation imellem applikationer)

Disse tanker kræver naturligvis en forklaring, som her følger. Argumentet for at understøtte tilstandsstyring på pakke niveau kunne fx være applikationer, der benytter et helt datalag, som fundament for en applikations tilstand. Et sådant data lag er ofte implementeret i eget namespace og det kunne derfor være interessant om man ved blot at annotere namespacet med en *StatefulNamespace* attribut kunne gøre hele data lagets tilstand tilgængelig for ABC frameworket.

Argumentationen for at understøtte tilstandsstyring på arkitektur niveau findes i den observation at de fleste applikationer i Enterprise størrelsen er distribuerede applikationer og har derfor behov for kommunikationen imellem de enkelte applikationer. Da der på nuværende tidspunkt ikke er taget højde for tilstandstyring i distribuerede applikationer (suspendere/aktivere applikationer på flere computere på én gang) vil dette projekt undersøge muligheden for at understøtte designet af sådanne arkitekturer og håber derved at kunne inspirere forskningen af dette.

### Model understøttet udvikling

ABC Frameworket er udviklet i C# på Microsofts .NET platform. Udviklingen af applikationer der understøtter .NET frameworket skal derfor enten udvikles i et .NET sprog eller indkapsles af en applikationen skrevet i et .NET sprog . Applikationer til .NET platformen udvikles for det meste i Visual Studio, det er derfor naturligt at udvide Visual Studio således at Visual Studio kan tilbyde model understøttet udvikling til ABC Frameworket.

I afsnittet blev Whitehorse teknologien præsenteret. Visual Studio understøtter altså allerede model understøttet udvikling. Det er derfor interessant at undersøge muligheden for at udvide elementerne i Visual Studio til at understøtte de behov der er identificeret i forbindelse med udviklingen af Enterprise applikationer til ABC Frameworket.

Figur 18 - ABC Niveau versus Visual Studio Element

På Figur 18 er sammenhængen imellem de elementer der er foreslået i analysen af ABC frameworket med de grafiske editorer der er tilgængelige i Whitehorse projektet. I det kommende vil det blive forklaret hvordan elementerne i Visual Studio ønskes udvidet således at de understøtter elementerne fra analysen af ABC Frameworket.

*Class Designer* ønskes udvidet med en eksplicit notation for *Stateful* attributter, klasser og namespace. Den nuværende implementering af *Class Designer* gør allerede muligt at designe klasser med annotationer, det er dog i en implicit form i det ikke fremgår af den grafiske notation.

*Application Connection Designer* ønskes udvidet med en ABC specifik forbindelses type. Det er i forbindelse med dette ønskeligt at undersøge hvilke muligheder og begrænsninger en ny forbindelses er underlagt for at dette kan tages i betragtning i forbindelse med forskningen af distribueret suspendering/aktivering.

I det kommende kapitel vil mulighederne for de ønskede undersøgelser blive undersøgt igennem en analyse af integrationsmuligheder i Visual Studio.

# Visual Studio Integration

I dette afsnit vil mulighederne for integration i Visual Studio blive gennemgået. Visual Studio er designet med rige muligheder for automatisering, tilpasning og udvidelse. Der findes fire forskellige muligheder for at udvide Visual Studio, disse er bruger tilpasning, makroer, add-ins og VSPackages.

|  |
| --- |
| Kompleksitet  Fleksibilitet  Tilgængelighed |

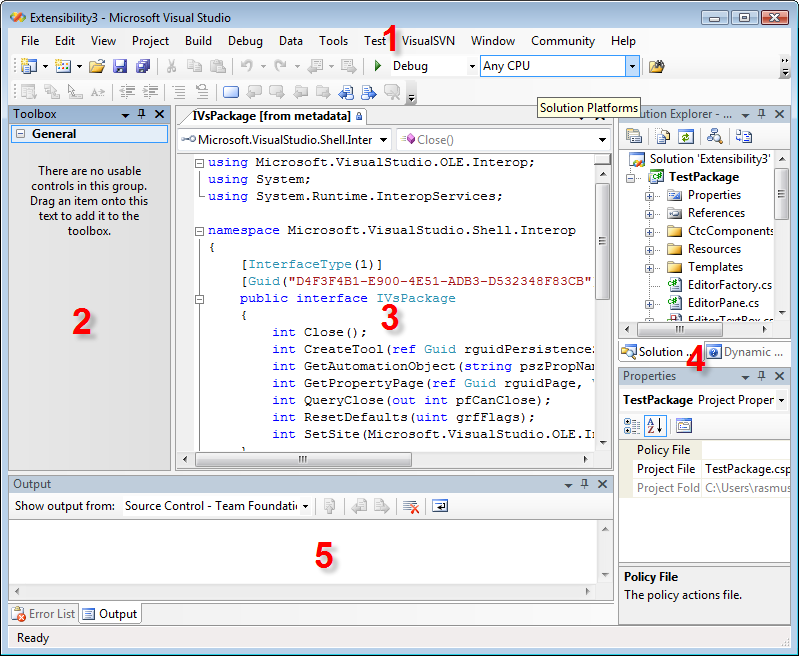
Figur - Intragrations pyramiden.

Som det fremgår af Figur 19 er der en sammenhæng i mellem fleksibilitet og kompleksitet i udvidelses mulighederne, til gengæld i kraft af at det bliver mere komplekst formindskes målgruppen.

I det kommende afsnit vil de forskellige integrationsmuligheder blive gennemgået. Der vil blive lagt vægt på gennemgangen af Add-ins og i særdeleshed Pakker, da det disse to der primært er interessant i forhold til dette speciale.

## Visual Studio UI Elementer

For at forstå hvordan man tilpasser/udvider Visual Studio er det nødvendigt at have et overblik over de forskellige UI elementer i Visual Studio. I de kommende afsnit vil de forskellige UI elementer blive introduceret.



Figur 20 - Elementer i Visual Studio

På Figur 20 er de vigtigste elementer i Visual Studio markeret. Disse elemeter er forklaret i nedenstående liste:

1. Menupunkt – Som kendt fra de fleste windows applikationer har Visual Studio også en menu linie, herfra foregår det fleste administrative opgaver via menupunkter.
2. Toolboxen – Bliver typisk anvendt af grafiske editors til at opbevare de forskellige elemeter, der kan trækkes/slippes ind på design grænsefladen.
3. Editor – Giver enten en grafisk eller tekstuel mulighed for at redigere i filer, klasser o.lign.
   1. Tool Window – Et visningvindue, der ofte bruges til at implementere vinduer der giver overblik over et valgt element eller projekt(er) .

## Visual Studio bruger tilpasning

Visual Studio bruger tilpasning er den som den normale bruger af Visual Studio foretager. Det er igennem denne tilpasning at brugeren kan konfigure skriftstørrelse, egenskaber for de forskellige værktøjer osv.

Disse tilpasninger bliver gemt i brugerens profil og brugeren har mulighed for at importere og eksportere vedkommendes tilpasninger.

Denne tilpasning bliver udmærket beskrevet i den integrerede hjælp i Visual Studio og vil derfor ikke blive uddybet nærmere.

## Makroer

Makroer repræsenterer den simpleste form for programmering af Visual Studio. En Visual Studio makro er et script, dvs. det afvikles uden at blive kompileret. En makro skrives i VB.NET og har derfor adgang til hele .NET frameworket. Der er endvidere adgang til alle kommandoer i Visual Studio, de såkaldte *Named Commands*.

En makro kan laves ved at benytte den indbyggede makrooptager i Visual Studio eller kan skrives vha. den indbyggede makro editor.

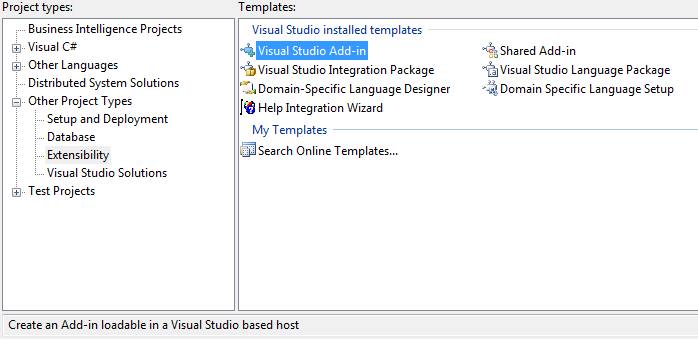
Der er en række ulemper ved makroer; makroer bliver ikke kompileret, dvs. de er et ringe udgangspunkt for kommercielt softwareudvikling, makroer kan desuden ikke oprette nye vinduer osv. i Visual Studio.

Makroer er altså en mulighed for at automatiser Visual Studio og giver derfor ikke en reel mulighed for udvidelse.

## Add-Ins

Add-ins er den første reelle mulighed for udvidelse af Visual Studio. Add-ins overkommer en række af de begrænsninger der findes ved makroer, bl.a. bliver Add-ins kompileret og de kan desuden oprette nye vinduer i Visual Studio.

For at arbejde med Add-ins er det nødvendigt at installere Visual Studio SDK. Når denne er installeret kommer der en række nye projekttyper frem i Visual Studio, bl.a. Add-in projekttypen, som vist på Figur 21.



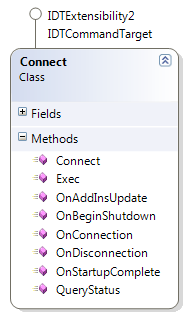
Figur 21 - Visual Studio SDK Projekttyper

I modsætning til makroer der udelukkende kan skrives i VB.Net så er det muligt både at skrive Add-ins i C#, VB.NET, J#, C++(Managed/Unmanged).

Add-ins kan afvikles både under makro fortolkeren, dog som kompileret kode og direkte imod Visual Studio.

### Objekt Model

For at forstå hvordan en Add-in virker er det nemmest at kigge på den objekt model som et Add-in skal overholde. Den nemmeste måde at få fat i denne information er at oprette et ny Add-in projekt og undersøge det generede kode.



Figur - Generet kode for Add-in

Som det fremgår af Figur 22 bliver der kun genereret en klasse når man laver et Add-in projekt. Klassen overholder 2 interfaces, som er værd at undersøge nærmere.

Det er igennem *IDTExtensibility* interfacet at Visual Studio notificerer et Add-in om dets tilstand. Visual giver Add-in udvikleren mulighed for at reagere når et Add-in tilføjes/fjernes (*OnAddInsUpdate*), når Visual Studio lukkes imens et Add-in afvikles (*OnBeginShutdown*), når en bruger tilføjer/fjerner et Add-in fra Visual Studio (hhv. *OnConnection/OnDisconnection*) og når et Add-in der skal loades når Visual Studio startes er startet (*OnStartupComplete*).

Det er igennem *IDTCommandTarget* interfacet at Add-in udvikleren har mulighed for at tilføje *Named Commands* til Visual Studio. Disse *Named Commands* er de samme som benyttes når der skrives makroer. Det altså muligt for et Add-in at blive automatiseret via makroer igennem dette interface. Interfacet kræver to metoder, en for at Visual Studio kan bede om at afvikle en *Named Command* (*Exec)* og en metode der giver Visual Studio mulighed for at spørge efter tilstanden af et Add-In (*QueryStatus*).

De vigtigste metoder er dog *OnConnection* og *Exec*. *OnConnection* metoden er vigtig fordi det er i denne metode at Visual Studio giver adgang til dens automatitions klasser igennem et objekt der implementerer både *EnvDTE.DTE* og *EnvDTE80.DTE2* interfacene, hvor *EnvDTE* er en forkortelse for *Environment Developement Tools Extensibility*.

*EnvDTE.DTE* og *EnvDTE80.DTE2* interfacene giver adgang til de fleste faciliteter i Visual Studio, såsom forskellige vinduer, andre Add-ins , makroer, åbne filer osv. Med adgang til disse interfaces er det næsten kun udviklerens fantasi der sætter grænse for hvilken funktionalitet en Add-in skal tilbyde.

## Pakker

Selvom Add-in gør de muligt at automatisere og udvide visual studio på utrolig mange punkter er der stadig en række opgaver, som Add-ins ikke kan klare. Det er her pakker eller *VSPackages* kommer ind i billedet.

Det er i særdeleshed når Visual Studio skal have tilføjet helt nye funktioner at pakker skal benyttes. Pakker er byggestenen i Visual Studio. Langt det meste af den funktionalitet som Visual Studio er født med kommer i form af pakker udviklet internt hos Microsoft.

Det er bl.a. igennem pakker at de forskellige udviklingssprog er implementeret, samt de forskellige grafiske designere.

I Visual Studio SDK findes der to forskellige typer projekter til at udvikle pakker med, integrations projektet og sprog projektet. Integrationsprojektet benyttes hvis der skal laves en normal udvidelse til Visual Studio og sprog projektet benyttes hvis der skal udvikles et nyt programmeringssprog.

Vi vil kun præsentere integrations projektet her, idet sprog projektet er uden for rammen af denne rapport.

I modsætning til Add-ins kan integrations pakker udelukkende udvikles i C++(Managed) og C#.

### Objekt Model

Med udgangspunkt i den kode der genereres for et nyt integrations projekt vil objekt modellen for pakker blive gennem gået.

|  |
| --- |
|  |

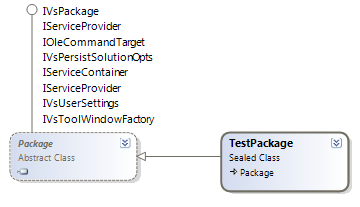
Figur - Pakke framework

Figur 23 viser den hvordan pakke integration med Visual Studio er opbygget. Det nederste lag indeholder Visual Studio i sin reneste form, en applikation der tilbyder udvidelse vha. en række COM interfaces. Oven på disse COM interfaces har er der en række COM interopbility klasser og ovenpå disse klasser har man bygget en framework til udvikling af pakker (MPF).

Integrations projektet giver mulighed for at bygge et værktøjsvindue, en editor og et menupunkt. Med adgang til disse 3 UI elementer er det muligt at konstruere de fleste udvidelses projekter. I de kommende afsnit vil hver selve pakken og de 3 UI elementers implementering blive gennemgået.

#### Pakke implementering

Ligesom en Add-in skulle overholde diverse interfaces for at kommunikere med Visual Studio så er der også en række interfaces en pakke skal overholde for at kunne integrere sig. Disse er dog en anelse mere komplekse end tilfældet ved Add-ins.



Figur - Pakke implementering.

For at implementere en pakke skal man blot lave en klasse der arver fra *Package* klassen fra MPF. *Package* klassen sørger for at implementere de nødvendige interfaces for at integrere med Visual Studio. Det er dog alligevel nødvendigt at kende betydningen af disse interfaces, idet det ofte er nødvendigt at override metoder fra disse interfaces. I det følgende vil de vigtigste informationer omkring disse interfaces blive beskrevet.

*IVsPackage* interfacet skal implementeres af alle pakker. *IVsPackage* interfacet svarer til forbindelsen imellem en pakke og Visual Studio. *IVsPackage* interfacet indeholder metoder til at oprette og lukke en pakke og tillader desuden pakken at registrere sig så andre pakker, Add-ins og Makroer kan benytte servicer fra pakken.

*IOleCommandTarget* svarer til *IDTCommandTarget* interfacet som benyttes ved Add-ins. Det giver en pakke mulighed for at tilbyde og eksekvere *Named Commands*.

*IServiceProvider* interfacet giver en pakke mulighed for at offentliggøre dets komponenter. Andre pakker, Add-ins og makroer kan igennem dette interface hente referencer til komponent instanser. *IServiceContainer* interfacet giver mulighed for at tilføje/fjerne services fra en pakke.

*IVsToolWindowFactory* interfacet giver en pakke mulighed for at oprette ToolWindows. For at holde styr på de forskellige ToolWindows har hvert ToolWindows et unikt ID og hver instans af et given ToolWindow har et unikt instans ID.

*IVsUserSettings* gør det muligt for en pakke at understøtte import/export af en pakkes indstillinger i en brugers profil. Hvis en pakke har en række indstillingsmuligheder er det igennem dette interface at persistering af disse bør foregå.

*IVsPersistSolutionOpts* interfacet giver en pakke mulighed for at gemme Solution specifikke indstillinger. Solution specikke indstillinger er dem der kun gælder for den pågældende Solution og vil derfor ikke være tilstede når der oprettes en ny Solution, i modsætning de indstillinger der er gemt i en brugers profil.

#### Menupunkt implementering

Det er muligt at implementere en række forskellige slags menupunkter i Visual Studio, de mest vigtige er vist på Figur 25 - Menupunkts typerFigur 25.

Menupunkt.emf

Figur - Menupunkts typer

De forskellige typer er forklaret i en nedenstående punkter, svarende til punkterne på Figur 25:

1. **Top Level Menu** – Svarer til et menupunkt, såsom File, Edit og View, der kendes fra de fleste Windows programmer.
2. **Cascading Submenu** – Et menupunkt i en Top Level Menu, der vises som tekst og en pil og har en række underpunkter.
3. **Toolbar** – En række ikon baserede knapper, der findes både under menuen og på de enkelte Tool Windows.
4. **Context menu** – En konstekt menu fremkommer typisk ved at højre klikke på element i Visual Studio, hvorved der vises en menu der passer til konteksten af det valgte element.

Fælles for de forskellige former for menupunkter er at de altid resulterer i udførslen af en specifik kommando.

At implementere et menupunkt for en pakke i Visual Studio er nok den komplekse integration fordi den benytter et specielt format. Formatet beskrives i en *Command Table Configuration* (.ctc) fil (se Figur 26).

|  |
| --- |
| CMDS\_SECTION guidPkg  MENUS\_BEGIN  // NewMenu,Relative to Group,Priority,Type,Name,Text  MENUS\_END  NEWGROUPS\_BEGIN  // NewGroup,Parent Group,Priority  NEWGROUPS\_END  BUTTONS\_BEGIN  // Command,Parent Group,Priority,Image,Type,Visibility  BUTTONS\_END  BITMAPS\_BEGIN  // Bitmap,Bitmap Index, Bitmap Index, ...  BITMAPS\_END  CMDS\_END  CMDUSED\_SECTION  // Command  CMDUSED\_END  CMDPLACEMENT\_SECTION  // Command,Group,Priority  CMDPLACEMENT\_END  VISIBILITY\_SECTION  // Command,GUID when visible  VISIBILITY\_END  KEYBINDINGS\_SECTION  // Command,when available,emulation,keystate  KEYBINDINGS\_END |

1. Figur 26 - *Command Table Configuration* fil

Denne fil kompileres med et specielt værktøj, *Command Table Compiler* (ctc.exe), der genererer en speciel ressource fil (.cto). Denne ressource fil kan herefter medtages en managed assembly (*VsPackage*), som en binær ressource[[5]](#footnote-6).

Som det fremgår af Figur 26 er en *Command Table Definition* opdelt i en række sektioner og undersektioner.

I *CMD\_SECTION* defineres menuer, menu grupperinger, buttons (de egentlige kommandoer) og bitmaps (hvis der skal hvis billeder ved knapper). Der er hierarki i denne opbygning, buttons kan have bitmaps og buttons defineres i forhold til menuer. Menuer defineres i forhold til menu grupperinger.

I *CMDUSED\_SECTION* defineres de kommandoer fra andre pakker, som denne pakke benytter sig af.

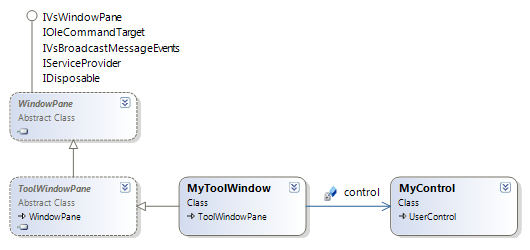
I *CMDPLACEMENT\_SECTION* defineres placering af menuer/menugrupper. Det er her muligt at placere egne grupper i eksisterende grupper og omvendt.

I *VISIBILITY\_SECTION* defineres det i hvilken kontekst de enkelte kommandoer skal være synlige. Fx giver det god mening at knapper der bruges til debugging ikke er tilgængelige når der skrives kildekode.

I *KEYBINDINGS\_SECTION* defineres shortcuts til kommandoer. Dette kunne fx være at CTRL+B skulle gøre tekst fed, som det kendes fra de fleste tekstbehandlings programmer.

#### ToolWindow implementering

For at implementere et nyt Tool Window i Visual Studio skal der implementeres to elementer, selve Tool Window’et og en kontrol der viser svarer til indholdet af dette Tool Window.



Figur - Tool Window klasser.

På Figur 27 vises de klasser der er brugt for at implementere et simpelt Tool Window. Som det fremgår af figuren findes i Managed Package Framework(MPF) en *ToolWindowPane* klasse der implementerer funktionaliteten til at integrere sig med Visual Studio som et Tool Window.

Igennem *WindowPane* klassen har MPF implementeret en række interfaces, som er nødvendige at forstå for at forstå implementeringen af Tool Windows. En række af disse interfaces er forklaret i tidligere afsnit og undlades derfor her.

*IVsWindowPane* interfacet gør det muligt at integrere et Tool Window i Visual Studio. Det er metoder til at oprette/lukke et Tool Window, samt metoder til at serialisere/deserialisere tilstanden af et Tool Window, således at et Tool Windows tilstand kan genskabes når Visual Studio starter op igen.

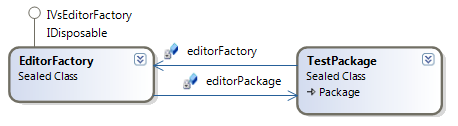
*IVsBroadcastMessageEvents* interfacet gør det muligt for et Tool Window at notificere andre elementer i Visual Studio omkring ændringer der er foretaget i et givent Tool Window. Omvendt giver det altså andre elementer muligheden for at lytte efter beskeder på dette Tool Window.

#### Editor Implementering

For at implementere en ny editor i Visual Studio er der brug for to ting, selve editor klassen (typisk en Windows.Forms klasse) og en factory klasse der kan oprette objekter af editor klassen.

En editor i Visual Studio bygger på separation af data og view (hhv. *DocData* og *DocView* i Visual Studio termer). Et view er endvidere opdelt i fysiske og logiske views. En editor kan tilbyde flere typer vinduer, et fysisk view fx en XML editor. Et enkelt fysisk view kan desuden tilbyde flere logiske view, fx HTML designerens *Design View* og *Source View*.

For at Visual Studio er i stand til at bede en pakke om det korrekte view er pakken nødt til at implementere en factory klasse, der overholder *IVsEditorFactory* interfacet, som vist på Figur 28.

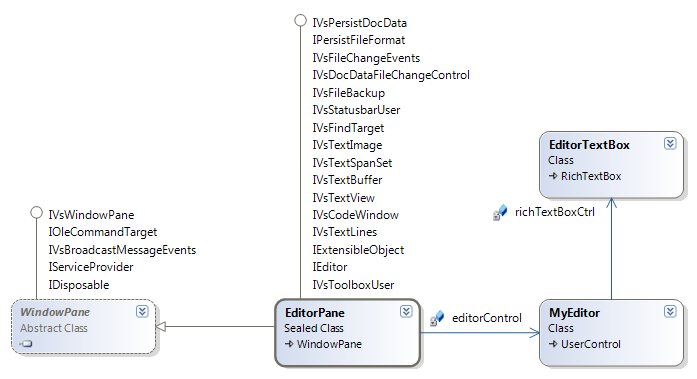


Figur - Editor Factory klasser.

*IVsEditorFactory* interfacet giver en pakke mulighed for at oprette/lukke/initialisere editor instanser, samt at koble logiske view til fysiske views.

Implementeringen af selve editoren er naturligvis afhængig af formålet med den pågældende editor, hvorvidt det er grafisk editor eller en tekstuel editor, antallet af fysiske view og herunder antallet af logiske views. Der er dog en række elementer som er fælles for alle editors, hvilket vil blive gennemgået i de kommende afsnit.

På Figur 29 vises de klasser der indgår i en typisk tekstuel editor. Der er behov minimum to klasser for at implementere en editor. En klasse der arver fra *WindowPane*. *WindowPane* klassen giver mulighed for at blive præsenteret som et vindue i Visual Studio. Denne klasse bør associeres med en *UserControl* klasse, som kan benyttes til at vise editor elementerne, fx en teksbox til en tekstuel editor eller som tegningsgrænseflade for en Visuel Editor.



Figur - Editor klasser

Som det fremgår af Figur 29 er der en lang række interfaces der skal implementeres for at implementere en editor. Et par interfaces er i forvejen implementeret i kraft af *WindowPane* klassen, men det er dog op til udvikleren at implementere de fleste interfaces selv.

##### Fælles interfaces

*IVsFileChangeEvents* interfacet giver en editor mulighed for at reagere på ændringer i filer eller filbiblioteker. Fx hvis brugeren sletter en fil i filsystemet, som en editor har åben kan denne editor lukke sig selv.

*IVsDocDataFileChangeControl* interfacet giver mulighed for at editor kan fortælle Visual Studio hvorvidt den skal notificeres igennem *IVsFileChangeEvents* for ændringer der er sket uden for den pågældende editor.

*IVsFileBackup* giver en editor mulighed for at indgå i Visual Studios backup rutine. Herved er det muligt for editoren at foretage backup af filer når Visual Studio mener at det er på tide.

*IVsStatusbarUser* interfacet giver en editor mulighed for at skrive i Status baren i bunden af Visual Studio. Det er altså muligt for en editor at notificere en bruger, hvis den fx er i gang med en tidskrævende operation.

*IVsToolboxUser* gør det muligt for en editor at notificere ejeren af toolbox elementer, når disse elementer benyttes i forbindelse med den pågældende editor.

##### Tekstuel Editor interfaces

*IVsPersistDocData* interfacet giver en editor mulighed for at persistere dets dokument. Det er både muligt at understøtte dokumenter der skal persisteres til filer og hukommelse.

*IVsFindTarget* interfacet giver mulighed for at understøtte Visual Studios find/replace funktionalitet.

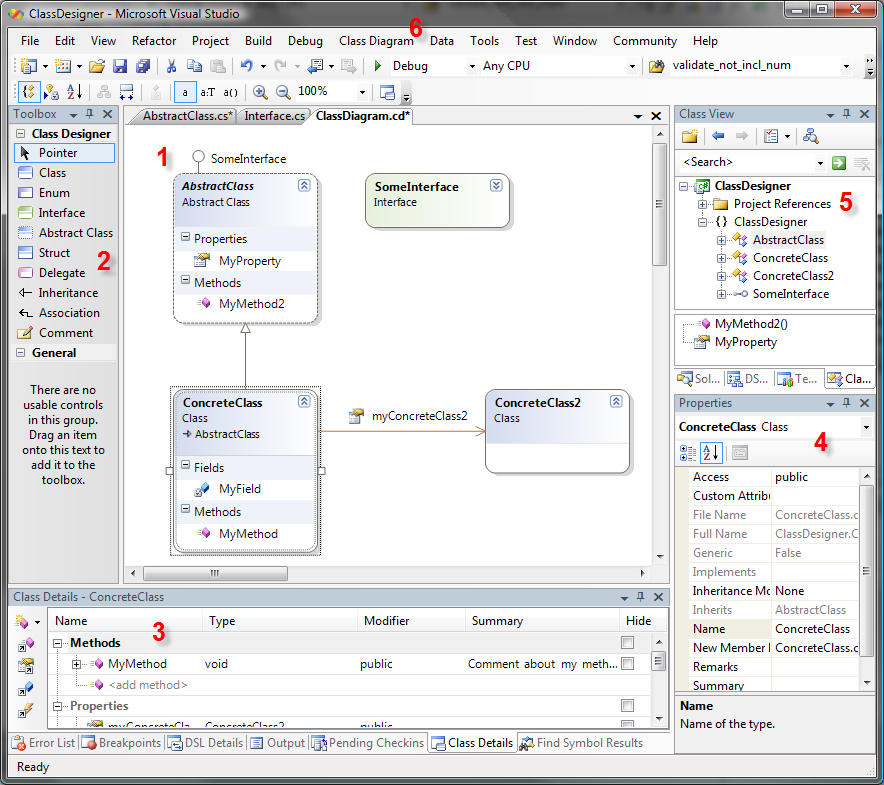
*IVsTextImage* interface, *IVsTextSpanSet*, *IVsTextBuffer,* *IVsTextView*, *IVsTextView, IVsCodeView*  og *IVsTextLines* gør det muligt at indgå i Visual Studios tekst servicer. En editor der overholder disse interfaces kan bl.a. tilbyde streng matchning, tekst buffer informationer og view information.

## Class Designer

*Class Designer* er et indbygget værktøj til model understøttet udvikling i Visual Studio. *Class Designer* er implementeret som en *VsPackage*. I det følgende afsnit vil de enkelte elementer i *Class Designer* blive gennemgået.

### Elementer

På Figur 30 er de forskellige elementer af *Class Designer* vist. Punkt 1 viser den grafiske editor, der er hovedelementet i *Class Designer*. *Class Designer* benytter en UML lignende grafisk notation for kode elementer. *Class Designer* benytter specielle *Class Diagram* filer (.cd). Disse er tilgængelige vha. ”*Add New Item”* dialog boksen i Visual Studio.



Figur - Class Designer.

Punkt 2 viser integrationen med Toolboxen i Visual Studio. Det er her muligt at trække/slippe disse elementer over på et *Class Diagram*. Med undtagelse af *Comment* elementet, er det kun muligt at hente elementer fra Toolboxen, der kan betragtes som objekter i objekt oriented sprog og forbindelser imellem dem.

Punkt 3 viser et Details view der er en del af *Class Designer*, i dette tilfælde detaljer for en klasse. Det er her muligt at for en overblik over klassens elementer, både de der fremgår af de grafiske notationer og dem der ikke gør.

I punkt 4 er der vist en integration i *Properties* viduet. For de elementer der er valgt i den grafiske editor er det muligt at rette egenskaberne her.

Punkt 5 viser *Class View*, som er en indbygget tekstuel visualisering af klasserne i et Visual Studio projekt. *Class Designer* kan arbejde sammen med *Class View* således at det er muligt at hente eksisterende kode elementer ind på et diagram.

### Udvidelse

Der findes desværre på nuværende tidspunkt ikke nogen API til at udvide *Class Designer[[6]](#footnote-7)*. Men udviklingsholdet bag *Class Designer* har bygget et værktøj til udvikling af Domæne Specifikke Sprog, DSL Tools, som bygger på samme framework som *Class Designer* er udviklet under. Dette værktøj giver bl.a. mulighed for at implementere modelleringsværktøjer til Visual Studio.

## DSL Tools

DSL Tools er et værktøj til Visual Studio, der giver mulighed for at bygge domæne specifikke værktøjer. Domæne specifikke værktøjer er det næste skridt i udviklingen af model understøttet udvikling, idet de typisk tillader en programmør at benytte grafiske notationer der er specifikke for problemdomænet.

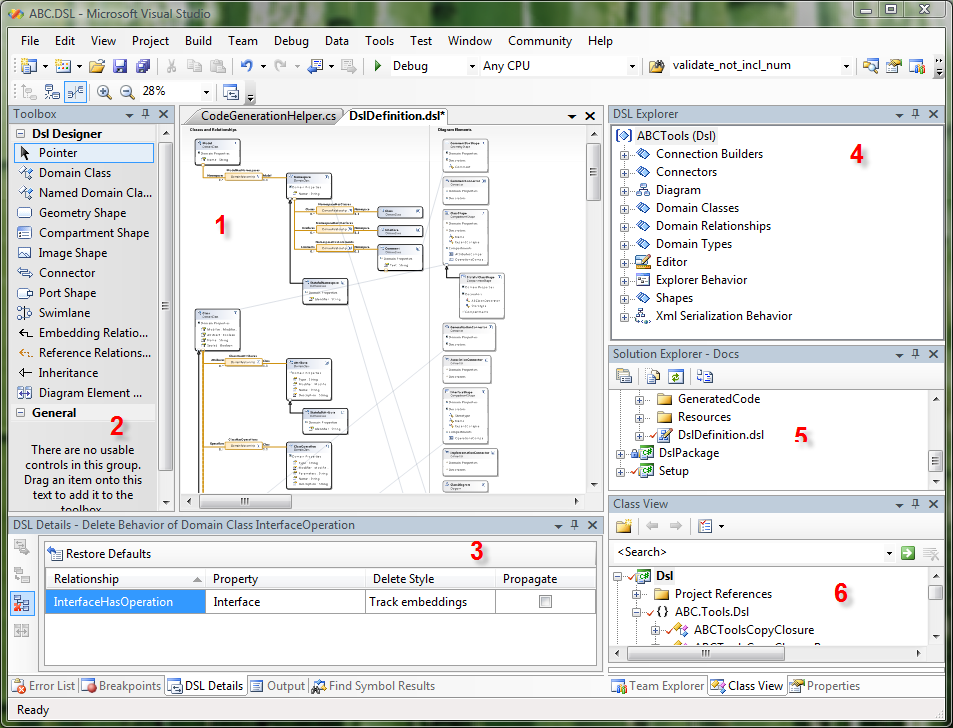
DSL Tools mest vigtige egenskab er at understøtte brugeren i at opbygge en domæne model, der repræsenterer brugerens problem domæne. Problemet domænet kan fx være modellering af applikationer til lyd tranformationer med dertilhørende grafiske notationer (filtre, forstærkere etc.) eller generiske applikationer med dertilhørende grafiske notationer (fx UML).

På Figur 31 er elementerne i DSL Tools vist. Punkt 1 viser den grafiske editor(DSL Designer), der benyttes til at opbygge domæne modellen for det domæne specifikke værktøj. *DSL Designer* benytter et specielt *DSL Definition* fil (.dsl), som er vist i punkt 5. *DSL Designer* udmærker sig at man ikke alene opbygger domæne modellen men at samtidig også opbygger de grafiske notationer til domæne modellen. Diagrammet er derfor opdelt i to svømmebaner, hvor domæne modellen er vist på den venstre side og de grafiske notationer er vist på den højre side.

Punkt 2 viser toolboxen med de forskellige elementer, der kan tilføjes til en *DSL Definition*. Elementerne kan opdeles i to klasser, de elementer der tilføjes til domæne modellen og de elementer der benyttes til den grafiske repræsentation.

Domæne modellen opbygges af to forskellige elementer, domæne klasser (*Domain Class og Named Domain Class*) og relationer (*Embedding Relationship* og *Reference Relationship*). En domæne klasse svarer til de elementer der findes i problemdomænet og relationer svarer til relationerne imellem disse.

De grafiske elementer er udvalg af prædefinerede grafiske element typer, der skulle dække de mest basale behov. Det fx muligt at repræsentere kasser, elipser og geometriske former vha. *Geometry Shape* og det er muligt at benytte egen grafik igennem *Image Shape*. Forbindelser imellem de grafiske elementer er repræsenteret af *Connector*, som giver mulighed for at lave streger i mellem figurer. For at forbinde de grafiske elementer med domæne modellen benyttes et *Diagram Element Map*.



Figur - DSL Tools.

Punkt 4 viser *DSL Explorer*, der giver tekstuel repræsentation af modellen. I forbindelse med *DSL Details* værktøjet (punkt 5) giver denne en fuld repræsentation af modellen.

De elementer der er vist i punkt kræver en nærmere forklaring, dette medvirker også til en bedre forståelse af DSL Tools.

## System Definition Model

*System Definition Model* (SDM) er modellen bagved Microsofts Whitehorse designere (se afsnittet ). For at forstå hvordan modellen kan udvides vil de dele af arkitekturen der kan udvides blive beskrevet først og herefter vil det blive beskrevet hvordan man udvider modellen.

### Arkitektur

SDM giver en række muligheder for at udvidelse. På Figur 32 er de elementer af SDM arkitekturen der kan udvides vist.

Figur - SDM Arkitektur elementer.

#### Objekter

Objekter i SDM svarer både til de logiske og fysiske elementer i *Distributed System Designers*. Et *System* objekt svarer til en deployment enhed. Et *System* kan bestå af andre SDM elementer fx applikationer eller andre *Systems*. Kommunikationen imellem *Systems* modelleres eksplicit som kommunikations relationer.

En *Resource* kan fx være en fil eller et certifikat. En *Resource* giver skal altid knyttes til et system og må ikke have afhængigheder af elementer uden for systemet.

En *Endpoint* svarer til et kommunikations interface på et system. Ved at benytte *Endpoint* er det muligt at lave kommunikationskanaler imellem systemer. Da en *Resource* ikke må have afhængigheder uden for et system skal der oprettes kommunikations kanaler, hvis en *Ressource* skal benytte services ved et andet system.

#### Relationer

Relationer i SDM bruges til definere relationer imellem *System*, *Resources* og *Endpoints*. Alle relationer går altid fra ét objekt til et andet, dvs. de er ikke bidirektionelle.

En *Containment* relation beskriver ejerskab, fx *System* S ejer *Ressource* R. En *Hosting* relation beskriver eksekvering miljø fx at en web applikation *System* afvikles under et web server *System.*

En *Communication* relationer beskriver en kommunikation imellem *Systems*. Hertil benyttes kompatible *Endpoints*. En *Communication* relation kan forekommen indenfor ét *System*.

En *Reference* relation beskriver afhængigheder imellem *Resources*. En *Reference* relation er kun kendt for den *Ressource* der benytter en anden *Resource*, dvs. ikke for den *Ressource* der bliver brugt.

En *Delegation* relation gør *Endpoint* tilgængelige for et *System* der er en del af et større *System*, fx hvis *System A* er en del af *System B* og *System A* har et *Endpoint* så kan dette *Endpoint* gøres tilgængeligt på *System B*.

#### Metadata

For at gøre det muligt at genere konfigurationer og foretage validering af de modellerede systemer kan der tilføjes en række metadata elementer til SDM.

*Settings* er gør det muligt at modellere konfigurations elementer. *Settings* er simple navngivne værdier. *Flows* benyttes ligesom *Settings* til at modellere konfigurations elementer, men i modsætning til *Settings* der er simple statiske værdier, så giver muligt mulighed for dynamiske værdier. Et *Flow* sender værdier igennem objekter og relationer og giver hermed mulighed for at lave transformationer af værdierne.

*Constraint* metadata gør det muligt at opstille krav for objekter eller relationer i SDM system. Dette giver mulighed for at validere at et samlet system opfylder disse krav og at advare brugeren hvis det ikke er tilfældet.

#### Udvidelse

For at udvide SDM skal der følges en temmelig kompleks proces. Processen og de involverede værktøjer er illustreret på Figur 33.

Figur - SDM udvidelses proces.

Udgangspunktet for enhver udvidelse af SDM er en speciel XML fil med endelsen *.sdm*. En *.sdm* kan indeholde både objekter, relationer og metadata deklarationer. En *.sdm* fil kompileres til et *.sdmdocument* vha. *SDM kompileren*(sdmc.exe), hvilket sker bl.a. for at verificere at filen er korrekt XML, overholder XML skemaet for SDM definitioner, overholder *Constraints* og *Flows*.

Hvis en *.sdm* fil benytter specielle typer skal der genereres såkaldte *Manager* for disse typer, hvilket gøres med *SDM Manager Generator* værktøjet (sdmg.exe). Dette producerer en fil med C# kode for manager klasserne, som skal kompileres med C# kompileren (csc.exe).

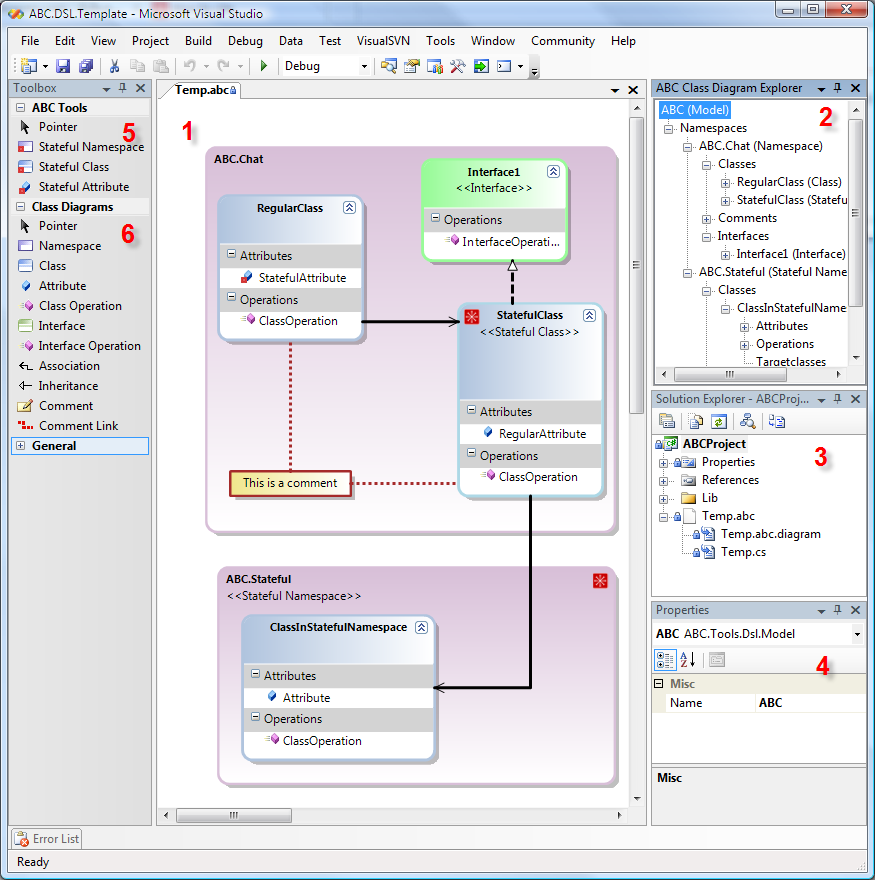
Hvis et *.sdm* dokument indeholder objekter benyttes *Prototype Generator* værktøjet (ProtoGen.exe) til at generere prototyper for objekterne. Prototyperne benyttes af Visual Studio til bl.a. at udfylde toolboxen med de grafiske notationer for de nye objekter.

# ABC Tools

Dette afsnit vil introducere ABC Tools, der er et værktøj der kan hjælpe udviklere med at udvikle applikationer til ABC Frameworket. I de følgende afsnit vil selve værktøjet og objekt modellen for værktøjet blive gennemgået.

## Introduktion

ABC Tools er implementeret som en *VsPackage* til Visual Studio, der giver udvikleren de samme muligheder som den indbyggede *Class Designer*. ABC Tools er vist i brug på Figur 34.



Figur - ABC Tools

Ud over at tilbyde udvikleren samme muligheder, som den eksisterende *Class Designer* tilbyder ABC Tools også mulighed for at eksplisit at designe følg.

* Statefull Components (attributter)
* Stafefull Classes
* Statefull Namespaces

De enkelte elementer som ABC Tools tilføjer til Visual Studio er nummererede på Figur 34. I de følgende afsnit vil de enkelte elementer i ABC Tools blive introduceret.

### ABC Class Designer

Punkt 1 er det grafiske modellering værktøj der minder om den grafiske modellering udviklere allerede kender fra *Class Designer*. Dette grafiske modelerings værktøj kaldes *ABC Class Designer*, og er implementeret som en grafisk editor i Visual Studio.

I modsætning til *Class Designer* er er namespaces eksplicitte design elementer, idet det er ønskeligt at kunne modellere statefull namespaces. Som det yderligere fremgår af figuren så er benyttes der specialle ABC annotationer på de grafiske elementer ved både statefull namespace, classes og attributter.

### ABC Class Diagram Explorer

Punkt 2 er et Tool Window der er unikt for ABC Tools. Dette Tool Window kaldes *ABC Class Diagram Exlplorer* og det giver en træstruktur visning over elementer på et *ABC Class Diagram*(se nedenfor). Ved at integrere sig i Properties vinduet (punkt 4) er det muligt for udvikleren at ændre egenskaber for de valgte elementer, såsom klasser, namespaces, metoder etc.

*ABC Class Diagram Explorer* er opbygget hierarkisk svarende til den underliggende struktur af koden. Hierarkiet starter ved namespaces på 1. Niveau, klasser, interfaces og kommentarer på næste niveau og operationer, attributter og associationer på næste niveau.

### ABC Class Diagram

Punkt 3 viser *Solution Explorer*, det medfølgende Tool Window, der giver et overblik over projekter og filer i en solution. Som det fremgår er der tilføjer ABC Tools en ny filtype med endelsen .abc. Dette er et *ABC Class Diagram*.

Et *ABC Class Diagram* er fundamentet for ABC Tools ligesom *Class Diagram* (.cd) er fundamentet for den indbyggede *Class Designer*. Et *ABC Class Diagram* kan tilføjes til et Visual Studio projekt ved at benytte den File -> Add -> New Item.

Et *ABC Class Diagram* benytter desuden en speciel kode generator til at genere kildekode (C# eller VB) ud fra modellen. Kode genereringen er forklaret nærmere i afsnittet **Error! Reference source not found.**.

### Toolbox

Punkt 5 og 6 viser ABC Tools integration i Toolboxen i Visual Studio. Det er herfra muligt at trække/slippe elementer over på et *ABC Class Diagram*. Punkt 6 viser de samme elementer som man finder i den indbyggede *Class Designer* og punkt 5 viser de elementer der er specifikke for ABC Tools.

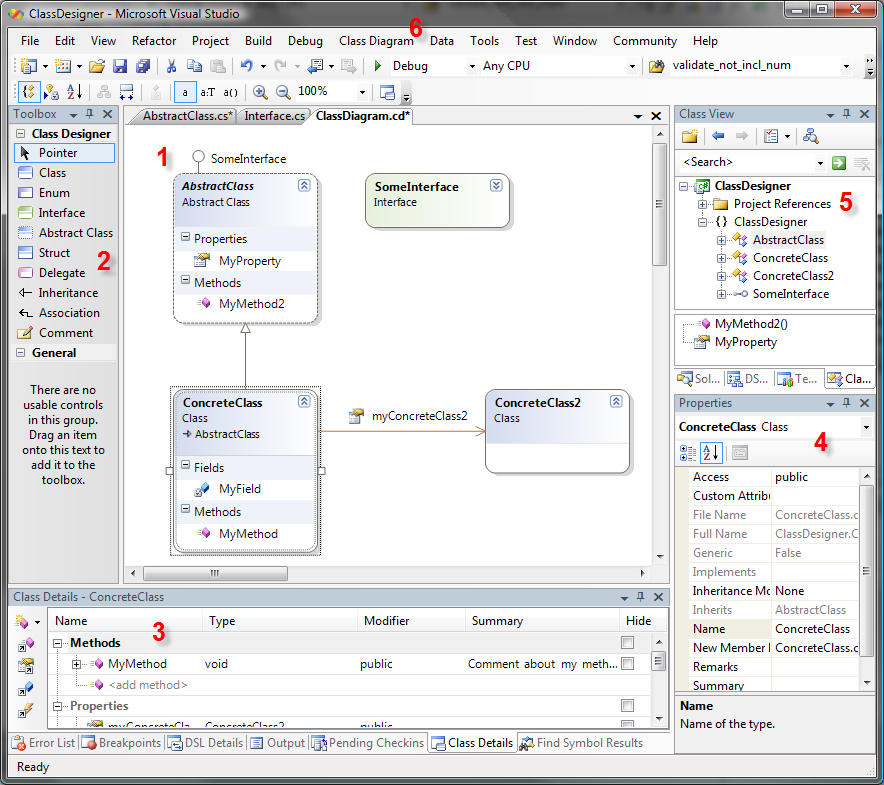
## Objekt Model

I dette afsnit vil den bagvedliggende objekt model for ABC Tools blive gennemgået. Modellen er bygget vha. DSL Tools (se **Class Designer**

*Class Designer* er et indbygget værktøj til model understøttet udvikling i Visual Studio. *Class Designer* er implementeret som en *VsPackage*. I det følgende afsnit vil de enkelte elementer i *Class Designer* blive gennemgået.

### Elementer

På Figur 30 er de forskellige elementer af *Class Designer* vist. Punkt 1 viser den grafiske editor, der er hovedelementet i *Class Designer*. *Class Designer* benytter en UML lignende grafisk notation for kode elementer. *Class Designer* benytter specielle *Class Diagram* filer (.cd). Disse er tilgængelige vha. ”*Add New Item”* dialog boksen i Visual Studio.



Figur 30 - Class Designer.

Punkt 2 viser integrationen med Toolboxen i Visual Studio. Det er her muligt at trække/slippe disse elementer over på et *Class Diagram*. Med undtagelse af *Comment* elementet, er det kun muligt at hente elementer fra Toolboxen, der kan betragtes som objekter i objekt oriented sprog og forbindelser imellem dem.

Punkt 3 viser et Details view der er en del af *Class Designer*, i dette tilfælde detaljer for en klasse. Det er her muligt at for en overblik over klassens elementer, både de der fremgår af de grafiske notationer og dem der ikke gør.

I punkt 4 er der vist en integration i *Properties* viduet. For de elementer der er valgt i den grafiske editor er det muligt at rette egenskaberne her.

Punkt 5 viser *Class View*, som er en indbygget tekstuel visualisering af klasserne i et Visual Studio projekt. *Class Designer* kan arbejde sammen med *Class View* således at det er muligt at hente eksisterende kode elementer ind på et diagram.

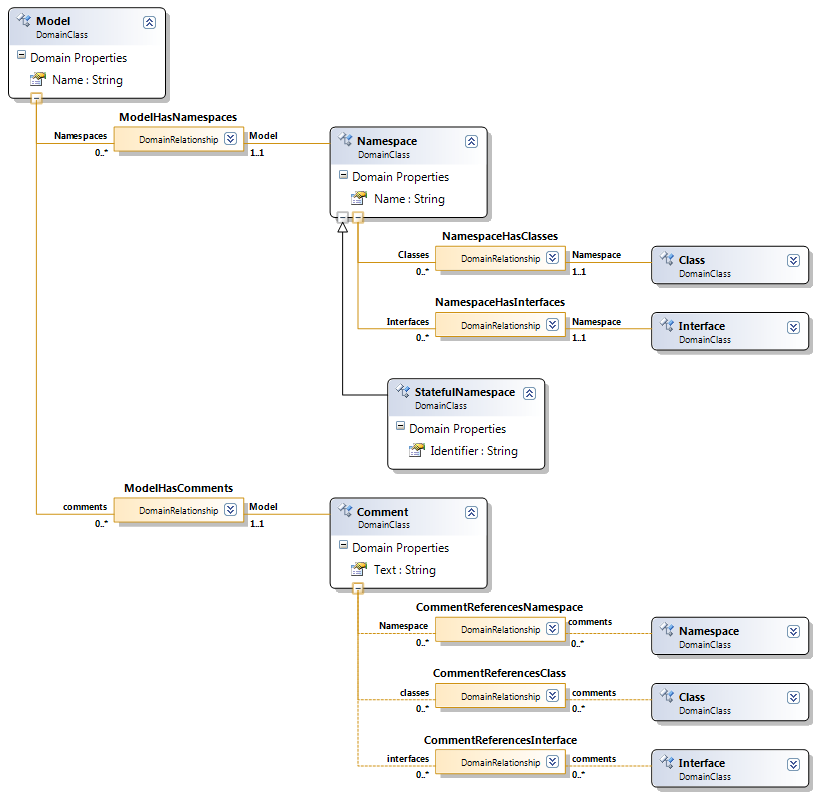
### Udvidelse

Der findes desværre ikke nogen API til at udvide *Class Designer*. Men udviklingsholdet bag *Class Designer* har bygget et værktøj til udvikling af Domæne Specifikke Sprog, DSL Tools, som bygger på samme framework som *Class Designer* er udviklet under. Dette værktøj giver bl.a. mulighed for at implementere modelleringsværktøjer til Visual Studio.

DSL Tools).

Modellen er opbygget i en hierarkisk struktur med ét rod element, *Model* (se Figur 35). *Model* har ikke nogen reel påvirkning på modellen foruden at give ét indgangspunkt til modellen. Modellen har dog et navn, men dette er udelukkende til kosmetiske formål.

En model består af en række namespaces og hvert namespace har præcist én model. Et namespace har et navn, som skal være unikt. Årsagen til at navnet må et namespace skal være unikt er at domæneklassen *StatefullNamespace* der nedarver fra *NameSpace* kræver at et namespace er unikt for at ABC frameworket kan afgøre hvorvidt klasser tilhører et namespace, der er markeret med *StatefulNamespace* attributten. Det er nødvendigt at modellere dette som en speciel domæne klasser for at kunne binde et eksplicit design element til den.



Figur 35 - ABC Tools objekt model del 1.

Domæneklassen *StatefulNamespace* har en identifier attribut, der svarer til den identifier ABC frameworket skal benytte for at gøre namespacet stateful.

*Namespace* domæne objektet har tre relationer. Den første relation er til domæne objektet *Class*, der svarer til en klasse. Et namespace kan have 0 til uendeligt mange klasser, men klasse skal have et namespace. Det er altså ikke muligt at benytte sig af default namespacet, som fx er indbygget i .NET frameworket.

Den næste relation er til domæne klassen *Interface*. Ligesom *Class* er det ikke muligt at have et interface, der ikke er tilknyttet et namespace.

Foruden namespaces har en model også en række kommentarer i form af domæne klassen *Comment*. En model kan indholde et vilkårligt antal kommentarer. En kommentar kan knyttes til tre forskellige domæne klasser, *Namespace*, *Class* og *Interface*. Fælles for disse tre domæne klasser er at de alle kan have et vilkårligt antal kommentarer tilknyttet.

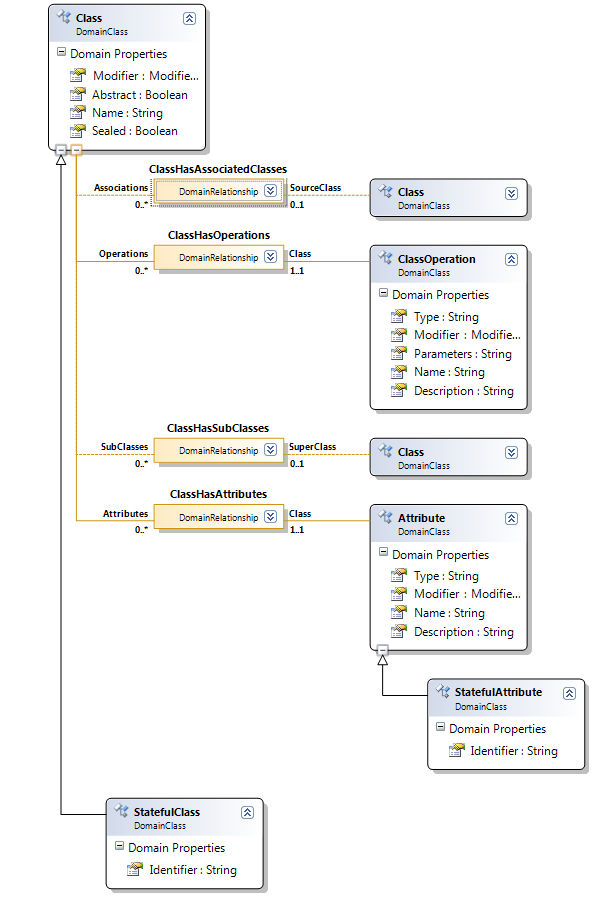
På Figur 36 er detaljerne for domæne klassen *Class* illustreret. En *Class* er har fire egenskaber; navn, modifier (eg. public/private etc.), hvorvidt klassen er abstrakt eller ej, og hvorvidt klassen er sealed eller ej. *Class* klassen har én subklasse, *StatefulClass*, der svarer til en klasse der er annoteret med *StatefulClass* attributten. Det er igen nødvendigt at have en domæne klasse til dette for at kunne lave et eksplicit designelement.

En *Class* har desuden fire associationer til domæne klasser. Den første association er til *Attribute* domæne klassen, der svarer til en medlemsvariabel. En klasse kan et vilkårligt antal medlemsvariable, men en medlemsvariabel tilhører altid én specifik klasse. En medlemsvariabel beskrives ved fire egenskaber; navn, type, *Modifier* (public/private etc.) og beskrivelse. Som det fremgår af Figur 36 har *Attribute* også en subklasse, *StatefulAttribute*, der giver mulighed for eksplicit design af *StatefulAttribute*.

Den anden association er til *ClassOperation*, der svarer til en metode. Ligesom *Attributes* kan en klasse have et vilkårligt antal metoder og en metode har altid kun én klasse. En *ClassOperation* er beskrevet ved fem forskellige egenskaber; navn, beskrivelse, type, Modifier (private/public etc.) og Parameters, som en er liste af strenge.

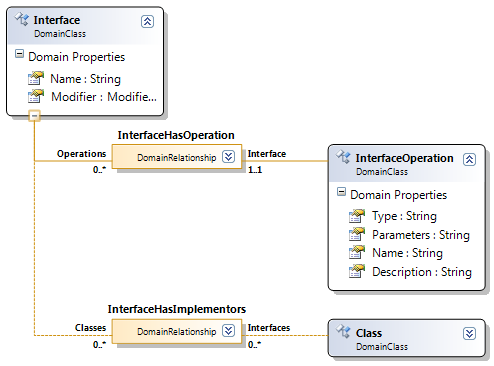
Den tredje association er til *Class* selv. Denne association udtrykker forholdet imellem super og subklasser. En klasse kan have højst én superklasse, men kan have en vilkårligt antal subklasser.

Den fjerde association er til *Class* selv. Dette association udtrykker muligheden for at modelere medlemsvariable som associationer. Her er det muligt for en *Class* at have et vilkårligt antal associationer og at indgå som kilde til en anden klasses associationer.



Figur 36 - ABC Tools objekt model del 2.

På Figur 37 er den sidste del af modellen vist. Det første element er domæneklassen *Interface*, der svarer til en interface. Et interface er beskrevet ved to egenskaber; navn og *Modifier* (public/private etc.).



Figur 37 - ABC Tools objekt model del 3.

*Interface* klassen har to associationer til andre domæne klasser. Den første association er til domæne klassen *Interface Operation*, der svarer en metode defineret i et interface. *InterfaceOperation* beskrives med de samme egenskaber som *ClassOperation*, dog har den ikke en *Modifier*, idet metoder på et interface altid er public.

Den anden association er til *Class.* Denne association beskriver forholdet imellem et interface og de klasser de implementerer dette. Et interfacet kan implementeres af vilkårligt mange klasser og en klasse kan implementere vilkårligt mange interfaces.

# Evaluering

Dette afsnit introducer evalueringen af ABC Tools. ABC Tools er blevet evalueret efter

# Diskussion

Dette afsnit introducerer en diskussion af de erfaringer der er opnået i forbindelse med udviklingen af ABC Tools, samt resultaterne af evalueringen.

# Konklusion

# Bibliografi

**Bardram Jacob E and Bunde-Pedersen Jonathan** A2BC - An Agent Approach to Activity-Based Computing [Report] / Centre for Pervasive Helthcare, Department of Computer Science ; University of Aarhus. - 2004.

**Bardram Jacob E., Bunde-Pedersen Jonathan and Mogensen Martin** Design of the ABC Framework, Version 4 [Report] / Department of Computer Science ; University of Aarhus. - 2005. - http://www.daimi.au.dk/~bardram/pvc/resources/abc.v4.design.pdf.

**Bardram Jakob, Christensen Henrik Bærbak and Bunde-Pedersen Jonathan** How to program using ABC [Online] // Activity-Based-Computing.org. - April 28, 2004. - November 2, 2006. - http://www.activity-based-computing.org/programming.html.

**Gibson Bill and Torone Alex** Visual Studio 2005 Team System: Designing Distributed Systems for Deployment [Online] // http://msdn.microsoft.com. - Microsoft Corporation, May, 2004. - August 16, 2006. - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsts-arch.asp.

**Hall Richard S. and Cervantes Humberto** Gravity: supporting dynamically available services in client-side applications [Conference] // Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. - Helsinki, Finland : ACM Press, 2003. - vol. I. - pp. 379 - 382.

**MacKenzie Matthew C. [et al.]** Reference Model for Service Oriented Architecture 1.0 [Report]. - [s.l.] : OASIS Open, 2006.

**Soroker Danny [et al.]** Pegboard: A Framework for Developing Mobile Applications [Conference] // MobiSys´06. - Uppsale, Sweden : ACM Press, 2006.

1. Implementering af ACD applikationer betyder at der laves/tilføjes et Visual Studio projekt. [↑](#footnote-ref-2)
2. Virtual Private Network [↑](#footnote-ref-3)
3. Hyper Text Transfer Protocol. [↑](#footnote-ref-4)
4. Simple Object Access Protocol. [↑](#footnote-ref-5)
5. For at Visual Studio kan finde resourcen med menupunkter skal ressourcen der indeholder .cto filen kaldes CTMENU. [↑](#footnote-ref-6)
6. Som forklaret på Microsofts forum (http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=816047&SiteID=1) [↑](#footnote-ref-7)