Frameworking - Strukturen und Verbindungen in einem Framework

Strukturen und Verbindungen in einem Framework

by Walter Kriha

Frameworking

by Walter Kriha

Published 19.April 1997

Table of Contents

List of Tables

Eine Bitte an die Leser dieses Buches:

Das Buch beruht auf praktischen Erfahrungen aus einem grösseren Framework Projekt und ist daher zwangsläufig einseitig. Es hat momentan den Status eines „first draft" und enthält mit Sicherheit etliche Teile die entweder falsch, unvollständig oder unverständlich sind.

Ich bitte daher alle Leser um einen Feedback in jeglicher Form. Schön wären auch Erweiterungen oder das Einbringen eigener Erfahrung.

Die Visualisierung komplexer Strukturen und Zusammenhänge bereitete mir bereits während der praktischen Implementation die grössten Probleme. Leider hängt gerade davon die Kommunikation im Team sowie zu Anwendern ab. Gegenwärtig versuche ich Ideen hierzu zu sammeln und wäre froh über Beiträge.

Chapter 1. MANAGEMENT SUMMARY

Jede Software lässt sich nach den verschiedensten Kriterien strukturieren, z.B. nach logischen Abstraktionen, physischen Ableitungsverhältnissen, Runtime-, Compile- und Linktime Abhängigkeiten, Generierung und Maintenance (Build, install) sowie nach den Erweiterungsmöglichkeiten und der Organisation des Source Codes.

Diese Strukturen stehen teilweise im Konflikt miteinander, z.B. führt eine logische Ableitungshierarchie durch die damit verbundenen physischen Kopplungen zu Wartungsproblemen und langen Generierzeiten.

Ein Grund des Scheiterns vieler Software Projekte – auch und gerade objektorientierter Applikationen – liegt in der Nichtbeachtung dieser Strukturen und ihrer Wechselwirkungen. Fatal an diesen Strukturen ist zudem, dass sie im Source Code unsichtbar bleiben und dass Strukturprobleme typischerweise erst zu einem relativ späten Zeitpunkt offenkundig werden.

Dieses Buch will Softwarestrukturen offenlegen, erklären, ihre Wechselwirkungen bestimmen und Verfahren angeben, wie sie besser visualisiert werden können.

Ein Framework eignet sich besonders gut zur Untersuchung dieser Strukturen denn Frameworks zielen genau auf den oben beschriebenen Mangel normaler Applikationen: Flexibilität und Erweiterbarkeit durch strukturelle Mechanismen. Ein Framework enthält dieselben Strukturen wie jede andere Software, macht sie aber gleichzeitig explizit, sichtbar und programmtechnisch greifbar als Voraussetzung für Erweiterbarkeit. Ein Framework enthält somit Code der sich seiner Austauschbarkeit bewusst sein muss. Frameworks sind daher essentiell reflektive, dynamische Systeme.

Der typische Ansatz die Komplexität einer Software durch Isolierung von Componenten und Packages zu reduzieren funktioniert gerade in einem Framework so einfach nicht: Innerhalb eines Frameworks SOLLEN Klassen und Componenten miteinander agieren um eine Lösung zu erreichen. Dieses Paper zeigt Wege auf, Kooperation trotz Separation zu erreichen.

Wissen um die technischen Aspekte eines Frameworks ist eine notwendige Voraussetzung für erfolgreiche Projekte. Trotzdem scheitern Projekte regelmässig trotz technischen Wissens. In der praktischen Erfahrung vermischen sich technische Strukturen und soziale Strukturen der Organisation der Arbeit zu einem komplexen Gebilde. Deshalb – nicht zuletzt aus eigener schmerzlicher Erfahrung – versucht diese Arbeit auch die soziale Strukturen der Erstellung und Verwendung eines Frameworks zu bestimmen. Dabei geht sie von drei Thesen aus:

1. Dass unsere technischen Denkbilder und Modelle sowohl von den sie begleitenden sozialen Strukturen bestimmt sind als auch umgekehrt neue technische Modelle neue Formen der sozialen Organisation bedingen.

2. Komplexe Projekte häufig an den sozialen Strukturen scheitern, die den technischen entweder nicht angemessen sind oder ihnen entgegenarbeiten.

3. Dass wir technische Interfaces auf minimale Berührungsflächen hin entwerfen sollten. Soziale Interfaces sich jedoch zu ihnen orthogonal oder netzartig verhalten und auf maximale Berührungspunkte hin organisiert sein sollten.

Der technische Teil der Arbeit zeigt ganz konkrete Lösungen für flexible, dynamische und adaptive Systeme mit der Betonung auf der möglichst vollständigen Betrachtung des Lebenszyklus eines Produktes und der dynamischen Abhängigkeiten zwischen seinen Teilaspekten.

Im sozialen Teil werden ebenfalls konkrete Lösungen für die Enführung neuer Technologie, die Organisation der Arbeit in komplexen Systemen und die Auswahl neuer bzw. der Umgang mit bestehenden Mitarbeitern aufgezeigt. Wenn dabei Kriterien für Kündigung und Selektion fallen dann nicht weil der Autor der Meinung ist, dass eine gewisse Brutalität für die Einführung neuer Technologie unbedingt nötig und wünschenswert ist sondern weil Kündigungen häufig der letzte Ausweg aus völlig festgefahrenen Denkbildern und falschen Organisationsformen sind. Es gilt die Ursachen dafür zu finden dass technische Modelle religiösen Charakter erhalten, die Mitarbeiter völlig unflexibel werden und wir den Produktionsprozess von Software auf eine Weise organisieren die Konfrontation und Politik statt bessere Produkte erzeugt.

Die Überlagerung technischer und sozialer Strukturen bestimmt letztlich den Erfolg eines Projektes.

Um diesen Zusammenhang deutlich zu machen trägt die Arbeit das Kunstwort „Frameworking" im Titel.

Ein Wort zu den Voraussetzungen:

Dies ist kein Buch über Design Pattern. Im Gegenteil, es setzt die Kenntnis von Design Pattern schlicht voraus. Frameworktechnologie ist ohne die technischen und kommunikativen Hilfen der Design Pattern kaum möglich.

Dies ist auch kein Buch zum C++ oder Java lernen. Wer die Standardwerke zu C++ nicht kennt, speziell die Problematiken um die physischen Abhängigkeiten und die Behandlung von Interface und Class dem werden die Schwierigkeiten dynamischer Lösungen mit C++ unverständlich bleiben..

Das gleiche gilt für Java. Kenntnis von JDK1.1, Class Loading, Reflection etc. wird vorausgesetzt.

Eine Basiskenntnis von distributed object Technologien wie CORBA oder DCOM ist ebenfalls sehr nützlich.

Nicht zuletzt soll dieses Buch als Basis für eine Serie von Systor internen Workshops zur Anwendung von Design Patterns und Frameworktechnologie dienen.

Chapter 2. WIESO EIN FRAMEWORK?

Selten fängt ein Projekt mit der Zielsetzung „Wir bauen ein Framework" an. Oft steht die Erkenntnis, dass ein Framework Desing nötig gewesen wäre erst am Ende eines Projektes fest. Wenn der erste Prototyp oder die erste Auslieferung zeigen, dass man wesentliche Gesichtspunkte wie Flexibilität und Erweiterbarkeit zu wenig bedacht hat. Üblicherweise ist dann bereits eine Menge Code entwickelt worden und das Projektteam steht mit dem Rücken zur Wand, da jede Änderung oder Erweiterung einen quadratischen Aufwand bedeutet.

Wir wollen am Beispiel eines Businessprojekts die Entwicklung zum Framework nachvollziehen – das „Framework-Denken" verdeutlichen. (Das Beispiel beruht auf einem vom Autor selbst entwickelten Framework für Document Imaging und Workflow).

Das Beispiel schliesst ganz bewusst auch die sozialen und organisatorischen Strukturen einer Framework-Entwicklung ein, die nach eigener Erfahrung einen grösseren Einfluss auf den Erfolg von Projekten haben als die technischen Strukturen.

Ausgangslage: Ein existierendes, konventionelles Softwareprodukt

Firma X hat ein Produkt „OLDSYS", eine Kombination aus Software und Hardware für Belegverarbeitung in verschiedenen Branchen.

Aufgaben von OLDSYS:

- Scannen von Belegmaterial. OCR Behandlung der Belege mit automatischer Validierung der Ergebnisse (Prüflogik).
- Anschliessende manuelle Nachbearbeitung der Ergebnisse durch Datentypisten.
- Weitergabe der Resultate an Mainframes bzw. Archivierung.

Das Paket ist sehr erfolgreich, hat aber einige Eigenschaften, die bereits jetzt oder in absehbarer Zukunft zu enormen Problemen führen werden, nichts zuletzt auf Grund der gestiegenen Kundenzahl. Das Kontrolling der Produkte findet kaum statt. Es ist nicht transparent welche Produktfehler in welchem Umfang durch die Servicegruppe abgefangen werden. Entwicklern gelingt es immer wieder, Unzulänglichkeiten der Software auf die Servicegruppen abzuschieben. Die Servicegruppen sind hoch motiviert und haben ein ausgesprochenes „Just do it" Denken.

Eigenschaften von OLDSYS:

- MS-DOS basiertes Toolkit mit branchenspezifischen Applikationen, eigenes Memory Management, eigene ISAM Datenbank, eigenes GUI unterstützt durch eigene Keyboard und Video Driver. Eigene Treiber zum Anschluss von Peripherie (Scanner)
- Modulkonzept: Module zur Verarbeitung werden durch einen eigenen Lademechanismus zur Laufzeit geladen und zu bestimmten Zeitpunkten vom Kernel aufgerufen. Die Module werden durch Einträge in ein Konfigurationsfile bestimmt.
- Flexibilität durch Konfigurationsfiles: Der gesamte Ablauf sowie die Logik von Belegprüfungen werden durch Konfigurationsfiles bestimmt.
- OLDSYS ist in C und teilweise in Assembler programmiert. Zentrale Datenstrukturen sind in Dokumenten beschrieben und werden in allen Modulen gekannt und verwendet.
- Neue Peripherie kann integriert werden, z.B. wenn neue Scanner Typen kommen. Integriert wird neue Peripherie im wesentlichen im Modell des ersten unterstützten Scanners.
- OLDSYS ist sehr performant, nicht zuletzt weil es ein eigenes GUI besitzt und direkt auf DOS bzw. unter

Umgehung von DOS arbeitet.

- OLDSYS ist kostengünstig, da die Konkurrenz momentan noch grössere und damit teurere Systeme zum selben Zweck einsetzt.

Trotz des Erfolges sieht das Management einige technische und soziale Probleme, die zum Wunsch nach einem Redesign von OLDSYS führen.

Problem mit OLDSYS aus der Management Sicht:

- OLDSYS wurde im wesentlichen von einer Person entwickelt, die das Unternehmen verlassen wird.
- Kenntnisse zu den Interna sind nur schwer erhältlich obwohl der Code ursprünglich sauber und übersichtlich programmiert war. Nach einigen Erweiterungen sind die Abhängigkeiten der Funktionen und Datenstrukturen nicht mehr klar. Änderungen oder Erweiterungen verursachen obskure Fehler in anderen Modulen.
- Die Kunden wollen Windows oder OS/2 bzw. Motif Unterstützung.
- Die Wartung/Entwicklung der eigenen Treiber multipliziert sich pro Plattform.
- Der Konfigurationsaufwand „skaliert" nicht, d.h. je mehr Kunden umso grösserer Konfigurationsaufwand da mit „copy and paste" gearbeitet wird. Neue Hardware führt zu einem überproportionalen Aufwand an Konfiguration.
- Neue Mitarbeiter tun sich sehr schwer mit der Konfiguration. Es entstehen viele Fehler beim Kunden. Der Gewinn pro zusätzlichem Kunden ist abnehmend durch gestiegenen Serviceaufwand.
- Änderungen dauern immer länger und führen zu obskuren Nebeneffekten. Das System selbst wird immer grösser. QA-Probleme sind vorhanden, z.B. bezüglich des Release Managements.
- Es sind keine automatischen Updates möglich, da die Änderungen von Hand durchgeführt werden. Es gibt immer mehr Speziallösungen für einzelne Kunden. Die alte Zweiteilung des Systems in Toolkit und branchenspezifische Applikationen existiert kaum noch. Neue kundenspezifische Applikationen entstehen durch „cut and paste" aus vorhandenen. Dies führt dazu, dass bei Fehlerbehebungen unzählige Module geändert werden müssen. Bei Sonderwünschen muss meistens so verfahren werden, dass sie Teil der System oder Applikationsbasis werden und somit VON ALLEN KUNDEN mitgeschleppt werden müssen.

Konventionelle Software degeneriert im Laufe der Zeit. Durch notwendige Änderungen und Erweiterungen wird die ursprünglich saubere Struktur zerstört.

Alles dies veranlasst das Management über ein neues Produkt „NEWSYS" nachzudenken. Zwei grundlegende Richtungen bestehen:

- Weiterentwicklung des bestehenden Systems
- Komplette Neuentwicklung

Die Entwickler des alten Systems bevorzugen den Gedanken der Weiterentwicklung. Die Geschäftsleitung – beunruhigt durch die bereits deutlich längeren Releasezyklen des alten Systems – entscheidet sich für die Neuentwicklung mit Integration von Komponenten des alten System. Dazu wird ein völlig neues Team angeworben und in der Firma installiert. Die Mitglieder des neuen Teams verfügen über keine Branchen- oder Domainkenntnisse und sehr unterschiedliche OO-Kenntnisse.

In einigen anfänglichen Meetings wird versucht mit einer Gruppe von älteren und neueren Mitarbeitern ein Anforderungsprofile für NEWSYS zu entwickeln. Die Vorschläge der älterne Mitarbeiter sind sehr konkret und detailiert, beziehen sich aber im wesentlichen auf inkrementelle Verbesserungen des bestehenden Systems. Der Prozess schläft nach einigen Monaten ein und wird anschliessend fast selbständig im Team von NEWSYS

durchgeführt. Bereits nach kurzer Zeit tritt eine Isolierung des NEWSYS Teams auf, die sich im Laufe der Zeit noch verstärkt, nicht zuletzt durch den Entschluss ein Framework zu bauen. Die Applikationsgruppen stehen dem Projekt zu Anfang wohlwollend gegenüber, nicht zuletzt auf Grund persönlicher und technischer Probleme mit der OLDSYS Gruppe.

Anforderungen an „NEWSYS":

- Erschliessen neuer Märkte durch Ausweitung des Dokumentenmodells
- Qualitätssicherung der Produkte
- Kompatibilität mit den alten Datenstrukturen, d.h. beim Kunden sollen OLDSYS und NEWSYS Versionen zusammen laufen können, z.B. die gleichen ISAM Datenbanken verwenden.
- Einführung von Mandantenfähigkeit.
- Plattformunabhängigkeit: OS/2, WIN32, UNIX
- Internationalisierbar
- Compiler-unabhängig
- Projektdauer ca. 1 Jahr mit frühen Prototypen und inkrementellen ersten Teilversionen (Die Firma hatte pikanterweise bereits lange vor Projektbeginn Dummys des neuen Produktes auf Messen gezeigt)
- Vorgaben: Entwicklung in C++ („damit der Umstieg leichter fällt", Verwendung eines GUI-Toolkits für plattformunabhängige User Interfaces.)

Das neue Projekt NEWSYS

Eine neue Sprache entsteht

Das NEWSYS Team begann mit der Analyse der Domain-Aspekte (was wird im Banken und Versicherungsumfeld benötigt) sowie einer Analyse des bisherigen Systems (Wie funktioniert OLDSYS?). Die Mitglieder des NEWSYS Teams hatten nur geringe Kenntnisse der Applikationsdomain und gar keine Kenntnisse des alten Systems. Dies stellte sowohl einen Vorteil wie auch einen Nachteil dar. Der Vorteil bestand darin, dass die Vorstellungskraft der Teammitglieder nicht durch die Kenntniss der Implementation des alten Systems von vorne herein eingeschränkt war.

Wie sehr die Macht der Denkgewohnheiten gefürchtet ist zeigte sich in einem spontanen Kommentar eines C++ Spezialisten der einen für uns einen C++ Kurs hielt, nachdem wir ihm von unserem Projekt erzählt hatten: „aber doch wohl nicht mit den alten (OLDSYS) Leuten!" Damals hatte ich diese Aussage als eher arrogant und peinlich angesehen, im Nachhinein – nach viel Lehrgeld – muss ich ihm zumindest teilweise Recht geben. Von Benjamin Lee Whorf gibt es ein (angenehm kleines) Taschenbuch mit dem Titel „Sprache, Denken, Wirklichkeit". Er gibt dort Beispiel wie sehr wir über die Begrifflichkeit unserer Sprache im Denken geleitet werden. Seine Analysen z.B. der Hopi Sprache und Kultur sind m.E. methodisch sehr fragwürdig, seine Beispiele aus dem Alltagsleben hingegen sind auf Grund eigener Erlebnisse sehr eingängig und gut nachzuvollziehen. Es mag lächerlich klingen, aber es ist ein grosser Unterschied ob ich einen Scheck als „Beleg" bezeichne oder als „Dokument". Dokument lässt einen viel weiteren geistigen Spielraum zu, z.B. über die verschiedenen physischen und logischen Arten mit denen ein Scheck realisiert werden könnte. Durch diese gedanklichen Alternativen erscheint ein konkretes Scheckformular nur als Spezialfall möglicher Scheck-Dokumente. Das Allgemeine (logischer Inhalt, Struktur) trennt sich vom Speziellen (physische Repräsentation, Format) und beides tritt klarer hervor.

Der Nachteil der geringen OLDSYS-Kenntnisse bestand darin, dass die Analyse des alten Systems entsprechend mühsam war.

Ziel der Anforderungsanalyse war wie gesagt die Ausweitung des Dokumentenbegriffs von „Beleg" (Scheck, Einzahlungsformular, Wahlzettel, Zeitabrechnung etc.) hin zu „Dokument".

Bald stellte sich heraus, dass typische Begriffe aus OLDSYS für die erweiterte Sichtweise nicht benutzbar

waren und deshalb durch allgemeinere Begriffe ersetzt wurden. Fortan sollten die alten Begriffe nur noch genau die Bedeutung besitzen, die sie im OLDSYS System hatten und auch nur noch dort angewandt werden. OLDSYS Mitarbeiter weigerten sich standhaft diese Begriffe zu verwenden und stellten in der Folge jede Mitarbeit ein.

Nichts spaltet eine Firma gründlicher als eine neue Sprache die aus einer neuen Sichtweise stammt.

In der Folge entstand zunächst ein Gemisch aus alten und neuen Begriffen das zu reichlich Verwirrung führte und das Ziel verfehlte, nämlich eine höhere Abstraktionsebene zu erreichen.

Von der Analyse über Design zur Programmierung

Das NEWSYS Team entschied sich frühzeitig, die OO-Methodik von Coad/Yourdon einzusetzen, „brave OO-Bürger" zu sein. Entsprechend dieser Methodik wurden Analyse und Design durchgeführt und ein Prototyp einer ersten Teilfunktionalität entwickelt. Es entstand ein logisches Domain-Modell das die Applikation in HIC (User Interface), PDC (Problem Domain) DMC (Data management) aufteilte. Basis- Funktionalität wurde in gemeinsamen Basis Klassen gesteckt.

Nicht erkannt wurde vom Team, dass speziell das Vorgehen von Coad/Yourdon sich speziell für kleinere und unabhängige Applikationen eignet, nicht jedoch für grössere Toolkits.

Der erste Prototyp

7 Monate nach Projektbeginn war ein erster Prototyp entstanden der zum grossen Entsetzen des NEWSYS Teams zwar objektorientiert programmiert war aber:

- Völlig unflexibel war
- Lange Compilier und Linkzeiten besass
- Änderungen zu einer Reinstallation führten
- Alles mit allem irgendwie unsichtbar zusammenhing OBWOHL nur Objekte verwendet wurden
- Das System viel unflexibler war als das (auf prozeduralen Callbacks) beruhende OLDSYS.
- Das System hochgradig plattformabhängig war.

Auch guten OO- Bürgern widerfährt Böses.

Die Ergebnisse einer objektorientierten Analyse können nicht einfach in eine Applikation umgesetzt werden. Die Logik der Applikation kann stimmen OHNE dass die Versprechen der OO-Bewegung (Kapselung, Wartbarkeit, Erweiterbarkeit etc.) eintreffen.

Das Signal des Prototypen an die firmeninterne Umwelt war eindeutig: Fehlschlag

Bereits während der Entwicklung des Protoypen beschäftigte sich der Leiter des NEWSYS Teams (der als einziger der Gruppe aus der Systementwicklung kam) verstärkt mit Opendoc und Fresco. Speziell Fresco, ein OO–Gui Framework im Source verfügbar- enthielt Mechanismen der Flexibilität (CORBA, Design Patterns). Der CORBA Kernel von Fresco wurde auf OS/2 portiert und der dahintersteckende – noch ziemlich unklare - Begriff des Frameworks in die Gruppe eingebracht. Damit entstand gleichzeitig eine Sichtweise die mehr von einem Systemkonzept als von herkömmlicher Applikationsprogrammierung ausging.

Problemanalyse des Prototypen:

- Die Auftrennung der Applikation in logische Domains verhinderte nicht ihre physische Verbindung. Logisch war die Applikation modular, physisch war sie ein Monolith.

- Die Compile und Linkabhängigkeiten können nur durch die Trennung von Interfaces und Implementations erreicht werden.
- Object Creation ist einer der grossen „cold spots" (unflexible Teile) in einem System
- GUI Componenten müssen dynamisch zur Laufzeit mit Applikationslogik verbunden werden.
- GUI Componenten verwenden ein Protokoll zur Kommunikation mit Model Objekten (logische Separation)
- GUI Componenten dürfen NUR über ein Interface mit Objekten des Models sprechen. (physische Separation)
- Konfigurationsinformation muss SICHER verwaltet werden können. Es ist ein symbolisches Type System für Meta Information nötig.
- Der Abstraktionsgrad der verwendeten Objekte war noch zu gering: z.B. spiegelte die „Beleg" Klasse darunterliegende Low Level Funktionalität in ihrem Interface. Damit wären nie generische Clients denkbar.
- Ein Metadaten-Modell für Objekte ist nötig. Information darf nicht kopiert werden. Statt z.B. Datenbank Schemata im Source zu codieren müssen sie aus Beschreibungen zur Laufzeit generiert werden können.
- Peripherie muss ohne Recompile oder Linken dynamisch ins System integriert werden können.
- Fremdprodukte dürfen nur gekapselt ins System eingeführt werden.
- Dynamisches Austauschen einzelner Klassen muss möglich sein (Wartung und Erweiterung)
- Neue Formen der Wissensweitergabe sind nötig, da bereits vier Personen gegenseitig die Implementationskenntnisse nicht mehr austauschen können: Design Patterns
- Das System muss zur Laufzeit durch Komposition von Objekten anpassbar sein, d.h. die Verbindung von Objekten muss am Ort ihrer Verwendung geschehen und nicht statisch zur Compilezeit („closer to actual usage time", Ron Resnick, distributed objects mailing list). Neue Konzepte wie Java-Spaces, Java-Infobus und allgemein Objektbus Technologie (z.B. iBus) oder Event Modelle wie der CORBA Event Service zielen ebenfalls auf dieses Problem.

Neben den logischen Strukturen einer Applikation gibt es eine Reihe anderer Strukturen (physische, Runtime, Extension etc.) die ALLE entscheidend zum Erfolg beitragen, jedoch in der objektorientierten Modellierung NICHT AUFTAUCHEN.

Ein besseres Verständnis der OO-Sprache entsteht

Der Prototyp hatte unser mangelhaftes Verständnis der physischen Aspekte von C++ sowie die Mängel im OO-Verständnis insgesamt (Interfaces, abstract data types) gnadenlos aufgedeckt.

Danach haben wir durch intensives Studium von Literatur und PD Source Code versucht unser System zu verbessern.

Die entscheidende Literatur war, ungefähr in dieser Reihenfolge:

- Stroustrup, diesmal auch die HINTEREN Seiten. (Interfaces etc.)
- Scott Meyers erstes Buch (low level aspekte von C++, multiple inheritance)
- Der Taligent Programming Guide (OO-System Design)
- Annotated Reference Manual
- Taligents Framework Buch
- Ron Ben Natans CORBA Einführung

- M.Ellis (Templates Behandlung)
- Strubbe, Das GUI als Situationskonzept
- Copliens Buch zu Idioms
- GOF Design Patterns
- Siemens Design Pattern Buch
- Beide Design Pattern Language Bücher
- Scott Meyers 2. Buch (Smart pointern, Proxies, VTable.)
- Weitere CORBA Bücher, Doug Schmidt Artikel etc.
- Lakos, physical Aspects of C++ (fundamental, für uns nur leider zu spät bzw. wegen bereits durchgeführter Interface/Implementation Trennung nicht mehr so relevant.

Sowie parallel dazu der C++ Report sowie das Object Spektrum und gelegentlich JOOP. Source Code Studium: Fresco

Was kann man danach?

Die schlimmsten Fehler vermeiden. (Dieser leicht zynische Kommentar resultiert aus der Erkenntnis, dass man selbst nach ausgiebigem Lernen in C++ häufig vor Alternativen steht, die alle nicht optimal sind und dass die physischen Abhängigkeiten doch einen enormen Einfluss auch auf das Design erhalten)

Ein neues Verständnis von OOA und OOD entsteht

Die objektorientierte Modellierung an sich hatte sich nicht als falsch herausgestellt. Sie musste jedoch in 2 Aspekten deutlich abgeändert werden.

Die OOA ist unvollständig und verkürzt

Die erst Erkenntis war, dass es nicht genügt nur die statischen Aspekte von Dingen der Wirklichkeit zu analysieren bzw. zu modellieren, d.h. die OOA wurde in erster Linie als unvollständig erkannt.

Zur Begründung:

Wenn man Analysemodelle von Business-Applikationen betrachtet fällt auf, dass sich die meisten auf den statischen Aspekt eines Objektes konzentrieren, z.B. wird ein Objekt „Beleg" mit seinen Attributen modelliert. Was nicht modelliert wird ist woher die Objekte kommen, wer sie erzeugt, welche Phasen sie durchlaufen, kurz: die Analyse beschränkt sich häufig auf die Teile, die letztlich als Records einer Datenbank enden. Ebensowenig wird berücksichtigt, dass sich wirkliche Objekte in ihrem Verhalten stark ändern können d.h. abhängig von ihrem Kontext sind ohne ihre Identität zu verlieren (d.h. Class und Identity sind unterschiedlich im Laufe der Zeit). Wirkliche Objekte können ihr Verhalten abstimmen auf den jeweiligen Sender einer Nachricht was die meisten OO-Programmiersprachen nicht ohne massive Eigenprogrammierung ermöglichen.

Resultat:

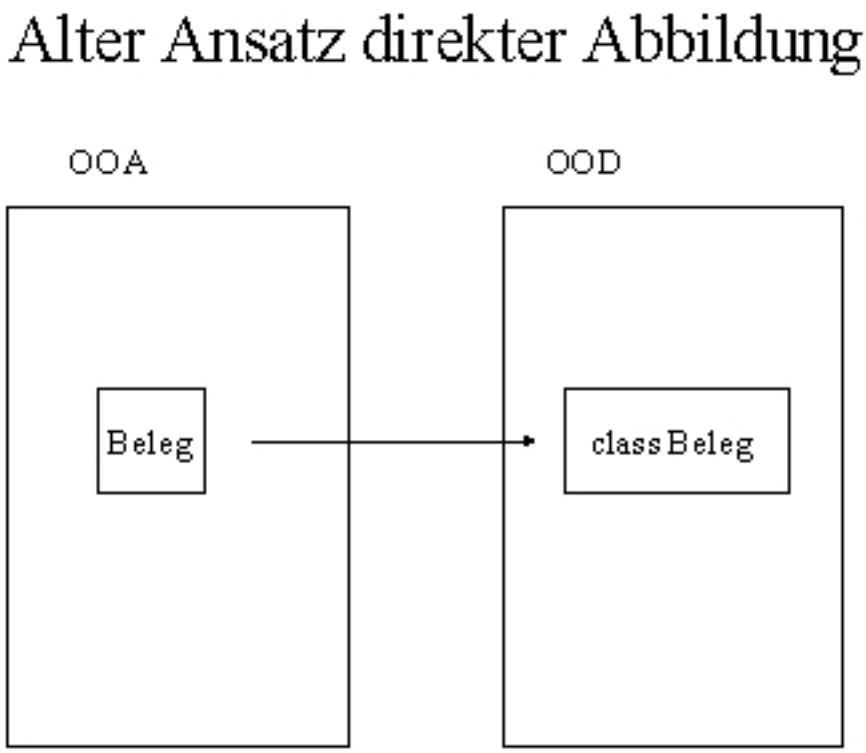
Die Möglichkeiten der momentanen OO-Sprachen und altgewohnte Sichtweisen aus der Datenwelt bestimmen unsere heutigen OO-Analyse die damit zwangsläufig stark verkürzt und unvollständig bleiben.

Die Konsequenz:

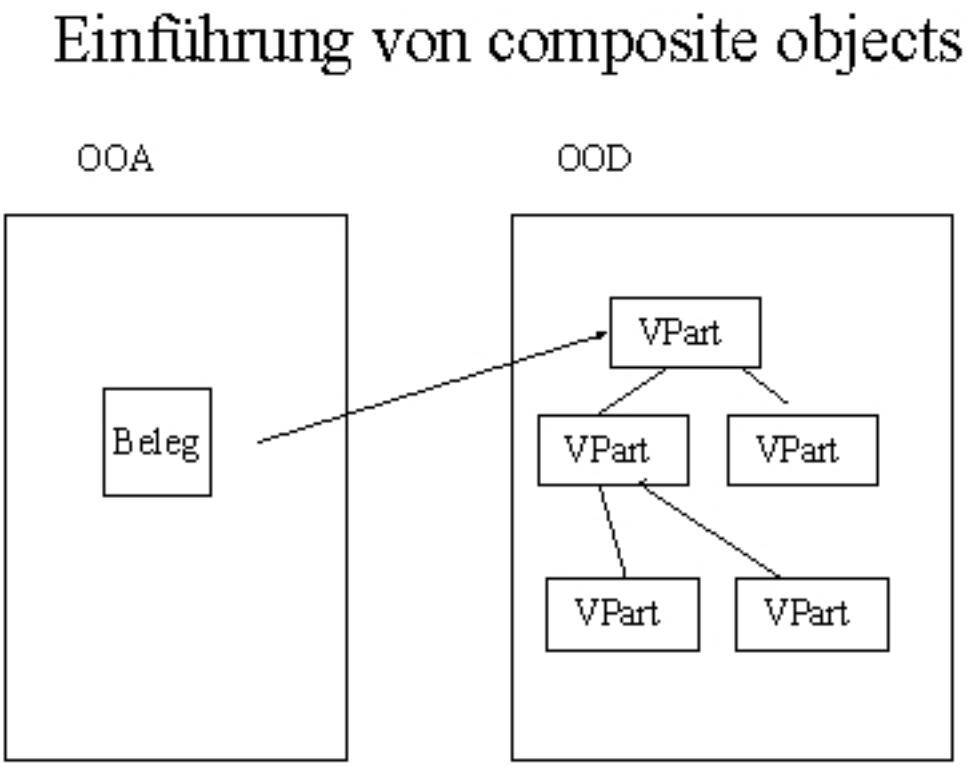
Die Objektaspekte, die in der Analyse unberücksichtigt geblieben sind, tauchen als Problem erst viel später im konkreten Design auf bzw. schlimmstenfall erst beim nötigen Redesign des Produktes weil z.B. versäumt wurde die massenhafte Integration neuer Objekte zu berücksichtigen.

Analyseobjekte und Maschine werden stärker getrennt

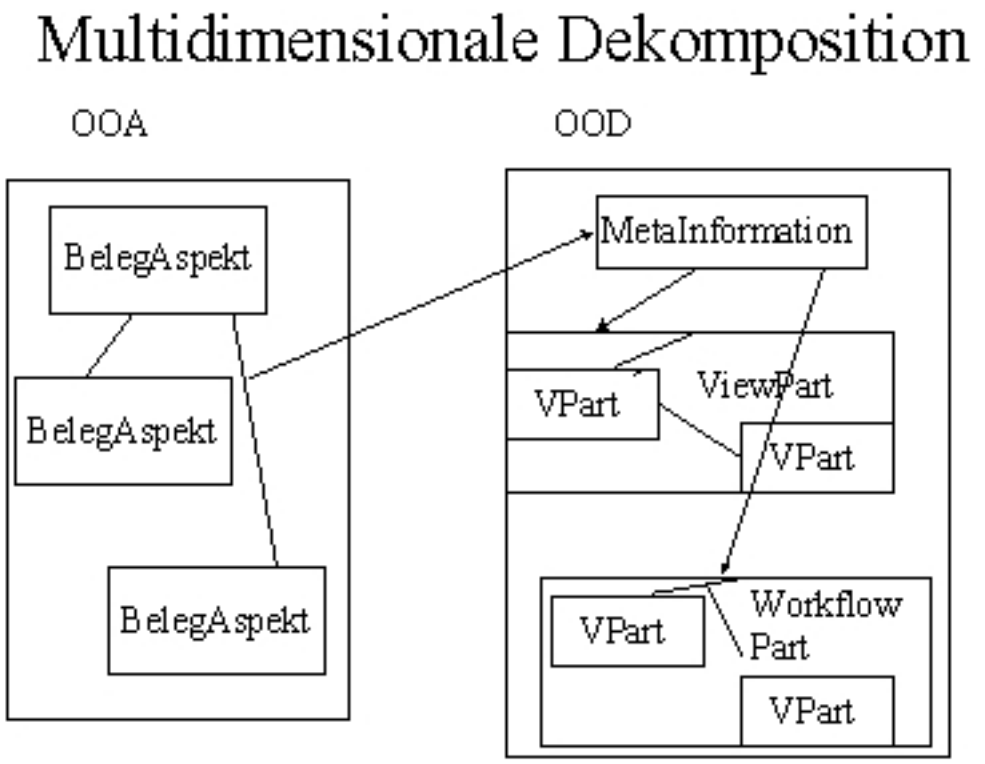
Das wahrscheinlich wichtigste Ergebnis der Problemanalyse war, dass nicht mehr von einem einfachen Abbildungsmodell der Analyseobjekte auf Design- und Implementationsobjekte ausgegangen wurde sondern über mehrere Stufen eine reflektive Architektur sowohl des Gegenstandsbereichs als auch der „Maschine" des Frameworks entstand. Diese Architektur lässt offen ob sowohl Gegenstandsbereich als auch Maschine in Form von Objekten analysiert werden und in welcher Form die Implementation (Abbildung) erfolgt.



Im ersten Ansatz waren wir von der direkten Abbildung ausgegangen. Dem liegt ein erkenntnistheoretischer Kurzschluss zugrunde: Wenn wir über Tische reden **funktioniert** unser Denken mit Hilfe von Tisch Objekten im Gehirn. Wir haben durch unsere Sprache zweifellos ein Objektmodell der Wirklichkeit zur Verfügung. Die Art und Weise der Verarbeitung dieses Objektmodells im Gehirn (Maschine) muss jedoch keineswegs in Form von Objekten erfolgen bzw. es können Objekte gänzlich anderer Art beteiligt sein. Dies führte zu einer erstem Zwischenstufe eines erweiterbaren Frameworks durch die Einführung von composite objects. Was auf der Ebene der Analyse ein Objekt war und meist auch eine Repräsentation der Wirklichkeit intendierte, wurde im Design abgebildet auf eine Objektbaum der zur Laufzeit dynamisch zusammengebaut wurde. Mechanismen aus Fresco wurden für die Architektur der composite objects übernommen (allgemein zum composite object pattern: [GOF96])



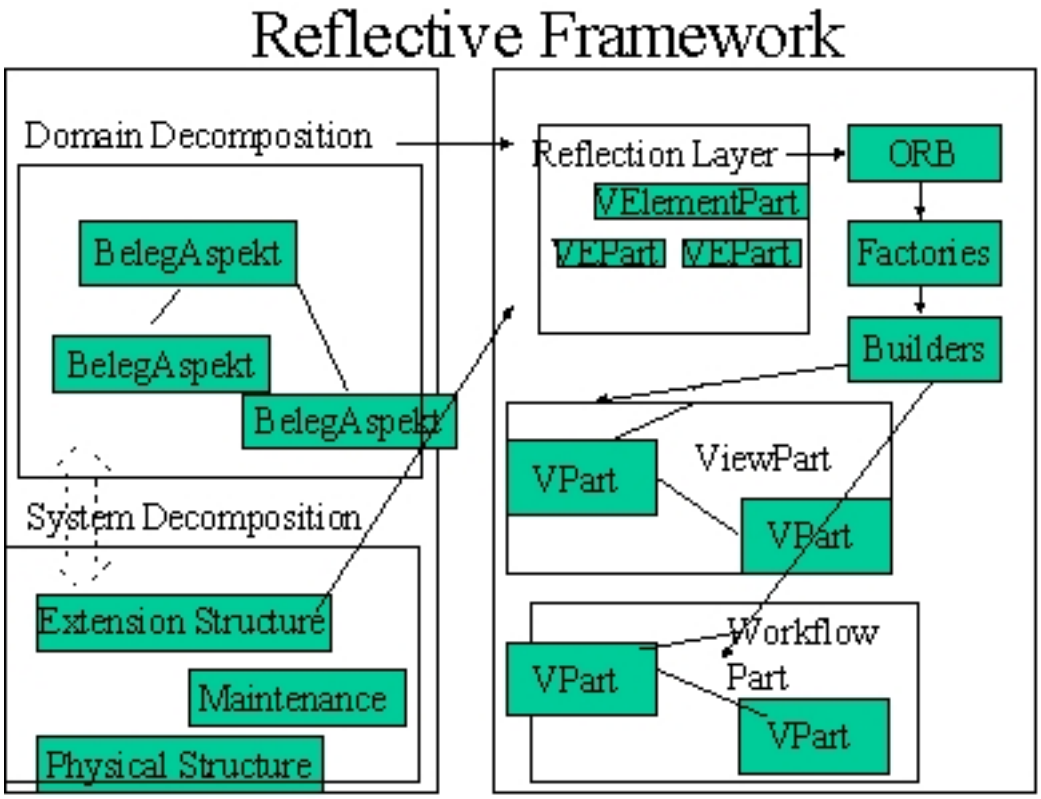
Damit wurde es möglich die verschiedensten Belegsorten aus wiederverwendbaren Teilen zusammenzubauen ohne dauernd neue Klassen definieren zu müssen. Dennoch gab es eine ganze Reihe von Funktionalitäten die auf den ersten Blick so aussahen als seien sie nur bei einigen Typen von Belegen anwendbar, z.B. Informationen und Funktionen zur physischen Ausprägung. In einem nächsten Schritt wurde versucht, ein generisches Framework durch sog. Multi-Dimensionale Decomposition der Problem Domain zu erreichen. Hierzu wurden die verschiedenen Aspekte von Belegen herausgearbeitet: logischer Aufbau, verschiedene Möglichkeiten physischer Representation, verschiedene Darstellungsmöglichkeiten (Views) und nicht zuletzt verschiedene pragmatische Bedeutungen wie z.B. das Vier-Augen-Prinzip im Bankbereich.



Jeder einzelne Aspekt wurde nun seinerseits als Dokument ausgedrückt und konnte je nach Bedarf mit anderen verknüpft werden. Die Art und Weise der Bildschirmdarstellung war jetzt z.B. nicht mehr eine Eigenschaft des Dokumentes selbst sondern in einem eigenen Dokument (Stylesheet, Viewdescription) festgehalten.

In einem letzten Schritt wurde dieses Verfahren jetzt auch auf das Framework selbst angewendet und damit der Schritt zum selbstreflektiven Framework vollzogen

Es fand eine Decomposition der Problem Domain Strukturen sowie der dem Framework eigenen Strukturen in Verbindung mit den Domain Strukturen statt. Anschliessend erfolgte mit Hilfe der Meta Information eine Coordination zur Laufzeit.



Die Metainformation enthielt das Konzept eines Belegobjektes sowie seiner Aspekte, repräsentierte also eine Abstraktion sowie konkrete Ausprägungen. Die Realisation funktionierte ganz anders und ohne direkten Bezug zu einem Objekt der OOA sondern vermittelt über die Metainformation.

(Das Diagramm zeigt nur einen kleinen Teil der beteiligten Strukturen und Aspekte!)

Sowohl Problem Domain als auch Framework selbst wurden Teil der Analyse. Die Ergebnisse flossen konstruktiv ein in die Meta Information des Reflection-Layers. Realisiert wurde der Reflection-Layer durch dieselben Vpart Klassen (V denotiert eine Interface Klasse) die auch zur Laufzeit zur Bearbeitung konkreter Dokumente verwendet wurden, d.h. die MetaInformation wurde ebenfalls ein Dokument.

Verschiedenste Bearbeitungen konnten damit zur Laufzeit konstruiert werden.

5) Generative und generische Ansätze

Die eben geschilderte Entwicklung ist bewusst auf generische Verfahren ausgelegt. Dass dies nicht die einzige Möglichkeit ist, soll in 2 Beispielen aus der Business orientierten Software Entwicklung gezeigt werden.

Im Bereich des Business Processing (z.B. Finanzdienstleistungen) dominieren meist zwei andere Verfahren.

- Quick and dirty.
- Modellierung mit anschliessender automatischer Code Generierung und Mapping zu existierenden Datenmodellen.

Das erste erstellt in einem ersten Schritt ein grobes Objektmodell der zu lösenden Aufgaben (z.B. eine Kundenbearbeitung) und beginnt dann gleich mit der Implementation in Form direkter Abbildung es Objektmodells in die Implementation. D.H. es wird so vorgegangen wie im unter 1) geschilderten Fall. Begründet wird das Vorgehen damit, dass man erstens nur für einen Kunden arbeitet, zweitens die Aufgabe sehr speziell und abgegrenzt ist und drittens der Umfang der zu entwickelnden Software überschaubar klein ist. In der Praxis lässt sich keine der Begründungen auf Dauer halten. Die Auffassung dass man nur für einen Kunden arbeitet vergisst zwei

Dinge: Gleiche Dinge werden dadurch in verschiedenen Abteilungen immer wieder neu entwickelt. Die Kosten solcher Spezialsoftware treiben selbst Banken und Versicherungen in die Arme von Outsourcing Unternehmen, d.h. man sägt am eigenen Ast. Ausserdem ist die Auffassung dass man speziellen Problemen mit speziellen Lösungen am besten begegnet einfach falsch. Auf Generalisierung zu verzichten bedeutet in das berüchtigte „Inventors Paradox" zu laufen. Es stellt sich nämlich oft heraus, dass eine generelle Lösung wesentlich kleiner und leichter ist als eine sehr spezielle.

Leider stimmt auch die zweite Annahme, dass die Aufgabe genau eingegrenzt ist nicht. Ich habe noch keinen Kunden ohne Änderungs- und Erweiterungswünschen erlebt. Dies ist aber gerade dann nicht mehr möglich wenn kaum Generalisierung betrieben wurde und die Software Lösung von speziellen Teilen nur so wimmelt. Weitere Wünsche führen nur zu weiterer Degeneration der Architektur.

Und was den dritten Punkt angeht: geringer Umfang des Software Projektes. Wer jemals z.B. mit C++ entwickelt hat wird festgestellt haben, dass eine Explosion des Codes sehr schnell eintritt und damit auch schnell zu langen Compile- und Linkzeiten führt.

Ein weiterer Grund für den Verzicht auf Generalisierung und Abstraktion ist darin zu suchen, dass neue Business Applikationen oft nur Frontends für bestehende Legacy Applikationen und Datenbanken sind. D.H. Letztlich wird nur das bestehende Datenmodell „objektifiziert". Daraus können natürlich nur sehr spezielle Klassen entstehen die kaum allgemein zu verarbeiten sind. Polymorphie kann dort gar nicht verwendet werden.

Diese Ansichten sind nicht unbedingt rein zufällig entstanden. Auf einem Arbeitskreis zu Framework Technologie wurde mir kürzlich versichert, dass gerade im Banken und Versicherungsbereich die Aufgabe der Objektmodellierung häufig von den ehemaligen Datenmodellierern erledigt wird.

Das zweite Verfahren mit automatischer Codegenerierung ist deutlich flexibler als das erste. Folgende Voraussetzungen müssen erfüllt sein, damit es erfolgreich ist:

- die Anzahl der Klassen muss überschaubar und ziemlich konstant sein.
- Die Kette der Codegenerierung muss ziemlich vollständig sein.

Ich hatte die Gelegenheit an einem Projekt zur Verwaltung von Finanzinstrumenten für einige Wochen mitzuarbeiten. In diesem Projekt wurden die Instrumente mit Paradigm Plus modelliert und über selbstgeschriebene Scripts anschliessend Code erzeugt bzw. Mit existierenden Basisklassen gemischt. Die Businesslogik war ebenfalls in einer descriptiven Sprache erstellt und der Validierungscode generiert. Damit konnten sogar kurzfristige Änderungen im Standard der Finanzinstrumente in kurzer Zeit bewältigt werden.

Selbst ein Konzept zur Erweiterung des Systems konnte so entwickelt werden, dass die manuellern Anpassungen gering blieben. Eine Stelle die von der automatischen Generierung nicht erfasst wurde war das Java basierte GUI. Zwar waren die Finanzinstrumente einander strukturell sehr ähnlich (sie wurden sogar mehr oder weniger als Superset auf der Datenbank abgelegt) jedoch sollten sie dem jeweiligen Verfahrensstandard gemäss unterschiedlich dargestellt werden. Dies führte sehr schnell zu einem sehr fehlerträchtigen und mühsamen „cut and paste" Ansatz. Die GUI Teile der Finanzinstrument liessen sich auch nicht durch Ableitung wiederverwenden da die Ausschnitte (Views) nicht die gleiche Granularität der existierenden Attribute besassen, d.h. wenn eine Methode A einen set bestimmter Attribute zurückbrachte war das nur für eine Klasse B richtig, jedoch nicht für die inhaltlich sehr ähnliche Klasse C.

Das Mapping zwischen GUI Views und Modell Klassen musste also unter anderen Gesichtspunkten erfolgen als sie bei der Klassenmodellierung gültig waren.

Für dieses Problem bieten sich zwei Lösungsansätze:

1. Vollständige Generierung des Mappings
2. Dynamische View Generierung zur Laufzeit aus Meta Information (analog Stylesheets)

Beide Verfahren setzen voraus dass der Verfahrensstandard der die Views letztlich bestimmt ebenfalls definiert bzw. Modelliert wird. Ist dies geschehen, können beide Verfahren eingesetzt werden.

Diesem eben geschilderten Schema folgt auch der von IBM kürzlich vorgestellte Component Broker. Auch dort werden mit OOA/OOD Tools Modelle (meist sog. „Business Objects") erstellt und durch ein inteaktives Tool (ObjectBuilder) mit einem mächtigen Framework verknüpft.

Vielleicht ist es eine Folge meines eher systemtechnischen Hintergrundes oder resultierend aus schlechten Erfahrungen mit den statischen Eigenschaften von C++ aber ich habe häufig ein schlechtes Gefühl bei einem derartigen Vorgehen und begründe es folgendermassen:

- Der systemtechnische Aspekt von Software Lösungen wird meist nicht analysiert oder modelliert. Man konzentriert sich auf die „Business Objects", die wiederum häufig reine Mappings zu Datenbanktabellen sind, d.h. im wesentlichen aus „getter „ und „setter" Methoden bestehen.

- Die Plazierung der Business Logik ist unklar: Soll sie in das Business Objekt oder als externer Verarbeitungsprozess funktionieren? Die Erfahrung zeigt, dass je mehr Business Logik in das jeweilige Objekt gelangt, desto weniger ist es für andere Verarbeitungsschritte geeignet. IBM spricht hier von „usage independent business logic" bleibt aber leider eine Definition dessen schuldig. Das Objekt verliert seinen Datencharakter. Die andere Alternative – externe Logik – stellt hingegen den Objektcharakter in Frage. Wozu brauche ich dann noch Business Objekte wenn die ganze Logik ohnehin nur durch externes Verarbeiten der Attribute des Objektes erfolgt? Zumindest hat dieses Verfahren den Vorteil dass es andere Workflows nicht grundsätzlich ausschliesst. Wartungsmässig ergeben sich jedoch gravierende Nachteile wie im nächsten Punkt deutlich wird:

- Der Aufwand den Interface Änderungen für die Maintenance eines gesammten Geschäftsablaufs darstellen wird unterschätzt, dies ist eine Erfahrung gerade auch aus grossen CORBA Anwendungen [MAFF97],[MAWBR97]. Wenn das Systems 24 Stunden am Tag online sein muss, z.B. weil es international genutzt wird, dann ist die automatische Generierung nur noch ein theoretisches Mittel der Wartung und Erweiterung. Damit aber ein System zu Laufzeit dynamisch ausgewechselt oder erweitert werden kann braucht es enorme Unterstützung durch Systemklassen die wiederum Einfluss auf den generierten Code haben. Die IBM geht hier mit dem Component Broker sicher einen interessanen Weg, gelingt es ihr doch die Komplexität des Storage Managements praktisch vor dem Applikationsprogrammierer zu verstecken. Leider wird aber auch hier kein „hot swap" von Services geboten.

Wann besitzen dynamische Verfahren Vorteile?

- wenn die Anzahl der Klassen grundsätzlich nicht begrenzbar ist (welche Typen von Dokumenten kann es z.B. geben?)

- wenn zur Laufzeit Instanzen neuer und unbekannter Klassen verarbeitet werden müssen.

- Wenn dynamische Reconfiguration gefordert wird.

- Wenn oft Änderungen an den Klassen erfolgen.

- Wenn die Änderungen einen enormen Installationsaufwand nach sich ziehen.

- Wenn ein Objekt sich nach aussen mit eingebautem Behavior zeigen soll (also im Sinne echter Objekte) dieses Verhalten aber dynamisch erstellt werden muss da es über die Laufzeit des Objektes wechselt bzw. das Objekt immer neuen und teils noch unbekannten Verarbeitungen unterzogen werden muss.

Gut denkbar sind auch Mischformen von generischen und generativen Verfahren. Z.B. kann das Objektmodell durch Scripts auch in ein Metamodell überführt und zur Laufzeit zur Verfügung gestellt werden. Leider habe ich noch kaum Modellierungen solcher System gesehen.

Noch ein kleiner Nachtrag zur GUI Problematik: Wie können Views auf Klassen gemappt werden?

Typischerweise verwendet man einen sog. GUI-Builder zum Aufbau von Bildschirm Masken und Layouts. Während es keine Frage ist dass das manuelle Arrangieren von Feldern ohne ein Layout Tool recht mühsam ist ergeben sich beim näheren Hinsehen doch einige kritische Punkte.

Woher weiss der Layout Designer was für GUI Elemente er selektieren muss? Es muss dafür irgendeine Spezi-

fikation geben. Wenn es sie gibt, warum wird sie nicht zur Generierung verwendet? Gibt es Gemeinsamkeiten der Views bzw. treten einzelne Views immer wieder auf? Wenn ja, können sie an beliebigen Stellen so referenziert werden ohne dass Verdoppelungen auftreten? Können einzelne Views voneinander erben? Wenn eine Änderung der View Vorschrift auftritt, müssen alle Ressource Files geändert werden? Und wenn, geschieht dies manuell? An einer zentralen Stelle? Woher weiss der Layout Designer wie lang Felder sein müssen? Tritt hier nicht eine Verdoppelung von Information auf, denn andere Stellen im System müssen das auch wissen?Wie kann es sein dass ich ein komplexes Layout mit einem Texteditor sehr einfach und schnell massiv abändern kann (vorausgesetzt das Ressource File ist editierbar) mit dem Layout Tool selbst aber mühsam alle Felder einzeln manipulieren muss? Kann es sein dass der GUI gesteuert Zugriff doch nicht immer der optimale ist?

Kurz und gut, dies sind nur einige ketzerische Fragen an die übliche Auffassung, dass das GUI eine Applikation „treibt". Mir scheint die Vorstellung von generischen Browsern wesentlich angenehmer, ganz im Sinne von Netscapes Devise „the information is the interface". Ähnliche Fragen lassen sich übrigens auch an einen Datenbank Designer richten. Sie betreffen letztlich den Unterschied zwischen der „Informationsarchitektur" und der „Implementationsarchitektur" [BIRMAN96] eines Systems, auf den weiter unten noch genauer eingegangen wird.

Radikale Abstraktion:"Alles ist ein Dokument"

Natürlich ist nicht alles ein Dokument, genausowenig wie das Web alle EDV-Probleme lösen wird. Es ist jedoch erstaunlich, wie weit eine so allgemeine Metapher wie „Dokument" tragen kann.

Die anfänglich nur zaghaft eingeführte Abstraktion „Dokument" für die Kernobjekte des Systems wurde nun auch auf andere Bereiche als wie die ursprünglichen „Belege" ausgeweitet.

Was ist ein Dokument? Ein dynamisch zusammengebautes Objekt, bestehend aus vielen anderen. Die Art und Weise der Zusammensetzung regelt wiederum ein Dokument.Was ist Systemkonfiguration? Ein Dokument. Was ist Workflow? – Eine Beschreibung, d.h. ein Dokument. Was ist Datenimport? – Eine Transformation eines Dokumentes in ein anderes, vermittelt über eine Beschreibung (Dokument) der notwendigen Abbildungen. Was ist eine Data-Management Komponente? Ein System das Dokumente persistent speichern kann, in Formaten die in Meta-Dokumenten festgehalten sind. Was ist ein View? Die dynamische Darstellung von Dokumenten mit Hilfe von Beschreibungen (Style-Sheets, View-Descriptions)

Durch die entstandene Vielfalt von Anwendungsmöglichkeiten des Dokumentenbegriff wurde plötzlich auch eine Kernaussage der Taligent Entwickler verständlich: zentrale Metaphern müssen sehr allgemein sein, z.B. People, Places, Things. Nur diese Allgemeinheit lässt einen unscharfen und damit vielfältigen Gebrauch zu. Vielleicht hat Wittgenstein mit seiner Definition der Bedeutung von Begriffen durch ihren Gebrauch doch etwas Richtiges getroffen.

Die vielfältige Verwendung des Dokumentenbegriffes hatte enorme Konsequenzen für die Implementation. Dieselben Dokumentenklassen konnten in immer neuen Zusammenhängen wiederverwendet werden. Die Entwickler gewöhnten sich schnell and die relativ überschaubare Zahl von Klassen und Methoden. Die Komposition der Dokumente konnte ebenfalls in anpassbare Klassen ausgelagert werden.

Eine weiter Erkenntnis war, dass in grossen Systemen das Verhalten der Klassen hinter die Bedeutung der Informationsarchitektur, d.h. der Daten (Dokumente) und der Metadokument zurücktritt.

Die vielleicht wichtigste Entscheidung war jedoch, das Dokumentenmodell auf das Framework selbst anzuwenden, d.h. viele Informationen die früher hart codiert im Source Code gelandet wären wurden nun als Dokumente extern gehalten und zur Laufzeit geladen und interpretiert. Flexibilität und vor allem Klarheit der Architektur wurden damit wesentlich verbessert.

Eine Anmerkung zu den sozialen Konsequenzen dieses Vorgehens: Die Durchführung der radikalen Abstraktion führte zum Bruch mit der ausschliesslich aus Mathematikern bestehenden Gruppe der Applikationsentwickler.

Wir bauen ein Framework

Nach den vernichtenden Ergebnissen des Protoypen entschied sich das NEWSYS Team auf ein Framework Design umzusteigen. Zum damaligen Zeitpunkt gab es praktisch keine verfügbaren Frameworks die man hätte kaufen können.

Die Geschäftsleitung lehnte den sofortigen Einsatz von CORBA (Orbix) ab (zu teuer, Zeitaufwand bei Einarbeitung, zunächst kein distributed computing geplant). In der Folge mussten vom NEWSYS Team wesentliche Broker-Mechanismen selbst implementiert werden. Grob geschätzt gingen in einem Zeitraum von 12 Monaten 80% der Zeit in die Grundlagenentwicklung.

Zur objektorientierten Applikationsprogrammierung ist eine Systembasis nötig, die in C++ nicht vorhanden ist. Lebenswichtige Services müssen entweder selbst entwickelt oder eingekauft werden. 80% der Arbeiten im ersten Jahr sind Grundlagen Technologien.

Eine überraschende Erkenntnis nach der Analyse der physischen Eigenschaften von C++ war, dass die Trennung von Interfaces und Implementationen ein Broker Konzept erzwingt, AUCH wenn die Applikation nur lokal laufen soll.

Zum Zeitpunkt der Entscheidung gab es keinerlei Erfahrung über die Zeitdauer einer Framework Entwicklung. Von Seiten der Applikationsgruppen kam Widerstand, da dort ein Toolkit mit feststehendem API erwartet wurde. Das NEWSYS Team stand somit vor dem Problem, dass einerseits (Geschäftsleitung) ein sehr flexibles System erwartet wurde, andererseits (Applikationsgruppen) ein festes API mit festen Implementationen vorausgesetzt wurde. Hinzu kam, dass die Grundprobleme von C++ (statisches Linken, keine Interfaces) nur schwer vermittelbar waren.

Hohe Kompetenz von Applikationsentwicklern hilft wenig beim Kampf mit Systemstrukturen der Applikation bzw. der Programmiersprache. Lifecycle Betrachtungen sind ebenfalls oft ein fremdes Konzept.

In der Folge stellte sich auch heraus, dass der zentrale Gedanke eines Frameworks: neue Funktionalität durch das Austauschen von „Puzzleteilen" zu erreichen von den Applikationsgruppen nicht mitgetragen wurde. Für die NEWSYS Gruppe war das Framework in erster Linie ein funktionaler Rahmen der Erweiterbarkeit mit bewussten Leerstellen. Für die Applikationsgruppen waren die Leerstellen nicht Möglichkeiten der Erweiterung sondern einfach mangelnde Funktionalität. Dieser Konflikt endete damit, dass die Mitglieder der Framework Entwicklung auch die Applikationen auf Basis des Frameworks entwickelten. Es hatte sich eine unüberbrückbare technische, sprachliche und kulturelle Kluft ergeben.

Ein in diesem Zusammenhang immer wieder aufgeführtes Argument war das des „Pragmatismus". Abstraktionen oder Reengineering wurde abgelehnt mit dem Hinweis, man sei ja „Pragmatiker". Im Laufe der Zeit wurde klar, dass damit eigentlich etwas ganz anderes gemeint war:

Pragmatismus in Software Projekten wird oft als Ausblenden der Lifecycle Problematiken verstanden. Wenn überhaupt Probleme erkannt werden (Softwäre wächst, Wartung und Installation) werden sie nach hinten verschoben.

Beispiel:

Die meisten „alten" Mitarbeiter neigten zu sehr langen Methodenrümpfen. Häufig versuchten sie lediglich bestehenden Code in neuen Klassen wiederzuverwenden. Restrukturierung und Faktorisierung des Codes wurde abgelehnt. Derartige Klassen sind nicht wiederverwendbar und nur schwer zu warten.

Sehr lange Methodenrümpfe sind ein typisches Kennzeichen von ehemaligen C Programmierern die versuchen, alten Code und alte Methodik im OO-Bereich weiter zu verwenden. Diese Methoden machen Klassen nicht wiederverwendbar und können nur global überschrieben werden. Die Methodenlänge ist durchaus als empirisches Kennzeichen des OO-Entwicklungsstandes im Projektmanagement verwendbar.

In einem mehr oder weniger bewussten Prozess entstand in den Monaten nach dem Prototypen ein Framework. Die weiter unten beschriebenen Strukturen sind eine Sichtweise aus dem Nachhinein!

Viele zentrale Design Pattern wurden aus FRESCO [LINTON] übernommen. Das Ergebnis ein Framework das das logische Modell der Trennung in verschieden Domains um weitere Dimensionen (im folgenden „Strukturen") erweiterte.

Ein Framework ist die Realisierung (im Sinne der UML 1.0) einer „Evolving System" Meta Architektur, die ein Produkt in seinem ganzen Lebenszyklus (Entwurf, Entwicklung, Erweiterung und Spezialisierung, QA, Installation, Konfiguration, Kunden, Wartung etc.) betrachtet. Diese Architektur ist das krasse Gegenteil einer „pragmatischen" Sichtweise, die vor den unvermeidlichen Änderungen eines Systems die Augen verschliesst.

Im Prozess der Realisierung eines Frameworks hebt sich die Trennung zwischen Applikationsentwicklung und

Grundlagenentwicklung auf. Mitglieder eines Frameworkteams müssen während der Arbeit umschalten können. Dies stösst bei manchen auf grosse Probleme im Selbstverständnis.

Grundgedanken des Frameworkings an einem Beispiel

Es wurde bereits erwähnt, dass object creation einer der gravierendsten cold spots eines Systems ist. Dies erklärt sich dadurch, dass im Moment der Instanziierung mehrere Strukturen des Frameworks gleichzeitig beteiligt sind:

- **WER** erzeugt
- **WANN**
- **WELCHE** Objekte
- **WIE?**

Cold Spot Analyse ist die Untersuchung eines Mechanismus daraufhin, welche Teile davon so festgelegt sind, dass sie OHNE Source Code Änderung nicht modifiziert werden können und worin diese Abhängigkeiten bestehen. Cold Spot Analyse verändert das WER, WANN, WELCHE und WIE und untersucht die Konsequenzen auf die logische, physische, runtime und extension sowie die Sourcecode-Struktur.

Wem die folgende Analyse übertrieben erscheint, hat noch nie eine grössere OO- Applikation mit C++ entwickelt.

Ausserhalb eines Framework Konzeptes entsteht bei der Arbeit mit C++ schnell eine fast paranoide Furcht vor Änderungen. Notwendige Verbesserungen im Design unterbleiben weil sie Auswirkungen auf viele Stellen des Source Codes hätten und ausserdem eine Reinstallation erzwingen würden.

Beispiel: Object creation cold spots und ihre strukturelle Dekomposition

X.hpp/cpp

#include <Z.hpp>

#include <Y.hpp>

#include <G.hpp>

Class X {

private:

G myG;

Y myY;

public:

DoIt () {

Z myZ = new Z();

..}

}

Eine multidimensionale Dekomposition zeigt, dass dieses harmlos aussehende Stückchen Source Code MEHRERE cold spots enthält, die verschiedene Strukturen betreffen, z.B.:

1. Einen physischen cold spot der erzwingt, dass X recompiliert werden muss wann immer etwas im Headerfile von G, Z oder Y geändert wird. D.H. auch bei der Einführung einer neuen privaten Variablen die den Client X überhaupt nicht interessiert da er sie ohnehin nicht erreichen kann. Wenn G, Z oder Y im .cpp geändert werden, muss der Client X neu gelinkt werden, da sich Adressen verschoben haben können.

2. Mehrere logische cold spots: Ohne Sourcecode Änderung ist es nicht möglich in X ein abgeleitetes Z‘ oder Y‘ zu verwenden. Es ist nicht möglich X als composite object zu behandeln das beliebige child implementations desselben Interfaces enthalten kann. Y ist ein festes Member von X. Es können nicht mehrere Y verwendet werden.

3. Einen cold spot bezüglich der Erweiterungsstruktur: Wenn ein bugfix für X,Y oder Z nötig wird, wird gleichzeitig ein neues Release nötig da alle Klassen miteinander verbunden sind. Die Klassen werden auch nicht distributed laufen können.

4. Einen sozialen cold spot: Die Klassennamen unterliegen keiner Konvention und geben keinerlei Hinweise auf die Eigenschaften der Klassen (Sind es Interfaces, default implementations, konkrete Klassen die nicht mehr abgeleitet werden sollten? Multiple Inheritance Klassen mit Besonderheiten bei Pointern?) Die Integration neuer Teammitglieder wird erschwert, zumal auch keinerlei Design Patterns erwähnt sind. Class members sind nicht deutlich unterschieden von automatic variables.

5. Ein cold spot bezüglich der Wartung und Auslieferung: Es gibt keine Unterstützung für automatisches Generieren einer Class Reference direkt aus dem Source Code.

6. Ein cold spot bezüglich der Sourcecode Struktur: Diese Stück Sourcecode ist NICHT portabel. Microsoft Plattformen brauchen z.B. spezielle calling declarations (declspec) die mit Macros gekapselt werden müssen. Andere Plattformen und Compiler brauchen möglicherweise andere Anweisungen.

7. Ein cold spot bezüglich des build Prozesses und der Source Code Struktur. Es gibt keine Macros vor der Declaration, in der Declaration und im Implementationsteil dieser Klasse. D.H. man kann keine neuen Methoden oder Funktionen automatisch einfügen durch Änderung eines Macros. Gerade Basisfunktionalität muss aber häufig „nachgerüstet" werden weil man sie am Anfang nicht gekannt oder verstanden hat.

8. Ein runtime cold spot: Es wird keine Metainformation zur Klasse generiert (Name, Ableitung etc.)

9. Ein cold spot bezüglich Quality Assurance: memory deallocation ist nicht spezifiziert, es werden keine Smartpointer eingesetzt. Dies ist besonders schlecht im Falle von Exceptions.

C++ unterstützt das Konzept sich entwickelnder Systeme nur ganz gering. Mit wenigen Statements werden physische Abhängigkeiten geschaffen ohne dass dies sofort bemerkt wird. Low Level Implementationsdetails der Programmiersprache beeinflussen das Applikationsdesign. Der ohnehin recht statische Charakter von Klassifizierungsystemen (und nichts anderen sind Klassenhierarchien) wird durch die Programmiersprache noch verstärkt.

Das nächste Beispiel zeigt einen typischen Framework Code in C++. Der Code versucht die eben angesprochenen Probleme zu umgehen.

SyPNContainerDoc.hpp/cpp

// Copyright, usage

// RCSid etc.

// InterfaceVersionID ALL automically generated!!!!!!!!!!!!!!!!!!!!!!!!!

#ifndef SYVDSTORAGEMANAGER_HPP

#include <SyVDStorageManager.hpp> // Storage Manager Interface from Data management Domain

#endif // include protection, reduces build time, generated automatically

#ifndef SYVPDOCPART_HPP

```
#include <SyVPDocPart.hpp> // Document Part Interface from Problem Domain
#endif
DeclareNormalImplementationClass(SyPNContainerDoc) // macro for reference types and
// smart pointers, declares an implementation for
// a Systor Interface
// the class declaration is a macro that sets platform dependent export/import statements
// depending on the PACKAGE declaration (SBV_Customer is set during build).
// The macro also generates meta class information.
// the comment uses a tag for automatic class reference generation
//- SyPNContainer
SyNClass1(SyPNContainerDoc, SyVPDocPart, SBV_Customer)t
//. This class provides a simple container for document
//. Parts .....
//. Design Patterns: composite object
//. preconditions, postconditions, warnings
{ // this macro will cover anything that is necessary in class declaration (e.g.
// class meta information etc.
NormalImplementationInDeclaration(SyPNContainerDoc)
public addChild(SyVPDocPart_ref) { } Any document part can become a member now, not just Y
doIt () {
// use configuration to find which class to use (could be a pattern for a trader too)
String_ref StorageManagerName = (Broker->lookup(„StorageManager");
// use Broker to get factory to get StorageManager. ONLY interfaces are known by X!
// use tmp macro to count down ref count on returned temporaries
SyVDStorageManager_ref = _tmp(Broker->DMCKit())->StorageManager(StorageManagerName);
..}
private:
G myG; // This is OK if and ONLY if G is declared and implemented IN THE SAME
// physical package as this source file!!!!! (component level insulation)
}
```

Das Beispiel sieht zunächst ziemlich kompliziert aus allerdings können die meisten Statements automatisch generiert werden. Wesentlich ist die „Künstlichkeit" des Source Codes, die aus der Anwendung von Coding Conventions stammt, die die Trennung der physischen Strukturen sicherstellen.

1. Die Klasse X (SyNPContainerDoc) kennt jetzt ausschliesslich Interface Klassen.
2. Sämtliche beteiligten Implementationen könnten zur Laufzeit ausgetauscht werden
3. Der Client kennt noch nicht einmal den konkreten Klassennamen z.B. des StorageManagers. Er erhält ihn aus einer Konfiguration bzw. als Default aus der DMC factory
4. Automatisch kann die Class reference generiert werden
5. Der Code is plattformunabhängig (export/import Statements werden über einen Macro gesetzt, Funktionen zum dynamischen Laden, sog. „Loader Functions" fehlen allerding noch).
6. Der Container part kann jetzt beliebige andere Dokument Parts aufnehmen.
7. Der Source Code entspricht automatisch Richtlinien NEWSYS und wird als template automatisch generiert.
8. Meta Information wird automatisch generiert. Weitere Methode die alle Implementations von Interfaces betreffen können automatisch eingefügt werden, z.B. Loader Functions zum Laden aus DLLs.
9. Der Client X muss nur noch compiliert oder gelinkt werden, wenn sich das Interface der verwendeten Klassen ändert. Das Composite Object Pattern sorgt dafür, dass dies selten der Fall ist.

C++ Code innerhalb eines Frameworks stützt sich stark auf generative Mechanismen, oft als Macros implementiert. Der Code ist hoch ritualisiert und geregelt. Die Entwickler müssen durch kleine Tools unterstützt werden. (NEWSYS beinhaltete z.B. einen Source Code Generator der Templates von Header Files, Implementations Files, Funktionen oder Applikationen erstellte).

Ein C++ Framework braucht umfangreiche Coding Standards die NICHTS mit der Optik des Source Codes zu tun haben. Die Coding Standards beinhalten das gesammelte Design Wissen des Framework Teams. Fremde Benutzer MÜSSEN diese Standards ebenfalls einhalten.

Die tatsächliche Komplexität der dahinterliegenden Macros und Mechanismen wird nicht sichtbar, ihre Kenntniss ist aber auch durch Tools nicht ersetzbar. Programmierer brauchen tiefe System- und Plattformkenntnisse.

Zur Bedeutung von Coding Standards siehe Kapitel Coding Standards.

Bevor in den folgenden Abschnitten die Strukturen eines Frameworks im Einzelnen behandelt werden hier noch das Beispiel von oben als Java Source:

```
/* SyPNContainerDoc.java

* Copyright, usage

* RCSid etc.

*/

package systor.doc.Implementations

import java.*

import systor.doc.* // document interfaces

import systor.broker.* // broker and factory interfaces

/**

*implements a container part for systor document parts by extending the interface class for document parts

*This class provides a simple container for document Parts
```

```
* Design Patterns: composite object

* preconditions, postconditions, warnings

*/

class SyPNContainerDoc implements SyPVDocPart

{

/**

* any doc part type can be added to this container doc part

*/

public addChild(SyVPDocPart) { }

/**

*method uses configuration information and broker and storage manager interfaces

*in a generic way.

*/

public doIt () {

String StorageManagerName = Broker->lookup(„StorageManager");

SyVDStorageManager myMngr = Broker->DMCKit()->storageManager(StorageManagerName);

..}

private G myG();

// a version identifier for evolving classes, see java serialization ap.

// generated by a serialver (do not confuse with source code id!):

static final long serialVersionUID = 1234567890123456L;

}
```

Dieser Java Source Code sieht weniger kompliziert aus als die C++ Version. Er enthält jedoch MEHR Funktionalität (eingebaute automatische Dokumentation, garbage collection) und hat durch das dynamische Linken nicht das Problem der „fragile superclass" bei dem nachträgliche Änderungen an Basisklassen umfangreiches Recompilieren und Linken aller abgeleiteteten Klassen erzwingt.

Bei Interface Änderungen sind die turn around Zeiten DRASTISCH kürzer als im C++ Fall, etwas das das verbesserte C++ Beispiel oben noch gar nicht berücksichtigt.

Die Mechanismen dahinter die der Programmierer verstehen muss sehen nicht nur weniger kompliziert aus, sie sind es auch. Mit anderen Worten: Die Integration von Teammitgliedern wird schneller gehen.

Noch wichtiger als die physischen Aspekte der Programmiersprachen ist jedoch die folgende Erkenntnis: Die Design Patterns „Broker" und „Factory" sind in Java nicht weniger wichtig als in C++. Sie sind nur leichter zu erzeugen und zu warten. Durch die weniger komplexe physische Struktur erlaubt es Java sich auf die logischen Strukturen und Design Patterns zu konzentrieren.

In diesem Fall bedeutet dies: Der Client braucht nicht die Implementation der Objekte zu kennen, mit denen er arbeitet. Kunden können Klassen spezialisieren und der Client wird ohne Änderung mit den neuen Klassen arbeiten können. Dieselben Pattern erreichen das gleiche Ziel in C++, nur dass sie dort noch zusätzliche Sprachprobleme lösen.

In Java ist die Behandlung physischer Eigenschaften leichter. Die logischen Strukturen, z.B. das Verstecken von spezifischen Objektimplementationen hinter Broker und Factories bleiben gleich. Die hot spots die ein Evolving System braucht werden durch die gleichen Design Patterns wie in C++ erzeugt. Braucht es noch mehr Gründe für einen an Design pattern orientierten Entwicklungsprozess?

Was also ist ein Framework?

Als ich noch ein blutiger Anfänger in der Unix Kernel Division der Siemens AG war und Device Treiber für die damals neuen Multiprozessor Systeme entwickelte, habe ich einmal einen sehr erfahrenen Freund bei Sequent gefragt, was man ein Multiprozessor System baut. Seine Antwort lautete: „Das ist ganz einfach, man muss nur alle globalen oder statischen Variablen mit Locks vor gleichzeitigen Zugriffen schützen." Die Kunst besteht natürlich darin, dies mit Hilfe von spezieller Hardware und geschickten optimierten Algorithmen so zu tun dass n Prozessoren auch ungefähr die n-fache Systemleistung erbringen.

Analog dazu bring Ken Birman distributed computing auf einen Punkt: es ist eigentlich trivial. Das Problem liegt nur darin auf zwangsläufig auftretende Fehler (Hardware,Software,Bedienung) so zu reagieren, dass Reliability und Security gesichert sind und die Performance noch akzeptabel bleibt.

In diesem Sinne definiere ich ein „Frameworking" als das Berücksichtigen aller Abhängigkeiten innerhalb eines Softwaresystems über den gesammten Lebenszyklus hinweg. Die Aufgabe besteht darin, diese Abhängigkeiten und deren Konsequenzen zu erkennen und Techniken zu entwickeln, die ein Degenerieren des Systems verhindern. Und die Kunst ist es, trotzdem in erträglicher Zeit zu einem Produkt zu kommen.

Deshalb geht der nachfolgende Teil genau auf die technischen Strukturen und ihre Verbindungen ein und versucht praktische Lösungen anzubieten.

Die Definition eines Frameworks ist noch nicht vollständig. Birmam (wie viele andere) hebt noch etwas entscheidendes hervor: Den Willen, z.B. Probleme der Reliability und Security ernst zu nehmen und technisch zu lösen. Er zeigt eindrucksvoll dass wir Softwareentwicklung auf gut Glück betreiben und hoffen dass unsere Programme keine Menschen umbringen. Eben dieser Wille scheint neben der Technik die zentrale Rolle zu spielen, wie auch im Falle von NEWSYS.

Die Erfahrungen im NEWSYS Projekt haben gezeigt, dass der Wille u.a. von 2 Faktoren abhängt:

- Persönlichkeitstruktur (Neugier, Motiviertheit, Flexibilität, Verständnis für andere, Kommunikationsfähigkeit etc.)
- Soziale Strukturen der Organisation von Software Produktion

Im Anschluss an die Behandlung der technischen Strukturen eines Frameworks wird daher auf die sozialen Strukturen eingegangen, die seinen Erfolg wesentlich mitbestimmen. Darüberhinaus versuche ich zu zeigen, dass zwischen technischen und sozialen Strukturen eine enge gegenseitige Abhängigkeit besteht und wie man sie günstig beeinflussen kann.

Der Titel „Frameworking" wurde von mir ganz bewusst gewählt um die Arbeit an einem Framework als technischen und sozialen Prozess hervorzuheben.

Natürlich hat Frameworking noch weitere Dimensionen. Neben der technischen und sozialen Dimension steht der Produktionsprozess von Software in einem Mikroökonomischen und einen Makroökonomischen Zusammenhang. Die mikroökonomische Dimension habe ich (vielleicht als Folge der Tatsache dass NEWSYS innerhalb einer kleinen Firma entwickelt wurde in der soziales und ökonomisches hautnah miteinander verbunden sind) einfach unter den sozialen Strukturen mit behandelt. Die Makroökonomische Dimension (hier geht es u.a. um die Frage wie lange sich Softwareentwicklung noch ausserhalb normaler Geschäftsbedingungen abspielen kann? Stichwort: Produkthaftung, Gewährleistung, Qualität, Sicherheit etc.) übersteigt jedoch meine Kompetenz bei weitem und sprengt den Rahmen des Papers, das auf praktische Hilfen beim Frameworking ausgerichtet ist. Meiner Meinung wird jedoch der Tag kommen an dem ein US Gericht die Absicherungsklauseln eines Softwarherstellers für ungültig erklärt und Schadensersatz in enormer Höhe zuspricht. Von diesem Tag an wird „Frameworking" nicht mehr nur nützlich und praktisch für den internen Produktionsprozess sondern eine absolute Notwendigkeit sein.

Die hier gegebene Definition von „Frameworking" vermeidet absichtlich eine Bindung an Objekttechnologie.

Zwar wurde NEWSYS unter Verwendung von Objekttechnologie entwickelt und deshalb liegt der Schwerpunkt des Papers auch darauf, aber dies ist keine notwendige Voraussetzung für ein Framework. Viele Operating Systems sind z.B. Nicht-OO Frameworks (Device Driver, pluggable Filesystems, Policies für User/Security, streamsbasierte protocol stacks etc. sind alles Puzzleteile und entsprechen dem Lifecycle Aspekt eines Frameworks)

Ausserdem, und dies ist eine persönlich gefärbte Aussage, habe ich nach wie vor Probleme mit Klassifikationssystemen. Sie sind immer unter gewissen Blickwinkeln und Annahmen entstanden. Und während sie sich im Alltag praktisch einsetzen lassen (und auch notfalls abändern) erweisen sich bisher objektorientierte Systeme beim Wechsel von Perspektiven als sehr unflexibel. Änderungen der Klassifikationen sind aufwendige Massnahmen. Hingegen sind strukturelle Transformationen z.B. von ausgezeichneten (mit Metainformation versehenen) Informationen/Daten relativ leicht und automatisch machbar. In einer Zeit wo sich Organisationen laufend umstrukturieren, d.h. ihre internen Klassifikationssysteme umbauen und flexibler gestalten ist eine Software auf Basis von starren Klassifikationssystemen problematisch.

Beispiel: Class Customer

Wenige Entwickler von Business-Software zögern eine Klasse „Customer" zu definieren. Mit etwas Glück erkennen sie rechtzeitig, dass die Klasse Customer besser nicht von der Klasse Person abgeleitet wird, da es juristische Personen als Customer gibt. [FOWLER] Aber wie sieht es mit dem ontologischen Gehalt von Customer aus? Customer sieht wie eine Basisklasse aus – ein sog. Business Object. In Wirklichkeit handelt es sich bei Customer bereits um eine Relation zwischen anderen Entitäten. Diese Relation kann sich mit der Zeit verändern. Neue Aspekte kommen hinzu. Eventuell wird aus einem Customer ein Partner oder ein Teil der eigenen Firma.

Momentan zeichnet sich ausserdem durch die Agent-Technologie ein Trend ab, der mehr die Kapselung und „Lebendigkeit" eines Objektes betont als die Position innerhalb einer Klassifikation. Unübersehbar ist auch der Trend in Richtung mehr Metainformation, sei es bei Objektdatenbanken oder beim Austausch zwischen Komponenten (Java‘s Infobus z.B. tauscht Daten aus).

Chapter 3. DEFINITION DER STRUKTUREN EINES FRAMEWORKS

Sind Strukturen wirklich?

Ich weiss es nicht. Momentan glaube ich, dass Strukturen ein Konstrukt unseres Verstandes sind und dass Entities der Realität Hinweise sind nach Strukturen zu suchen, sie zu konstruieren.

Es gibt einige Belege dafür: Strukturen, speziell software Strukturen sind unsichtbar. Wenn sie es nicht mehr sind dann entweder weil sie jemand im Source Code visuell ausgezeichnet, „getaggt" hat oder weil wir mittlerweile Erfahrung gewonnen haben die es uns gestattet, anhand von Source Code Stücken sofort davon betroffene Strukturen zu erkennen.

In bezug auf Strukturen in Frameworks haben wir folgende Situation:

Framework Strukturen sind:

- unsichtbar
- schwer zu erklären und zu visualisieren
- gefährlich für ein Projekt wenn sie ignoriert werden (unbeschadet ihrer unsichtbaren Qualität).

Wie wir an den Source Beispielen oben gesehen haben, kann ein einziges Statement zu mehreren Strukturen gleichzeitig gehören ohne dass irgendetwas in diesem Statement auf diese Tatsache hinweisen würde. Mit einem Statement kann man seine sorgfältig entworfenen logischen Abstraktionen korrumpieren, die aufwendig kreierten physischen Trennungen ruinieren und Erweiterbarkeit und Wartung schwer einschränken.

Zumindest in ihren Konsequenzen sind Software Strukturen real.

Der härteste Teil bei der Behandlung von Software Strukturen in Frameworks ist, sensitiv für die Hinweise in Source Codes zu werden die anzeigen, dass bestimmte Strukturen davon betroffen sein könnten. Object Creation statements sind ein Beispiel dafür. Include Files ein anderes.

Und wenn man diese Hinweise entdeckt hat ist es wichtig deutlich zu machen, zu welchen Strukturen sie gehören und sie entsprechend auszuzeichnen, zu markieren. Coding Standards mit Regeln für Namensgebung etc. sind ein Platz dafür. Sonst werden Neulinge sie nicht wahrnehmen.

Und wenn wir das alles getan haben, kommt das nächste Problem: Sind diese Strukturen miteinander verknüpft und wie kann man dies visualisieren?

Es ist nicht schwer zu zeigen, dass Strukturen in Frameworks miteinander verknüpft sind:

Moderne Software muss reflektiv sein, mit anderen Worten, sie muss über sich selbst Auskunft geben können. Ein Teil der dazu notwendigen Information entsteht während des Build Prozesses, z.B. über package und Komponentennamen, Versionen, Interface Versionen etc. Die Build Struktur muss diese Informationen gewinnen und sie zurück in die Source Struktur leiten (z.B. über compiler defines). Elemente der logischen Struktur werden zur Laufzeit von der Runtime Struktur benutzt um diese Informationen zu verarbeiten, z.B. auf Interface mismatches hinzuweisen. Wenn das Framework selbst zur Laufzeit neue Klassen generieren soll, wird die Verbindung zur Source Struktur noch enger.

Die schwere Frage ist nochmals: Wie erklären und zeigen wir diese Verbindungen und Abhängigkeiten?

Framework Strukturen geben uns 3 Probleme zu lösen:

1. Wie finden und erklären wir Strukturen und ihre Konsequenzen?
2. Wie machen wir sie explizit, d.h. programmatisch und visuell sichtbar?
3. Wie machen wir ihre Verbindungen ebenfalls explizit und programmatisch/visuell sichtbar?

Technische Strukturen

Einige der wichtigsten technischen Strukturen eines Frameworks sind:

1. analytische Struktur
2. reflektive Struktur
3. logische Struktur
4. physische Struktur
5. runtime Struktur
6. Erweiterungsstruktur
7. Source Struktur
8. Generierungs Struktur
9. Usage Struktur

Theoretisch könnten alle verschieden Strukturaspekte in einem einzigen Objekt Modul (file) zusammengefaßt werden. Erweiterungen (kundenspezifische, branchenspezifische oder einfache Fehlerbehebungen würden in diesem File stattfinden. Danach würde neu kompiliert und gelinkt und ein neues Release wäre fertig.

Dies ist bereits bei der Verwendung von C nicht praktisch und C++ stellt darüber hinaus besondere Anforderungen an die physische Struktur.

Die physische Struktur von grösseren C++ Paketen hat wiederum Einfluß auf die logische Struktur, was sich z.B. in besonderen Designtechniken wie Interface/Implementation Trennung oder bestimmten Hilfsklassen wie Factories und Brokern zeigt.

Bezeichnend ist, dass zwischen den einzelnen Strukturen Abhängigkeiten bestehen. Z.B. Sollten die Elemente der physischen Struktur ebenfalls als Klassen modelliert (reflektiert) werden. Dies setzt jedoch Input von der Source Struktur voraus.

Analytische Struktur

Die Komplexität der analytischen Struktur hängt von der Zielrichtung des Frameworks ab. Hier kann man unterscheiden zwischen erweiterbaren und generischen Frameworks.

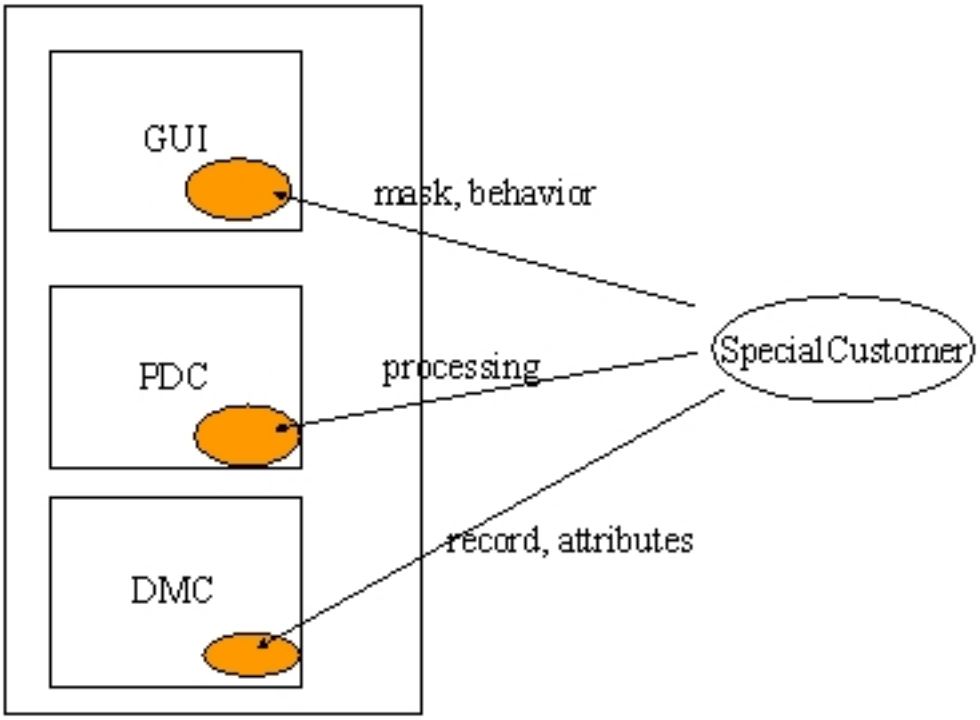
Erweiterbare Frameworks

Erweiterbare Frameworks lassen in ihren Strukturen Leerstellen, die Anwender durch Einfügen der fehlenden Stellen für ihre Zwecke spezialisieren können. Hier ist das Framework EINFACH, die Arbeit der Anwender üblicherweise relativ komplex.

Der Abstraktionsgrad ist noch nicht sehr hoch, d.h. der Gegenstandsbereich wird teilweise direkt in der Maschine abgebildet. Neue Typen des Gegenstandsbereichs erfordern oft neue Verarbeitungsobjekte im User Interface, im Datenbank Management etc. Dynamische Casts zur Behandlung spezieller Objekte sind teilweise nötig.

Typischerweise gibt es keine Meta Infomationsmodelle. Der Anteil wirklich generischer Module, d.h. Module die einen grossen Gegenstandsbereich ohne extra Anpassung verarbeiten können ist gering. Damit hat die ana-

lytische Struktur grossen Einfluss auf die physische und extension Struktur.



A simple Framework based on the puzzle metaphor with hot spots for customization

Erweiterbare Frameworks werden durch die Puzzle Metapher gut beschrieben. Zur Laufzeit sind sie Instanzen spezieller Applikationen. Die analytische Struktur eines erweiterbaren Frameworks leistet im wesentlichen eine „hot spot" Analyse auf Objektlevel.

Genau das Konzept der Flexibilität auf Objektlevel ist jedoch auch die grösste Schwäche des Puzzle-Modells. Wie das Diagramm zeigt, spaltet sich ein Objekt in die verschiedensten Aspekte auf und betrifft ganz unterschiedliche Strukturen. Das Puzzle-Modell funktioniert nur, wenn der Aspekt den das Objekt ausdrückt auf logischer Ebene GENAU LOKALISIERBAR ist. Dies ist typischerweise der Fall in Sub-Frameworks wie z.B. einem Converter Framework Die Unterstützung eines neuen Text Formates ist eines der Paradebeispiele für „Plug and Play" durch Objektaustausch. Anwendungen im Source Code dazu finden sich im ET++ und MET++ Framework [ACKERMANN]

Generische Frameworks

Generische Frameworks sind in ihrem Aufbau sehr KOMPLEX, erlauben aber das einfache Behandeln einer grossen Zahl unterschiedlicher Gegenstände. Sie sind generische Maschinen zur Bearbeitung von Gegenständen der Realität. Eine solche Maschine setzt einen hohen Grad an Abstraktion des Gegenstandsbereiches voraus. Die analytische Struktur muss diese Abstraktionen liefern.

Innerhalb der analytischen Struktur findet eine Dekomposition der Gegenstände der Realität in Grundstrukturen statt. Das Framework arbeitet auf diesen Grundstrukturen. Es gibt in einem Framework nichts was z.B. direkt einem bestimmten Dokument Objekt entsprechen würde. Es gibt keine direkte Entsprechung zu einem speziellen Use Case.

Als generische Maschine kann ein Framework keinen konkret auf spezielle Gegenstände bezogenen Code beinhalten. Überhaupt ist es nicht gut in Framework Code direkt Gegenstände der Realität abzubilden. Es gibt im allgemeinen kein singuläres Programmobjekt das einem Objekt der Realität entsprechen würde (schlicht deshalb weil die Objekttechnologie bisher in keinster Weise echte Objekte abbilden kann und ihnen zudem die ontolo-

gische Qualität (noch) fehlt). Als Folge dessen ist auch ein dynamisches Casten meist nicht sinnvoll – WORAUF SOLLTE DENN GEKASTET WERDEN wenn es keine 1:1 Entsprechungen gibt?

Generische Frameworks bestehen aus generischen Modulen die Verarbeitungsanweisungen nicht im Source Code einprogrammiert haben sondern aus Meta Informationen beziehen. Gegenstände der Realität sind in Form von Metamodellen beschrieben auf denen alle Module des Frameworks zugriff haben. Diese Metamodelle sind üblicherweise selbstbeschreibend.

Zur Laufzeit erfolgt im Framework eine Umsetung der Meta Informationen in Cluster und Bäume aus konkreten Programmierobjekten des Frameworks. Der Unterschied zwischen Meta Information als Abbildung der Realität und dessen Umsetzung in Programmobjekte bleibt deutlich sichtbar.

Ein Beispiel:

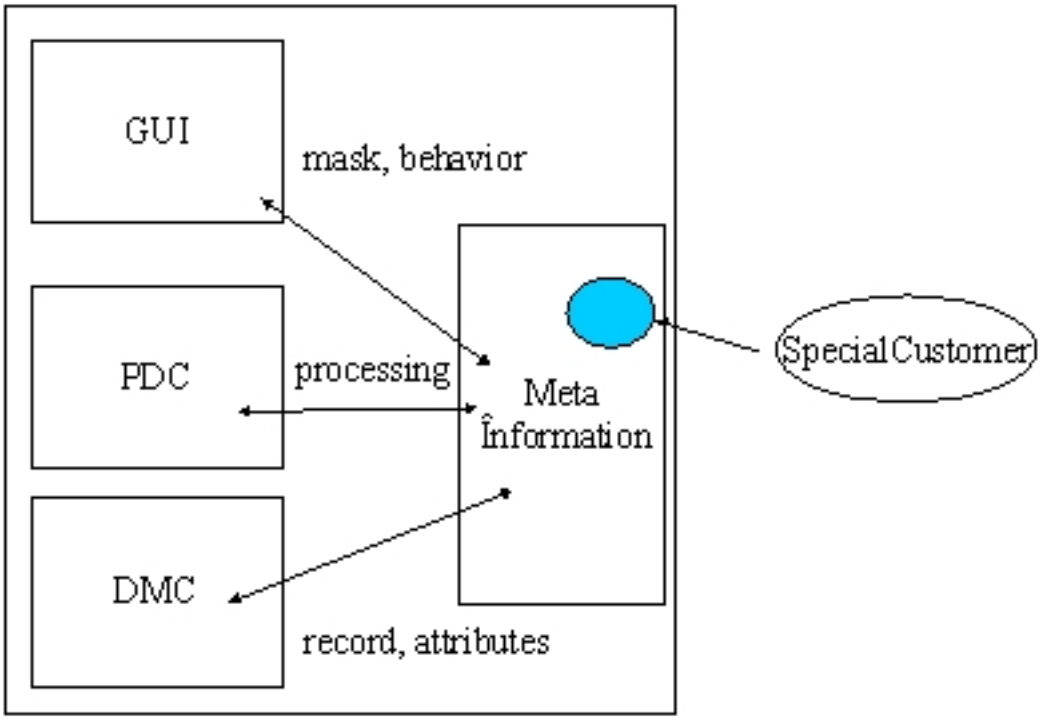
Es gibt ein Objekt „Kunde" des Gegenstandsbereichs „Banking Business". In einem generischen Framework findet sich keine Klasse „Kunde", keine Klasse „Kundenrecord" in Headerfiles, kein GUI ressource file für eine „Kundenmaske". Was das generische Framework benötigt um ein Objekt „Kunde" des Gegenstandsbereichs zu verarbeiten ist Meta Information über ein solches Objekt.

Diese Metainformation existiert nur ein einziges Mal und ist selbstbeschreibend. Das Framework kann aus dieser Metainformation Views für das GUI sowie datenbank schemata generieren. Es kann ein „Kunden" Objekt des Gegenstandsbereiches durch ein composite Object bestehend aus generischen Objekten des Frameworks repräsentieren.

Neue Anforderungen, z.B. Spezialisierungen von „Kunde" sind plötzlich durch Änderungen der Meta Informationen möglich und erzwingen keine neuen Module (Puzzle Teile)

Hauptzweck der analytischen Struktur ist Abstraktion durch Dekomposition. Eine Besonderheit dabei ist, dass dabei MetaInformation entsteht die programmatisch sichtbar werden muss, d.h. das Framework muss darauf programmatischen Zugriff haben.

Dies findet in der reflektiven Struktur (s.u.) statt. Das wiederum bedeutet, dass die meta information NICHT Dokumentation im Sinne zusätzlicher Schriftstücke sein kann. Sie muss zur Laufzeit zur Verfügung stehen, gepflegt und angepasst werden. Meist geschieht das mit Hilfe des Frameworks selbst, d.h. mit Hilfe von logischen und physischen Elementen die aber zur Reflektiven Struktur des Frameworks gehören.



A generic framework that represents objects with meta information.

Ganz ähnlich einem „aspect weaver" [AOP] setzt das Framework zur Laufzeit eine Instanz von sich so zusammen, dass das Objekt des Gegenstandsbereiches bearbeitet werden kann.

Die analytische Struktur eines generischen Frameworks erarbeitet die Meta Informationen. Die Granularität der „hot spots" ist nicht mehr nur auf Objekt Level sondern lediglich durch die Mächtigkeit der Meta Information beschränkt. Generische Frameworks gleichen eher Betriebssystemen als speziellen Applikationen.

Reflektive Struktur

Die reflektive Struktur ist die Reflexionsebene [KZIK] eines Frameworks. Reflektiert werden zwei Dinge:

1. Gegenstandsbereich (Business Domain) in Form von Meta Information die in der analytischen Struktur erarbeitet wurden.
2. Die eigenen Software Strukturen und Mechanismen

Somit ist nicht nur die Bearbeitung von Business Objekten eine Aufgabe für ein Framework sondern auch die eigenen Software Strukturen die über die Flexibilität und Erweiterbarkeit entscheiden.

Typische Reflektive Elemente sind das Interface bzw. Implementation Repository und Meta Struktur Beschreibungen wie der Aufbau eines Records einer Datenbank.

Die Reflektive Struktur eines Frameworks bietet einen KONSTRUKTIVEN, PROGRAMMATISCHEN Zugriff auf die Ergebnisse der Analytischen Struktur.

Logische Struktur

Unter logischer Struktur eines Software Paketes versteht man die Faktorisierung des Problembereichs ohne Beachtung physischer Komponenten. (z.B. „Domains/Services", Base, HIC, Schnittstellen von außen (Bibliotheksverwendung) und innen (Einfügen von applikationsspezifischen Erweiterungen) etc.

Wie wichtig es es, die logischen und physischen Aspekte GEDANKLICH zu trennen zeigt sich daran, dass im NEWSYS Projekt die logischen Domains weiterhin verwendet werden konnten (HIC,PDC,DMC). Sie waren auch keineswegs falsch gewesen sondern lediglich auf undurchsichtige Weise mit physischen Aspekten vermischt worden.

Immer unterscheiden zwischen der logischen Komponente XX als Service oder Domain und den sie realisierenden physischen Packages XX.exe, XX.dll, BranchXX.dll, CustomerXX.dll, WhatEverXX.dll, WhatEver.exe!

Die logische Stuktur enthält unter anderem:

- Zentrale Abstraktionen, z.B. das verwendete Dokumentenmodell
- Metaobject Protokolle
- die Ableitungsverhältnisse (Basisklassen, Defaultimplementationen, Applikationsspezifische Ableitungen)
- das Objekt Modell (Root Klasse, Object Request Broker, Hilfsklassen, Konkrete Klassen)
- die verwendeten Design Patterns (möglichst als programmatische Entities!)
- die Verwendungsregeln z.B. bezüglich des Memory Managements und Reference Countings
- Konfigurationen als Meta Informations Dokumente
- Logische Aspekte der Interface/Implementation Trennung (HotSpot Design)

In NEWSYS entstand Dokumentation in Design Pattern Struktur zu:

Dokumentenmodell (Parts)

ORB, Memory Management und Object Modell(Superfactory NSys)

Implementation von Factories,

der Integration von Fremdprodukten durch Wrapper und Adaptoren

Meta Information (System Config, Documents, Database schemas)

Sub-frameworks (data management, Entity Management)

Software Wartung mit Design Patterns

Überraschenderweise liessen sich Design Patterns sogar zur Software Wartung einsetzen. So konnte in einem Fall durch die Verwendung des Bridge patterns eine Implementation eines Composite Message Patterns durch zwei getrennte Implementationen ersetzt werden OHNE dass sich Änderungen am Interface ergeben hätten. Die physischen Auswirkungen blieben auf ein Package begrenzt. Kein anderes Package musste compiliert oder neu gelinkt werden.

Dies zeigt andererseits nur, dass die Trennung logisch – physisch eine gedankliche ist. Beide Strukturen lassen sich auch aufeinander anwenden.

Viele Teile der logischen Struktur richteten sich an Standards aus, deshalb sei an dieser Stelle auch auf die Dokumentation zu CORBA und ORBIX sowie auf die einschlägige SGML Literatur verwiesen.

Ein Hinweis auf die Zugehörigkeit einer Klasse oder Funktion zu einer logischen Domain ergibt sich aus dem jeweiligen Kürzel am Anfang des Namens. z.B. sind alle SyBxxxxxx.xxx Teil der logischen BASE.

(zu den Namens- und anderen Konventionen von NEWSYS siehe unten: Coding Standard)

Physische Struktur

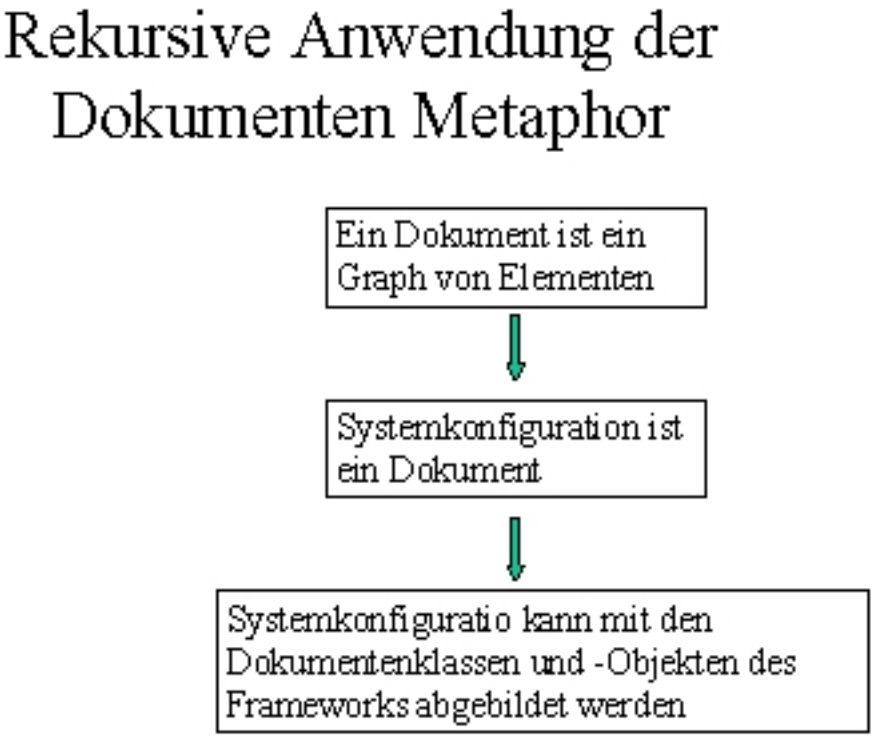
Unter der physischen Struktur eines Software Paketes versteht man

- Compile- und Linktime Abhängigkeiten von Objekt Modulen
- Aufteilung von Repositories, Configuration Information in physische Entities
- Trennungen von Interfaces und Implementationen
- Default Implementationen
- Object Creation / Factories
- Die Zusammenfassung von Objekt Modulen zu Komponenten (Packages) sowie die Definition deren Funktionalität. Damit verbunden ist die Organisationsweise des Source Code Trees.
- Die Definition bestimmter Packages als zum Basissystem gehörig bzw. applikationsspezifische Erweiterung
- Compile- und Linktime bzw. Runtime Abhängigkeiten der einzelnen Packages untereinander
- Die Auslieferungsform (Dlls, libraries, exes, config files etc.)
- Performance Aspekte (z.B. Degeneration von Implementationen durch Zahl oder Umfang der Objekte)
- Scaling Structure: Wo sind cold spots die bei Erweiterungen physikalisch nicht skalieren?

Bei der physischen Struktur kommt deutlich eine selbstbezügliches Element des Frameworks zur Geltung.

Damit die physiche Struktur ihre Leistungen erbringen kann, stützt sich sich ihrerseits wieder auf Mechanismen der logischen Struktur (Klassen, Regeln etc.) Dieses Element der Selbstbezüglichkeit taucht immer wieder auf, z.B. wenn Konfigurationsfiles als Dokumente aufgefaßt werden, damit sie mit denselben Mitteln des Frameworks für allgemeine Dokumenttypen bearbeitet werden können OBWOHL sie das Framework selbst beschreiben.

So verwendet das Konfigurationsframework innerhalb von NEWSYS genau wie eine Belegapplikation die Dokument Parts des Frameworks. Dies verringert die Zahl der Klassen und Interfaces enorm und bringt leichtere Handhabbarkeit und Polymorphie im Einsatz der Klassen:



Beispiel:

Die eingangs erwähnten Serviceprobleme mit Konfigurationsfiles unter OLDSYS. Bei der manuellen Bearbeitung der Dokumentenbeschreibung entstanden häufig syntaktische und semantische Fehler. OLDSYS war nun seinerseits ein System zur Bearbeitung der Zeichen, Syntax und Semantik in Belegen. Wenn man jetzt selbstbezüglich die Dokumentenbeschreibungen unter OLDSYS als Beleg auffaßt, dann müßte OLDSYS alle Mechanismen bereits besitzen, um mit Hilfe von Prüfroutinen (die teilweise belegspezifisch noch erstellt werden müßten) eine Bearbeitung der EIGENEN Dokumentbeschreibungen zu ermöglichen.

Der ohnehin nötige Editor zum Bearbeiten der Belege könnte dann auch zur Bearbeitung der eigenen Konfigurations Informationen verwendet werden. Den Entwicklern von OLDSYS war diese Analogie nicht aufgefallen und es wurde versucht, einen extra Editor zu entwickeln statt den vorhandenen genereller zu machen.

Unter OLDSYS scheiterte dieses Konzept zusätzlich an der engen Kopplung von Datenhaltung und Editieren.

Das Prinzip der Selbstreferenz in Verbindung mit einem hohen Abstraktionsgrad ist eines der wichtigsten Design Pattern auf System/Architektur Ebene. Gelungene Systeme zeichnen sich durch einen hohen Grad an Selbstreferenz aus.

(vgl. z.B. das Stream Design Pattern das an vielen Stellen in Unix gleich verwendet wird. Das erleichtert sowohl die Implementation als auch die Benutzung.)

Wesentlicher Bestandteil der physischen Struktur eines Frameworks sind die Eigenschaften der zugrundeliegen-

den Programmiersprache. Für C++ Programmierung bedeutet dies u.a. die Kenntnis von:

1. Bedeutung, Struktur und Verwendung der Vtable bei single und multiple inheritance
2. Behandlung der vom Compiler generierten Methoden (Wann und wie werden sie generiert, wo legt sie der Compiler hin)
3. Aufbau von Objektlayout (compiler abhängig)
4. Konsequenzen aus dem compilerabhängigen binären Layout
5. Konsequenzen aus Source Changes auf das binäre Layout
6. Name mangling und dynamisches Laden
7. DLLs mit C++ Modulen
8. Probleme der statischen Initialisierung
9. Konsequenzen der Verwendung von inlines, templates und default parametern auf den Software Lifecycle.
10. memory allocation operatoren und debugging möglichkeiten bei memory leaks
11. Compilerabhängige Behandlung von Templates
12. Möglichkeiten der Garbage Collection und Meta Object Protokolle für C++

Ein fertiges Framework schützt seine Benutzer vor vielen physischen Details. Während seiner Entwicklung allerdings müssen physischer Eigenschaften der Programmiersprache explizit gemacht und in ihren Konsequenzen verstanden werden. Viele Applikationsprogrammierer machen diese schmerzliche Erfahrung erst wenn es zu spät für ein Projekt ist.

Literatur dazu: [LACOS],[MEYERS1/2], [TALIGENT1], [M.ELLIS], [J.ELLIS]

Runtime Struktur

In diesem Paper wird die Runtime Struktur als Unterstruktur der physischen Struktur aufgefasst, quasi als deren dynamischer Aspekt.

Konzepte der Runtime Struktur beinhalten:

- Boot Prozess
- Fehlermeldungen

- Ressource Allocation

Extension Struktur

Die Extension Struktur beinhaltet alles was zur Anpassung des Frameworks bzw. zur Erweiterung (kernel, branches, customer specific) nötig ist. Dazu gehört auch die Software Wartung im Fehlerfall bzw. automatische Updates bei neuen Releases.

Source Code Struktur

Die Source Code Struktur regelt:

- wie neuer Source Code entsteht,
- wo er plaziert werden muss,
- welchen Konventionen er unterliegt,
- wie Source Code angepasst wird
- Plattformunabhängigkeit des Source Codes.

Source Code Control

Concurrent Source Code Management beschreibt nur einen winzigen Teil dieser Struktur. Im Zusammenhang mit konkreten Produkten wie PVCS oder CVS/RCS tauchen meist zwei Fragen auf:

1. Filesystem basierte Lösung oder echter client/server Betrieb
2. Kann ein Entwickler ein File für Zugriffe sperren, z.B. weil er Modifikationen durchführt.

Aus Gründen des Zugriffsschutzes ist ein echter client/server Betrieb vorzuziehen, z.B. mit dem public domain Tools CVS/RCS. Ein weiterer Vorteil dieser Lösung ist, dass die tatsächliche Position von Files auf dem Server unsichtbar bleibt, d.h. es können Reorganisationen durchgeführt werden ohne dass Clients davon etwas merken.

Ein „modify mode" mit file locking setzt voraus, dass die Zeitspanne die das File gelockt bleibt abschätzbar und kurz bleibt. Ansonsten blockiert eine umfangreiche Änderung andere Entwickler die einen anderen Aspekt des Files ändern wollten.

Ohne modify mode können zwei Entwickler gleichzeitig Änderungen durchführen und derjenige der als zweiter ein commit durchführen will muss erst die Änderungen des ersten in seine neuere Version einführen. Dieser Nachteil ist jedoch kombiniert mit dem Vorteil, dass sich beide Entwickler während der Modifikationen (die ja auch länger dauern können) nicht gegenseitig blockieren.

Source Code Organisation

Qualität und Reuse von Componenten hängt zentral von automatischen Builds und Installations ab, die plattform- und compilerunabhängig sowie ohne jeden administrativen Aufwand auf Seiten eines Applikationsentwicklers funktionieren müssen. Diese wiederum sind auf eine sinnvolle Source Struktur angewiesen. Neue Mitarbeiter müssen sich schnell auf der eigenen Maschine, auf Maschinen von Kollegen bzw. dem Source Server zurechtfinden können.

Die Source Struktur muss jederzeit umbaubar sein, wenn sich z.B. die Zugehörigkeit von Sources zu Packages oder Domains ändert ohne dass dies zu Aufwand auf Seiten der Entwickler führt. Meist wird zur Gliederung der Source Struktur eine Directory Struktur verwendet. In der Praxis werden diese Directories im Laufe eines Projektes mehrfach umorganisiert.

Die Source Struktur muss nicht physische Komponenten spiegeln, z.B. dass alle Sources einer Komponente innerhalb eines Directories liegen. Der Build Prozess sollte von der Struktur des Source Codes unabhängig sein aber auf Regeln der Source Gliederung vertrauen können, z.B. dass Interfaces getrennt von Implementationen untergebracht sind etc.

Wenn das Framework zur Laufzeit selbst neue Classes generieren soll, muss die Source Code Struktur auch als Repository dienen können.

Die Flexibilität der Extension Strukur hängt ebenfalls stark von der Source Code Struktur ab.

Die Organisation des Source Codes definiert Grenzen:

1. Zugehörigkeit zu logischen Domains

2. Interfaces und Implementations
3. Framework Kernel Sources
4. Branchenspezifische Sourcen
5. Customer spezifische Sourcen
6. Systemabhängige Sourcen

(Die OMG unterteilt ihre Funktionalität z.B. nach ORB, Services und Facilities, Profiles und Components)

Generative Struktur

Ein Framework Design versucht nicht nur durch generische Verfahren Änderungen zu vermeiden sondern antizipiert sie. Der source code „an sich" wird zum Thema einer Framework Entwicklung, eine Erfahrung die übrigens jedes grössere Projekt macht wenn die Anzahl der Source Files hundert übersteigt und manuelle Verfahren sehr aufwendig werden.

Eine weitere Erfahrung aus grösseren Projekten ist, dass viele Anpassungen sehr einfacher Natur sind aber eine grosse Zahl von Files betreffen. Für diesen Problembereich eignen sich generative Verfahren besonders gut.

Das Problem der Source Code Anpassungen wirft zwei Fragen auf:

1. Wie können sie vermieden werden?
2. Falls sie doch unvermeidlich sind, wie können sie automatisiert werden?

Eine Regel für grössere Source Changes in einem Framework lautet: Systematische Änderungen durch manuelle Anpassungen aller Files SIND VERBOTEN. Es muss ein Tool verwendet oder notfalls geschrieben werden, um die Anpassungen automatisch durchzuführen.

Beispiel 1, Vermeiden von Source Code Änderungen:

Die Anforderung lautet, dass alle Implementationen von Interface Klassen bestimmte Metainformation im Interface zur Verfügung stellen müssen, z.B. den eigenen Classname, die Source Version ID und die Interface ID.

Der Classname ist im Source Code bei der Class declaration definiert, z.B.

in X.hpp:

Class X : public Y {..}

Er ist jedoch nicht PROGRAMMATISCH verfügbar, d.h. er ist nicht als String definiert. Eine Möglichkeit besteht darin, einen Macro zu verwenden um die gewünschte className() Methode zu generieren. Dies kann nur INNERHALB der class declaration geschehen, da der Macro keine Block Grenzen durchbrechen kann.

Class X : public Y {

SY_BASE_CLASS_NAME_METHOD(X)

}

wobei Macro folgendermassen definiert ist:

#define SY_BASE_CLASSNAME String_ref className(void) { return toString(X) };

Die Implementation des Macros befindet sich im Basis System Header des Frameworks, d.h. es wird ein kompletter Recompile nötig werden. Dies ist sofort erkennbar da die Namenskonvention den Macro gleichzeitig mit

seiner logischen Domain auszeichnet. Der Macro selbst unterliegt der automatischen Dokumentation, d.h. er wird automatisch als Funktionalität mitdokumentiert.

Die Rückgabe der Source Version ID (z.B. der RCSID des Revision Control Systems) ist bereits etwas komplizierter, da sie nur im .cpp enthalten ist, d.h. sie kann nicht über eine inline Methode zurückgegeben werden. Sowohl in der Class declaration ist eine neue Methode nötig als auch im .cpp File eine neue Implementation dieser Methode.

In X.hpp:

Class X : public Y {

SY_BASE_CLASS_NAME_METHOD(X)

SY_BASE_VERSION_NUMBER_DECL(X)

}

in X.cpp:

SY_BASE_VERSION_NUMBER_IMPL(X)

Wobei die VERSION_NUMBER Macros folgendermassen deklariert sind:

#define SY_BASE_VERSION_NUMBER_DECL(X) String_ref versionNumber();

und

#define SY_BASE_VERSION_NUMBER_IMPL(X) String_ref X::versionNumber()

{ return toString(RCSID);}

Die Interface ID stellt klar, zu welcher Interface Version eine Implementation kompatibel ist. Sie kann im Build Prozess generiert werden und wird bei der Compilation über ein Define in den Source Code hineingereicht.

(CC –DINTERFACEID=123456767 –DPACKAGE=MyPackage)

Class X : public Y {

SY_BASE_CLASS_NAME_METHOD(X)

SY_BASE_VERSION_NUMBER_DECL(X)

SY_BASE_INTERFACE_ID(X) }

in X.cpp:

SY_BASE_INTERFACE_ID_IMPL(X)

Wobei die Macros folgendermassen deklariert sind:

#define SY_BASE_INTERFACE_ID_DECL(X) String_ref interfaceID();

und

#define SY_BASE__INTERFACE_ID_IMPL(X) String_ref X::interfaceID()

{ return toString(INTERFACEID);}

Es können noch weitere Macros hinzukommen, z.B. für Persistence oder Garbage Collection, oder die Generierung von Exception Implementations. d.h. es ist sinnvoll die Erweiterungsmöglichkeiten des Source Codes als HOT SPOTS zu definieren und zu implementieren:

Dazu werden Macros definiert, entsprechend den typischen Stellen wo Änderungen im Source passieren:

SY_BASE_BEFORE_DECLARATION_CLASSTYPE(Classname, BaseClass,Package)

SY_BASE_CLASS_DECLARATION_SINGLE(Classname,BaseClass,Package)

SY_BASE_CLASS_DECLARATION_MULTI[1-n](Classname,Base[1-n] ,Package)

SY_BASE_IN_DECLARATION_CLASSTYPE(Classname, BaseClass,Package)

SY_BASE_IN_IMPLEMENTATION_CLASSTYPE(Classname, BaseClass,Package)

SY_BASE_AFTER_DECLARATION_CLASSTYPE(Classname, BaseClass,Package)

X.hpp:

SY_BASE_BEFORE_DECLARATION_CLASSTYPE(X,Y,Package)

SY_BASE_CLASS_DECLARATION_SINGLE(X,Y,Package)

{

SY_BASE_IN_DECLARATION_CLASSTYPE(X,Y,Package)

.....

}

SY_BASE_AFTER_DECLARATION_CLASSTYPE(X,Y,Package)

X.cpp:

SY_BASE_IN_IMPLEMENTATION_CLASSTYPE(X,Y,Package)

Die einzelnen hot spot macros beinhalten dann z.B. die obigen einzelnen Marcros:

#define SY_BASE_IN_DECLARATION(Classname,BaseClass,Package)

SY_BASE_CLASS_NAME_METHOD(ClassName)

SY_BASE_VERSION_NUMBER_DECL(ClassName)

SY_BASE_INTERFACE_ID(INTERFACEID)

.....

Wenn jetzt zusätzlich eine Methode zur Rückgabe des Package Namens gewünscht wird dann braucht nur der hot spot macro um einen weiteren Macro erweitert werden. Alle anderen Sources bleiben unberührt. Fremdprodukte wie OO Datenbanken und Garbage Collectors brauchen häufig extra Methoden.

In diesem Schema ist eine weitere Indirektion – und damit eine weiterer hot spot – möglich: Die Macros die im hot spot macro enthalten sind müssen keineswegs dort auch definiert sein. Sie können sogar zur Compile Time vom Build System erst generiert werden und damit sehr speziell auf bestimmte Packages zugeschnitten sein.

Vermeiden oder Eingrenzen von systematischen Source Code Änderungen wird wesentlich erleichtert wenn das Framework in der Source Struktur bereits hot spots definiert, die dann automatisch erweitert werden können. Die Änderungen im Source Code werden daher in der analytischen Struktur des Frameworks bereits vorgesehen.

Die automatische Source Code Erweiterung beruht wiederum auf der Unterstützung durch die Generierungsstruktur des Frameworks (Bereitstellung von Meta Information) sowie den Coding Conventions (Namensgebung und Tools zur Erstellung von Source Templates die die hot spot macros mit den richtigen Parametern enthalten.

Generative Tools sorgen dafür dass entstehender Source Code gleich dem Framework Coding Standard entspricht. (siehe newsrc Tool unter „Coding Standards and Tools")

Das eben geschilderte Vorgehen setzt voraus, dass Änderungen antizipiert wurden und Mechanismen zur automatischen Anpassung bereits existieren. In vielen Fällen lassen sich Anpassungen jedoch nicht vorhersehen. Auch für diesen Fall müssen Mechanismen im Framework existieren. Was wäre wenn die hot spot macros aus dem eben geschilderten Beispiel noch nicht im Source Code enthalten wären?

Beispiel 2, Automatisierung unvermeidlicher Änderungen durch einfache Tools.

Speziell im Unix Bereich sind eine Reihe von Tools entstanden, mit denen Texte automatisch modifiziert werden können, z.B. Stream Editoren wie sed oder das awk tool oder Macro Präprozessoren wie m4. Unix Kernel Builds verwendeten sie häufig. Diese Tools sind heute auf allen Plattformen in public domain Versionen erhältlich. Zeitweise wurden auch C Präprozessoren eingesetzt, die jedoch im Zuge der ANSI Standardisierung ihren Wert als charakter based Macro Prozessoren verloren, da sie plötzlich teilweise die Syntax der Programmiersprache kannten.

Die hot spot macros liessen sich mit Hilfe dieser Tools relativ einfach und automatisch nachträglich einfügen. Dazu würde im NEWSYS Framework die Build Struktur um einen Processing Step erweitert.

Voraussetzung einfacher Tools ist immer eine im wesentlichen kontextarme Struktur der Texte. Coding Conventions die ein Taging Schema vorschreiben bzw. bereits generieren erleichtern den Einsatz einfacher Tools sehr.

Beispiel 3, Automatisierung der Anpassungen durch komplexe Workbenches wie Together++, OEW, ParadigmPlus.

Selbst rein kosmetische Anpassungen von Source Code sind dem C++ Programmierer ein Greuel, da typischerweise ganze Tage damit verloren gehen, nur ein anderes Namensschema einzuführen - ohne irgendwelche funktionalen Änderungen. Type changes in Interfaces oder neue Interface Methoden lösen ebenfalls ganze Lawinen von Anpassungen bei den Implementationen aus.

Zu den Erfahrungen im NEWSYS Framework mit komplexen Workbenches siehe unter „Coding Standards und Tools". Speziell für den Bereich der Source Struktur lassen sich einige Regeln für die Verwendung solcher Tools aufstellen, wenn sie innerhalb eines Frameworks stattfinden soll.

1. selbst komplexe und integrierte Tools müssen sich ihrerseits integrieren lassen, z.B. in einen Build process.
2. Tools müssen eine „offene Implementation" bieten, d.h. ihre Mechanismen selbst müssen modifiziert und erweitert werden können. Könnte z.B. die obige Erweiterung des Source Codes (unter Mithilfe von Meta Information aus dem Build Process) innerhalb einer Workbench durchgeführt werden?
3. Stellen die Tools ihre Ergebnisse auch in einer Form zur Verfügung, die anderen Tools gestattet sie auszuwerten?
4. Können die Tools auch modifizierte Stellen weiter bearbeiten oder überschreiben sie diese jeweils neu?

Damit ein Tool innerhalb eines Frameworks verwendet werden kann, muss es viel OFFENER sein als im Rahmen einer normalen Applikation.

Beispiel 3, automatische Anpassung durch einen eigenen Preprocessor

Dieses Konzept wurde z.B. innerhalb von Fresco angewendet. Als CORBA basiertes System war ein IDL Compiler nötig. Dieser Compiler wurde dann so erweitert, dass er auch die Anpassung von Implementationen bei Interface Änderungen leisten konnte. In einem weiteren Schritt wurde die Implementation des Compilers so geöffnet, dass er an gewissen Stellen des Source codes nur noch Macros einfügte. Die Macros selbst konnten AUSSERHALB des Compilers definiert werden. Dies entspricht der Auftrennung eines Compilers in Parser und Code Generator mit offengelegter Implementation.

Der Vorteil dieses Verfahrens liegt darin, dass der Preprocessor die Programmiersprache versteht und deshalb wenig auf extra Tags angewiesen ist. Ein derartiges Tool ist für ein grosses Framework Projekt unverzichtbar.

Wichtig ist, dass der Preprocessor auch über ein Callback API verfügt, d.h. den Aufbau eines Abstract Syntax

Trees über Events einem Client mitteilen kann, der diese Events dann mit eigenen Aktionen verknüpft.

Plattformunabhängigkeit

Plattformunabhängigkeit bedingt unter Umständen eine fallweise Unterscheidung von Implementationen als „common code" mit conditionaler Compilierung oder als separate Componente unter einem OS spezifischen Source Tree. Fehler in der Einteilung wirken sich sehr stark auf die Portierbarkeit aus.

Macros die zur Plattformunabhängigkeit beitragen gehören ebenfalls zur Source Code Struktur.

Ein Framework behandelt den eigenen Source Code nach den Regeln adaptiver Systeme (siehe Design Patterns for Evolving Systems, [SEITERS]. Adaptive Systeme zeichen sich durch 2 Prinzipien aus:

1. Vermeidung von Änderungen
2. Kontrollierte und möglichst automatische Transformation des alten Systems in eine neues wenn Anpassungen unvermeidlich sind. Dies schliesst Definition des Umfangs und der Konsequenzen von Anpassungen ein.

Viele Beispiele für generative Aspekte der Source Code Struktur finden sich in Fresco.

Generierungs Struktur

Der Build Process muss andere Strukturen mit Metainformationen versorgen können, z.B. über Versionen, Abhängigkeiten, Self-Identification und automatisch generierten Metaclassen die physische Element wie Packages bzw. DLLs reflektieren. Im Buildprocess sind wesentliche Informationen enthalten, die jedoch in den gängigen IDEs versteckt und damit anderen Strukturen kaum zugänglich sind.

Der Build Process definiert:

- Package Namen
- Class Evolution
- Versionen von Packages und Komponenten
- Abhängigkeiten (statisch und dynamisch) von anderen Packages
- Konfigurations Informationen, Ressourcen, 3rd party products
- Compile time Umfeld (z.B. Framework Release, Base Package Release, etc.)
- Installation und Verwaltung von Konfigurationsfiles gehören ebenfalls dazu.

Usage Struktur

Zwischen einem Komponenten Ansatz als kleinste Einheit der Flexibilität/Reuse wie er z.B. von Lacos vertreten wird und einem Framework Ansatz der Interfaces und Implementations von Beginn an trennt (und Broker verwendet) scheint zunächst ein Widerspruch zu bestehen. Er löst sich aber auf, wenn man die unterschiedlichen Formen des Gebrauchs eines Framewors betrachtet. In der Praxis wird oft zwischen

Black Box Reuse

und

White Box Reuse

unterschieden. Black Box Reuse bedeutet die Benutzung des Frameworks durch einen Client, der über die Interna des Frameworks nicht Bescheid weiss und stattdessen über das Black Box API die Funktionalität benutzt. Der Client erstellt keine Ableitungen von Framework Interfaces sondern nutzt die komplexe Funktionalität eher in der herkömmlichen Form einer Library oder eines Toolkits.

Das Komponenten Modell des Reuses scheint mir besonders für diese (in der Praxis WICHTIGERE)Form der Framework Nutzung geeignet zu sein. Hier bleibt der Client vor einem wesentlichen Teil der Komplexität des Frameworks geschützt. Das Konfiguration Framework eines Frameworks sollte auch auf dieser Ebene Flexibilität ermöglichen (siehe unten)

Die Programmierregel für ein Komponenten Modell der physischen Kapselung lautet: Eine Klasse A darf ein Objekt einer anderen Klasse B NUR DANN direkt instanziieren (d.h. der header file der verwendeten Klasse wird includiert) WENN DIE KLASSE B TEIL DESSELBEN physischen Packages wie die Klasse A ist. Zu beachten: Auch Ableitungen über Komponent oder Package Grenzen hinweg verletzen diese Regel.

Damit entsteht wie bereits oben geschildert zwar ein klarer cold spot (niemand kann ohne Source Code Änderung an dieser Verwendung von B durch A etwas ändern. Allerdings muss auch nicht alles in einem Framework flexibel sein.

Resultat:

Nicht jeder Mechanismus und jede Stelle eines Frameworks muss änderbar bzw. austauschbar sein. Eine Konsequenz der Übernahme des Framework Denkens ist in der ersten Zeit ein totales Streben nach völliger Allgemeingültigkeit auf Seiten der Team member. Dies ist im wirtschaftlich nicht vertretbar und verzögert konkrete Lieferungen immer weiter („man müsste hier und dort unbedingt noch flexibler werden ..."). Die Abgrenzung notwendiger Flexibilität gegenüber „nice to have" ist eine der wichtigsten Aufgaben im Management eines Framework Projektes.

Mit einem physischen Komponenten Modell allein lässt sich der eigentliche Zweck eines Frameworks, nämlich kooperierende Klassen zur Lösung verschiedener Probleme einzusetzen nicht erreichen. Mit Hilfe von Broker, Factories und Repositories wird eine Zusammenarbeit von Klassen erreicht OHNE dass sie physisch in zusammenhängen (und damit nach der obigen Regel in EINER Komponente sein müssten).

White Box Reuse bedeutet „Puzzle" Teile des Frameworks durch eigene zu ersetzen. Dies ist am einfachsten, wenn die Reuse Granularität des Frameworks bis auf binäre Classes hinunterreicht. Damit kann der Client eine Framework Class durch einen Eintrag im Implementation-Repository durch eine eigene ersetzen bzw. neue Implementationen hinzufügen.

Das bedeutet, dass sich Komponenten Modell und Framework/Broker Modell ideal ergänzen.

Soziale Strukturen

Zwei Warnungen vorab:

1. der Begriff „sozial" wird hier im wesentlichen als terminus technicus gebraucht und hat zumindest im ersten Ansatz NICHTS mit der umgangssprachlichen Bedeutung von „gut/nett sein, Hilfe, Gerechtigkeit etc." zu tun.

2. Im Gegensatz dazu verwende ich Begriffe der Sozialwissenschaft wie Rolle, Identität etc. in einer eher umgangssprachlichen Bedeutung.

Auf der technischen Seite eines Frameworks lässt sich deutlich ein Abrücken vom Modulgedanken als gliederndes Element erkennen. Strukturen und Aspekte wurden als übergreifende, quer durch Software Schichten laufende Verbindungen erkannt. Dies hat Konseuenzen auch für die soziale Organisation der Software Entwicklung (siehe Organizational and Social Frameworks, [WKR97/1]). Auch hier versagt das „Kästchendenken" als Form der Reduktion von Komplexität.

Die 3 wichtigsten sozialen Strukturen einer Framework Entwicklung sind

1. Rollenstruktur
2. Wissensstruktur
3. Organisationsstruktur

Rollenstruktur

Framework Programmierung hat die herkömmliche Rollenverteilung der Informatik in Applikations- und Systemprogrammierer aufgelöst. Wer Teile eines Framework durch das externe API nutzt, handelt als Applikationsprogrammierer. Wer das Framework im Sinne von White Box Reuse nutzt, tut dies als Systemprogrammierer mit allen damit verbundenen Konsequenzen bezüglich Competence Level, Zeitaufwand und Sicherheit.

Diese Rollen vereinen sich in der Praxis jedoch in einer Person. Applikationen und Framework entstehen parallel, denn ohne Feedback durch die Anwendung entsteht keine allgemeines Framework.

Die Rolle als Prinzip der Reduktion von Komplexität fällt weg. Stattdessen wird das parallele Erfüllen verschiedenster Rollen vom Entwickler verlangt. Die rollenmässige Auftrennung der Entwicklung in Analytiker, Designer und Programmierer funktioniert nicht, weil sie die Verbindungen der Strukturen hinter den Rollen versteckt.

Die NEWSYS Entwicklung hat gezeigt, dass nicht jeder Entwickler bereit ist, den „Hut" dynamisch zu wechseln. Der Sinn eines Frameworks flexible und erweiterbare „Templates" von Lösungen (natürlich auch vorgefertigte Bausteine) anzubieten stösst auf eine Erwartungshaltung die nach festen APIs verlangt.

Wissensstruktur

Ein weiteres grosses Problem der Frameworkentwicklung ist die Verteilung der Wissenstrukturen innerhalb des Framework Teams wie auch hin zu Anwendern (Entwicklern).

Auch das resultiert aus der Tatsache, dass ein Stück Source Code viele verschiedene Aspekte beinhalten kann und demzufolge viele Strukturen gleichzeitig betrifft, im schlimmsten Fall erfolgt ein „cross cutting" über das gesamte Framework (z.B. Error Handling, Multithreading, Exceptions)

Eine Aufteilung der Arbeiten und Wissensgebiete in einzelne Bereiche führt auf Grund der internen Zusammenhänge des Frameworks schnell zu Doppelarbeiten oder Implementierungen die die zentralen Konzepte verletzen.

Selbst in kleinen Entwicklerteams (4-5 Personen) ist die Wissensweitergabe bereits ein Problem. Regelmässige interne Workshops und eine Dokumentation auf Design Pattern Basis sind ein Muss.

Hin zu externen Anwendern muss eine deutliche Reduktion der Komplexität stattfinden, gestützt auf visuelle Tools. Dies haben auch Systor interne Forschungen zum Thema Frameworks ergeben [RKA,TGO96].

Verstehen, Finden und Manipulieren von Funktionalität MUSS über visuelle Tools und Repositories ermöglicht werden. Wer die interne Komplexität des Frameworks immer durchscheinen lässt, wird massive Akzeptanzprobleme bekommen.

Neben der Verteilung des technischen Wissens ist das nötige Wissen um die fachliche Domain des Frameworks unentbehrlich und MUSS auf praktischer Erfahrung beruhen.

Die Darstellung von Framework-Mechanismen in flachen Klassendiagrammen allein ist kein Mittel. Funktionsweise eines Frameworks liesse sich am besten innerhalb einer VRML World darstellen. Ganz entscheidend ist auch, dass die Framework Componenten reflektiv über sich selbst Auskunft geben können. (siehe auch Generierungsstruktur)

Ein möglicher Ausweg aus dem Dilemma, dass die Reduktion von Komplexität auf Modulebene nicht mehr machbar ist wird im Bereich der sog. Aspekt-oriented Programming diskutiert. Entwickler können dort jeden Aspekt bzw. jede Struktur EINZELN in einer anderen Sprache programmieren. Ein „aspect weaver" tool verknüpft dann die Aspekte zu einem Programm.

Organisationsstruktur

Politische Voraussetzungen

Dass Framework Entwicklungen dauern länger und teurer sind als herkömmliche Entwicklungen einzelner Applikationen hat sich mittlerweile herumgesprochen. Sie setzen Erfahrung und eine Menge Geld voraus.

Weniger bekannt ist, dass sich der Entwicklungsprozess stark vom herkömmlichen unterscheidet. Besonders in Zeiten der Rapid Development Believers und ihren GUI toolkits benötigt es MUT, eine Framework-Entwicklung zu starten. Die Entwickler des NEWSYS Frameworks z.B. waren konstant in einer Auseinandersetzung mit Kollegen die mit RAD tools viel schneller zu Ergebnissen kamen als mit dem sich erst entwickelnden Framework

Auch mehrere Fehlschläge mit RAD tools (typischen 2 Tier tools mit Datenbankzugriff oder GUI Buildern mit extra Editoren) liessen die Konflikte nicht verstummen. Das Ergebnis der RAD tools war immer schnell und nach 4 Monaten nicht mehr wartbar, weil sich die Entwickler z.B. mit einem Wald von GUI callbacks konfrontiert sahen (in die sie ihre Applikationslogik integriert hatten). Kleinste Änderungen brachten obskure Nebeneffekte.

Ein weiteres Grundproblem mit Frameworks ist die Tatsache, dass ein Framework nicht nur das Framework Team betrifft sondern plötzlich die Basis auch für andere Teams darstellt. Teams die keineswegs eine so enge emotionale Bindung zum Framework besitzen wie dessen Entwickler. Das unter Informatikern so gut bekannte „not invented here" Syndrom führt in der Praxis sehr schnell zur Ablehnung, vor allem wenn die Firma grösser ist und entsprechende Geldmittel zur Verfügung stehen.

Ohne starke politische Rückendeckung ist ein Frameworkprojekt nicht durchführbar. Umso mehr als die Entscheidung die Lösung der Probleme im Frameworking zu sehen NICHT ausschliesslich rational begründbar ist (Thomas Kuhn, Zur Struktur wissenschaftlicher Revolutionen).[KUHN]

Wenn RAD-Tool A nicht das gewünschte Ergebnis bringt könnte man sich statt für ein Framework Design genausogut für einen weiteren Versuch mit RAD tool B entscheiden. Der Umschwung zum Framework-Denken ergibt sich erst aus der systematischen Beschäftigung mit dem Lifecycle einer Software und ihren technischen Bedingungen.

Eine wichtige Komponente für die Entscheidung zum Framework liegt in der Erfassung realer Kostenstrukturen der herkömmlichen Software Entwicklung. Diese ist oft so unterentwickelt wie das Kontrolling von PC Kosten. Ein internes Kontrolling das auch Wartungskosten und Kosten späterer Zusatzentwicklungen erfasst – im Vergleich zu dem was möglich wäre in einem Framework Konzept – ist meist nur ein Wunschtraum.

Eine wichtige Frage innerhalb der Organisationsstruktur ist auch, ob das Framework nur intern Verwendung finden oder ob es als Paket auch verkauft werden soll. Mit der Perspektive eines öffentlichen Produktes erweitert sich der Horizont der Entwickler deutlich. Gleichzeitig ergeben sich grosse Marketing- und Vertriebsprobleme, da dort die Kompetenz für ein derartiges Produkt fehlt. Oft stehen die Entwickler eines Frameworks selbst auf Messen und beantworten auch die Fragen der Kunden in eigener Regie.

Der Entwicklungsprozess selbst

Der Entwicklungsprozess eines Frameworks läuft anders ab als in herkömmlichen Applikationen und hat Auswirkungen auf die Organisationsstruktur.

Im einzelnen sind die Unterschiede:

Applikationsprogrammierung und Frameworkprogrammierung verschwimmen. Dies wurde auch beim Projekt NEWSYS festgestellt. Organisatorisch blieb die Trennung in Applikations- und Grundlagen Entwicklung jedoch bestehen. Dies führte teilweise zu enormen Reibungsverlusten und letztlich zur Übernahme auch der Applikationsentwicklung durch die Framework Entwicklung.

Eine organisatorische Splittung in Framework und Applikationsentwicklung ist während der Entwicklung des Frameworks nicht durchführbar.

Der Entwicklungsprozess ist parallel und nicht sequenziell: Framework, Applikationen und Tools bilden eine Einheit und entstehen zusammen. Dies erfordert das Einplanen von Zeiten für Toolentwicklung und grosse Flexiblität auf Seiten der Entwickler.

Entwicklung neuer Projektmanagement-Strategien ist nötig

Fortschritte sind oft kaum sichtbar. Frameworking is ein typischer bottom up prozess der von der Umwelt manchmal als Stillstand oder unwesentlicher Fortschritt gesehen wird. Ein typisches Beispiel:

Das NEWSYS team macht eine happy hour. Voller Stolz zeigen die Entwickler zwei Textfelder in einem Fenster. Wird der Inhalt des einen Textfeldes geändert, passt sich der Inhalt des anderen Textfensters an. Für Aussenstehende ist die Begeisterung völlig unbegreiflich („so etwas macht mein GUI tool X in 2 Minuten!")

Was tatsächlich auf Seiten des Frameworks passiert ist, ist folgendes:

- Das Model-View-Controller Pattern (MVC) wurde implementiert
- Applikationslogik ist getrennt vom GUI über Interfaces
- GUI spezifische Applikationslogik wird dynamisch über das strategy pattern mit GUI-Komponenten verknüpft. Die Logik selber ist in SGML Konfigurationen validierbar enthalten und kann dynamisch geladen und geändert werden.
- Das proprietäre GUI ist gekapselt mit Objekten des HIC framework. Dadurch ist die Update-logik von MVC möglich. Das GUI braucht nicht mehr durch pollen die Synchronisation der Bildschirminhalte mit den Modellinhalten sicherstellen.
- Aller Komponenten sind Interfaces und werden nur über Broker und Factories erreicht.
- Einzelne Packages sind jederzeit austauschbar.
- GUI-Teile kennen lediglich high level composite object parts des Models und funktionieren generisch.

Die Framework-Entwicklung kann erst spät etwas zeigen. Bei allem was im Framework entwickelt wird müssen die geringeren Wartungskosten bzw. Kosten für Erweiterungen auch berücksichtigt werden.

Einbeziehen von Anwendern wird gerne eingesetzt um Projektziele zu fokussieren auf reale Bedürfnisse. Im NEWSYS Projekt waren die Erfahrungen damit eher negativ. Z.B. wurden die Service Gruppen aufgefordert, ihre Probleme mit dem gegenwärtigen OLDSYS einmal zu spezifizieren. Trotz mehrfacher Versuche kam dies nicht zustande. Die Untersuchung und Abstraktion der Wartungsprobleme blieb dem Frameworkteam überlassen.

Abstraktion – eine grundlegende Tätigkeit bei der Entwicklung eines Frameworks - ist ein mühsamer Prozess der nicht überall beliebt ist. Formale Ausbildung hat fast keinen Einfluss auf die Bereitschaft zur Auseinandersetzung mit grundsätzlichen Problemen der Software Entwicklung.

Einbeziehung von Kunden. Hier tritt das Problem auf, dass der Kunde seine Probleme sehr konkret und meist als Use Case für das ältere System bringt. Diese Use Cases müssen abstrahiert und als Problem der generischen Maschine des Frameworks übergeben werden.

Software Interfaces, die Teilung der Arbeit und einige hartnäckige Probleme der Software Entwicklung

Software Entwickler denken gerne in Modulen und Schnittstellen. Sie scheinen Komplexität zu verringern. Ausserdem ermöglichen sie aus Sicht der Organisation des Entwicklungsprozesses selbst die Aufteilung der Arbeit auf verschiedene Personen. Die Aufteilung erfolgt unter dem Gesichtspunkt der gegenseitigen Unabhängigkeit. Scheinbar ganz natürlich teilen wir auch die Arbeit entlang solcher Schnittstellen und zwar sowohl dem einzelnen Entwickler gegenüber (GUI Spezialist, Datenbank Spezialist etc.) als auch zwischen Gruppen (Applikation, System, Kernel etc.).

Kaum haben wir die Teilung der Arbeit vollzogen stehen wir jedoch einem neuen Phänomen gegenüber: Entwickler scheinen sich plötzlich nur noch für ihren Bereich zu interessieren. Die Zusammenarbeit der einzelnen und der Gruppen untereinander muss nun aufwendig organisiert werden. Die Software Interfaces sind plötzlich nicht mehr nur Aspekte, künstliche Trennungen eines komplexen Ganzen sondern sie sind sozial untermauert,

sind eherne Grenzen geworden.

Vielleicht erklärt das einige seltsame Verhaltensweisen in der Software Entwicklung. Allen voran das Klammern an Diagramme und Modelle als wären sie „das Ding an sich". Fortlaufende Verwechslung des Bezeichners mit der Realität dessen was bezeichnet wird.

Ein Beispiel:

Ich war kürzlich Gast auf einem Arbeitskreis für Framework Technologie. Dort wurde versucht ein „Layer" Schema für Frameworks zu entwickeln (Eigentlich mehr oder weniger ein sog. Multi-tier Modell). Ein älterer Kollege hatte den Einwand erhoben, dass in der Realität diese „Layers" teilweise nicht so scharf trennbar seien. Probleme bereite vor allem der Übergang von einem zentralen Modell auf ein dezentrales. Diese Aufteilung zwinge teilweise zu einer anderen Gliederung. Diese stellte natürlich das gesammte Konzept einer generellen layer Architectur für Frameworks in Frage. Was taugt so ein Schema wenn es nicht für verschiedene Systemarchitekturen passt? Entsprechend heftig wurde der Einwand des Kollegen zurückgewiesen (man könnte auch sagen die Realität wurde im Sinne der layer bzw. tiers zurechtgebogen). Als ich dem Kollegen zu Hilfe kommen wollte und bemerkte, dass die Layer bzw. Tiers des Modells ohnehin nur logische Krücken für das Denken seien und ihre jeweilige Realisierung in einem konkreten Produkt ganz anders aussehen könne, wurde mir mit der Frage entgegnet, ob ich schon mal Software Wartung gemacht hätte.

Eine nötige Funktionalität == Ein Interface == Ein Layer == ein Modul == eine Gruppe bzw. Tätigkeit

Ich denke es lohnt sich die Argumentation an dieser Stelle genauer zu beleuchten. Zu meiner grossen Überraschung gingen viele Beteiligte stillschweigend von einer direkten Abbildung logischer Gliederungen auf physische Gliederungen aus. Ebenso stillschweigend hatten sie eine konkrete Systemarchitektur (in diesem Fall lokale Smalltalk Applikationen) angenommen) und netter weise passte das layer Modell sehr gut auf diese spezielle physische Systemarchitektur.

Ich möchte an dieser Stelle auf die Problematik von Modellen kurz eingehen:

[SIEMENS96] zeigt sehr schön dass das Layer Modell häufig zu unschönen Workarounds zwingt, d.h. die Architektur läuft Gefahr im Laufe der Zeit zu degenerieren. Häufig werden Interface Erweiterungen unterer Layer vorgenommen die im nachhinein zu Problemen führend. Man denke an den Missbrauch des ioctl Systemcalls auf Unix Systemen um z.B. bestimmter X Server Funktionalität performanter anbieten zu können oder im allgemeinen Applikationen einen Zugriff auf Systemressourcen zu gestatten ohne die Kernel/User Layers zu verletzen. [BIRMAN96]) zeigt die teilweise religiöse Verklärung der ISO/OSI Modells und seine mangelnde Brauchbarkeit für distributed objects.

Und zur Zeit dominiert das sog. 3-tier Modell die Zeitschriften. Pikanterweise entsteht mit intelligent Agents gerade jetzt eine Technologie die zu diesem Schema überhaupt nicht passt: Ein Agent kann sich je nach Situation einmal mit einem GUI melden, das andere Mal eine Komponente innerhalb eines Services darstellen und dabei kein eigenes GUI offenbaren. Der Agent hat offensichtlich alle „tiers" in sich vereint. [JAVABEANS97/2].

Der eigentliche Fehler liegt darin, die Modelle als gleichzeitig logisches wie auch physisches Design aufzufassen. Besser ist es von einer Projektion des logischen Modells auf jeweils ganz verschiedene physische Strukturen auszugehen. Eine Lösung kann z.B. entweder ganz auf einem Client, als Client/Server oder als alleinige Server Lösung implementiert werden. Das logische Modell spezifiziert globale Funktionalität, nicht physische Implementation.

Hardware Designer unterscheiden hier sehr viel genauer in die logische Funktionalität die in Blockschaltbildern gezeigt wird und dem konkreten Aufbau der Funktionalität in Form von diskreten Boards, ASICs etc.[I-LOGIX97]

Die Verteidigung des Layer Modells hat inhaltliche Schwächen aber das reicht nicht aus um die Vehemenz der Verteidiger zu erklären. Ein weiterer Grund wird aus der Bemerkung des Kollegen über die Software Wartung deutlich: Nicht nur geht das Modell von einer impliziten Systemarchitektur aus, es setzt auch eine bestimmte Form von Arbeitsteilung beim Umgang mit der Software voraus: Verantwortlichkeit ist per Modul oder Layer geregelt. Man kommt sich nicht ins Gehege. Man kann separate voneinander werkeln.

In welcher Beziehung stehen aber die ältesten Probleme der Software Entwicklung zu so einem Ansatz der Trennung und Isolierung? Wirklich unangenehm sind Fragen der Reliability, Quality, Security, Performance (Threading), Maintenance, Extendability. Wieso sind gerade diese Probleme so unangenehm? Ganz einfach –

sie lassen sich nicht „Layern", entziehen sich eines einfachen „separation of concerns" Ansatzes. Es gibt keinen Performance Layer oder Security Layer. Es gibt kein „quality module". Diese Dinge durchziehen die gesammte Architektur, egal wie sie in Module oder Componenten aufgeteilt sind. Sie sind eigentlich in der Verantwortung aller und damit in der arbeitsteiligen Umwelt in niemandes Verantwortung.

Als Folge ihres netzartigen Charakters erfordern sie intensive Kommunikation. Und nicht nur Kommunikation sondern auch Abstimmung, Teilen und als Folge davon auch Vertrauen. Keine leichten Forderungen in einer hochgradig arbeitsteiligen Welt.

Wieso stelle ich diese Fragen gerade in einem Kapitel mit dem Thema „Organisationsstrukturen"?

Weil die Probleme bei der Entwicklung des NEWSYS Frameworks genau daraus resultierten. Wir – d.h. „die Neuen" haben auf zwischenmenschlicher Ebene so ziemlich alles versucht um die Kollegen ins Boot zu ziehen – erfolglos. Wir haben um Mitarbeit gebettelt, nie ein schlechtes Wort über das bestehende System gesagt und alle Informationen offengelegt, relgelmässig Workshops abgehalten – alles ohne jeden Erfolg. Ich glaube deshalb dass es Massnahmen auf Organisationsebene braucht um neue Technologien einzuführen.

Die Verbindung von Softwarestrukturen und sozialen Strukturen an Beispielen:

- Software Qualität

Kaum eine Softwarefirma die nicht unter mangelnder Qualität eigener und fremder Softwareprodukte leidet. Viele installieren Gruppen bzw. ganze Abteilungen zur Qualitätssicherung (QS) und Kontrolle mit dem Ziel, die schlimmsten Fehler möglichst früh abzufangen. Das Verhältnis zwischen Entwicklung und QS ist geprägt von gegenseitiger Verachtung und Vorurteilen.

Mittlerweile ist auch klar, dass das eigentliche Ziel, nämlich Qualität möglichst früh, d.h. dort wo die Software entsteht sicherzustellen, mit dieser Trennung in Entwickler und QS nicht erreicht werden kann. Nocht schlimmer ist diese Trennung im Falle von neuer Technologie wie z.B. OO-Frameworks weil hier herkömmliche QS-Mitarbeiter meist die Technologie überhaupt nicht mehr verstehen. Qualität als im Produktionsprozess nachgeordnete Grösse funktioniert nicht und schon gar nicht wenn die Aufgabe „Qualität" sozial aufgeteilt ist in verschiedene Gruppen.

Was kann man tun?

Manche Firmen versuchen die Kluft zwischen Entwicklung und QS durch gleichzeitige Mitgliedschaft von Personen in beiden Gruppen zu überbrücken, d.h. jemanden zu 50% in der Entwicklung und zu 50% in QS arbeiten zu lassen.

Was sind die impliziten Annahmen dahinter?

Man will letztlich das Qalitätsgedanken dadurch Einfluss in die tägliche Arbeit finden, in dem verschiedene Denkbilder verschmolzen werden. Bezeichnenderweise will man dies durch Verschmelzung der sozialen Rollen erreichen.

Noch effektiver wäre es wenn man das Denkbild der Entwicklerrolle selbst um Qualitätsgedanken anreichern könnte. Techniken zur Qualitätssteigerung sind heute bekannt. Wer z.B. Memory Management Fehler erzeugt indem er keine Smart Pointer bzw. Garbage Collection einsetzt begeht einen echten Fehler im Engineering, keinen simplen „Bug". Techniken zur Qualität und Reliability müssen in Coding Standards festgeschrieben sein, dann entsteht Qualität bereits von Beginn an. (s.u. Coding Standards)

Ein konkretes Beispiel aus der NEWSYS Entwicklung:

Die Verwendung von Smart Pointern war ein ständiger Streitpunkt gegenüber den Entwicklern des OLDSYS Systems. Weder Workshops noch Literatur waren in der Lage hier Verständnis zu schaffen. Die grundlegende Einstellung wurde einmal in der Aussage eines Gruppenleiters deutlich: „Memory Management ist eine Frage der Disziplin und des Könnens, nicht von irgendwelchen Smart Pointern". (Pikanterweise entdeckten wir natürlich auch in seinem Code memory leaks ..)

Wie gesagt, Frameworking bezieht sich auf den gesammten Lebenslauf eines Produktes und weiter untern werden zahlreiche Techniken gezeigt um die Qualität eines Softwareproduktes bereits in der Entstehung zu sichern.

Wenn die Strukturen einer Softwareentwicklung bekannt sind, gibt es keine Entschuldigung mehr wenn sie nicht berücksichtigt werden.

- Weiterbildung

Woher kommen denn die vielen starren Denkbilder und das starre, einseitige Verhalten von vielen Softwareentwicklern?

Wir haben bereits gezeigt, dass die starren Denkbilder durch die soziale Trennung in Form der Arbeitsteilung zwischen Gruppen gefördert werden. Die Trennung geht jedoch innerhalb von Gruppen noch weiter. Wir haben den GUI-Spezialisten, den Datenbank Guru oder die Datenmodelliererin. Oder noch schlimmer: der Novell-Spezialist, der Oracle Guru usw. Wodurch entstehen diese Spezialisten?

Kürzlich bin ich um Mithilfe bei einem Ausbildungsplan für neue Mitarbeiter im Bereich Java gefragt worden. Ziel war es den idealen Einstieg für Universitätsabgänger im Bereich Internettechnologie und Java zu definieren. Die Fragestellung war, ob der Einsteiger besser mit den I/O Klassen oder mit Applet Programmierung anfangen sollte oder ob Networking gleich eine grosse Rolle spielen sollte.

Ich habe mir erlaubt den Ausbildungsplan um zwei wesentliche Punkte zu erweitern.

1. Parallel zum Erwerb technischer Kompetenz in der Programmiersprache Jave müssen Kenntnisse in Design Patterns und in sprachunabhängiger Objekttechnologie und Theorie erworben werden. Wenn diese Parallelität nicht gewahrt ist erhalten wir einen Mitarbeiter, der OO mit Java gleichsetzt – einen „Java-Mann" dessen Grenzen seiner Vorstellung gleich mit den Grenzen von Java sind.

2. Wichtiger als die Art und Weise des Einstiegs ist, dass jemand auf die Komplettheit der Ausbildung achtet über mindestens 2 Jahre. Anschliessend muss dafür gesorgt werden, dass keine ausschliessliche Spezialisierung auftritt. Dies kann durch laufenden Wechsel der Arbeitsgebiete geschehen. Wenn dies unterbleibt tritt uns der Mitarbeiter nach 4-5 Jahren in voller Borniertheit als Fachspezialist gegenüber und wir stehen wieder vor den damit verbundenen Kommunikationsproblemen. (Angesichts des Tour de France Siegs von Jan Ullrich wurde über ihn im Fernsehen gesagt: „Was für ein kompletter Sportler". Gemeint war, dass er sowohl in der Ebene als auch in den Bergen oder beim Zeitfahren annähernd gleich fähig ist. Wieso gilt das nur für Sportler?

- Phasen des Entwicklungsprozesses: Analyse, Design und Implementation

Im folgenden soll kurz die These von den sozialen Strukturen als Ursache des Scheiterns vieler Softwareentwicklungen vorgestellt werden.

Wieso gilt heute das Wasserfallmodell für die Software Entwicklung als gescheitert? Es scheint in seiner Trennung von Analyse, Design und Implementation doch die wesentlichen Schritte des Entwicklungsprozesses zu beinhalten. Es scheitert meiner Ansicht auch nicht technisch – d.h. es ist ihm technisch nichts immanent, was zu einem Scheitern der Entwicklung führen müsste – es scheitert vielmehr weil es eine technische Strukturen mit sozialen Strukturen in sehr problematischer Weise verknüpft.

Begründung:

Die Basis für den Cocktail sozialtechischer Probleme wird mit der Arbeitsteilung geschaffen, fein unterteilt nach Analytikern, Designern und Implementatoren (gemeinhin „Codierschweine" genannt. Dass dies keine nur horizontale Arbeitsteilung ist zeigt sich darin, dass die Gehalts- und Prestigeverteilung rein vertikal ist: Analytiker bekommen am meisten Gehalt und Prestige. Bekommen Designer noch Business Cards?

Den einzelnen Arbeiten wird der Inhalt der auszuführenden Rolle genau festgeschrieben. Es entstehen Identitäten die sich genau mit diesen Rollen identifizieren. An der Spitze ein Gefühl der Überlegenheit, verbunden mit einer Abscheu vor konkreten, „Design oder Implementationspezifischen" Dingen, den Niederungen des Alltags. Am Ende das Gefühl derjenige zu sein, der den ganzen Unsinn der Analytiker und Designer letztlich ausbaden muss bzw. durch eigenen Vorstellungen so hinbiegen muss, dass das Produkt letztlich fertig wird. (Es

sind Fälle bekannt wo Implementationsgruppen am Ende ihr eigenes Design entwarfen, da das offizielle angeblich nicht zu gebrauchen war).

Für jede der drei Rollen werden eigene Wertsysteme, Rituale und Rechtfertigungen entwickelt. Irgenwie muss ja der soziale Unterschied gerechtfertigt werden. Für die ganz unten verstehen „die ganz oben" einfach nicht, dass die eigentliche Arbeit ganz unten geleitstet wird und umgekehrt.

Und letztlich unterliegt dem Ganzen noch eine zeitliche Struktur der Form, dass Analyse vor Design und Design vor Implementation zu geschehen hat. Daraus ergibt sich die Notwendigkeit kompletten Wissens für das Wasserfallmodell. Wenn dieses komplette Wissen nicht auf allen Ebenen gegeben ist, dann sorgen die sozialen Schranken dafür, dass jede Korrektur den Anstrich von Fehler, Versagen hat. Rückmeldungen von unten nach oben sind schwierig und immer gleich ein Politikum. Wissen über den Zusammenhang von Analyse, Design und Implementation entsteht nirgends.

Häufig werden die Arbeitsschritte auch noch an verschiedene Firmen vergeben. Damit wird eine klare Strukturierung der Verantwortlichkeiten angestrebt, im Regelfall entsteht jedoch an den Übergängen enorme politische Spannung. Wasserfallmodelle enden meist in Form des Fingerzeigens. Die wichtigste Tätigkeit in Wasserfallmodellen besteht darin alle Gespräche, Vereinbarungen etc. genau zu dokumentieren um bei Bedarf Munition für die politische Auseinandersetzung zu haben.

Obwohl das Modell heute nicht mehr so oft angewandt wird spielen sich ähnliche Szenen doch immer wieder zwischen Gruppen ab, die an einer gemeinsamen Entwicklung beteiligt sind. Opfer sind meist genau die technischen Probleme, die eine übergreifende Sichtweise gebraucht hätten. Nur ist die Strukturierung hier eine horizontale statt wie im Wasserfallmodell eine vertikale.

Wie kann vermieden werden, dass die vertikale Struktur des vorgehens nach Analyse,Design und Implementation in eine ebenso vertikale soziale Struktur einmündet? Die Regel lautet dazu ganz einfach, dass wer eine Analyse erstellt auch am Design und in der Implementation mitarbeitet. In der Praxis sind dies erfahrene Softwarearchitekten an deren Seite jüngere Kollegen und Kolleginnen an allen Phasen mitarbeiten. Die praktische Erfahrung aus der Implementation sorgt wiederum für den notwendigen Feedback den auch der Architekt braucht.

Schwieriger als die vertikale Strukturierung ist die horizontale zwischen verschiedenen Gruppen die eben in das berüchtigte „Fingerzeigsyndrom" mündet. Hierzu gibt es mehrere Möglichkeiten: Wenn z.B. eine Gruppe ein Basismodul erstellt, das von einer zweiten Gruppe weiterverwendet wird, dann kann man Mitarbeitern der ersten Gruppe die Aufgabe geben, innerhalb der zweiten (Client)-Gruppe den Entwurf und die Implementierung der Interfacelogik zu übernehmen. Analog dazu sollen Mitarbeiter der ClientGruppe am Basismodul mitarbeiten.

Eine weitere Möglichkeit besteht darin, das Problem anders zu strukturieren. Statt in 2 grosse horizontal geschnittene Schichten kann man es vielleicht in mehrere dünnere vertikale „Stäbe" unterteilen und die Mitarbeiter darauf verteilen.

Ziel ist es der jeweiligen Gliederung des Problembereichs eine zu ihr orthogonale oder sogar netzartig verteilte soziale Struktur im Entwicklungsprozess zu überlagern. Während wir die Software Strukturen häufig so anlegen, dass die Berührungsflächen (Interfaces) zwischen Teilen (Modulen) möglichst klein sind um Wartung und Austausch zu verbessern sollten wir die sozialen Strukturen (Aufgaben) so wählen dass möglichst viele Berührungspunkte untereinander entstehen.

Wie weit kann das gesagte verallgemeinert werden? Analoge Verhältnisse wie im Wasserfallmodell herrschen z.B. an den Schnittstellen zwischen System- und Applikationsprogrammierern, zwischen Applikation und Netzwerk. Und ich denke sie existieren in vielen anderen Softwaregliederungen ebenfalls. Vielleicht geraten die Softwareentwickler doch einmal in das Blickfeld der Ethnologen und Anthropologen – lohnen würde es sich. Der zu erwartende Einwand lautet natürlich: „Aber die verstehen doch gar nichts von Software!". Dem kann man entgegnen, dass man für dieses Argument erst einmal nachweisen müsste, dass die Denkbilder und Strukturen der Softwareentwicklung keinen religiösen oder andersartig ritualisierten Charakter besitzen. Keine leichte Aufgabe. Ein Beispiel wie die Beachtung sozialer Strukturen das Software Design beeinflusst findet sich in [SIMONSEN/KENSING97]. Sehr interessante gemeinsame Projekte von Informatikern und Ethnologen finden sich auch im Umfeld der collaborativen Systeme von Xerox PARC sowie Xerox Research in Grenoble und Cambridge. [http://www.parc.xerox.com , http://ww.xrc.xerox????]

Noch ein Schmankerl zu den Denkbildern der Software Entwickler:

Angeregt durch den Siegeszug der Design Pattern die ja bekanntlich ihren Ursprung in den Werken des Ar-

chitekten Christopher Alexander haben, machte jemand den Vorschlag, doch einmal einen Architekturstudenten einzuladen und ihn über einige Wochen einen Blick auf die Verfahren der Softwareentwicklung werfen zu lassen. Dem Vorschlag wurde sofort entgegnet, (gleichlautend wie oben) dass dieser ja gar nichts von Software verstehe und dadurch nur eine Last sein würde, weil man ihm dauern alles erklären müsste. Nachdem sichergestellt war, dass seine Tätigkeit niemanden behindern würde und man ohnehin nichts wesentliches von ihm erwarten könnte kam als nächste Frage: „Könnte es nicht auch eine Frau sein?"(1997)

- Austauschen von Personen und Plätzen

Die Einführung neuer Technologie geschieht in der Softwareentwicklung gerne nach einem bestimmten Muster: Zuerst werden neue Mitarbeiter gesucht, möglichst solche die bereits ein wenig Erfahrung mit den neuen Themen haben. Anschliessend stellt man fest, dass das alte Environment nicht die richtigen Voraussetzungen bietet und man errichtet eine neue Organisation, oft ausserhalb der alten Firma. Nach zwei Dingen muss man hier fragen:

1. Wieso glaubt man, dass von den bisherigen Mitarbeitern keiner in der Lage ist, die Anforderungen zu erfüllen?
2. Wiese glaubt man, dass die bisherige Infrastruktur nicht für die neue Technologie geeignet ist?

Wer nachbohrt erfährt dass das wichtigste Kriterium eines neues Mitarbeiters das Teilen der gemeinsamen Vision von der neuen Technologie ist. Offensichtlich traut man gerade hier den bereits vorhandenen Mitarbeitern nicht. Als Grund für den Standortwechsel wird oft angegeben, dass man mehr Flexibilität braucht (Öffnungszeiten, Materialbeschaffung, IT-Support). Im Grunde fängt man am neuen Standort an das aufzubauen, was am alten schon existiert aber anscheinend nicht genügt. Gerade negative Erfahrungen mit dem bestehenden IT-Support führen in aller Regel schnell zur Forderung: Wir brauchen ein eigenes IT, Wir brauchen ein eigenes ....

Natürlich bauen wir jetzt die gleiche Struktur die wir in ein paar Jahren wieder als zu unflexibel ansehen werden. Damit ist nicht gemeint, dass Umstrukturierungen und neue Organisationsformen sinnlos sind. Bedenklich ist nur die Tatsache, dass sie scheinbar notwendig sind um aus den festgefahrenen sozialen und mentalen Strukturen wieder herauszukommen. Dies bedeutet aber auch, dass wir von der „lernenden Organisation" noch weit entfernt sind.

Bleiben noch zwei Fragen zu klären:

1. Was tun wir mit „Altgedienten" Mitarbeitern bei der Einführung neuer Technologie wie z.B. im Falle des Frameworking und
2. wie selektieren wir neue Mitarbeiter.

Die Kriterien für diese Entscheidungen folgen aus den technischen und sozialen Anforderungen eines Framework Projektes als da sind:

- Hohe Kommunikationsfähigkeit da die Granularität des Systems weit unterhalb von Modulen und Paketen liegt und damit permanente Abstimmungen nötig sind.
- Ein Gefühl für Zusammenhänge und übergeordnete Verantwortung da der ganze Lebenszyklus einer Software in den Blickpunkt tritt.
- Erfahrung im System- wie auch im Applikationsbereich, da die Grenzen zwischen beiden verschwimmen. Flexibilität statt Rückzug auf enge Aufgabengebiete.
- Abstraktionsfähigkeit gepaart mit praktischem Handeln, Bereitschaft zu Analyse, Design und Implementation.
- Ständige selbstständige Weiterbildung, Kenntniss neuester Technologie

- Ausgewogenes Profil der Mitarbeiter bezüglich Ausbildung und Geschlecht. Vermeidung von inzuchtartigen Zusammenballungen bestimmter Ausbildungen bzw. von nur männlichen Gruppen.

Dies hört sich zunächst relativ schwammig an, doch jeder erfahrene Projektleiter hat sich mit der Zeit einen Katalog von Fragen und Beobachtungen erarbeitet die es ihm erlauben, einen Kandidaten relativ schnell auf diese Punkte zu prüfen bzw. existierende Gruppen zu untersuchen.

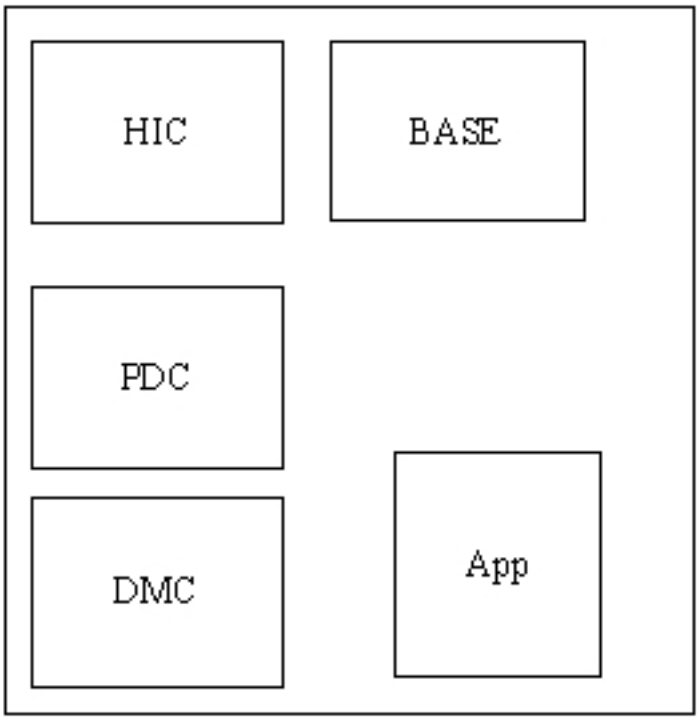
Neue Organisationsformen der Softwarearbeit: Virtuelle Organisation, Consulting

Im folgenden wird auf einzelne technische Strukturen des NEWSYS Frameworks vertieft eingegangen.

Chapter 4. DIE LOGISCHE STRUKTUR DES NEWSYS FRAMEWORKS

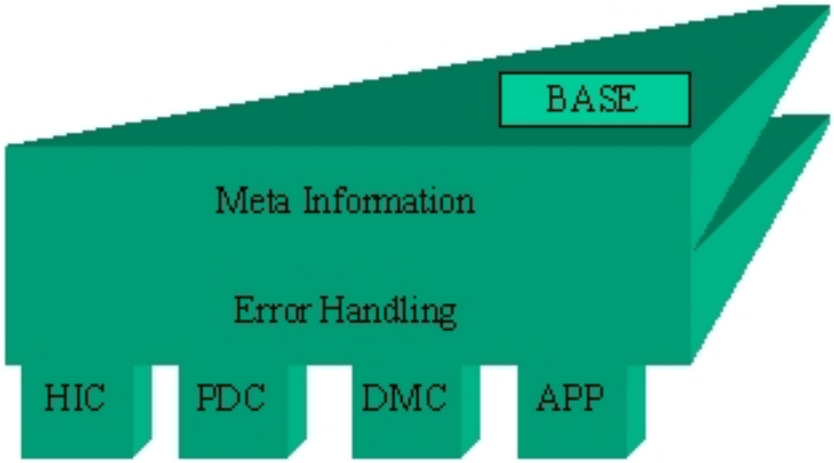
Domains

Die gröbste Gliederung von NEWSYS besteht aus 5 Domains die zusammen das Framework auf logischer Ebene repräsentieren und jeweils unterschiedliche Services zur Verfügung stellen.



Diese Domains sind lediglich **logische** Gliederungen der Apekte die sich einem bestimmten Gebiet zuordnen lassen. Damit ist keinesfalls festgelegt durch welche Module oder Packages diese Services tatsächlich zur Verfügung gestellt werden. Ebenfalls unterstellt dieses Schema **kein Layering**. D.h. die Verbindungen zwischen diesen Domains erlauben 2- oder 3-tier Architekturen. Zu den Problemen des Layer Architectural Design Patterns siehe [SIEMENS].

Orthogonal zu dieser Gliederung verlaufen Domainübergreifende Strukturen wie Fehlerbehandlung und Exceptions, Internationalisierung, User Kontrolle etc. die teilweise wie im Falle der Fehlerbehandlung und Internationalisierung untereinander wiederum verwoben sind. (Fehler müssen in der gewählten locale angezeigt werden). Gedanklich sind diese Funktionalitäten vor allem in der BASE Domain angesiedelt, sie können aber auch anderen Domains zugeordnet werden. User Kontrolle kann z.B. durchaus eine Funktionalität des Models (PDC) sein.



Aufgaben, Interfaces und Protokolle jeder Domain

An dieser Stelle wird ganz bewusst zwischen den Begriffen „Interface" und „Protokoll" unterschieden. Beide werden häufig auf Funktionen angewendet, die einem Client zur Verfügung gestellt werden.

Protokolle sind der prozedurale Teil der Benutzung von Interfaces- sowohl interner Interfaces wie z.B zu einer Base-Class als auch externe (public) Interfaces. Sie können leider in den meisten Programmiersprachen noch nicht programmatisch definiert werden, d.h. das vielbesprochene „behavior" von Klassen oder Clustern von Klassen ist immer nur implizit im Source Code enthalten. Beide Begriffe sollten streng getrennt werden, damit dieser Mangel nicht verdeckt wird. Das Einhalten von Protokollen durch Anwender ist eines der grössten Probleme bei der Erweiterung des Frameworks (siehe dazu die Visualisierung eines Template/Hook Protokolls durch Tags im Kapitel zu Coding Standards)

Die Konsequenzen dieser Trennung von Interface und Protokoll an einem Beispiel:

Eine Interface Klasse V wird folgendermassen spezifiziert:

Class V {

Boolean init(args);

Boolean doIt(args);

Void reset();

}

Eine Implementation N1 implementiert dieses Interface wie folgt:

Class N : public V {

Boolean init(args); // soll nur einmal aufgerufen werden

Boolean doIt(args); // kann nur einmal aufgerufen werden

reset(): // tut nichts

}

Eine weitere Implementation N2:

Class N2 : public V {

Boolean init(args); // soll nur einmal aufgerufen werden

Boolean doIt(args); // kann mehrmals aufgerufen werden, vorausgesetzt

// reset() wird dazwischen aufgerufen

reset(): // tut nichts

}

Eine weitere Implementation N3:

Class N3 : public V {

Boolean init(args); // kann mehrfach aufgerufen werden, reset() dazwischen

// nötig.

Boolean doIt(args); // kann mehrmals aufgerufen werden, vorausgesetzt

// init() wird dazwischen aufgerufen

reset(): // wenn gerufen, lässt die init argumente gültig für den

// nächsten doIt() Aufruf

}

Weitere Kombinationen sind denkbar, mit jeweils anderer Semantik und anderen Verpflichtungen für den Client.

ALLE Kombinationen sind jedoch mit dem in V spezifizierten Interface bezüglich der physischen Struktur verträglich. Worin sich die Implementationen unterschieden ist das PROTOKOLL. Da die Klasse V keine Protokoll spezifiziert (momentan ohnehin nur in Form von Kommentaren in der Interface Definition möglich) entstehen inkompatible Implementationen.

Zu diesem fundamentalen Problem hat die OMG ein RFP veröffentlicht und die Firma ICON hat dazu einen Vorschlag eingereicht [CATALYSIS].

Nach den Regeln der automatischen Generierung von Dokumentation hat jede Klasse im Deklarationsteil eine ausführliche Beschreibung ihrer Funktion im Framework zu liefern.

NEWSYS REGEL: Jede InterfaceKlasse ist verpflichtet, ihr dahinterliegendes Protokoll im Header File zu spezifizieren und über die automatische Dokumentation zugänglich zu machen, solange keine programmatischen Möglichkeiten bezüglich Protokollen bestehen.

Der NEWSYS Coding Standard enthält darüber hinaus eine Regel bezüglich minimalen Interfaces zur Vermeidung von semantischen Zweideutigkeiten in Protokollen. (s.u. Programmierregeln)

Achtung: Nicht nur Methoden von Interface-Klassen gehören zum Interface, auch normale Header Files in denen z.B. Konstanten für Parameter eines Protokolls gesetzt werden, gehören logisch zum Interface obwohl sie oft in einem separaten Headerfile Sy.D.... .h sind. Damit stellen sie ein großes Problem für die Kapselung dar: Wenn z.B. eine Implementation eine neue Konstante benötigt, muß dann das Interface geändert werden? Hoffentlich nicht, besser ist es, dann auch eine neue Implementation des clients zu erstellen und beide Implementa-

tionen teilen ein privates Header File mit gemeinsamen Definitionen. Die .h-Files bei den Interfaces werden weiter unten nicht extra besprochen.

Die konkreten Packages mit Implementationen unterliegen der weiter unten beschriebenen Source Code und Extension Struktur.

Exkurs: Verschiedene Möglichkeiten der Überwachung von Protokollen

Ausgangspunkt ist die Beobachtung, dass die meisten Klassen bezüglich ihres Protokolls, d.h. ihre „richtigen" Verwendung eher auf gut Glück hin implementiert werden als mit systematischer Fehlerbehandlung. Dies ist verständlich, steigt doch mit jeder Methode der Aufwand, den eigenen State zu sichern, d.h. Sicherheit auch dann zu garantieren, wenn die Methoden in der falschen Reihenfolge aufgerufen werden.

Diesem Problem kann man auf 2 Wegen begegnen. Der konventionelle Weg besteht darin, durch mehr Disziplin bei der Implementation der Klassen die Sicherung des States zu erreichen. Davon abgesehen dass das sehr mühsam und schwierig ist, belastet es auch die Performance des Gesamtsystems enorm. Dieser Ansatz sieht eine Klasse grundsätzlich isoliert, in einem unbekannten und potentiell feindlichen Umfeld tätig.

Ein anderer Ansatz (aus der Hardware Entwicklung) läge darin, auf interne Prüfungen weitgehend zu verzichten. Stattdessen publiziert der Klassenautor das zulässige Protokoll und vertraut darauf, dass die Clients es einhalten. Dies wiederum setzt voraus, dass es Tools gibt, die die Einhaltung des Protokolls garantieren können, sei es zur Compilezeit oder teilweise erst zur Laufzeit.

Das Validierungstool müsste zumindest auf Komponentenebene vollständige Informationen zur Verfügung haben, d.h. wir bewegen uns weg vom einzelnen File als Basis der Compilierung. Das Tool müsste weiterhin die zeitlichen Abläufe verstehen können. Dies bedeutet einen Schritt weg von der Compilierung und hin zur Simulation. Der Nutzen eines solchen Verfahrens wäre enorm: Implementationen von Klassen werden einfacher und performanter.

Ein Nebenergebnis dieser Überlegungen ist, dass das File als bisherige minimale Entwicklungseinheit gleichzeitig zu grob (inkrementelles compilieren auf Methodenebene) als auch zu klein (Sichere Verwendung durch Clients) ist.

BASE

Aufgaben der Base Domain

Die Base Domain soll wie der Name sagt Basisdienste allgemeiner Art für andere Domains zur Verfügung stellen SOWIE bestimmte Interfaces und Protokolle innerhalb des Frameworks festlegen.

Klassen der BASE sind Basisklassen für die logische und physische Struktur des Frameworks.

Insbesondere fallen darunter:

- Typdefinitionen (Basis typen, Typedefs für fremde Klassen, Typedefs von Basisklassen als zukünftige eigenständige Klassen)
- Macros zur Klassendefinition und Erweiterung, generative Mechanismen
- Konfigurationsdienste
- Exceptions (System und User Types)
- Actions, ActionCallbacks und Templates
- Observer/Observed Mechanismen
- Reference Counting
- Listen

- Iteratoren, Visitoren, Selektoren
- Smart Pointer
- Root Klasse als erste Ableitung von CORBAObject.hpp
- Request-Mechanismus und Template
- Threads, Mutex, Semaphore, Condition Variables, Thread Factory
- Application Error und Error Handler Klassen
- Dokumentenmodell: Definition der Parts und Basisparts.
- SGML Interfaces, SgmlApplication und Element Klassen, ParserEventGenerator Interface
- Composite Message Interface und Mechanismen
- Value Klasse mit Conversion methoden
- Interaktor und InteraktorStrategy Interface und Mechanismus als Kapselung von externen Aktoren
- Systemspezifische Mechanismen (File handling, queues, SysUtilFactory, Process management)
- Object Request Broker Dienst
- Package Interface Definition
- Class Loader
- Package private templates for lists and other primitive container structures
- BASE service factory
- Mechanismen der Plattformunabhängigkeit
- PartBuilder Interface
- PartTraversal Interface

DMC

Aufgaben der DMC Domain

Die DMC hat die Aufgabe, Zugriffe auf persistente Dinge durch ein einheitliches Interface zu kapseln.

Darunter fallen insbesondere OLDSYS Datenstrukturen (Beleg, Image, Isam) aber auch zukünftige Datenbankschnittstellen. Dadurch wurden Pflegemodule möglich, die unabhängig von der physischen Form eines Dokumentes (Auch Datenbankinhalte sind Dokumente) ein Dokument darstellen und modifizieren können. Hier wird deutlich, daß die NEWSYS Dokumentenmetapher sinnvoll auch auf Datenbanken angewendet werden kann. HIC Views können über ein einheitliches Datenmodell auf beliebige Daten zugreifen.

Auch die DMC verwendet das composite object prinzip der Parts, da nicht alle Datenstrukturen auf relationale abgebildet werden können und deshalb eine Graphenstrukur nötig ist.

- Type Definitions
- StorageManager und StorageElemente
- StorageObject Factory

- Translatoren
- PNKeyGeneratoren
- Requests
- Image Handling (CORBA) for remote OCR
- OLDSYS wrapper interfaces (Kompatibilität)
- Data management factory

PDC

Aufgaben der PDC Domain

Zur PDC gehört, was den eigentlichen Problemlösungsaspekt von Applikationen ausmacht (In diesem Sinne sind BASE, DMC und HIC nur Helfer der PDC).

- Type Definitions
- SGML Framework (ParserEventGeneratoren, Parser und Entity Manager, SGMLfactory)
- Dokumenttypen (Stapel, Belege, Felder)
- Workflowtypen (Feldgruppen, übergeordnete Prüfobjekte)
- Prüfobjekte
- Hilfsklassen und Routinen
- Security und Encryption
- PDC main factory
- Sub factories für parts und andere model objekte
- Factory Interface zu Workflow Parts
- OCR Interfaces and implementations, multithreaded and distributed (CORBA)

HIC

Aufgaben der HIC Domain

Die HIC regelt die Kommunikation mit dem User nach dem Modell-View (Controller) Prinzip.

View Information muß dynamisch dazu verwendet werden, einen Bildschirm zu dem jeweiligen Dokument aufzubauen. Dazu gehören z.B. die gewünschten Felder sowie die Aktionen die den Menues hinterlegt sind.

Die Architektur der HIC geht dabei von folgenden Voraussetzungen aus:

1. Der Human Interface Component kommt in einem Framework KEINE BESONDERE Stellung zu.
2. Nicht der User der frei Dokumente editieren kann ist das Ziel sondern User und Logik des Dokuments

zusammen erstellen und VALIDIEREN Dokumente. Die typische „Paintbrush" Metapher bei der Funktionalität über ein GUI direkt angeboten wird, ist z.B. in einem Bankenumfeld unbrauchbar. Dies ist ein Ergebnis, das sich aus dem Workflow Charakter des Frameworks ergibt

3. Die flexibelsten Human Interface Componenten sind nach dem Browser Prinzip organisiert. Nicht Methodenaufrufe verbinden HIC und Model sondern ein definiertes Protokoll. Damit ist der Austausch bzw. die Erweiterung des HIC problemlos möglich.

4. Die Human Interface Component muss applikationsspezifisch erweiterbar sein. Das bedeutet NICHT, dass Model Elemente in das GUI rutschen, sondern dass applikationsspezifisch GUI Elemente integriert werden können. Ein PlugIn Konzept wurde über das Strategy Desing Pattern realisiert.

5. Die Human Interface Component muss dynamisch und applikationsspezifisch aufgebaut werden können. Dokumente können beliebige Struktur beinhalten. Feste Masken sind daher unbrauchbar. Aus SGML Beschreibungen (Style Sheets) werden zur Laufzeit Views erstellt in Abhängigkeit vom Dokumenttyp, der bearbeitet werden muss. Editoren können dabei Userspezifisch oder Taskspezifisch gewählt werden.(zum User Interface als Document: SGML and Distributed Applications: [FUCHS])

6. Bearbeitung der Dokumente sowie Bearbeitung des Ablaufs der Dokumentbearbeitung erfolgen dynamisch durch Austausch von Strategies. Es gibt keinen zweistufigen Prozess wie er typisch für GUI Builder mit GUI Editoren ist.

7. Es dürfen KEINE Methodenpointer als Callbacks verwendet werden. Actionen sind ausschliesslich über Action Objects zu realisieren. Methoden orientierte Ansätze scheitern schnell in C++ sobald Multiple Inheritance eingesetzt wird. Sie sind ausserdem durch die einheitliche Signatur zu wenig flexibel.

8. Zur Implementierung des HIC wird ein plattformunabhängiges GUI Toolkit verwendet. Dieses stellt lediglich Grundfunktionen (lexical items) für Objekte und Interaktionen zur Verfügung. Die HIC kapselt diese in sog. Contexts und kann sie mit Strategies verbinden, die die Semantik beinhalten.

9. Eine generische Datenschnittstelle (Dokument Part Metapher) sowie ein generischer Request Mechanismus erlauben es, generische HIC-Module zu erzeugen, die beliebige Dokumente bearbeiten können.

10. Die HIC kann sich zu einer völlig selbständigen Komponente entwickeln, mit eigenem Modell und Storage Management wie z.B. ein typischer WWW-Browser. D.h. eine Aufteilung in verschiedene „Tiers" kann sich innerhalb einzelner Komponenten wiederholen.

Die Funktionalität des HIC umfasst:

- Type Definitions
- User Interface Hilfsklassen
- Action Classes, ActionCallback Class Templates
- UIContexte (Basis und Templates)
- Strategies
- Iteratoren/Visitoren
- Geometry Handling
- Popups und Menus
- Image Handling
- Dynamische Views
- HIC main factory
- Factories for contexts and strategy objects

- GUI toolkit wrapper classes

APP

Aufgaben der APP Domain

Die Applikationskomponente stellt allgemeine Dienste für Applikationen „als Dinge" zur Verfügung. Damit wird errreicht, daß Applikationen aus Konfigurationsinformation heraus dynamisch gebaut und gestartet werden können. (z.B. ergibt das Verknüpfen einer Frontend Klasse mit einer Server Klasse und einer Action Klasse einen speziellen Server bzw. eine Batch-Applikation)

- Batchthread
- Server
- Frontend
- Messager
- Protocoll
- Action

Wenn Klassen „sehr speziell" sind

Manchmal sieht es so aus, als liesse sich keine sinnvolle Interface Klasse definieren da die Klassenmethoden im wesentlichen Tabelleninhalte oder Parameter physischer Geräte etc. sind. Entscheidend sind die Argument- bzw. Returntypes der Methoden.

Werden z.B. viele Werte mit GetXYZ() zurückgegeben und die Werte sind alle Strings, lohnt es sich über ein symbolisches Interface nachzudenken (composite message design pattern) hier kann der Client über EINE methode z.B. getDataProtocol() ein Objekt der Klasse Dataprotocol erhalten und aus diesem mit getValue(„Token") spezielle Werte erhalten.

Dieses design pattern eignet sich wie gesagt vor allem bei gleichen Value Typen, da natürlich nur eine Klasse von Values zurückgegeben werden kann.

DataProtocol kann dann natürlich durch weitere getXYZ() methoden erweitert werden, die jeweils verschiedene Typen zurückliefern können. Dies wird z.B. in der ResultSet Klasse des Java JDBC Datenbankinterfaces getan.

Hilft alles nichts, kann trotzdem ein allgemeines Interface OHNE die speziellen Methoden definiert werden. Clients die unbedingt die speziellen Methoden brauchen, müssen dann einen dynamic cast durchführen. Es sind jedoch im allgemeinen nur ganz wenige Clients, die dies tun müssen.

Grundsätzlich gilt, dass gerade Klassen die aus einem Reengineering von „alten" Projekten stammen sehr mit „alten" Methoden überfrachtet sind, sie haben sozusagen das alte Design geerbt. Auch die speziellste Klasse hat generelle Typen, so ist z.B. eine Datenbank durchaus einmal als Typ Dokument aufzufassen. Ein spezieller Beleg einer Domain ist ebenfalls ab einer gewissen Ebene nichts anderes als ein Dokument.

Exkurs: zwischen statischem Typing und dynamischer Interpretation

Betrachtet man die Entwicklung von JAVA innerhalb des letzten Jahres fällt auf, dass die verwendete Technologie immer wieder pendelt zwischen einer Betonung von statischem Typing und der Verwendung von interpretativen Verfahren. So kannte das erste Release der GUI-Komponente (AWT) keine statisch getypten Events. Das Dispatching erfolgte über die Interpretation von Symbolen zur Laufzeit. Dies stellte sich in zweifacher Hinsicht als problematisch heraus. Erstens beeinträchtigte es die Performance und zweitens entstanden viele Fehler durch

Verwendung der falschen Symbole, die vom Compiler ja nicht geprüft werden konnten.

Das Java Development Kit 1.1 brachte eine Trendwende hin zu einem statisch typisierten Eventmodell. Speziell im Falle der Java Beans ist dieses statische Modell eng an Werkzeuge gekoppelt, die automatische Adapterklassen generieren, die die jeweiligen statischen Eventtypen kennen. Gleichzeitig wurde allerdings auch ein Reflection API eingeführt womit Objekte zur Laufzeit analysiert werden können.

In der neuen Spezifikation des Java Infobus Konzepts dominiert wiederum die dynamische Interpretation. Die Klasse DataItem (analog zum NEWSYS DataProtokoll) erlaubt es symbolische Informationen über ein einfaches generisches Interface zu transportieren.

Auch im CORBA Bereich finden wir beide Verfahren, z.B. im Event-Service. Und interessanterweise waren auch die Entwickler des Component Broker Connectors der IBM gezwungen ganz unten, d.h. am Interface zum Datenbank-Cache, ein symbolisches Interface (DataAccessObject) einzuführen, während die übergeordneten Klassen (Business Object und DataObject) statische Types sind.

Erfahrungen aus Message-Bus Systemen zeigen dieselbe Problematik: Wenn ein System wartbar bleiben soll, scheint ein gewisses Mass an symbolischer Information (mit anschliessender Interpretation zur Laufzeit) unvermeidlich.

Chapter 5. DIE PHYSISCHE STRUKTUR DES NEWSYS FRAMEWORKS

Bei der physischen Struktur unterscheiden wir grob in statische und dynamische Struktur. Die statische Struktur umfaßt jede Art von Ableitung einer Klasse von einer Basisklasse sowie die direkte Verwendung anderer Klassen innerhalb einer Klasse oder Funktion (d.h. die verwendeten Klassen sind bekannt „by size", ihre Header MÜSSEN includiert werden.

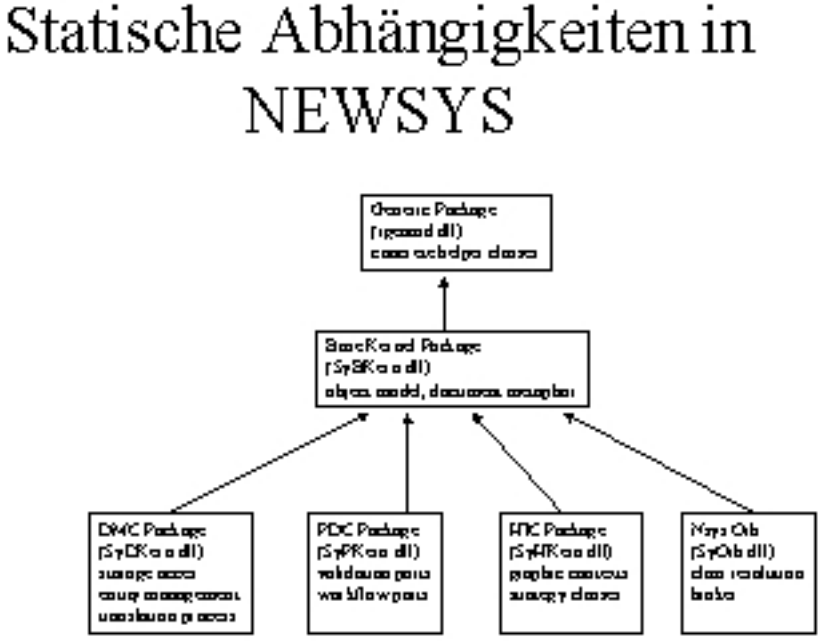
Die dynamische Struktur beinhaltet die Verwendung anderer Klassen über Interface Definitions/Protokolle, vermittelt durch Factories oder den Object-Request-Broker. Dabei wird nicht der Header einer konkreten Klasse includiert sondern die Interface Definition (Sy.V... Klasse). Die gewünschte Objektimplementation kommt dann aus einer Factory oder dem ORB, der Client kennt den konkreten Aufbau der verwendeten Klasse NICHT - d.h. die Factory könnte ihm auch eine andere abgeleitete Klasse desselben Interfaces geben (Polymorphie). Nur die dynamische Verwendung anderer Klassen ermöglicht das Austauschen von Implementationen (auch z.B. zu Wartungszwecken).

Die dynamische Struktur entsteht NICHT NUR durch die Trennung von Interfaces und Implementationen. Wenn lediglich zu jeder konkreten Klasse eine Interface Klasse geschrieben wird, spiegeln sich Implementationsdetails im Interface und es entsteht ein System aus unzähligen sehr speziellen Interfaces. Ein generisches Bearbeiten von solchen Klassen ist auf Grund der verschiedenen Interfaces nicht möglich.

Die Trennung von Interfaces und Implementationen ist die physische Voraussetzung einer logischen Struktur. Die logische Struktur definiert EIN Interface für viele mögliche Implementationen. Wenn es zu einem Interface nur eine MÖGLICHE Implementation gibt, dann ist die Trennung von Interface und Implementation nur aus statischen physischen Gründen erfolgt und nicht um eine Abstraktion und Verallgemeinerung zu ermöglichen.

Statische physische Struktur

Darunter fällt die Art und Weise von Ableitungen die wiederum die jeweiligen physischen Abhängigkeiten bedingen. Eine Ableitung von einer Basisklasse ist ein STATISCHER Vorgang, der Compilieren und Linken bedingt.



Dazu kommen Abhängigkeiten des DMC Packages von OLDSYS Kompatibilitäts Packages wenn sich Interfaces dort ändern sollten.

Wichtig ist, daß alle Komponenten vom Base Package statisch abhängen, jedoch UNTEREINANDER nicht statisch abhängig sind. Das bedeutet, daß z.B. die Änderung einer Implementation in der PDC keine neues Compilieren oder Linken anderer Packages nach sich zieht.

Anpassungen dort sind erst nötig, wenn sich z.B. die Interfaces/Protokolle der PDC ändern und die HIC solche verwendet. Für Zwischenreleases ist es deshalb wichtig, diese Interfaces einzufrieren.

Zur Schicht der konkreten Hilfsklassen besteht eine Linktime Abhängigkeit aller Komponenten. Dies wirkt sich in der Praxis aber nicht gross aus, da der Coding Standard festlegt, dass niemand konkrete Klassen ableiten darf. Die Komponente selbst ist eine dynamische Link Library (DLL) die zur Laufzeit automatisch beim Startup hinzugeladen wird.

Interfaces ("V" class tag)

Alle Interfaces sind von der Root Klasse abgeleitet (== CORBA Object Class)

Interfaces sind **implizite** Protokolldefinitionen ohne Implementation. Sie werden realisiert über Interface Klassen mit pure virtual methods. In NEWSYS befinden sich Interfaces immer in „INTINC" Verzeichnissen und sind meist über inline methods für constructor, destructor und assignment operator implementiert.

(In CORBA werden diese Header vom Interface Compiler aus den IDL Definitionen automatisch generiert). Bei einer kompletten inline Implementation schafft die Ableitung von pure virtual Classes keine link Abhängigkeit, im Fall von NEWSYS jedoch besteht durch die Root Klasse eine link Abhängigkeit aller Packages von BASE.

Default Implementations ("B" class tag) von Interfaces

Default Implementations sind erste Implementationen von Interfaces. Implementiert werden allgemeingültige Basismethoden. In NEWSYS befinden sie sich immer in „INC" und „SRC"

Normale Klassen ("N" class tag), abgeleitet von Interfaces/Default Implementations

Normale Klassen sind von Default-Implementationen abgeleitet und implementieren jeweils spezielle Funktionalität. Sie befinden sich im jeweiligen Package unter „INC" und „SRC".

Sie werden normalerweise NICHT weiter abgeleitet

Konkrete Klassen ("C" class tag)

NICHT von Root abgeleitete Klassen - auch von fremden Bibliotheken wie z.B. Generic -. Sie werden NIE weiter abgeleitet, da ihre Methoden im allgemeinen NICHT VIRTUAL sind.

Physische Konseqenzen der Interface/Implementation Trennung

Innerhalb eines Frameworks wird mit Protokollen gearbeitet statt mit konkreten Implementationen. Dies verhindert **compile und link Abhängigkeiten** unter den Klassen.

Clients kennen bei dieser Architektur üblicherweise (d.h. ohne dynamic cast) nur noch das Interface von objekten. Dies hat zur Folge dass die Instanziierung der Objekte an anderer Stelle erfolgt, z.B. in Factories. Damit verschwindet aus dem Client auch der cold spot des new Statements.

Ein einfaches Kopieren von Objekten ist damit ebenfalls nicht mehr möglich. Gut sind sog. Clone() Methoden bei denen das zu kopierende Object selbst eine Kopie von sich selbst erstellt. Besser ist ein MetaObject proto-

coll bei dem durch die Verwendung eines Reflection APIs ein weiterer Objektzugang eröffnet wird. Ein Beispiel dafür ist das Reflection API des neuen Java Development Kit Version 1.1 [BAI/WKR97]

Der Begriff des Attributes wandelt sich ebenfalls: Üblicherweise bezeichnet er Member einer Klasse. Kennt ein Client jedoch nur das Interface einer Klasse, sind die Member Attribute der Klasse für ihn unsichtbar. Stellt das Interface Methoden wie getAttribute(), setAttribute() zur Verfügung, so bedeutet das NICHT, dass diese Klasse tatsächlich solche Attribute als member hat. Die Klasse erklärt mit diesen Methoden nur noch, dass sie ein Attribute X versteht, z.B. als eine LOGISCHE property ihres states. Intern kann die Klasse alle möglichen Typen von Membern haben und z.B. ein Property X auf einen Membertyp Y abbilden.

Dynamische physische Struktur (Runtime Struktur)

Granularität der physischen Struktur

Darunter fallen das BENUTZEN von fertigen (binären) Klassen oder Objekten aus bestimmten Packages sowie die Packages selber.

Was kann der Inhalt von Packages sein?

1. Klassen, die von anderen Klassen dieses Packages compile- oder linkabhängig sind
2. Eine Anzahl verschiedener Implementationen eines Interfaces die alle einzeln ausgewählt werden können und voneinander völlig unabhängig sind
3. Zwischenformen der obigen beiden.
4. Bugfix Klassen zu den verschiedensten anderen Packages die aus Gründen der Wartung in einem neuen Package zusammengefasst wurden.

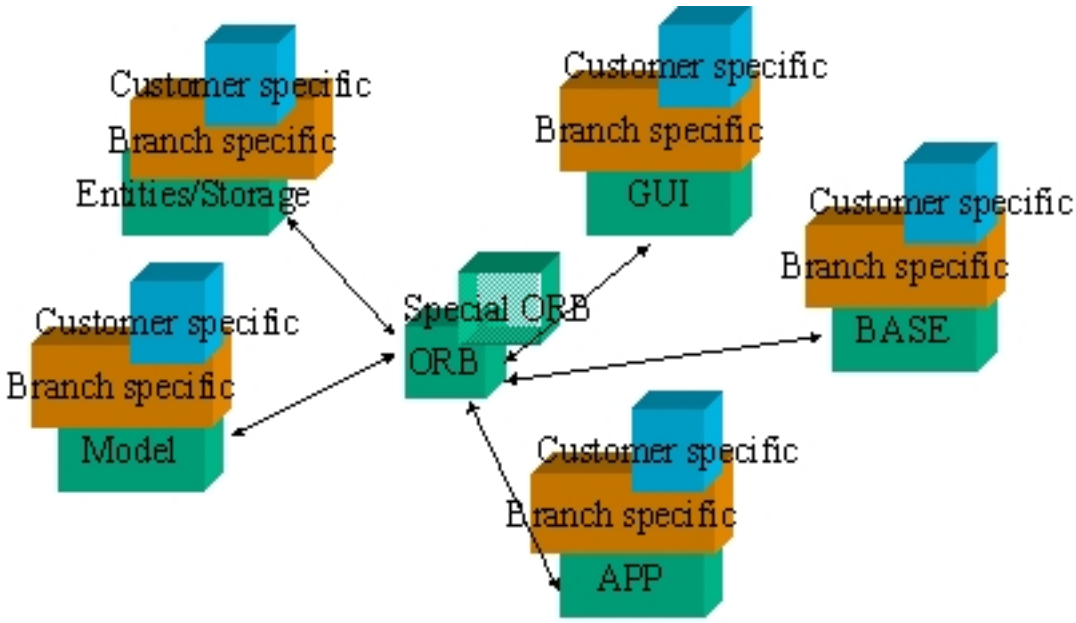
Ergebnis: Die physische Struktur eines Frameworks muss Reuse bis auf die einzelne Klasse hinunter gewährleisten, sonst lassen sich bestimmte Mechanismen wie z.B. Composite Object Design Patterns nur schwer realisieren. Die physische location einer Klasse muss beliebig sein.

Dies ist nur möglich wenn das Framework über einen dynamischen Lademechanismus für Klassen und Packages verfügt, was wiederum Naming und Resolve Mechanismen voraussetzt.

Wenn Interface und Implementationen getrennt sind, können beliebige Implementationen aus Packages on Demand geladen werden ohne daß compilieren oder linken nötig ist. Dies steigert die Flexibilität enorm und gestattet auch einfachere Wartung.

Ohne dieses Konzept müssen bei C++ grundsätzlich immer ganze Produkte ausgetauscht werden statt einzelner Packages oder Klassen und ein Dazuladen von Kundenmodulen zur Laufzeit wäre unmöglich.

Die dynamische Struktur von NEWSYS: Die grösseren Packages haben üblicherweise mindestens eine factory.



Clients erhalten ALLE Objektimplementationen über den ORB bzw. über von diesem erhaltene Factories. Packages können sich über den ORB GEGENSEITIG nutzen mit AUSNAHME der HIC die von Packages ANDERER Domains nicht genutzt werden darf (Prinzip der Trennung von User Interface und Modell). Natürlich nutzen zusätzliche Packages der HIC Domain beliebige andere Packages wie auch das HIC Basispackage selbst.

Factories

Da Interface Klassen nur Aspekte von Protokollen sind, können sie auch nicht instanziiert werden. Die konkreten Implementationen erhält ein Client meist aus „Factory" Klassen. Diese Factories sind als Baum organisiert an dessen Spitze der Object Request Broker (ORB) NSys steht.

Packages

Packages sind typischerweise dynamisch (symbolisch) ladbare DLLs mit deren Hilfe der ORB Wünsche nach Objekten befriedigen kann. Packages enthalten Implementationen von Interfaces. Entweder sorgt eine Factory innerhalb des Packages für bequemen Zugriff oder einzelne Loaderfunctions der Objecte gestatten das symbolische Laden. INNERHALB eines Packages sind alle Addressen von Funktionen oder Objekten entweder relativ zu der Package Factory (und können nur über Methoden der Factory errreicht werden) oder sie müssen symbolisch geladen werden. Voraussetzung für das symbolische Laden ist ein Eintrag im Implementation-Repository BootCore.cfg. Nur wenn dort zu einem Token der Name einer Loaderfunction sowie das zugehörige Package (dll) angegeben sind, kann die Methode CORBA_bind() von NSys dieses Objekt laden und zurückgeben.

Wichtig: Die Verwendung von DLLs in C++ Applikationen OHNE ein symbolisches Ladekonzept reduziert lediglich die Grösse von Applikationen. Keinesfalls ist damit die Möglichkeit gegeben, fehlerhafte DLLs ohne neues Linken des Clients austauschen zu können. Natürlich kann eine DLL neu erzeugt und ausgetauscht werden ohne den Client zu ändern. Dies führt in aller Regel zu mehr oder weniger sinnvollen Meldungen des Betriebssystems und zum Abbruch der Applikation.

Anzumerken ist, daß die Sourcecode-Struktur der Packages durchaus verschieden von der Domainstruktur sein kann. Z.B. befindet sich das Loader-Package unter NEWSYS/OS/$(Architecture)/loader, gehört aber zur Base Domain.

Die meisten Packages beinhalten auch eine Factory-Klasse zum dynamischen Einfügen weiterer Implementatio-

nen (siehe unten: Erweiterungsstruktur). Die Factory-Klasse heißt typischerweise [Domain][Type]Kit

Ebenfalls gehört eine Packageheader Klasse zu jedem Package. Sie besitzt ein Interface das jedes Package ALS OBJEKT greifbar macht (Release Stand des Packages, in welchem NEWSYS Release generiert etc.) Die Methoden werden über Macros automatisch generiert, da z.B. als Releasestand des Packages einfach die RCSID (Versionsnummer) des Packageheader.cpp files genommen wird. (s.u. Sourcecode Struktur)

Über das PackageHeader Interface kann z.B. der Object Request Broker prüfen, ob ObjektImplementationen aus Packages geladen werden, die einem anderen globalen NEWSYS Release entstammen oder die mit einem älterne Base Package gelinkt wurden. Wenn es im Teil der System Config ein Element gibt, in dem Packages mit linktime Abhängigkeiten aufgeführt sind, dann kann diese Prüfung durch den Loader per Konfiguration erweitert werden.

Maßgebend für die Package Namen und die benutzte Source Struktur ist die NEWSYS Verzeichnis- und DLL Struktur, die ein Teil des Coding Standards ist. Hier nur ein Beispiel für den Unterschied zwischen Logischer Domain und der sie implementierenden Packages:

BASE Domain

Für drei Packages der Base Domain gilt, daß zu ihnen eine Link- Abhängigkeit der anderen Packages besteht und daß sie statisch beim Programmstart geladen werden. Alle anderen Packages werden dynamisch geladen und Referenzen symbolisch aufgelöst. Services dieser drei Packages sind jedoch zum Booten des Frameworks nötig und können deshalb nicht selbst by symbol geladen werden.

- BaseKern.dll implementiert das Object Modell, Event Services, Memory Management Service. Enthalten ist auch BaseKit, die Factory dieses Packages
- Loader.dll enthält die plattformspezifische Implementation des class und function loaders.
- NSys.dll enthält den Object Request Broker und liefert first initial Referenzes auf weitere Factories sowie Object references einiger dauernd benötigter Objekte (config service, DataProtocol etc.) Jede Applikation kann eine eigene ORB Klasse einfach ableiten und dadurch den Boot Prozeß kontrollieren und konfigurieren. Per Default lädt der ORB alle Objekte on Demand.

Die Interfaces dieser Packages sind unter:

NEWSYS/Kernel/Base/interfaces.

NEWSYS/NSys/interfaces

Die Implementationen unter:

NEWSYS/Kernel/Base/implementations

NEWSYS/OS/$(Architecture)/loader/implementations

NEWSYS/NSys/implementations

Alle Packages gehören LOGISCH zur gleichen Domain, sind aber physisch in unterschiedlichen Strukturen weil sie z.B. plattformabhängigen Source beinhalten.

Object Request Broker (ORB): NSys

Zur Rolle eines ORBs in C++ Frameworks

Häufig wird ein ORB mit Distributed Computing in Verbindung gebracht. Die Besonderheiten von C++, speziell das statische Erstellen der Vtable sowie das Fehlen echter Interfaces erzwingen aber oft den Einsatz eines ORBs auch für lokale Anwendungen. Anders ist eine Trennung in Interfaces und Implementationen mit flexiblem und dynamischen Austausch von Implementationen nur schwer möglich.

Fällt das für distributed computing nötige Marshaling/Unmarshaling weg, reduziert sich die Funktion eines ORBs auf den Lookup von angeforderten Implementationen sowie auf das Anbieten einiger zentraler Services oder Factories aus Gründen der Bequemlichkeit. Ein ORB ist daher leicht selbst zu erstellen.

Im NEWSYS Framework spielt die Klasse NSys die Rolle des ORBs (separate Unterlagen). Hier sei nur erwähnt, daß der ORB die zentrale Vermittlungsstelle der dynamischen Struktur von NEWSYS ist. Über ihn erhalten Clients die gewünschten Objekte. Strenggenommen ist er Teil der Base Domain. Da er jedoch am Anfang häufiger modifiziert wurde, erschien es besser ihn als separates Package zu implementieren. Zum ORB gehört auch ein Implementation Repository das alle Informationen über existierende Implementationen von Interface Klassen beinhaltet. Der Zugriff kann über Token, Classname oder applikationsspezifische Kriterien erfolgen.

Gekoppelt mit dem ORB ist auch der Bootstrap des Frameworks. Client Applications initialisieren zuerst den ORB und haben dann Zugriff auf weitere Objekte.

Clients können leicht eigene Ableitungen des ORBs erzeugen um z.B. spezielle Initialisierungen während des Bootstrap durchzuführen oder den Object lookup zu spezialisieren.

Die NSys ORB Klasse implementierte ein Template Hook Pattern, um einzelne Funktionalitäten des Bootens gezielt abändern zu können ohne die logische Reihenfolge zu beeinträchtigen.

Damit auch Spezialisierungen des ORBs verwendet werden können, registriert sich ein ORB über ein setORB(thisOrb) methode in der Base des Frameworks. ALLE Komponenten kennen eine globale Funktion getORB() die eine Referenz auf den ORB verfügbar macht.

Die ORB Referenz könnte auch in thread specific storage gehalten werden, dann könnten mehrere ORBs gleichzeitig im Framework tätig sein. Bei NEWSYS wurde das als unnötig angesehen.

Grundfunktionen eines lokalen ORBs

1. statische Initialisierungsfunktion mit Registrierung in Base Package.
2. Get und Set Methoden für toplevel Factories (die im wesentlichen die Factories der Top Level Packages verfügbar machen, sowie die Systemkonfiguration.
3. Boot() Template Methode sowie Hooks (bootEnvironment, bootSystem, bootDataManagement, bootHIC , bootApplication etc.)
4. ResolveClass() und LoadObject Methoden
5. Convenience Methoden für Konversion einiger Interface Typen in konkrete Typen.

Beispiel eines Implementation Repositories:

Table 5.1.

Token	**ClassName**	**Package**
BasicWorkflow	PTPNBasicWorkflow	PTPWflow.dll
DMCFactory	PTDNDMCKit	PTDKern.dll
DefaultRequest	PTPNRequest	PTPKern.dll

Durch Austausch von Package Namen kann jederzeit die Implementation gewechselt werden.

Applikationen können den ResolveImplementation-Hook des ORBs jederzeit überladen und eigene Strukturen definieren. Das Implementation Repository ist ein SGML-File und sichert daher die Eindeutigkeit von Tokens und lässt beliebig komplexe Namespaces und Gliederungen zu.

Das Implementation Repository ermöglicht eine andere Verwendung auf Klassenebene OHNE auf die Ebene des Source Codes zurückzufallen.

Die Granularität kann dabei von der Applikation selbst bestimmt werden, denn es lassen sich in das Repository genauso die PackageHeader Classes der Packages eintragen. Damit werden die Packages als solche fassbar.

Loader-Functions

Für das dynamische Laden der Implementation zur Laufzeit wurde in allen Objektimplementationen eine sog. Loaderfunction mit festgelegter Signatur implementiert.

OHNE Verwendung von CORBA Tools wie z.B. einem IDL Compiler muss eine Einschränkung für die Signatur von Construktoren definiert werden, da die Signatur der Loaderfunction einheitlich sein muss. In der Praxis hat sich das als nur geringe Einschränkung erwiesen.

Die physische Struktur der NEWSYS Konfiguration, Dokumentenbeschreibungen und Workflow Informationen

Die Art und Weise wie NEWSYS eigene Konfigurationselemente sowie Dokumenten und Workflow Information behandelt, unterscheidet sich grundlegend von OLDSYS. Da das Verfahren sehr gut auf verschiedene Formate erweiterbar ist, intern den Benutzern jedoch immer das gleiche Interface liefert, kann man hier von einem eigenen Subframework innerhalb von NEWSYS sprechen.

NEWSYS behandelt alle Formen symbolischer Information über die gleiche Dokumentenmetapher als „strukturierte Dokumente"

NEWSYS verwendet SGML Techniken wie General- und Parameterentities um das Kopieren und Duplizieren von Information zu vermeiden, was sonst große Wartungs- und Zuverlässigkeitsprobleme mit sich bringt. Damit kann gleichzeitig erreicht werden, daß harte Pfadangaben etc. aus den Systemkonfigurationen verschwinden. Der SGML Parser un Verbindung mit dem Entity-Manager löst Entity-Referenzen zur Laufzeit auf. Installation, Wartung und automatische Updates werden dadurch erleichtert bzw. erst möglich. Die strukturelle Validität der NEWSYS Systemdokumente wird bereits durch den Parser erzwungen.

NEWSYS verwendet ISO Standards für seine Dokumente wie auch deren Meta- Beschreibungen, damit handlesübliche SGML Tools zur Bearbeitung eingesetzt werden können. (Beispiel: Near&Far’s Tools zur graphischen Analyse und zum editieren von Dokument Type Descriptions.

NEWSYS kann das gesamte symbolische Environment des Systems bzw. einer Applikation BEIM STARTUP der Applikation erzeugen. D.H. eine Änderung EINER Stelle in einer Konfigurationsentity wird SOFORT für alle Applikationen wirksam. Dies ist möglich, weil Information referenziert statt kopiert wird.

Da das Includieren über den Entity manager erfolgt, ist sogar das Includieren von Information die sich auf anderen Servern oder im WWW irgendwo befindet möglich. Es muß z.B. als Identifier lediglich eine URL (Protocol://Server/Parameter angegeben werden und die Installation eines Kunden würde zum Startup eine Verbindung zum Hersteller aufbauen und das entsprechende Konfigurationsfile vom Web Server laden.

Das Kozept des Entity-Managements erlaubt darüber hinaus auch das Auflösen von Referenzen zur Laufzeit, d.h. ein dynamisches Abändern der Konfiguration wäre ebenfalls möglich.

Selbstverständlich ist auch das einzelne Parsen und Prüfen aller Konfigurationen mit Hilfe des nsgmls.exe Parsers möglich. Dieser erzeugt einen Zwischencode der ebenfalls von NEWSYS gelesen werden kann.

OLDSYS-Konfiguration

Systemkonfiguration: harte Drivenamen, fixe Pfade, nicht dynamisch überschreibbare globale Konstanten, In-

formation zur Peripherie gemischt mit logischen, d.h. hardwareunabhängigen Informationen.

Programmkonfiguration: harte Drivenamen, fixe Pfade, Workflow information gemischt mit anderen Informationen und nur über Kopieren an andere Stelle verwendbar.

Dokumentkonfiguration: Sie zeichnete sich durch eine Mischung unterschiedlichster Informationsarten aus.

physische Information über ein Dokument

Scanner Information

logische Information (Prüflogik)

View Information (GUI-Konfiguration)

alles in einer Datei. Anpassungen für Kunden, Peripherie oder Workflow führen zum Kopieren. Änderungen werden durch die große Zahl leicht geänderter Kopien immer schwieriger. Automatische Updates im Feld sind unmöglich.

Die Einführung neuer Gliederungskriterien wie z.B. Mandanten war auf dieser Basis kaum möglich.

NEWSYS Konfigurationsframework

Das Konfiguration Framework ist kein in sich abgeschlossener Teil des Framework sondern benutzt die im Framework vorhandenen Mechanismen zur Bearbeitung von Dokumenten. Seine Hauptmerkmale sind:

- Aufsplittung (Faktorisierung) unabhängiger Teile in „sgml entities"
- Bezug auf Symbole über „formal public identifiers"
- logische Dokumentenbeschreibungen in „entities"
- physische Dokumentenbeschreibungen in „entities"
- Systemdefinitionen (Pfade, Drivenamen) in „entities"

Währen der ORB Mechanismus Clients vom Wissen über den Standort von Objekten (und ihre physische Implementation) befreit, ermöglicht das Konfigurationsframework die Unabhängigkeit der Clients von Systemdetails wie Pfaden, Drives, Ressourcen etc. Alle diese können symbolisch adressiert werden. Auflösung der Namen und Adressen sowie Instanzierung sind Aufgaben des Frameworks. (Vgl. dazu auch das JDBC Datenbank-Framework in Java)

Start einer Applikation:

Anmerkung: Mit „Applikation" ist ein gewisser Workflow gemeint und nicht im herkömmlichen Sinn ein Programm in Form eines .exe Files. NEWSYS „Applikationen" sind im allgemeinen lediglich Konfigurationsdateien in SGML.

Aufruf: NewSys.exe MyApplication.sgm

1. Generische Klassen der Application-Domain booten NEWSYS Framework (Broker, Base DLLs)
2. NEWSYS erzeugt ein symbolische Runtime Environment „config". Dabei handelt es sich um etwas ähnliches wie den CORBA Naming Service jedoch mit erweiterter Funktionalität wie z.B. mehrere gleichlautende Token/Objekt Paare pro Context und der Möglichkeit von update events für alle Contexte. Darüberhinaus besassen die Contexte Suchmethoden.

Die Initialisierung dieses Environments erfolgt folgendermassen:

- NSys lädt Config Object mit Parameter appl.sgm über Methode der Base Factory.
- Die Base-Factory lädt ein Config Object mit Parameter appl.sgm
- Das Config-Object fordert einen ConfigBuilder aus der BaseFactory an
- Der ConfigBuilder holt einen ParserEventGenerator aus der SGMLFactory mit argument appl.sgm
- Der SgmlKit erzeugt den richtigen ParserEventGenerator für .sgm files
- Der ParserEventGenerator lädt app.sgm und ruft den SGML-Parser
- Der SGML-Parser validiert appl.sgm und gibt Ergebnisse an den ParserEventGenerator
- Der ParserEventGenerator ruft den Config Builder mit Ergebniss-Events auf
- Der Config Builder baut einen Part-Graph aus den Events auf.
- Das Config-Object initialisiert sich mit Part Graph
- Die Base-Factory gibt das fertige Config-Object zurück
- NSys merkt sich diese Config als „current config". Diese current config ist jederzeit zur Laufzeit erweiterbar durch merge() methoden. Information zum Environment kann in beliebigen Formaten vorliegen, da der jeweilige ParserEventGenerator diese immer in die gleichen SGML-Events transformiert.

3) generische Applikatione Klassen fordern vom NSys Broker Action Objekte an und starten sie als Thread oder Prozess. D.H. es gibt kein Applikation.exe Programm sondern ein oder mehrere DLLs mit Spezialisierungen der NEWSYS Klassen sowie ein symbolisches Environment Application.sgm.

Bei der Erzeugung des symbolschen Runtime Environments wird das Parser Pattern verwendet. Config-Objekte operieren nur auf Graphen von Parts. Physische Formate und syntaktische Validierung erfolgen im Zusammenspiel von Parser und ParserEventGenerator. Neue Config Formate bedeuten neue ParserEventGeneratoren. ConfigBuilder ist vom Type SGMLApplikation, d.h. ParserEventGeneratoren kennen dieses Interface und können an solche Objekte Events weiterreichen.

Wie sieht die logische, physische und runtime Struktur dieser Config aus?

logische Struktur: Ein Graph aus Elementen, validiert durch den Parser

Dokument-Type-Descriptions (DTDs) und Element-Models legen die logische Struktur einer Konfiguration fest:

Was muß in einer legalen Config enthalten sein? Auf welche Weise und Wo muß es enthalten sein?

Der Parser validiert diese logische Struktur wie ein c/c++ Compiler ein sourcefile parst und validiert. Und ganz ähnlich wie dieser andere Files dabei „included", ruft der SGML Parser den Entity Manager wenn er auf eine Symbolreferenz stößt, die nicht bereits vorher definiert wurde.

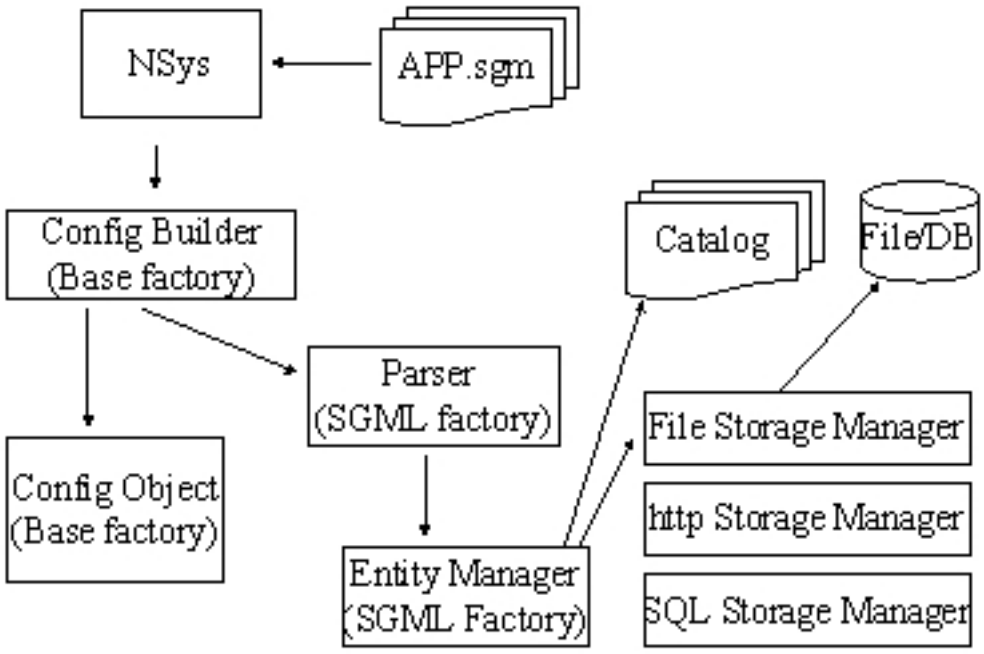
Wie der c/c++ Compiler arbeitet der SGML Parser letztlich auf der LOGISCHEN Struktur die ein Ganzes bildet, egal ob dieses logische Ganze bereits in einem File zusammengefügt war oder erst durch das Einfügen vieler „Includes" oder in SGML: „Entities" entstanden ist. Eine Besonderheit der SGML Entities ist jedoch, dass sie durch den Entitymanager notfalls zur Laufzeit erst erzeugt werden können, z.B. durch eine Datenbankanfrage oder durch ein „auschecken" der Information aus einem Source Code Control System. Üblicherweise sind Compiler leider kaum so offen gebaut, dass man eine solche Funktionalität nachrüsten könnte.

physische Struktur: Eine Anzahl von Entities

Entities (ähnlich den c/c++ include files) beinhalten die Dokument Type Descriptions, Element Modelle (== Element Definitionen) und Daten. Diese können aus anderen Dokumenten includiert werden, d.h. Information muss nicht kopiert werden.

Runtime Struktur: Wie entsteht ein Config Object?

Wie wird aus einem Anfangsdokument (Appl.sgm) ein Config Object OHNE dass zwischen den Beteiligten Klassen unnötige Compile- oder Linkabhängigkeiten entstehen? Die Lösung liegt darin, Object Allocation immer über Factory Interfaces bzw. Broker durchzuführen wobei die beteiligten Klassen ausschliesslich Interface Definitionen voneinander kennen. Im folgenden Diagramm stellen die Pfeile den Ablauf der Aufrufe dar.



- appl.sgm referenziert über symbolische Namen(Formal Public Identifier) eine bestimmte Entity
- ConfigBuilder aus der BaseFactory erhält Document Name und holt Parser aus SGML Factory
- SGML Parser findet symbolische Referenz in app.sgm und fordert seinen Entity manager auf, die Referenz aufzulösen.
- CATALOG enthält ein Mapping von Symbol auf Entity
- Entity Manager braust durch CATALOG und findet Entity Location. Er übergibt Entity an Storage manager.
- Storage Manager lädt Entity und gibt sie zurück.
- Entity Manager gibt Parser die offene Entity.
- Parser fügt Entity Inhalt in Dokument ein
- Config erhält alle Inhalte OHNE überhaupt zu merken, daß sie aus vielen Teilen entstanden sind bzw. diese Teile ERST ZUR LAUFZEIT ÜBERHAUPT ENTSTANDEN SIND (z.B. durch Datenbank Zugriffe)
- ConfigBuilder bekommt Events vom parser mit dem Inhalt des Dokuments, holt sich C++ Dokument Part Objekte aus Factories, füllt den Inhalt der Dokument Teile ein und verknüpft die Parts zu einem Composite Object.

Hinter dem Konzept der "formal public identifier" steht nichts anderes als das Grundproblem der Informatik:

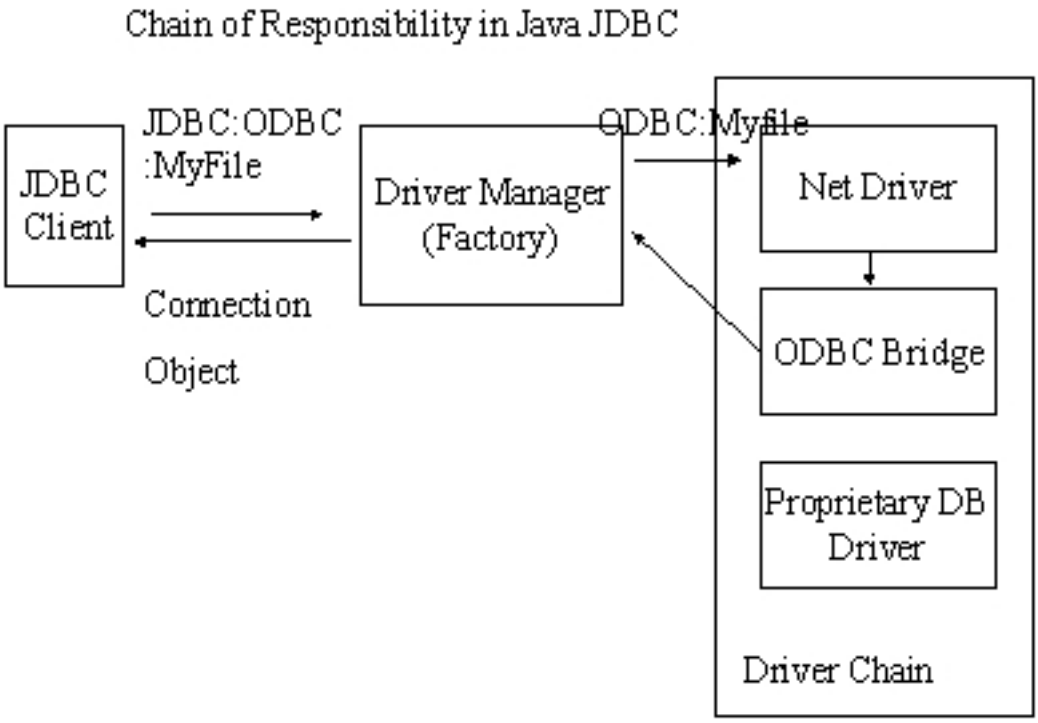
Was bedeutet ein Name? Wie finde ich das, worauf sich ein Name (Link) bezieht?

In Hytime, dem ISO Hypertext Standard findet dieses Konzept breite Anwendung. Die neueste Version des SGML Parser Toolkits von James Clark (SP-Parser) unterstützt auch das Konzept der „architectural forms", die in diesem Zusammenhang eine große Rolle spielen.

Performance und Skalierungsstruktur am Beispiel des Chain of Responsibility Patterns (COR)

Die logische Strukur eines COR ist einfach: Ein Client verwendet eine Factory um z.B. ein Connection Object zu erhalten, das ihn mit einer Datanbank verbindet. Er spezifiziert seinen Request symbolisch, d.h. er gibt keine physischen Charakteristiken der Datenbank an (z.B. Oracle, über TCP/IP, von Server Foo). Aufgabe der Factory ist es, die Verbindung zur richtigen Datenbank herzustellen OHNE Kenntnis des Sinns des symbolischen Request und mit der Möglichkeit Datenbanken dynamisch hinzuzufügen, OHNE die Factory ändern zu müssen.

Diese Anforderungen lagen auch dem JDBC Konzept von Java zu Grunde:



Die Driver werden entweder von der Factory bei der Instanziierung der Factory geladen oder ein Client instanziiert einen Driver selbst, der sich daraufhin bei der Factory registriert. Kommt ein Client Request, leitet ihn die Factory (mit Ausnahme des ersten Teil der URL) und den ersten Driver der Kette weiter. Erkennt der Driver die URL als seine an, gibt er ein Connection Object zurück, das die Factory weiter an den Client reicht.

In diesem Beispiel wäre dies der 2.Driver der Chain. Fühlt sich kein Driver zuständig, wird eine Exception erzeugt. Jeder Driver enthält einen Interpreter der die spezielle URL Semantik seines Drivers versteht.

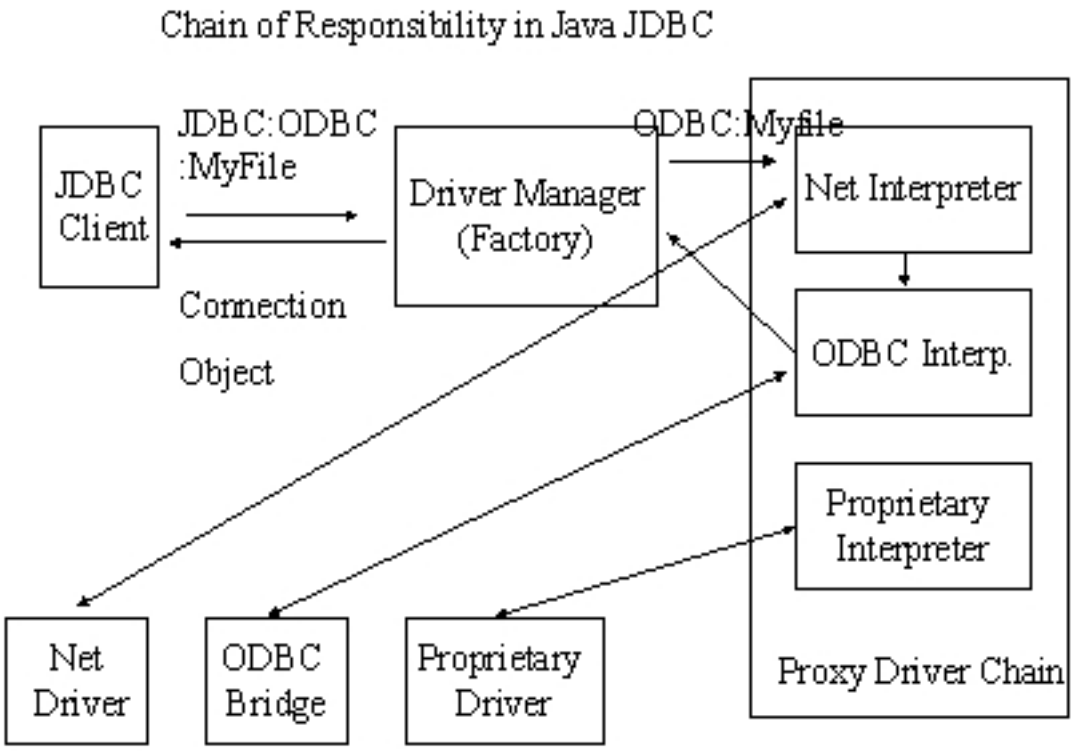
Diese Implementation des logischen Patterns hat folgende physische Konsequenzen bezüglich Scalierung und Performance:

1. Memory-Usage: Alle Driver müssen geladen werden, denn der Interpreter sitzt im Driver
2. Driver-Resolution-Performance: Je mehr Driver registriert sind, desto länger kann es dauern bis der richtige Driver gefunden ist.

Sowie die logische Konsequenz, dass bei zwei Drivern mit gleicher URL der erste Driver in der Kette genommen wird.

Wenn der logische Teil der Extension Struktur die Anzahl der Driver als hot spot definiert hat, (d.h. man rechnet mit einer grösseren Zahl von Drivern), dann skaliert diese spezielle Implementation des COR Patterns nicht.

Eine physisch skalierende Implementation wäre z.B. diese:



Hier wird die Interpretation der URL von Driver Proxies vorgenommen, die nur im Falle eines Matches den entsprechenden Driver laden.

Falls die Interpretation selbst zu aufwendig wird muss ein request-pattern basierter Dispatch Mechanismus verwendet werden, bei dem ein weiterer Teil der inneren Struktur der URL in die Factory verlagert wird.

Chapter 6. DIE ERWEITERUNGSSTRUKTUR DES NEWSYS FRAMEWORKS

EXKURS für Framework Neulinge: Ein einfaches Beispiel für das Metapattern „Framework" worin sein technischer aber auch sozialer Charakter zum Ausdruck kommt

Ausgangspunkt:

Bei einem Laserdrucker der Firma Haltbar & Preiswert sollen zusätzliche Schrifttypen unterstützt werden.

1. Anlauf

Die Entwickler werden von der unverschämten Forderung des Managements vollkommen überrascht und gehen nach einigem Geschimpfe an die Arbeit. Die Software des Laserdruckers wird um weitere Schrifttypen erweitert, neu in Eproms gebrannt, getestet (die ganze SW) und in die Produktion gebracht. Zuvor wurden die Handbücher erweitert, neu gedruckt und an Vertrieb und Kunden verteilt.

Anschließend erfolgt zu dem ganzen Vorgang ein Management Review, wobei der Entwicklungleiter die Hoffnung zum Ausdruck brachte, daß das Produktmanagement in Zukunft VORHER Bescheid geben soll wenn sie ZUSÄTZLICHE Funktionalität benötigten (Der Widerspruch blieb unbemerkt).

Das Produkt Management brachte zum Ausdruck, daß das Ganze ja wohl ziemlich umständlich sei für ein paar neue Schrifttypen und daß die Geschäftsleitung dem Vertrieb deutlich machen soll, daß in Zukunft den Kunden solche spinnigen Ideen auszureden bzw. gar nicht erst einzureden seien. Das Ganze lohne sich doch nicht.

Die Dokumentationsabteilung wollte wissen, ob das die letzte Revision der Handbücher gewesen sei, da der vorgesehene Order zu diesem Drucker mittlerweile voll sei!

Der Mitarbeiter in der Kopierabteilung kündigte nach der 5. Korrektur des Handbuchs und wechselte zu McDonalds mit der Begründung, dort wäre die Produktpalette wenigsten immer gleich besch.. .

2. Anlauf

Ein findiger Entwickler kommt auf die Idee, die neuen Schrifttypen in einen EXTRA Prom zu setzen und interessierten Kunden zu liefern.

Reaktion des für den Laserdrucker zuständigen Gruppenleiters: Sollen unsere Kunden wohl an unserem H&P Laser herumlöten und schrauben? Das kann nicht gutgehen. Der ist viel zu komplex. Die machen nur Mist. Der ist dafür nicht ausgelegt. Geht nicht. Kunden können keine Proms handhaben, die verbiegen bloß die Beinchen und machen sie statisch kaputt. Geht nicht. Wir sind hier die Profis und solche Erweiterungen brauchen Profis.

Nach einigen Überlegungen kommt der findige Entwickler auf die Idee, daß er ein LEERES INTERFACE braucht. Eine Leerstelle im System, die später ausgefüllt werden kann, z.B. indem der Kunde dort etwas später erworbenes hineinsteckt. Die Handhabbarkeit der Proms ließe sich durch ihre Verpackung in Cartridges aus Plastik vereinfachen. Der ganze Drucker müßte an zwei Stellen geändert werden: die physische Struktur müßte um ein Cartridge Interface erweitert werden. Die logische Struktur müßte um den logischen Teil des Cartridge Interfaces erweitert werden.

Er geht mit diesen Ideen zum Gruppenleiter und blitzt ab. Der hat noch die Kosten der letzten Schrifttypen Änderung und den Rüffel des Entwicklungsleiters im Gedächtnis. Er legt dem Entwickler die Unmöglichkeit seiner Ideen überzeugend dar („wissen sie denn was Änderungen kosten???") und schickt ihn nach Hause, nicht ohne ihm vorher noch väterlich auf die Schulter zu klopfen und für seinen Einsatz zu danken.

3. Anlauf

Der Zufall will es, daß der neue CEO der Firma H&P von diesem Vorfall erfährt (Beim Cocktail erzählt ihm der Entwicklungsleiter, mit was für verrückten Ideen die Entwickler manchmal kämen und daß sie überhaupt keine Ahnung von den Kosten hätten)

Der CEO der Firma war vor kurzem aus der Automobilbranche zu H&P gewechselt, angelockt durch ein süßes Aktienpaket und hatte von Druckern ungefähr so viel Ahnung wie von Automotoren - nämlich gar kein. Dennoch hatte er in diesem Moment ein Erlebnis schlagartiger Wiedererkennung: Diese „Cartridges"??? kamen ihm irgendwie wie Stecker vor. Und Stecker hatte er schon gesehen: in seinem Auto! Da war ein Zigarettenanzünder. Und seine Frau hatte mal eine elektrische Luftpumpe für das Gummiboot ihrer Kinder gekauft. Und eine Leselicht zum besseren Kartenlesen im Auto (Sie kaufte immer solches überflüssiges Zeug) Auf jeden Fall konnte man diese Dinge dort reinstecken, wenn man den Zigarettenanzünder herausnahm. Und wenn er sich recht erinnerte machte seine alte Automobilfirma eine beträchtliche Menge ihres Umsatzes mit solchen Dingen, die man nachträglich irgendwo in den Autos reinstecken oder anklemmen konnte!! Und von Umsatz verstand er etwas.

Am nächsten Tag rief er den Entwicklungsleiter zu sich und fragte, warum in dem Laserdrucker kein Zigarettenanzünder sei. Wo sollten denn die Kunden ihr Zubehör reinstecken?? In seiner Verzweiflung rief der Entwicklungsleiter den Gruppenleiter und der den findigen Entwickler dazu und dem gelang es, den CEO vom Zigarettenanzünder auf einen Cartridge Slot umzustimmen.

Auf jeden Fall verlangte der CEO vom Entwicklungsleiter, daß in Zukunft mehrere Stecker in die Drucker eingebaut würden. Und kurz bevor er die Beiden wegschickte erinnerte er sich noch an etwas wichtiges: In seiner alten Automobilfirma wären die Stecker VON BEGINN AN eingebaut!!!

4. Anlauf: Alternative A

Der findige Entwickler versucht, die nötigen physischen und logischen Schnittstellen einzubauen. Schnell stellt er fest, daß das momentane Design dies nur sehr schwer zuläßt. Es muß ein Redesign von Null an her. Und die Stecker müssen von Beginn an eingeplant sein.

Der Entwicklungsleiter ist ob der Kosten entsetzt, aber da der CEO nicht locker läßt, gibt er widerwillig den Auftrag zum Redesign. Der Gruppenleiter ist überaus skeptisch und verweigert jegliches Einbringen seines Architekturwissens.

Der findige Entwickler baut das logische und physische Cartridge Interface ein und während dieser Arbeit kommt ihm eine weitere Idee: Wäre es nicht auch sehr schön, wenn die Hostanbindung des Druckers ebenfalls über einen solchen Stecker funktionieren würde? Leider stellt sich schnell heraus, daß das Font Cartridge Interface dafür nicht so geeignet ist, es müßte ein controller slot sein. Die Integration wäre dadurch möglich, daß der Laser beim Booten an eine Memory Adresse schaut und wenn sich dort ein bestimmtes Muster und eine Startadresse befindet ruft er diese Adresse zu bestimmten Zeitpunkten auf: beim Booten, wenn es Fehler gibt, bei einer neuen Seite, etc. Dazu wäre aber ein neues Design des Motherboards und der Systemsoftware nötig!

Dadurch verzögert sich das Redesign, der Entwicklungsleiter sieht die Kosten steigen und cancelt das Projekt schließlich. Wenn solche Designs so aufwendig wären, wäre die Time to Market dahin, es sei besser etwas weniger flexibles zu haben aber dafür schneller am Markt zu sein. Und außerdem bräuchte er - der findige Entwickler - die Dokumentation ja nicht ändern. Das sei Aufgabe der Dokumenten- und Kopierabteilung.

Dem CEO erklärt er das Scheitern durch unüberwindbare technische Probleme die ER übrigens bereits bei Beginn vorausgesehen hätte. Ein Zigarettenanzünder hätte eben doch nichts mit dem Laserdrucker von H&P zu tun, obwohl seine - des CEOs - Idee natürlich einfach großartig gewesen sei.

4. Anlauf: Alternative B

Es treten dieselben Probleme wie bei Alternative A auf. Der Entwicklungsleiter will das Projekt canceln aber wie es der Zufall will, hatte die Frau des CEOs am Vortag einen automatischen Zigarrenanzünder zum Geburtstag ihres Gatten erstanden, den man in den Zigarettenanzünder einstecken konnte.

Völlig begeistert von diesem Gerät beschließt der CEO, daß hinter der Stecker Idee wirklich etwas stecke und gibt weitere Mittel für das Projekt frei.

Mit einiger Verzögerung entstand daraus ein Laserdrucker mit verschiedenen Erweiterungsmöglichkeiten für Fonts, Sprachen, Hostanbindungen und sogar ein ganz allgemeines Interface für weiß Gott welche zukünftigen Anwendungen. Diese Erweiterungen werden separat entwickelt, getestet, dokumentiert und verkauft. Der Gruppenleiter beschloß aus diesem Vorfall zu lernen, ging heim und zerlegte sein Auto auf der Suche nach Steckern, Klemmen etc.

Mittlerweise produzierten sogar schon andere Firmen Teile die man in den Drucker hineinstecken konnte.

Wenig bekannt hingegen ist, daß sich unten am Bodenteil des Druckers eine kleine Klappe befindet, hinter der ein 12 V Zigarettenanzünder sitzt.

In diesem kleinen Exkurs zum Frameworking sind eine ganze Reihe von Faktoren angedeutet, die das Scheitern oder Gelingen eines solchen Projektes ausmachen, und es handelt sich dabei nicht ausschließlich um technische. Eine kleine Liste der Fehlermöglichkeiten:

- Unterschätzung der Zeit für Faktorisierung (finden der richtigen Abstraktionen)
- Überdehnung der Aufgaben des Frameworks (Typ Eierlegende Wollmilchsau)
- Fortdauernde Ausdehnung der gewünschten Funktionalitäten.
- Unterschätzung der nötigen finanziellen Ressourcen.
- Unterschätzung der nötigen internen Ausbildung der Mitarbeiter.
- Soziale und Gruppenprobleme, Ängste und Ablehnung auf Seiten einiger Mitarbeiter.
- Mangels Erfahrungswerten besitzt das Management kaum Kriterien für Planung und Kontrolle.
- Unterschätzung der sprachspezifischen Probleme, speziell bei C++ (binäre Abhängigkeiten, lange comile und linkzeiten etc..
- Unterschätzung der Zeit für Evaluation.
- Mangelnde Eigenwerbung (Evangelists)
- Keinen Mentor einzusetzen obwohl alle in der Gruppe Neulinge im Frameworking sind bzw. sogar noch Neuling in C++ und Objektorientiertheit insgesamt.
- Mangelnde Entwicklungsumgebungen (Source Code Control, automatische Builds)
- Fehlende strikte Konventionen innerhalb der Entwicklung (Source Struktur unübersichtlich, Klassennamen nicht selbstsprechend, z.B. wg. falscher Rücksichtnahme auf 8.3 Filename Notation)
- Das Fehlen kleiner Tools die die Entwickler bei der Einhaltung der Konventionen unterstützen
- Das Fehlen von Makros die gemeinsame Funktionalität automatisch in neue Klassen einbringen. Je größer das Softwarepaket wird, umso wichtiger wird der generative Aspekt bei der Software Produktion
- Fehlen automatischer Dokumentation der Referenzen (gut RTF, besser HTML, noch besser SGML/XML)
- Design von Flexibilität und Erweiterungsmöglichkeiten ungenügend oder an den falschen Stellen eingesetzt.
- Mangelnder Einsatz von existierenden Fremdprodukten (Sie können nicht einfach eingeführt werden sondern müssen geschult und ihr Einsatz durchgesetzt werden.)
- Unterschätzung der Probleme plattformunabhängiger Software (Habe ich einen Spezialisten pro Plattform?)

Grundbedingungen der Erweiterbarkeit

Neben der Wiederverwendbarkeit von Teilen und dem vorgegebenen Ablaufrahmen ist eines der entscheidenden Merkmale von Frameworks die Erweiterbarkeit, sei es durch interne Programmierer oder durch Kunden.

Die logische Struktur von Erweiterbarkeit besteht bei objektorientierten Frameworks in der Ableitung vorgegebener Defaultimplementationen. Die dadurch entstehenden Objekte entsprechen den vom Framework vorgegebenen Interfaces und können somit auch vom Framework polymorph verwendet werden.

Eine mögliche Implementation - vollkommen verträglich mit der logischen Struktur des Frameworks - wäre die

kundenspezifische Ableitung neuer Klassen die anschließend mit dem ganzen Framework gelinkt werden (analog zum Anlauf 1 im obigen Exkurs). Die Packages die das Framework ausmachen würden dadurch ständig wachsen und sich ändern. Ein Alptraum in Bezug auf Test und Wartung sowie Handhabbarkeit.

Innerhalb der physischen Struktur des Frameworks muß es deshalb Mechanismen geben, die eine Erweiterung des Frameworks in gekapselter Form gestattet, ohne daß das Framework selbst geändert werden muß. Diese physischen Mechanismen stützen sich selbstverständlich selbst wieder auf logische Mechanismen.

Bevor diese Mechanismen jedoch eingebaut werden, muß eine Analyse der Problem Domain festlegen WELCHE TEILE erweiterbar sein sollen.

Faktorisierung der Applikation Domain

Es gibt nicht DAS allgemeingültige Framework. Jedes Framework hat sowohl relativ starre Teile (cold spots) wie auch sehr flexible und austauschbare Teile (hot spots). Eine Analyse der Applikation Domain ist unbedint nötig um herauszufinden, wo Flexibilität nötig ist bzw. wo sie die Kosten erhöht aber wenig bringen wird. In diese Entscheidungen müssen Time To Market Überlegungen genauso eingehen wie Strategische Überlegungen zum zukünftigen Geschäftsbereich und Volumen, kurz um die zukünftige Entwicklung der Firma. Fehler an diesen Stellen lassen sich im Nachhinein kaum mehr ausmerzen. Entweder wird das Framework zu groß (und kommt damit oft auch zu spät) oder es werden wichtige Stellen der Flexibilität übersehen.

Bei einem Framework für Dokumentenbearbeitung und Workflow existieren typischerweise folgende Hot Spots:

- Zugriffe auf Daten (Datenbanken, flache Files, WWW etc.) allg: Entity management
- Unterschiedliche Peripherie (Scanner, Drucker, Bildschirme)
- Verschiedene Operating Systems
- Verteilte Services (remote recognition und image processing, compute services)
- Unterschiedliche Dokumenttypen beliebiger Komplexitat und Form (Texte, Bilder, Video, Audio etc.)

Logische Strukturen der Flexibilität und Erweiterung

Innerhalb der logischen Strukturen lassen sich 2 große Mechanismen der Flexibilität ausmachen:

1. Erzeugung neuer Typen, z.B. von Dokumenten, durch Ableitungen (und deren Austausch über Mechanismen der physischen Struktur)
2. Das Zusammenfügen von bereits vorhandenen Typen (Composite Object Pattern, Lego Prinzip).

Es zeigt sich daß erst die Verbindung beider Konzepte Erweiterbarkeit eines Frameworks schafft. Nicht alles läßt sich durch Komposition erreichen und ohne Ableitung muß Code massiv dupliziert werden. Noch schlimmer: Default- oder Basisklassen legen in einem Framework auch VERHALTEN fest bzw. schreiben es vor. Die Verpflichtungen aus einem Interface/Protokoll können in C++ nicht explizit gemacht werden, d.h. wer eine Methode neu implementiert und damit eine Basismethode überschreibt muß die SEMANTIK des Verhaltens verstehen und NEU implementieren. Dies wird in C++ als das INTERNE Protokoll/Interface bezeichnet und ist gerade in einem Framework das auch für das Verhalten einer Applikation einen Rahmen festlegt von entscheidender Bedeutung.

Wird z.B. die Kindliste eines Parts modifiziert, ruft die Methode der Basisklasse der ContainerParts die Methode Modified() auf, eine Methode der allerersten Basisklasse der Parts. Diese wiederum ruft NotifyObservers() auf, eine Methode der Root Klasse, die dadurch alle Observer dieses Parts von einer wichtigen Änderung unter-

richtet. Damit wird Verhalten festgelegt, auf das Clients vertrauen. Neuimplementierungen eines Parts ohne Ableitung eines Default Parts sollten sich identisch verhalten!!!

Problem: Das Erben von Verhalten und Implementation in C++ bringt auf Seiten der logischen Struktur des Frameworks große Vorteile, auf Seiten der physischen Struktur jedoch compile und link Abhängigkeiten die den Nutzen der gewonnenen Wiederverwendbarkeit in Frage stellen.

Wer im Hinblick auf seine Applikation Domain den Fehler macht und das falsche Prinzip der Felxibilität einsetzt, wird z.B. durch die sog. „**Explosion der Klassen**" bestraft. Bei Dokumenten ist z.B. nur ein Lego Prinzip sinnvoll - die Realisierung neuer Dokumenttypen und Elemente ausschließlich über statisches Ableiten führt in kürzester Zeit zum völligen Chaos.

Erst muß analysiert werden, aus welchen Teilen Dokumente typischerweise bestehen und wie Dokumente strukturiert sind. Anschließend werden über das Composite Object (container/contained) Interface diese Teile zu jeweils neuen Dokumenten dynamisch erzeugt. Tauchen völlig neue Dokumenttypen auf, werden sie aus vorhandenen Typen abgeleitet. Ein Dokument ist NIE EIN Objekt sondern ein Baum aus verschiedenen Elementen. Nur so lässt sich die Vielfalt tatsächlicher Dokumente repräsentieren.

Ein Beispiel: Dokumentenmodell eines Belegverarbeitungssystems im Vergleich zu einem allgemeinen Dokumentenverarbeitungs Systems.

Dokumentenmodell des Belegsystems mit cold spots:

Ein Beleg wird repräsentiert durch:

struct Document {

FieldArray pFieldArray;*

Image_pointer pVorderseitenBild;

Image_pointer pRückSeitenBild;

Image_pointer pBitonalesBild;

Beschreibungs_pointer pBeschreibung;

}

Ein Belegcontainer durch:

struct Stapel {

DocumentArray pDocumentArray;*

}

Dieses Modell ist zugeschnitten auf eine Belegverarbeitung bestimmter Art in Verbindung mit bestimmer Hardware. Wenn dann noch das Speichersystem diese Strukturen fix kennt und darauf ausgerichtet ist, dann entsteht eine sehr schnelle jedoch auf solche Dokumente zugeschnittene Software.

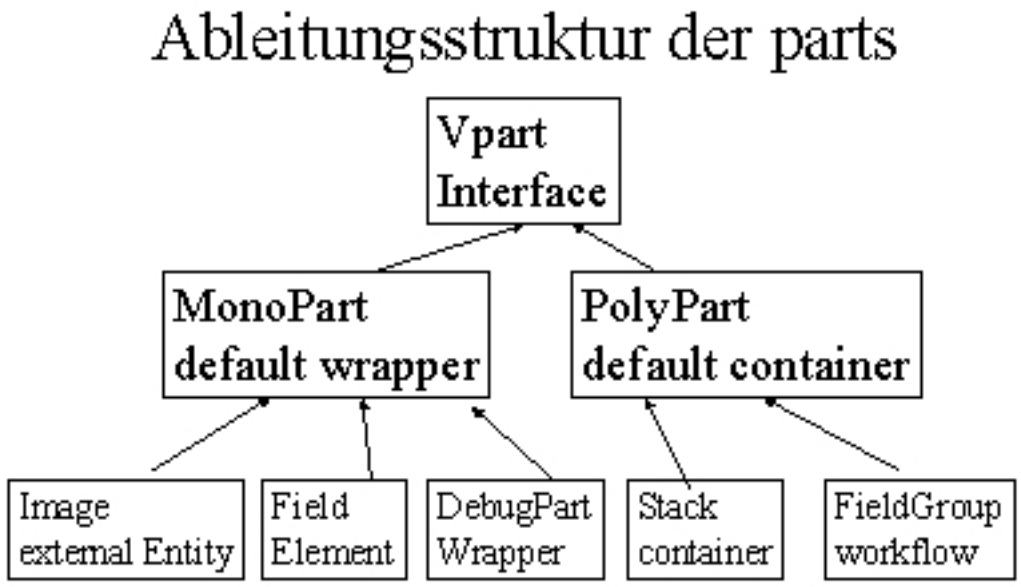
Neue Formulartypen die z.B. aus mehreren Belegen pro Formular bestehen lassen sich in dieses Modell nur schwer integrieren. Wie soll der innere Zusammenhang der Belege pro Formular ausgedrückt werden? In OLDSYS wurde versucht dies durch Kunstgriffe wie z.B. die Einführung von Teilcontainern zu beheben. Diese waren jedoch nicht vorgesehen gewesen und führten zu aufwendiger ad-hoc Programmierung.

([BMEYER97] enthält interessante Zahlen über Typen von Änderungen und damit verbundener Softwareaufwand. Erstaunlich ist, dass Änderungen der Datenstrukturen zwar fast alltäglich sind, die meisten Programmierer jedoch zu programmieren als ob dies der grosse Ausnahmefall wäre.)

Dokumentenmodell des allgemeinen Dokument Processing Systems:

Ein Dokument wird repräsentiert durch einen Graphen aus beliebigen Elementen, genannt parts. Auf nichttextuelle Parts wird über Links zugegriffen, im Dokument ist nur die Strukturinformation dazu vorhanden

Es gibt ContainerParts und MonoParts. Alle Parts teilen eine gemeinsame Schnittstelle und lassen sich zur Laufzeit beliebig verknüpfen. Reichen die vorhandenen Parts nicht aus, werden neue abgeleitet und können jederzeit auch mit spezieller Intelligenz versehen werden. (Internes Traversal Pattern, intelligente Dokumente)



Ein auf dem Composite Object Design Pattern beruhendes Runtime System wirft ein interessantes Problem auf: wie sollen welche Objekte zusammengebaut werden? Irgendwo im System muss dieses Wissen vorhanden sein, am besten nicht hart codiert innerhalb einer Programmiersprache sondern leicht zugänglich und änderbar. Es braucht eine Spezifikationsmethode die mindestens die gleiche Mächtigkeit besitzt wie das Composite Object Pattern, d.h. beliebige Strukturen in Form von Bäumen oder Graphen ausdrücken kann.

Gleichzeitig muss diese Wissen aber auch SICHER verwaltet werden können. Der Aufbau eines Dokumentes muss dort beschrieben sein und Verletzungen dieser Struktur müssen entweder erkannt werden oder dürfen vom Runtime System gar nicht erst zugelassen werden.

Die Standard Generalized Markup Language (SGML) oder die Exended Markup Language (das ist die SGML Variante für das WWW) bieten genau das. SGML wurde in NEWSYS dafür eingesetzt.(zu SGML siehe [WKR96/1], zu XML siehe [KHARE/RIFKIN97]

Statische Definition der Elementstruktur eines konkreten zusammengesetzten Dokumentes: (vereinfachtes SGML).

In Worten beschreibt die Definition folgenden Zusammenhang:

Es wird ein Dokumenttyp „KaugAnträge" (Antrag auf Konkurs-Ausfallgeld) definiert, der sich auf der ersten Ebene in „Anschreiben und eine beliebige Menge von Elementen des Typs Kaugantrag (jedoch mindestens einer) zusammensetzt. Diese Elemente wiederum setzen sich aus weiteren Elementen zusammen usw. Es entsteht ein Baum.

<!Doctype KaugAnträge (Anschreiben, Kaugantrag+)>

<!Element Anschreiben (Firmenbriefkopf,Allgemein,Gruß)>

<!Element KaugAntrag (KaugAntragSeite1, KaugAntragSeite2)>

<!Element KaugAntragSeite1 (PersAngaben,FirmenAngaben,Unterschrift)>

<!Element KaugAntragSeite2 (Gehalt,Zeit,Unterschrift)>

<!Element Firmenbriefkopf .....

<!Element Allgemein ....

<!Element Gruß ....

<!Element PersAngaben (Adresse, Name, Vorname, Alter)>

<!Element Adresse (Stadt, PLZ, Strasse)>

Die Strukturelemente eines Dokumenttyps KaugAnträge werden definiert. Dies macht die logische Elementstruktur dieses Dokumentes aus.

Unabhängig davon gibt es einen Dokumenttype „Beleg" der folgendermaßen strukturiert ist:

<!Doctype Beleg - - (VorderSeitenBild?,RückSeitenBild?,BitonalBild?)>

<!Element (VorderSeitenBild,RückSeitenBild,BitonalBild) (PageSpec)

<!ATTLIST (VorderSeitenBild,RückseitenBild,BitonalBild)

picture ENTITY #IMPLIED

„

„*>*

Der Dokumenttype „Beleg" beschreibt physische Charakteristiken der physischen Formulare hingegen beschreibt der Dokumenttyp KaugAnträge lediglich die logische Struktur von physischen Dokumenten.

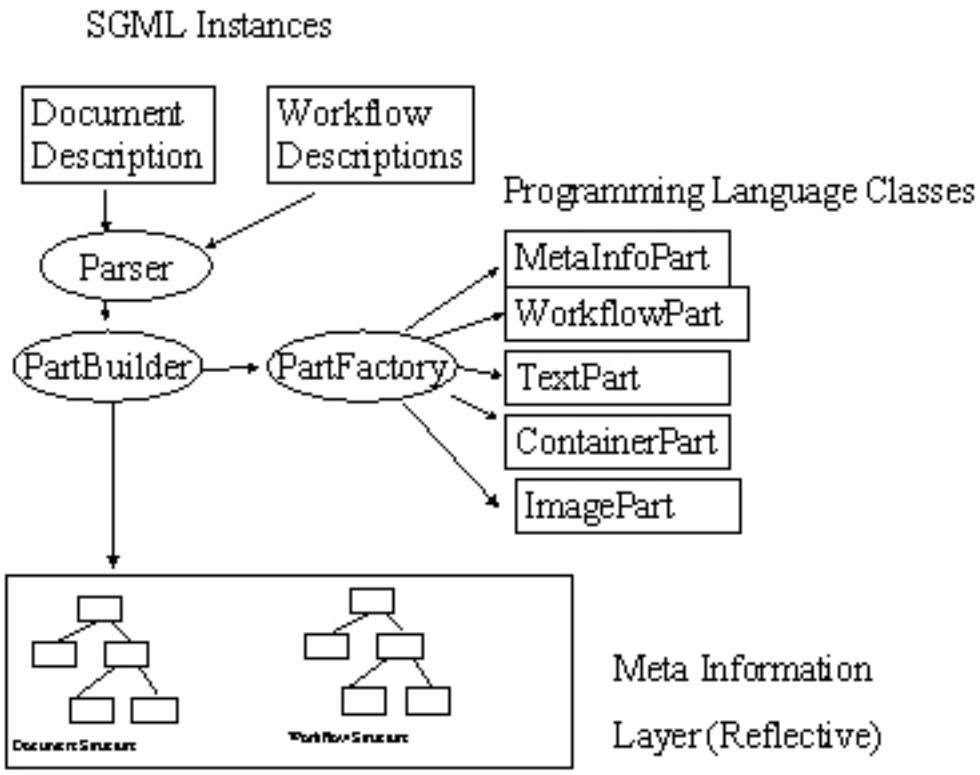
Zusätzlich steht in der Verarbeitungsbeschreibung zu diesem Dokument, daß den Elementen Anschreiben, KaugAntrag1 und KaugAntrag2 jeweils ein Dokumenttyp Beleg zugeordnet ist.

D.H. einfach ausgedrückt, daß der Dokumenttype Kauganträge aus einem Formular Anschreiben mit Vorder und Rückseite (man hätte auch einen nur Vorderseiten-Beleg konfigurieren können) und einer Menge von Anträgen besteht. Jeder Antrag besteht aus 2 Blättern, also 4 Seiten) und daß jeweils Blatt eins und zwei zusammengehören.

Verbinden der logischen Struktur mit echten Klassen eines Frameworks

1) Aufbauen der MetaInformation

Aus diesen Informationen kann dann ein sog. Partbuilder zur Laufzeit aus vorhandenen Parts einen Baum aufbauen, der dieses Dokument Kauganträge durch ein Composite Object aus C++ Part Objekten repräsentiert.

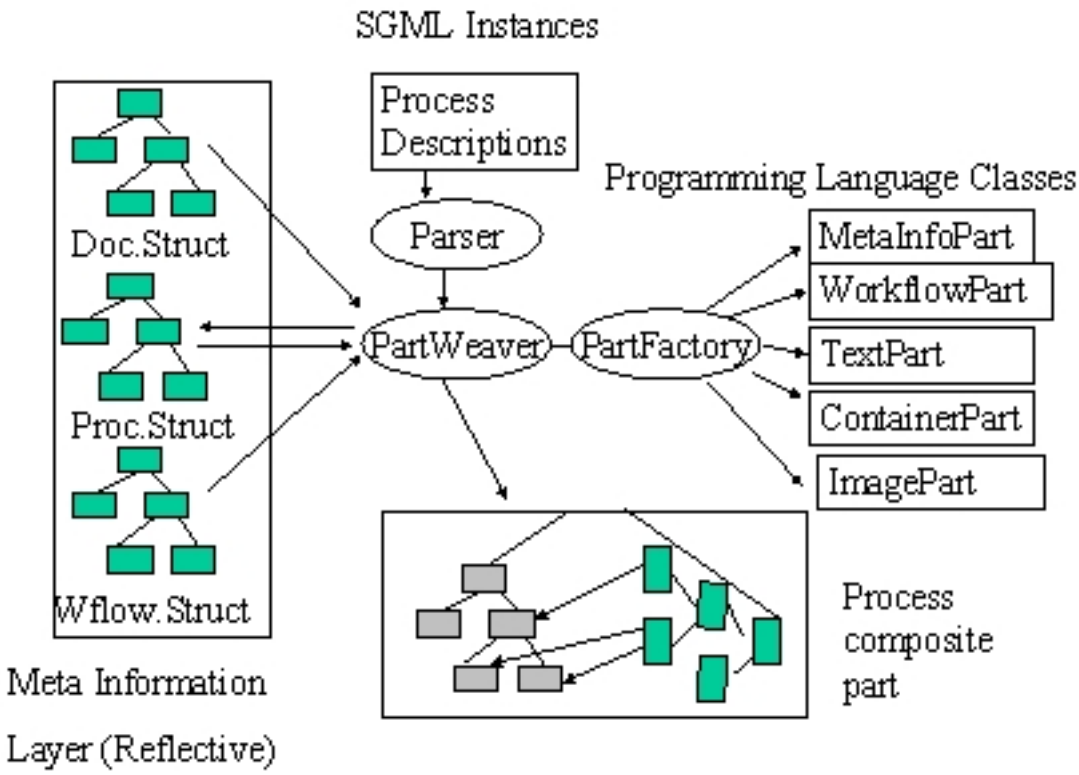


2) Aufbauen eines Process Parts als composite object durch den PartWeaver

Der PartWeaver bildet Meta Information in Klassen der Programmiersprache ab und verknüpft sie unter den verschiedenen Aspekten die in der Metainformation gespeichert sind (Dokumentbeschreibungen, Validierungen, Workflow Beschreibungen, Prozessbeschreibungen etc).

In Wirklichkeit sind Workflow Elements und Document Elements in verschiedenen Graphen (Aspekten) wobei Workflow-Elemente Referenzen auf Document-Elemente haben, jedoch nicht umgekehrt (dies würde zu viel Verarbeitungslogik in die Dokumente bringen) Der Client kann je nach Anwendung das Composite Object als Black Box betrachten oder Informationen über seine Struktur erlangen.

PartBuilder und PartWeaver sind nur Implementationen von SGMLApp Interface und sind ihrerseits in eine Builder Factory enthalten.

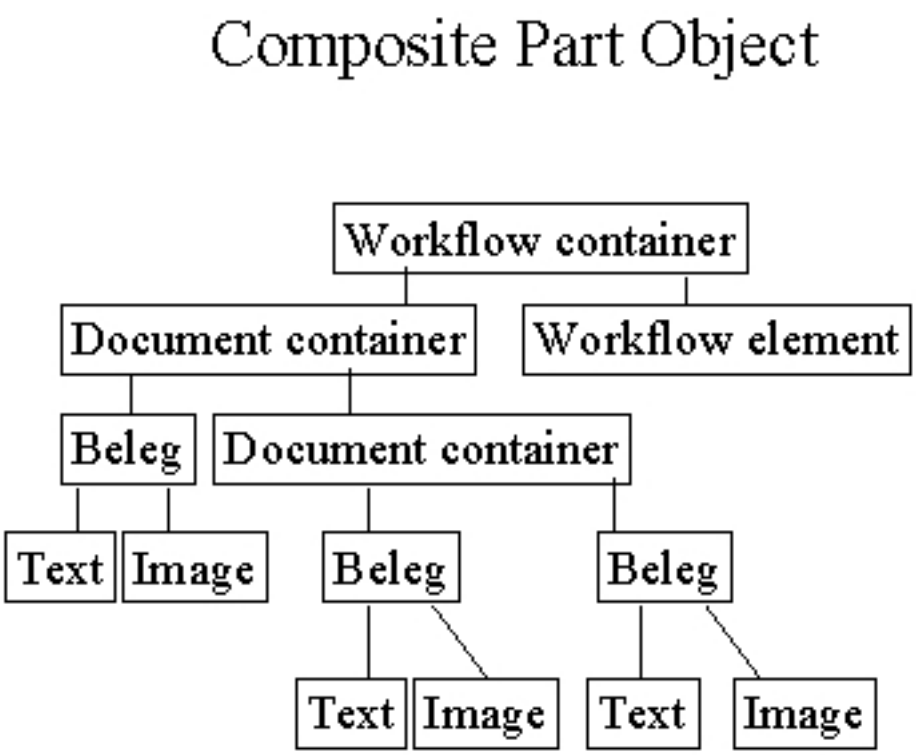


Zur Laufzeit kann dann dieser Graph durchgegangen werden und auf seine Validität geprüft werden. Selbstverständlich muß auch das Speichersystem diese Flexibilität besitzen, wie sie z.B. für Bento (Grundlage von OpenDoc) oder Structured-Storage (MFC) typisch ist.

Physische Charakteristiken und Workflow Information wird ebenfalls zur Laufzeit in diesem Baum integriert. Diese Information ist ebenfalls wieder in Form von Dokumenten gehalten. (Selbstbezügliches Prinzip)

Laufzeit Struktur des zusammengesetzten Dokumentes als directed acyclic graph (DAG)

Hier handelt es sich um die „flache" Sicht. Clients können diesen DAG extern traversieren und manipulationen an seiner Struktur vornehmen, z.B. Workflow Elemente einfügen oder entfernen. Typische Clients sind Dokument oder Workflow Editoren.



Nicht sichtbar in diesem chart sind die Links von Workflow Elementen zu documents und ihren Teilen. Aus Beschreibungen der Documente und des Workflows (in SGML) werden zur Laufzeit zueinander orthogonal stehende Graphen aus Implementationen von Parts. Clients die auf dieser komplexeren Struktur arbeiten wollen verwenden Methoden des Part Interfaces die ein INTERNES Traversieren ermöglichen, d.h. sie behandeln das ganze Composite Object (das in seiner Zusmmensetzung einem Dokumentenspezifischen Workflow entspricht) als „black box" und sprechen im wesentlichen mit dem top part des compositums. Interne Struktur des Compositums bleibt ihnen verborgen (macht clients unabhängig) und das Compositum selbst kann eigene Intelligenz einsetzen um z.B. bestimmte Prüfungen durchzuführen oder einen gewissen workflow zu starten. Erst in diesem Modus wirkt sich die Kapselung von Intelligenz und Struktur im composite object design pattern voll aus.

Typische Clients sind hier branchenspezifische Editoren die einen Workflow voraussetzen, z.B. die Bearbeitung von kritischen Dokumenten in Banken (Schecks, Einzahlungen etc.) Hier kann der User nicht frei editieren sondern muss durch einen validierenden Workflow geführt werden.

Z.B. werden die Workflow-Parts Änderungen an der Struktur des Dokuments (sie erhalten natürlich Nachricht von solchen) nicht zulassen. Der branchenspezifische Editor wird die Workflow-Elemente des Kompositums überhaupt nicht anzeigen, da sie für den Bearbeiter in diesem Arbeitsschritt nicht änderbar sind.

Sämtliche Parts sind aus der Konfiguration heraus mit Workflow (Prüf) Information versehen, sei es durch Einfügen von Prüfobjekten oder durch Aktivieren eigener Methoden. Der Status des obersten Dokument-Container Parts wird z.B. erst dann auf gut gehen, wenn die Stati seiner Kinder gut sind.

Browser und Editoren können jetzt auf diesem Graphen arbeiten und Änderungen vornehmen, deren Gültigkeit jedoch von den beteiligten Parts intern geprüft werden.

Über den Beleg-Part erhalten die Browser und Editoren die Bilder und können diese manipulieren. Die Graphstruktur erlaubt das Bearbeiten beliebiger Dokumente.

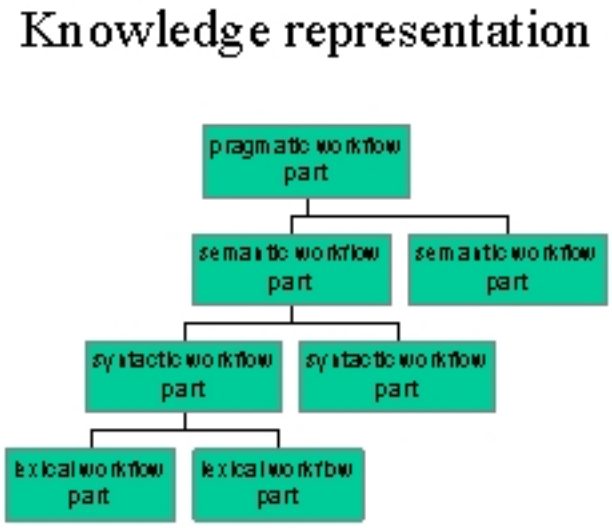
Dies war ein Beispiel für Erweiterung und Flexibilität durch Zusammensetzung. Bei anderen Punkten wird man sich für die Erweiterung durch Ableitung entscheiden. Hier entscheiden die Mechanismen der physischen Struktur, ob eine Änderung des Frameworks selbst (durch compilieren, linken oder dynamisches Einbinden) erfolgt oder ob die Erweiterung in einem Branchen Package landet, wo sich wiederum die Frage des compilieren, linken oder dynamisch Einbinden stellt.

Anmerkung zu orthogonalen Graphen

Bei dem geschilderten Composite Object Design Pattern mit externem und internem traversieren handelt es sich um eine Art Meta-Pattern für Frameworks. Fresco und (M)ET++ verwenden z.B. dieselbe Architektur für andere Zwecke.

NEWSYS spezifisch ist das Konzept der othogonal verlinkten Graphen mit gleichzeitiger aktiver Verbindung zum User Interface. D.h. zu jedem Zeitpunkt konnten entweder vom User oder von der internen Workflow Logikänderungen am Dokument vorgenommen werden. Die Graphen waren dabei verantwortlich für die ununterbrochene Validität des Dokumentes. Dokument Views wurden durch update Methoden des Observer Patterns benachrichtigt. Graphinterne und Intergraph- Kommunication lief über Messages.

Die einzelnen Workflow-Parts waren je nach ihrer speziellen Zielrichtung in unterschiedlicher Höhe des Graphen untergebracht:



Diese Hierarchie spiegelt die Aufteilung des Gegenstandsbereiches „gescanntes Dokument", dessen Validität aus den vier Kriterien besteht:

- Pragmatik: z.B. Vier Augen Prinzip bei Banken
- Semantik: z.B. Zusammenhang zwischen Bankleitzahl,Kontonummer u. Adresse
- Syntaktik z.B. Hat Kontonummer das richtige Format?
- Lexikalik z.B. Hat die OCR Erkennungsfehler signalisiert?

Übergeordnete Parts haben damit die Möglichkeit externes Wissen zur Korrektur unterer Parts einzusetzen bzw. Zusammenhänge zu erkennen. Neben der Kommunikation durch Messages konnten Parts Informationen in einem Dokumentglobalen Namespace deponieren bzw. sich dort für bestimmte Informationstypen registrieren lassen. Hat ein Part eine Information deponiert, bekamen die registrierten Parts eine Update meldung. Dieser Mechanismus (ähnlich dem CORBA naming service) ist gerade in C++ frameworks wichtig, da er die dauernde Anpassung von Methoden durch zusätzliche Parameter unnötig macht.

Physische Strukturen der Flexibilität und Erweiterung

Erweiterungen der physischen Struktur: Neue Parts

Ein Beispiel zur Erzeugung neuer Typen von Parts:

An diesem Beispiel soll auch der Unterschied zwischen Framework Kernel und Branchenspezifischen Erweiterungen diskutiert werden.

Es soll ein neuer Typ von Part gebaut werden, da die vorhandenen diese spezielle Funktionalität nicht leisten bzw. diese durch neue Prüfobjekte (== Intelligenz des Parts) alleine nicht erreicht werden kann.

(Dies behandelt auch den Fall eines nötigen Bugfixes bei einem vorhandenen Part)

1. Frage: Wie allgemeingültig ist der Part?

Grundsätzlich unterscheidet die physische Struktur von NEWSYS 3 Ebenen der Allgemeingültigkeit:

- Für fast alle Anwendungen nützlich
- Für bestimmte Branchen nützlich
- Für einen bestimmten Kunden nützlich

Fall 1: Der Part ist allgemein verwendbar.

Jetzt stellt sich das Problem, wie der neue Part in die physische Struktur des Frameworks eingebaut werden soll.

Möglichkeit 1: Erweiterung des Default Part Packages das Teil des physischen Kernels ist

Vorgehen: Ableiten, kompilieren, linken mit den vorhandenen Objekten dieses Packages und eine neue dll erstellen.

Nachteil: Die alte Dll musste angefaßt werden, Dokumentation und Service,Wartung sind betroffen. Eine neue umfangreiche Liefereinheit ist entstanden die aus bereits getesteten und einem neuen Teil besteht. Der neue Teil kann nur in dieser DLL getestet werden.

Möglichkeit 2: Der Part kommt in eine eigene DLL. Das Implementation Repository wird um einen Eintrage für diesen Part in dieser DLL erweitert. Die Part Factory der PDC sucht beim Booten nach Partdlls und verknüpft sie über die Append(PartFactory_ptr) Methode.

Wenn ein Client den neuen Part benötigt, führt die PartFactory ein Chain of responsibility Pattern durch und lädt den neuen Part.

Vorteil: die alte dll bleibt unberührt und die vorhandenen Objekte können dynamisch vermehrt werden. Auf die gleich Weise lassen sich Bugfixes als Ersetzen von Objekten durchführen, ohne daß vorhandene DLLs berührt werden.

Dieses Schema ließe sich sogar sehr einfach um ein Dynamisches Nachladen BEI LAUDENDEM SYSTEM erweitern. Dazu brauchte der NSys ORB lediglich einen Thread, der auf einem Socket auf Nachrichten wartet. Ein kleines Programm (analog dem ORBIX Putit) nimmt als Argument den Namen der neuen Dll und gibt textuelle Information darüber über den Socket an den ORB Thread. Dieser läßt über den Loader das Objekt laden und führt im Falle einer Factory den Append(Factory_ptr) bei der bereits geladenen Default Factory durch. Alternativ könnte man das Implementation Repository editieren und über das Hilfprogramm den ORB Thread zu einem rescan des Repositories auffordern.

Fall 2: Der Part ist branchenspezifisch

Sämtliche bei Fall 1 geschilderten Möglichkeiten stehen auch hier offen. Lediglich das Erweitern des Default Part Kits im Kernel sollte unterbleiben. Stattdessen kann eine branchenspezifische DLL erweitert werden bzw. eine zusätzliche erstellt werden.

Fall 3: Der Part ist kundenspezifisch

Analog zu Fall 2, nur daß das Erweitern branchenspezifischer oder default dlls unterbleiben sollte. Stattdessen kann eine kundenspezifische Dll erweitert bzw. eine zusätzliche erstellt werden.

Einführung von MetaObject Protokollen (Reflection Design Pattern)

Im Laufe der Entwicklung von NEWSYS wurde das SGML Sub-framework immer stärker für die validierte Erzeugung und Verwaltung von Metainformationen eingesetzt. Zielpunkte der Metainformation waren:

1. Erweiterung oder Änderung von Basis Funktionalität (z.B. über Einträge im Implementation Repository)
2. Nutzung der Metainformation für generisches Processing (dynamische Views, dynamisches Erzeugen und Auswerten von Datenbanken etc.)
3. Deskriptive Applikation Domain Languages
4. Automatische Validierung von Abläufen

Abänderung von Basis Funktionalität

Eine wesentliche Voraussetzung von Flexibilität ist die Möglichkeit, über ein Reflection API Zugriff auf interne Mechanismen des Frameworks zu erhalten und diese erweitern bzw. abändern zu können. Jedes Framework enthält noch sog. „cold spots" die mit den normalen Mitteln (Ableitung oder Composition) nicht leicht zu ändern sind.

Ein flexibles Framework wird auch Dinge wie Collaboration von Klassen zur Erzeugung eines Ablaufes in Klassen ausdrücken und zugänglich machen.

Ziel ist hier die „Open Implementation" zu erreichen.

Schöner noch wäre es, die dahinterliegenden Design Patterns auch programmatisch fassen zu können. Die UML1.0 bietet jetzt die Möglichkeit z.B. solche Design Patterns als „parametrisierbare Collaboration" Klasse zu fassen.[UML1.0]

Nutzung von Meta Information für generisches Processing

Ein reflektives Moment des NEWSYS Frameworks ist die symbolische Repräsentation der Applikationsinformation in SGML Dokumenten statt im Source Code. Alle Teile des Frameworks können Metainformationen über Dokumente aus Dokumenten gewinnen und daraus z.B. einen speziellen View aufbauen oder ein Datenbank Schema generieren. Symbolische Requests verwenden diese Metainformation zum Mappen von Tokens und Views auf Values.

Dass dies möglich ist, zeigt nicht zuletzt auch das von Christoph Sutter entwickelte FISH, flexibles Infomationssystem für Hypermedia, [SUTTER97].

Unabhängigkeit von Implementationen durch Descriptive Application Domain Languages

Im NEWSYS Framework werden Dokumentbeschreibungen, Workflow Spezifikationen etc. häufig als „Konfiguration" bezeichnet. Dies ist jedoch ein historisch bedingtes Erbstück aus dem OLDSYS Produkt. In Wirklichkeit handelt es sich um Instanzen kleiner, rein descriptiver Applikation Domain Languages. Die Erstellung einer SGML Document Type Description ist ähnlich der Erstellung einer Applikation Domain Language.

Diese Languages bilden einen Aspekt der Applikation Domain ab und zwar unabhängig vom Runtime System rein deskriptiv.

Beispiel: Archivierung von Dokument und Prüfprozeduren

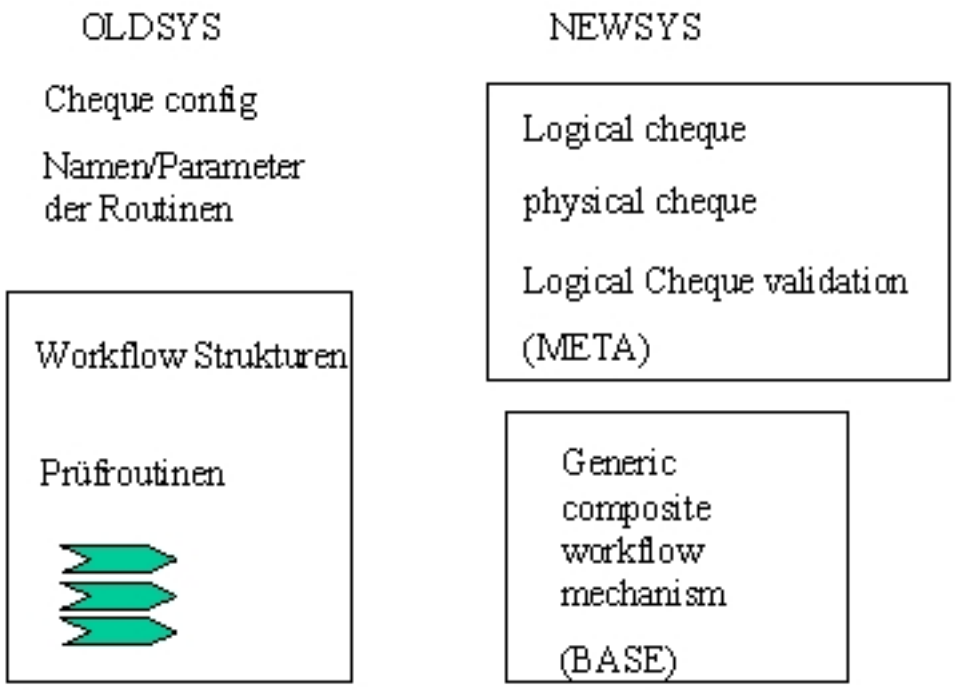
OLDSYS enthielt einen Mechanismus, um die binären Prüfprozeduren zusammen mit den Dokumenten zu speichern. In der Praxis wurde dieser Mechanismus aus verschiedenen Gründen nie benutzt. Einer davon war

sicher, dass die Wahrscheinlichkeit, dass ein OLDSYS Runtime System in 2 Jahren noch die binären Prozeduren würde ablaufen lassen können, wohl als sehr gering angesehen wurden.

Java Programmierer werden an dieser Stelle die Plattformunabhängigkeit von Java hervorheben sowie die physische Flexibilität des dynamischen Linkens. Demgegenüber lässt sich nur hervorheben, dass typische Archivierungszeiten z.B. in der Versicherungsbranche über 100 Jahre sein können!

Mit NEWSYS konnten die Prüfprozeduren DESKRIPTIV erstellt werden. Über mapping Documente wurden sie mit programmatischen Objekten des Frameworks verknüpft. Die Konsequenz war, dass ein Workflow ganz anders implementiert werden konnte, die Spezifikation des Workflows davon aber unberührt blieb.

Dies war gleichzeitig einer der Punkte die zum dauernden Konflikt mit den Mitgliedern des OLDSYS Teams führten. Das Bemühen des Framework Teams zu logischen Beschreibungen der Validität eines Dokumentes zu gelangen, um Freiräume für die Implementation zu erhalten stiess auf völliges Unverständnis. Für OLDSYS Entwickler sass die Logik eines Dokumentes im Source Code, nicht in abstrakten Beschreibungen.



Validierung von Implementationen durch Metainformation

Von der Implementation unabhängige Beschreibungen der Struktur von Dokumenten und der Logik ihrer Bearbeitung sind sicher aus einem Wartungs- und Flexibilitätsgedanken heraus wichtig.

Der wirkliche Vorteil dieses Verfahrens liegt jedoch in der plötzlich möglichen automatisierten Überprüfbarkeit der Implementationen.

Wenn die Logik einer Bearbeitung wie in OLDSYS im Source Code versteckt ist, dann sind die Kriterien für eine Prüfung der Logik entweder im Kopf des Entwicklers oder in Dokumenten zur Prüflogik versteckt und damit für eine automatische Prüfung der Implementation nicht zugänglich.

Beispiel: automatische Prüfung der Validierung von Dokumenten

Voraussetzungen:

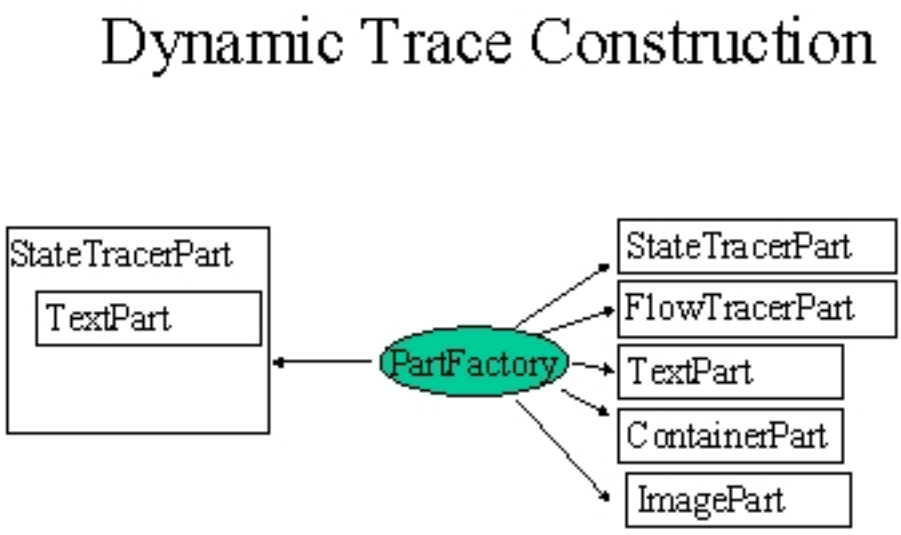
1. Spezifikation der Semantik eines Dokuments als MetaInformation

2. Runtime Support zur Informationsgewinnung

Ad 1) Die Kriterien der Semantik eines Dokuments werden in einem MetaDokument erstellt. Dies kann die Form einer State Tabelle haben in der der Entwickler für jeden Bearbeitungsschritt legale Zustände des **logischen** Dokumentstatus definiert.

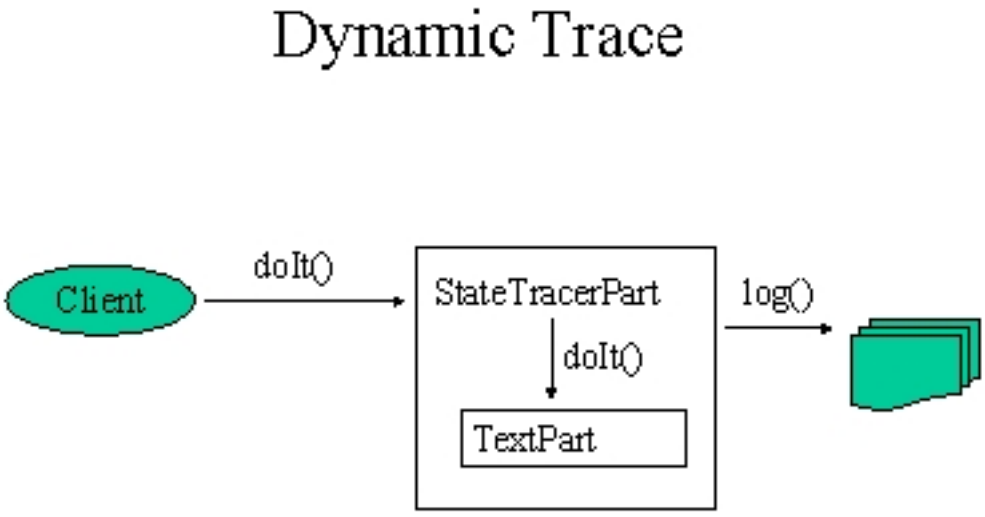
Ad 2) Das Problem der Debug Information ist typischerweise, dass sie aus Performance Gründen nur während der Entwicklung im Source Code aktiviert ist und damit nicht einfach durch eine Option eingeschaltet werden kann oder dass die falschen Daten ermittelt wurden, die gerade für den momentanen Fehlerfall nicht aussagekräftig sind.

Der Einsatz des Wrapper Design patterns löst beide Probleme. Eine Wrapper Klasse besitzt das selbe Interface wie die zu wrappende Klasse, d.h. in diesem Fall wie die Dokument und Workflow Parts. Zur Konstruktion Zeit entdeckt z.B. eine Partfactory, dass für den verlangten Part ein Wrapper Part definiert ist.(d.h. DYNAMISCH!)



Sie erzeugt beide Parts und fügt den zu wrappenden Part in den Wrapper Part ein. Ab diesem Moment ist die Tatsache, dass hier 2 Objekte existieren für Clients nicht mehr sichtbar.

Der Wrapper Part empfängt automatisch alle Methodenaufrufe an den eingeschlossenen Part und leitet sie an den Part weiter. Davor und danach hat er die Möglichkeit Ergebnisse zu protokollieren (die er z.B. vor und nach dem Methodenaufruf vom eingeschlossenen Part abgefragt hat). Damit kann z.B. die Statusänderung eines Parts verursacht durch einen Methodenaufruf protokolliert werden.



Das entstandene Protokoll kann jetzt durch ein Tool aufbereitet werden, z.B. kann geprüft werden, ob zu irgeneinem Zeitpunkt ein Partstatus nicht mit dem in der MetaInformation spezifizierten Status übereinstimmt.

Das Ergebnis ist ein validierter Workflow, z.B. Firma X kann einem Kunden aus dem Bankenbereich nachweisen, dass ihr Workflow zur Scheckbearbeitung den rechtlichen Richtlinien der Scheckverarbeitung genügt.

Natürlich lässt sich mit dem Wrapper Design Pattern auch ein rein technisch orientiertes Debugging in einem Framework durchführen. Wrapper Parts sind ebenfalls über das Implementation Repository erzeugbar, d.h. sie sind ohne physische Konsequenzen austauschbar und neue können hinzugefügt werden.

Wrapper Parts können natürlich auch ausserhalb eines konkreten Debuggings als Trigger auf bestimmte Fehlerbedingungen eingesetzt werden (Watchdog) ohne dass der Code von existierenden Parts geändert werden muss.

Chapter 7. DIE SOURCE STRUKTUR DES NEWSYS FRAMEWORKS

Die Grundstruktur wurde bereits im Kapitel über die Erweiterbarkeit angesprochen. Besonders die physische Struktur ähnelt der Source Struktur in ihrer Aufteilung auf die „Profiles" Kernel, Branchen und Customer. (Übrigens finden sich ähnliche Aufteilungen auch bei CORBA)

Der Framework Kernel

Er beinhaltet neben den Interface Definitions für alle Domains auch Default Implementationen von Klassen die einerseits für das Framework selbst nötig sind als auch zur Ableitung zur Verfügung stehen und ein default Verhalten implementieren.

Kernel/Base:

ein Beispiel:

Interface: SyBVPart.hpp (interface directory)

erste Default Implementation: SyBBPart(hpp,cpp)

weitere Default Implementationen: SyBBMonopart (= Contained)

SyBBPolypart (== Container)

SyBBDebug (DebugWrapper für Parts)

Kernel/PDC

Packages dieser Domain leiten von Base Implementationen und Interfaces ab, z.B. spezielle Dokument Parts wie Pages, Stack, Field, Fieldgroup etc.

Kernel/DMC

Getreu dem Prinzip zusammengesetzter Dokumente realisiert die DMC Domain Zugriffe auf persistente Objekte ebenfalls durch Ableitungen der Part Klasse, z.B.

StorageManager, StorageObjekt (PolyParts == Container)

Translator (Monopart = Wrapper)

Kernel/HIC

Entsprechend dem jeweiligen zusammengesetzten Dokument (Graph von verschiedenen Parts) baut die HIC mittels Context und Strategy Objekten den jeweils gewünschten View dynamisch auf. Context kapseln low level GUI Funktionalität und liefern bereits erste semantische Ergebnisse an die jeweilige Strategy. Die Strategy verknüpft verschiedene Contexte (z.B. ImageKontext, Fieldkontexte, PopupMenuContext) zu einer komplexen Verarbeitung wie z.B. Nacharbeiten.

Branchenspezifische Sourcen

unter NEWSYS/Branches/..... finden sich Implementationen, die nur für bestimmte Branchenanwendungen sinnvoll sind.

Packages branchenspezifischer Domains haben drei Möglichkeiten Kernel Packages zu nutzen:

1. Sie können von Kernel Packages ableiten. Dadurch entsteht eine Link und Compile Abhängigkeit von den Kernel Packages bei gleichzeitigem maximalen Reuse von Implementation Code

2. Sie können von einer Interface Klasse ableiten und das Protokoll komplett neu implementieren. Diese Lösung bringt maximale Unabhängigkeit bei minimalem Reuse von Implementation Code.

3. Sie können von einer Interface Klasse ableiten aber aus einer Factory eine Base Implementation bekommen und die Teile der Implementation an diese Delegieren. Diese Lösung kombiniert Unabhängigkeit (keine Compile/Link Abhängigkeiten ausser bei Interface changes) mit Reuse des Implementation Codes durch Delegation.

Die drei Möglichkeiten des Package übergreifenden Implementation Reuse gelten für alle Extensions. Aus Wartungsgründen sind nur 2) und 3) empfehlenswert.

Systemspezifische Sourcen

Der Begriff „Systemspezifisch" bzw. „ betriebssystemspezifisch" ist doppeldeutig. Er bezeichnet einmal Funktionalitäten einer LOGISCHEN Domain System, ähnlich der Base Domain. Ein Beispiel dafür sind Transport Endpunkte wie Sockets, Named Pipes, Message Queues etc.

Zum anderen bezeichnet er Funktionalitäten die vielleicht auf verschiedenen Plattformen verschieden implementiert werden müssen, OBWOHL die Funktionalitäten selbst keineswegs zur logischen Domain SYSTEM gehören. Mit „verschieden implementiert" ist gemeint, daß die Art und Weise der Implementation SO SEHR UNTERSCHIEDLICH IST, daß kein gemeinsames Source File sinnvoll ist wenn man nicht einen Wald aus „#ifdefs" einführen will. Aus dieser Doppeldeutigkeit entstand einige Konfusion bezüglich der physichen Teile die unter NEWSYS/OS gehören. Deshalb wiederholt sich die allgemeine Domainstruktur (Base, Dmc, Pdc, HIC unter NEWSYS/OS.

Innerhalb des NEWSYS Frameworks werden betriebssystemspezifische Klassen und Funktionen auf folgende Weise intergriert:

Zunächst wird für den gewünschten Service ein Interface in Form einer V Klasse definiert. Dieses Interface befindet sich im interface directory der jeweiligen Domain. Die Implementationen des Services befinden sich in den Directories w32 und os2 unter NEWSYS/os. Für den Framework Kernel ist die Tatsache unsichtbar, daß eine Objektimplementation Betriebssystem spezifisch ist. Die Generiergungsstruktur (imake, cvs) stellt sicher, daß die entsprechenden Packages für jede Plattform erstellt werden.

Historische Besonderheit: OLDSYS Kompatibilitätskomponenten die ursprünglich unter MS-DOS entwickelt wurden beinhalten plattformabhängige Teile. In einem ersten Ansatz (der falsch war) wurden diese Teile unter NEWSYS/OS/os2 eingehängt. Dies ist jedoch aus 2 Gründen unsinnig: Erstens sind nur sehr wenige Teile darin plattformabhängig die sich leicht mit einigen Macros beseitigen lassen. Zweitens gehören sie NICHT zur SYSTEM Domain.

Mittlerweile sind diese Teile mit wenigen Macros plattformunabhängig gemacht worden, womit jeglicher Grund für ihre Existenz unter NEWSYS/OS weggefallen ist. Sie sollten unter Kernel/DMC eingehängt werden.

Regeln für Entities unter NEWSYS/OS:

Wenn ein Service System oder Base Charakter hat ist er ein Kandidat für die logische Domain BASE, sein Interface gehört unter Kernel/Base/intinc. Kann er mit GERINGEM Aufwand plattformunabhängig in einem Source File gemacht werden, sollte seine Implementation auch wirklich in einem File zusammengehalten werden. Wo soll dieser Service physisch in die Source Struktur eingehängt werden?

Die dauernden Erweiterungen zentraler Packages wie z.B. BaseKern.dll ist schädlich. Besser ist es, die Implementation unter NEWSYS/Os/common/Base einzuhängen. Eventuell sollte das Sysutil Kit erweitert werden oder der Service bekommt sein eigenes Package.

Wenn ein Service KEINEN System oder Base Charakter hat aber dennoch völlig unterschiedlich implementiert werden muß: Sein Interface kommt unter das interface directory der jeweiligen logischen Domain. Seine Implementation kommt unter NEWSYS/OS/$(Architecture)/Domain/implementations für die jeweils unterstützten Architekturen, wobei Architecture für UNIX ¦ OS2 | W32 steht.

Solange es geht sollte man COMMON SOURCE CODE anstreben. Dies vereinfacht die Wartung erheblich!!!

Da das Laden von Komponenten über das Implementation Repository symbolisch und dynamisch erfolgt, ist es dem Framework egal, wieviele Packages existieren bzw. welche Services in welchen Packages stecken. Das Generiertool sorgt bei neuen include Pfaden ohnehin dafür, daß mit einer einzigen zentralen config/cf/xxx Änderung alle Projekte automatisch updatebar sind.

Chapter 8. DIE GENERIERUNGSSTRUKTUR DES NEWSYS FRAMEWORKS

Ein plattformunabhängiger multiuser Entwicklungsprozess eines Frameworks sowie dessen Installation muss sich auf einen Generierungsprozess stützen der die folgenden Kriterien erfüllt:

1. DEFINIERT
2. QUALITÄTSGESICHERT
3. AUTOMATISIERT
4. MULTI-USER FÄHIG
5. FLEXIBEL
6. PLATTFORMUNABHÄNGIG

Im NEWSYS Framework stellt kein Programmierer irgendwelche Compiler/Linker options, Libraries oder Include Pfade ein. Alle Options sind in automatisch für ihn generierten Makefiles enthalten.

Jeder Programmierer kann mit einem Kommando ALLE oder einzelne Komponenten erzeugen, auch diejenigen die er nicht selbst geschrieben hat.

Das Problem integrierter Entwicklungsumgebungen

Eine Erfüllung der obigen 6 Kriterien war mit den zur Verfügung stehenden Tools (IBM C-set, Visual Age, Visual C++) nicht möglich. In einem selbst erlebten Fall dauerte es 1 Woche bis ein bestimmtes Release eines Prototypen von einem anderen Mitarbeiter erzeugt werden konnte. Er musste dazu ca. 30 Dlls erzeugen sowie Konfigurationen anpassen deren überwiegende Zahl für ihn völlig bedeutungslos war.

Detailkritik:

Ad 1) Definiertheit der Umgebung

Basis eines jeden Produktionsprozesses ist die Definition von Sources, Werkzeugen, Abläufen, Ergebnissen und Installationen. Diese Definition hat an EINER Stelle zu erfolgen. „Private" lokale Definitionen zerstören den einheitlichen Produktionsprozess. Die genannten IDEs lassen eine globale Definition des Produktionsprozesses (unter Berücksichtigung lokaler Gegebenheiten wie abweichende Drives oder Volumes) nicht zu. Z.B. war es mit dem IBM Workframe nicht möglich, bestimmte Defines für alle Entwickler auf allen Maschinen zu setzen.

Ad 2) Qualitätsgesicherte Produktion

Ein qualitätsgesicherter Produktionsprozess setzt die Definiertheit (siehe 1) und automation voraus. Alle betrachteten Tools zwangen zu interaktiven Anpassungen und Eingaben. Inkompatibilitäten der Module durch falsche Optionen oder Tools waren die Folge. Es war implizites Detailwissen zur Produktion nötig.

AD 3) Automation des Produktionsprozesses

Ein Produktionsprozess darf keine interaktiven Eingaben enthalten aus 2 Gründen: Um Fehler zu vermeiden und Zeit zu sparen. Es ist nett wenn man durch GUIs Optionen wählen KANN. Wenn fortwährend Optionen gewählt werden MÜSSEN (z.B. beim Aufsetzen eines neuen Projektes im gleichen Buildprozess) entsteht ein Wartungsalptraum.

GUIs sind nicht die richtige Lösung für einen Produktionsprozess

Wenn z.B. der Produktionsprozess definiert an einer Stelle geändert wird, muss EIN Kommando genügen um ALLE Projekte die sich lokal auf Maschinen befinden automatisch andzupassen.

Nicht nur Projects, auch die insgesammt nötigen Tools müssen automatisch installiert werden.

Ad 4) Multi – user fähig

Keine der genannten IDEs unterstützt Multi-User Entwicklungen. Jeder Entwickler setzt GUI gestützt seine eigene Entwicklungsumgebung auf. Es gibt keinen Automatismus zum erzeugen fremder Module auf der eigenen Maschine.

Ad 5) Flexibel

Grössere Entwicklungen benötigen immer wieder spezielle Schritte bei der Generierung, teilweise auch als Workaround um Limits der IDEs (Anzahl Files, Ablauf Definition nur durch File Extensions bestimmbar, Environment Grösse zu knapp etc.). Plattformunabhängigkeit zwingt ebenfalls zu grosser Flexibilität. Z.B. musste der Build auf Microsoft Plattformen plötzlich den Package Namen über ein CC – define in den Source Code reichen. Ohne ein flexibles Tool bedeutet dies ein Anpassen aller Projekte!

Konvertierungen und anderes Processing muss sich an jeder Stelle des Produktionsprozesses enfügen lassen, ohne dass alle Projekte geändert werden müssen.

Lokale Besonderheiten auf Maschinen dürfen keine Rolle spielen (IBM Workframe enthielt z.b. die Source Pfade ABSOLUT!

Alle Definitionen müssen SYMBOLISCH sein. Sinnvolle Default Mappings zu harten Pfaden etc. existieren, können aber lokal überschrieben sein.

Der Produktionsprozess muss in der Lage sein verschieden Releases oder Modes (Profiled, Debugged, Produktion) auf EIN Kommando hin zu erzeugen, OHNE dass lokale Einstellungen bei Projekten erfolgen. Dies wiederum setzt eine definierte Source und Installationsstruktur voraus die es erlaubt, verschiedene Versionen oder Modes von Modulen GLEICHZEITIG zu verwalten.

Ad 6) Plattformunabhängig

Bei Microsoft ohnehin kein Thema. IBM Tools längst noch nicht auf allen Plattformen obwohl ein Weg dorthin sichtbar ist. Ein besonderes Problem stellen kleine Tools und Scripts dar, wenn sie beim Produktionsprozess eingesetzt werden. Die GNU Utilities helfen hier plattformübergreifen. Wichtig ist die verwendete Script Sprache. Sie sollte ebenfalls plattform unabängig sein, was z.B. bei Rex (visual rexx etc.) nicht unbedingt der Fall ist bzw. war.

IDEs sind typischerweise zugeschnitten auf einen einzelnen Entwickler. Wenn z.B. das Framework oder die Applikation im Laufe der Zeit immer modularer wird (was eigentlich gut ist) steigt gleichzeitig der Aufwand bei der Übersetzung und Installation. 50 Services in dynamik link libraries oder exe files bedeuteten z.B. beim IBM Workframe 50 verschiedene Projekte mit harten Source Pfaden. Mal genommen mit der Anzahl der Entwickler kann man sich schnell ausrechnen, dass die Wahrscheinlichkeit dass alle Optionen auf allen Maschinen bei allen Entwicklern gleich sind, gleich NULL ist. Obskure Runtime Probleme sind die Folge.

Die Entwicklungsumgebung für Frameworks oder grössere Applikationen muss auf Knopfdruck automatisch installiert werden können. Die einzelnen Projektfiles gehören zum Source Code. Selbst bei der ohnehin zu empfehlenden Trennung von Interface und Implementationsklassen bleibt ein Rest von compile oder linktime Abhängigkeiten.

Letzten Endes waren wir bei NEWSYS gezwungen auf das imake tool aus der XWindow Umgebung zurückzufallen - sehr zum anfänglichen Missfallen der GUI orientierten Kollegen - die jedoch nach reerfolgichen automatischen Generierungen schnell begeistert waren.

Wichtig für Benutzer eines Frameworks - egal ob reiner Benutzer (black box) oder Erweiterer (white box): Nur ganz wenige Entwickler kennen die internen Abhängigkeiten der Module und Packages untereinander. Deshalb muss auf automatische Weise sichergestellt werden, dass die benötigten Module auch generiert werden.

Die meisten sog. IDEs unterstützen weder grössere Gruppen von Entwicklern noch bieten sie einen automatischen und daher qualitätsgeprüften Entwicklungsprozess. Die komplette Kontrolle des eigenen Produktionsprozesses ist unabdingbar. Ressourcen sind dafür einzuplanen.

Problemfall Repositories und die Zukunft von Generierungswerkzeugen

Zur Zeit ist noch der grösste Teil der im Entwicklungsprozess verwendeten Tools Filebasiert. Zunehmend werden jedoch von Toolherstellern Repositories eingesetzt, z.B. im VisualAge Java der IBM. Grund für die Einführung eines Repositories ist der Wunsch, die Granularität von Änderungen kleiner als nur pro File zu bestimmen, z.B. übersetzt ein inkrementeller Kompiler nur die Methoden, die sich tatsächlich geändert haben. Dies bringt eine schnellere Übersetzungszeit. Zusätzlich kontrolliert das Repository noch Abhängigkeiten zwischen Klassen, z.B. wenn sich eine Methode einer Klasse ändert, wird der Programmierer automatisch auf betroffene abhängige Klassen hingewiesen.

Auch hier wird der Zwang zur dynamischen Verwaltung von Informationen sichtbar da sonst Äbhängigkeiten (die als Links ausgedrückt werden können) nicht direkt verfolgt werden können.

Diese Entwicklung ist sicherlich zu begrüssen im Sinne grösserer Sicherheit der Generierung. Problematisch wird es für einen automatischen Build, wenn kein Zugriff auf das Repository in 2 Richtungen möglich ist. Ein API zum Repository muss gestatten, die Informationen auszulesen bzw. neu zu setzen. Sonst ist eine Integration der Information in übergeordnete Repositories nicht möglich.

Eine weitere Schwierigkeit von Repositories liegt darin, dass sie plötzlich auch sämtliche Aufgaben eines Source Code Control Systems übernehmen müssen (Multi-User Fähigkeit etc.). Ein Tool wie CVS oder Clearcase „sieht" nur noch ein File (der Inhalt des Repositories), d.h. bisher übliche Massnahmen zur Verwaltung von Änderungen (z.B. in Verbindung mit einem Error Tracking System) funktionieren nicht mehr. Nur das Repository weiss über Änderungen bescheid.

Man kann sich vorstellen auf welche Schwierigkeiten man in einem auf distributed objects basierenden System stösst, mit IDL Spezifikationen, Java und C++ Klassen, Runtime Environment in Form von Naming Service (abgebildet auf DCE) und Modellierungstools wie Rose oder Paradigm. Fast jedes dieser Tools besitzt ein eigenes Repository. Wie sollen diese Repositories synchronisiert werden ohne massiven manuellen (und daher fehlerträchtigen) Einsatz?

In einer solchen Umgebung ist es bereits ein grosses Problem die Konsistenz des Systems sicherzustellen. Es müssen dazu:

- Äbhängige Files in verschiedenen Programmiersprachen neu übersetzt werden.
- Konfigurationsfiles für das Runtime System erzeugt werden
- Einträge in diversen, zur Runtime zur Verfügung stehenden Repositories vorgenommen werden
- Eventuell parallel verschiedene Versionen gleichzeitig laufen können.

Darüber hinaus wird es in Zukunft noch viel wichtiger sein, VOR einer Änderung ihre Auswirkungen bestimmen zu können. Wie kann dies geschehen wenn noch nicht einmal klar ist, WO Änderungen anfangen? Fängt der Entwicklungsprozess mit einem OO-Modell an? Fängt er auf Grund von Legacy Applikationen mit einem Entity-Relationship Modell existierender Datenmodelle an? Fängt er mit GUI Ressource Files an?

Ich weiss auf diese Fragen zur Zeit keine Antwort. Die Vorstellung eines zentralen Repositories mit ALLEN Inhalten ist attraktiv, in der Praxis jedoch nicht machbar. Nötig wäre ein Workflow System der Generierung, basierend auf Tool-unabhängigen Informationen und dynamisch mit Behavior verbindbar. Voraussetzung dafür sind APIs zu den Tools und Repositories sowie eine Sprache die in der Lage ist, beliebige Inhalte und ihre Verbindungen auszudrücken: SGML/XML. Dies widerspricht natürlich dem Interesse der Hersteller, die statt modulare Verwendbarkeit nur allumfassende IDEs anstreben.

WARNUNG: Wer Tools mit Repositories anschafft, sollte sich über die dadurch entstehenden Abhängigkeiten vom Hersteller sowie die Auswirkungen auf den Generierungsprozess im Klaren sein.

Source Code Verwaltung

Der Source Code des NEWSYS Frameworks wird auf einem zentralen Server mit den plattformunabhängigen Tools RCS und CVS verwaltet. Entwickler arbeiten mit lokalen Bäumen, die durch CVS upgedatet und verwaltet werden können. Zu beiden Tools gibt es umfangreiche Dokumentation. Source Code Verwaltung muss vom Build Prozess aus erreichbar sein, z.B. um geänderte Versionen auf Kompatibilität der Interfaces zu untersuchen.

automatische plattformunabhängige Generierung

Dazu wird das Tool Imake verwendet. In der NEWSYS Version 1.0 stehen die zentralen Config Files unter NEWSYS/config/cf, in der Version 2.0 (OS2 und W32) ist Imake ein eigenes Projekt und die anderen Projekte überschreiben in ihrem config/cf Verzeichnis lediglich was unbedingt nötig ist. Dieses Konzept folgt einem Vorschlag aus dem Buch zum Imake.

Alle für NEWSYS nötigen Fremdprodukte, soweit sie im Hause selbst generiert werden, befinden sich im automatischen Generierprozeß.

Dieser Prozess umfasst:

- Standard Library mit konkreten Hilfsklassen
- SP SGML Parser Toolkit
- Imake Produktion tool selbst
- NEWSYS Framework

In jedem Verzeichnis das automatisch bearbeitet werden soll, befindet sich ein sog. Imakefile - die Projektbeschreibung. Er enthält keine plattformabhängigen Optionen sondern gilt für alle Plattformen gleich. Plattformabhängige Optionen stehen in speziellen globalen config Dateien. Aus ihnen und den Anweisungen des Imakefiles sowie den Regeln des imake.rules files wird ein Makefile automatisch generiert der hochgradig plattform,host und user abhängig ist. Dieser Makefile erzeugt das oder die Targets die im Imakefile beschrieben sind.

Der wesentliche gedankliche Unterschied zwischen Imake und IDEs ist, dass Imake ganz im SGML Sinne eine blosse Beschreibung der Materialen, Tools und Abläufe in strukturierter Weise gestattet. Das Imake Tool bearbeitet diese Beschreibungen und erzeugt Makefiles. An KEINER Stelle werden Definitionen des Build Prozesses verdoppelt sondern die Abläufe werden IMMER aus diesen Definitionen erzeugt. Es gibt keine Teilung des Produktionsprozesses in Dokumentation der Definitionen und Projektfiles die diese Definitionen enthalten (verdoppeln). Die Definitionen selbst sind immer verfügbar für die Definition neuer Abläufe und verschwinden nicht in obskuren binären Project Files.

So wie das Source Control System dafür sorgt, dass es von jeder Version eines Sources nur EINE Entity gibt, so sorgt das Imake System dafür, dass vom Source des Produktionsprozesses selbst (Definitionen und Regeln) nur EINE Entität existiert.

Ein auf strukturierten Beschreibungen basierter automatischer Generierungsprozess ist übrigens kompatibel mit der Verwendung von IDEs. Dazu gibt es zwei Möglichkeiten:

1. Entwickler verwenden die automatisch generierten Makefiles als Input zur Erstellung von toolspezifischen Projekt files. Dies ist ein manueller Prozess, der in der Praxis jedoch nur für ganz wenige Projekte nötig ist, da ein Entwickler meist nur an ein oder zwei Packages arbeitet.

2. Das automatische Generiertool erzeugt gleichzeitig die toolspezifischen Projektfiles. Dazu ist die Kenntnis der Struktur der Projektfiles nötig. Dieser Vorgang kann völlig automatisiert werden, z.B. können unter OS/2 Rex files erzeugt werden, die die Registrierung der Objekte im Workframe vornehmen.

Jeder Projektmanager sollte die Effektivität des Generierungsprozesses regelmässig testen. Ein guter Zeitpunkt dafür ist z.B. die Einstellung eines neuen Mitarbeiters.

Der Härtetest: Kann ein neuer Mitarbeiter am ersten Tag eine Version der Applikation oder des Frameworks durch EIN Kommando generieren? Auf einer bis dahin bezüglich Entwicklungsumgebung LEEREN Maschine, d.h. kein Source, keine Compiler/Tools, keine Ressourcen?

Konfigurationsfiles

Sie sind Source Code genauso wie Ressource Files oder Header Dateien. Sie werden im Source Code Control System verwaltet und im Build Prozess konvertiert, bearbeitet und installiert.

Dazu gehören:

- Implementation Repository
- Applikationsspezifische Daten
- Dokumenten Beschreibungen
- Workflow Beschreibungen
- StyleSheets für Views
- Database Schemata

ALLE sind in SGML zu schreiben. Sie sind Bestandteil des Source Codes und werden beim Build prozess geparst. WEHE jemand schreibt nicht geparste Konfiguration Files in die Source Verwaltung!!!

Source Code und Documentation Tools

newsrc.cmd hilft bei der Erzeugung neuer Source Files

i2rtf.exe erzeugt aus Header files automatisch Dokumentation im Rich Text Format

i2html.cmd erzeugt aus Header files automatisch Dokumentation im HTML Format.

Unter Imake/config/util befinden sich weitere kleine tools die bei der Generierung nötig sind und von dort automatisch geholt werden.

SGML Parser kann Konfigurations Files auf syntaktische und teilweise semantische Richtigkeit prüfen.

Alle Source Files haben den NEWSYS Coding Conventions zu entsprechen. Diese regeln Memory Management, Naming und Packaging etc. Class References werden automatisch erzeugt. Teil der Coding Conventions ist die Liste der Pflichtlektüre.

JEDE DOKUMENTATION ZU ARCHITEKTUR UND MECHANISMEN HAT DIE VERWENDETEN DESIGN PATTERNS EXPLIZIT ZU DOKUMENTIEREN. DESIGN PATTERNS SIND INTEGRALER BESTANDTEIL DES FRAMEWORKS.

Es gibt KEINE Ausnahme!

Versionen

Standardmässig leistet ein Source Code Management Tool wie z.B. RCS/CVS speziell im Hinblick auf die Besonderheiten von C++ wenig Dienste. Eine Änderung drückt sich lediglich in einer neuen RCSID des Source Files aus. Damit ist nichts gesagt über den Scope und die Konsequenzen der Änderung:

- Betraf die Änderung INTERFACES?
- Müssen Repository Inhalte geändert werden?
- Wird ein neues Package nötig?

Sinnvoll wäre ein Tag analog der Java VersionUID mithilfe dessen der Programmierer bestimmen kann, ob die neue Version einer Klasse Interface-kompatibel zur alten ist. Der symbolische Lademechanismus des Frameworks kann dann prüfen ob eine Instanz der neuen Klasse zum gegenwärtigen Zustand des Frameworks passt.

Wenn keine Interface Änderung vorliegt, kann die neue Klasse bedenkenlos zur Laufzeit instanziiert werden, da die Interface/Implementation Trennung jede Kenntnis des Clients über Methodenaddressen (ausserhalb der virtuellen Interface Methoden) verhindert. Und selbst diese sind nur als relativer Offset bekannt, d.h. die Position einer Klasse innerhalb eines Packages kann sich verschieben bzw. mithilfe der Repository Einträge sogar in ein ganz anderes Package verlagern.

Der Build Mechanismus könnte leicht so erweitert werden, dass er prüft ob eine Änderung sich auf das Interface auswirkt und dann automatisch eine geänderte VersionUID erzeugen.

Bezeichnenderweise waren auch die Taligent Leute gezwungen den Buildprozess durch eigene Tools und Compiler incrementeller zu machen. Globale Source Ids die nur die Tatsache einer Änderung signalisieren waren zu grob. Ebenso der Make Mechanismus der lediglich das globale Datum einer Änderung berücksichtigt.

Gerade in Bezug auf die Wartbarkeit wird der Vorteil von Java durch das dynamische Binden zur Laufzeit besonders deutlich. Dieses Verhalten könnte in C++ ebenfalls implementiert werden, nur müsste der dynamische Lader dann auch die Vtable der Klasse sowie interne Referenzen innerhalb der Klasse auf EIGENE virtual functions anpassen. Dieses Vorgehen wäre nicht portabel und aufwendig.

Runtime Dependencies

Da Objekte nicht mehr statisch miteinander gelinkt werden sondern einander über Factories oder Broker anfordern, kann es passieren, dass durch einen Konfigurationsfehler eine Klasse zur Laufzeit nicht vorhanden ist. Der Default Mechanismus des Frameworks generiert in diesem Fall eine Exception (analog CORBA_bind() exceptions) und der Client kann diese fangen und verarbeiten.

Wünschenswert wäre es, wenn der Buildprozess die Abhängigkeiten vorher festhalten und zur Verfügung stellen könnte. Beim Boot der Applikation könnte dann das Framework sicherstellen, dass alle gewünschten Klassen zur Laufzeit auch vorhanden sind oder einen Fehler mitteilen.

Chapter 9. FREMDPRODUKTE IM FRAMEWORK

Damit sind Softwarepakete gemeint, nicht Tools wie Compiler etc.

Regeln für Fremdprodukte

Folgende Voraussetzungen gelten für Fremdprodukte damit sie im Rahmen des NEWSYS Frameworks verwendet werden können.

Voraussetzungen des Einsatzes:

- Plattformunabhängigkeit (OS/2, W32, Unix)
- 32 Bit Technologie
- Kapselung durch Interfaces, soweit nötig und machbar.

Für die Integration von Fremdprodukten gelten folgende Regeln:

Regeln der Verwendung:

1. Sind Änderungen nicht auszuschliessen, müssen die Fremdprodukte über Interfaces gekapselt werden.
2. Auf keinen Fall dürfen fremde konkrete Klassen oder Interfaces AUSSERHALB des sie verwendenden Packages sichtbar sein.
3. Nur EIN Package darf eine fremde Komponente wrappen.
4. KEINE statischen fremden Libraries dürfen direkt verwendet werden.
5. Die Fremdprodukte werden im Buildprozess definiert und verwendet.
6. Ihr Standort bleibt dem Applikationsprogrammierer verborgen.
7. Notwendige Build options werden einmal im zentralen Repository des Builds gesetzt.
8. Sourcen von Fremdprodukten werden der eigenen Versionsverwaltung und Produktion unterstellt.
9. Fremde Typen werden GRUNDSàTZLICH durch eigene Typedefs oder Klassen gewrappt. Dies ermöglicht eigene Erweiterungen OHNE den Source des Fremdproduktes ändern zu müssen.
10. Bei Klassen aus Fremdprodukten gilt: Es muss eine USE Relation verwendet werden, KEINE Ableitung.

Standard Bibliothek mit konkreten Hilfsklassen

Praktisch jede Applikation oder Toolkit benötigt konkrete Klassen wie Container, Listen etc. Alle Teile von NEWSYS sind berechtigt, diese Klassen jederzeit zu verwenden. Einzige Bedingung ist daß vorher ein Typedef auf eine NEWSYS Klasse vorgenommen wird, damit bei der vollständigen Implementation in IDL eine Wrapper Klasse definiert werden kann.

Liegt die Bibliothek im Source Code vor, wird sie auf dieselbe Weise generiert mit Imake wie auch NEWSYS und andere Produkte.

GUI Toolkit

Das NEWSYS Framework verwendet zur plattformunabhängigen Bearbeitung von graphische Events eine externe Klassenbibliothek.

Konkrete Probleme bei der Integration eines sog. GUI Builders waren:

- ein low level C- Callback Interface, das vom Framework Team selbst objektorientiert gekapselt werden musste.
- Context Klassen des Frameworks kapselten die primitiven graphischen Elemente und Events um vorgefertigte Bausteine zu liefern, die dem System höherwertige Information zur Verfügung stellen. (Prinzip des Situations Konzeptes, Uni Freiburg). Die Verbindung zum Model-View-Controller Konzept des Frameworks wurde ebenfalls durch Kapselung erreicht.
- Dynamische Verbindung zwischen Graphischem Ereignis und Semantischer Aktion musste selbst entworfen und implementiert werden.
- Die graphische Klassenbibliothek stellte sich als zu schwach in Bezug auf image processing heraus und unterstützte zusätzlich kein Multithreading.
- Der sehr bequeme GUI Editor konnte nicht verwendet werden, da die HIC Teile des Frameworks Views DYNAMISCH für beliebige Dokumente aufbauen mussten (oder für Datenbank requests).

Die GUI Bibliothek wird generiert sondern auf dem ComponentServer installiert und durch Imake automatisch includiert. Neue Versionen der GUI Bibliothek (DLLs) erzwingen neue Versionen der HIC Packages des Frameworks da die Bibliothek keine Interface/Implementation Trennung kennt.

SGML Parser Toolkit und Parser

Das SP Toolkit von James Clark beinhaltet einen SGML Parser und Entity Manager. Das NEWSYS Configuration Framework verwendet dieses Toolkit zum Validieren von Dokumenten.

SP wird im Hause generiert auf den Plattformen OS/2 und W32. Das Toolkit behandelt plattformunabhängig Zeichensätze (unicode u.a.) und ist selbst durch zusätzliche StorageManager Typen erweiterbar. Momentan wird unterstützt: File Storage, HTTP, Filedescriptor etc.

Chapter 10. FRAMEWORK STRUCTURES UND CODING STANDARDS

Welche Rolle spielt ein Coding Standard in Bezug auf die Strukturen eines Frameworks?

Ein Resultat der Strukturdefinitionen war, dass uns Strukturen im Framework drei Probleme aufgeben:

1. Wie findet und erklärt man Strukturen und ihre Konsequenzen?
2. Wie machen wir sie explizit, d.h. programmatisch und visuell sichtbar?
3. Wie machen wir die Verbindungen zwischen ihnen explizit, d.h. programmatisch und visuell sichtbar?

Mittlerweile sind wir in der Lage, Framework Strukturen zu finden und zu erklären, aber bis jetzt befindet sich dieses Wissen nur in unserem Kopf. Eine systematische Entwicklung braucht mehr. Neben individuellem Training und Weiterbildung ist ein Coding Standard das erste Tool, das unser Design- und Architekturwissen sichtbar macht.

Ein coding standard in einem Framework Projekt hat NICHTS mit der Optik des Source Codes zu tun. Er ist stattdessen eine Realisierung des objekt-orientierten Wissenstandes des Teams und deckt eine grosse Zahl von micro design Fragen ab. Der Coding Standard enthält die Programmierregeln, abgeleitet aus der Erfahrung mit dem Framework. Und er enthält Markup (Auszeichnungen, visible tags) die versuchen Stellvertreter für die unsichtbaren Strukturen zu sein.

Ein Coding Standard füllt die Lücke zwischen Design/Architektur Level und der tatsächlichen Codierung.

WIESO CODING STANDARDS?

Der coding standard eines mittleren Projektes nimmt schnell die Form eines dicken Buches an. Im ersten Moment erscheint er überaus restriktiv, man muss sich richtiggehend einleben. Hinzukommt, dass der Coding Standard kontinuierlich mit Framework und Tool zusammen weiter entwickelt werden muss. Eine dauernde Pflegeaufgabe.

Ist dieser ganze Aufwand wirklich nötig? Zum Verständnis habe ich einige - schmerzhaft erworbene - Erfahrungen meines letzten Framework Projektes zusammengefasst. Sie sind im Gegensatz zu den obigen Strukturen stark auf C++ bezogen.

Aus eigenen Erfahrungen schätze ich, dass die Einarbeitung eines neuen Mitarbeiters in eine grössere C++ Applikation ca. 1 Jahr und bei einem Framework bis zu eineinhalb Jahren dauern kann, bis der Mitarbeiter in der Lage ist, Designentscheidungen selbständig zu treffen unter Beachtung ALLER Konsequenzen auch für Auslieferung und Wartung.

Toolunterstützte Coding Standards sind hier eine grosse Hilfe. Sie entlasten den Entwickler von einem Teil der Entscheidungen, sichern dass die entstandenen Sourcen verträglich mit dem System sind und dass bestimmte Entscheidungen "defensiv" vorbelegt sind. Beispiele sind: Macros für Forward Declarations die die const Problematiken lindern, Plattformunabhängigkeit sichern und die Gefahren der automatischen Methodengenerierung durch den Compiler verringern.

Die Erstellung eines Frameworks bringt eine Welle von Konventionen und Absprachen mit sich, die Teil des Coding Standards werden müssen.. Aufgabe des Coding Standards ist es, gerade diese internen Verpflichtungen und Vorgehensweisen deutlich zu machen und ihre Einhaltung so weit möglich zu erzwingen.

Oft stellt man fest, dass bestimmte Bereiche in C++ Projekten völlig unscharf oder gar nicht definiert wurden. Gerade Systemaspekte wie Memory Management, Object Sharing und Multithreading werden nicht durch Regeln und Mechanismen erfasst.

Coding Standards sind verpflichtet in ALLEN Design Fragen eine Antwort zu liefern. Was in einem C++ Projekt nicht definiert ist, wird schiefgehen und lässt sich nur mit grossem Aufwand reparieren.

Coding Standards im C++ Bereich haben folgende Aufgaben:

1. Entlastung der Programmierer durch generative Mechanismen,
2. Sicherung der Qualität
3. Ermöglichen des Einsatzes von Pre- und Postprocessing Tools
4. Explizit machen der internen Verpflichtungen und Absprachen bei der Verwendung oder Erweiterungen von Teilen der Applikation bzw. des Frameworks durch Namensgebung und Tags.
5. Dokumentation der physischen Struktur des Produktes, vom der Organisation des Source Codes bis hin zu ausgelieferten Packages und Komponenten sowie deren Konfiguration.
6. Als Check Liste für Code Reviews und QA Massnahmen.

Coding Standard und Tools

Meine Erfahrung in einem grösseren Framework Projekt bezüglich Coding Standards und Tools war , dass zwischen beiden eine enge wechselseitige Abhängigkeit besteht.

C++ erzwingt strikte, teilweise unangenehme Konventionen bei der Programmierung, wenn Reuse, Wartbarkeit und Verständlichkeit ein Ziel sind. Ohne Tools die z.B. Templates fuer Header und Implementation Files erzeugen bleibt die Akzeptanz der Coding Standards gering und fehlerhaft.

Andererseits ist der Einsatz von Tools wiederum stark vom Coding Standard abhängig:

Je rigoroser und formaler die Coding Standards sind, desto leichter können Tools eingesetzt werden. Aus regel- und strukturlosem Matsch kann auch das beste Tool keine weitergehende Information erzeugen. Gerade das automatische Prozessing ist aber bei C++ wichtig, da kleine Änderungen oft grosse Source Code Auswirkungen haben. Der Einsatz von Entwurfsumgebungen wie Together++ und OEW++ hatte im Projekt NEWSYS nicht den erhofften Zuwachs an Flexibilität gebracht. Diese Tools hatten Probleme im Reverse Engineering Bereich, Schwierigkeiten mit der Integration von IDL Präprozessoren und waren generell nicht übermässig stabil. Oft blieb nur der Rückgriff auf kleine Tools wie stream Editoren oder REXX scripts.

Ohne Tools kommt ein Coding Standard nicht zum Leben. Tools müssen die Anwendung des Coding Standards erleichtern bzw. automatisieren. Andererseits können ohne die Konventionen und Namensregeln des Coding Standards einfache Tools kaum angewendet werden.

Beispiel: Ein kleines, vom Framework Team selbst entwickeltes interaktives Tool zur Erzeugung von Header und Implementation Files (sowie Interface Klassen).

Das Tool frägt folgende Facts ab:

- Interaktive Abfrage der Art des gewünschten Source Codes (Interface, Default, Normal Implementation, Function, Applikation etc.)
- Abfrage der Domain/Package Zugehörigkeit, da sie in den File und Klassennahmen einging.
- Abfrage der Eigenschaften (public typdefs, inline functions gewünscht)
- Abfrage ob header oder src file gewünscht werden.
- Abfrage der Kurzinfo zu der jeweiligen Klasse

Aus diesen Angaben erzeugt das Tool folgendes:

- korrekte Filenamen und Klassennahmen, entsprechend der Konventions.
- Korrekte source code information (/* Log */ etc. unterschiedlich nach file type (mit oder ohne Versions String.)
- Korrektes Copyright
- Korrekte Kurzinfo zu der Klasse
- Einfügen der Tags zur automatischen Dokumentation
- Default Construktor und Assignment operatoren je nach Typ (Interface oder Implementation Class, per default auf protected oder private gesetzt damit sie nicht vergessen werden können.
- virtual Destruktor
- Einfügen von Macros zur Runtime Type Information und Loader Functions
- Verwendung der richtigen Macros für Class declarations (siehe Dll erzeugung)
- include bestimmter Package Header files für die Verwaltung physischer Packages.
- Gleichzeitig hat newsrc die neu erzeugten files sofort automatisch im concurrent versioning system (CVS) registriert.

Ohne dieses Tool werden die Konventionen nicht eingehalten. Es fehlt noch das nachträgliche automatische Einfügen weiterer Methoden.

Jedes eingesetzte generative Tool sollte auch bereits erstellte Targets nochmals neu überarbeiten können. Gerade wenn der Source Code umfangreicher wird, sind rekursiv generative Verfahren unumgänglich.

Im NEWSYS Framework wurde das Tool an einem Wochenende in Vrexx programmiert und danach kontinuierlich erweitert, gemeinsam mit der Entwicklung der coding standards. Weitere derartige Mini Tools waren: implementation2interface (erzeugt aus einer ersten Implementation eine abstrakte Interface Klasse) sowie das umgekehrte interface2implementation.

Dank dieser Tools konnte Metainformation sowie die Class Reference aller Klassen automatisch erzeugt werden. Die automatische Dokumentation verwendete eine angepasste Version des i2rtf tools aus Fresco (erzeugt rich text format) und später i2html (erzeugt html doc.) Alle diese Tools waren sehr einfach und schnell geschrieben bzw. angepasst.

Der Beitrag kleiner selbstgefertigter Tools zur Produktivität der Framework Entwicklung ist unschätzbar. Projektleiter müssen für Entwicklung und Pflege solcher Tools Zeit einplanen.

Noch besser: Der Einsatz eines eigenen Preprocessors. Fresco z.B. enthält einen idl compiler im source code, der public domain ist. Er wäre das ideale Tool zur Verwaltung des eigenen Source Codes und könnte leicht angepasst werden.

Tools und Konventionen gegen Memory Leaks und Pointer Fehler

Jedes grössere Projekt braucht eine explizite Memory Management Policy. Sie muss klar dokumentiert sein.

Wenig brauchbar sind Kommentare in Methoden von der Art: Client muss delete sagen bzw. darf nicht delete sagen etc. Ein solches Vorgehen zwingt den Klient immer genau zur Betrachtung der Interna verwendeter Klassen.

Besser ist eine einheitliche Policy unter Einsatz von Reference Counting und Smart Pointern (die enge Kopplung der beiden Techniken ist nicht vermeidbar, siehe Exceptions). Z.B. kann folgende Regel gelten: Wenn der Empfänger parameter einer Methode über die Laufzeit der Methode hinweg festhält, muss der Reference Count erhöht werden. Return Werte von Funktionen sind dagegen immer bereits um eins erhöht worden.

Dieses Vorgehen ist generell nicht schlecht (z.B. auch bei Corba eingesetzt), jedoch auch nicht der Weisheit letzter Schluss.

Mehrere Probleme entstehen:

Clients vergessen die Freigabe von Ressourcen und es entstehen Memory Leaks. Dem kann durch den konsequenten Einsatz von automatisch generierten Smart Pointern entgegengewirkt werden.

Objekte anderer Bibliotheken die nicht über dieses Ref Counting verfügen müssen anders behandelt werden. Gerade dabei entstehen subtile Memory leaks (allocated mit new und array operator, freigegeben ohne und umgekehrt)

Hier hilft der Einsatz z.B. von Purify. Durch Instrumentierung der Binaries gelingt es Purify Memory Allocation Fehler sogar in Systembibliotheken zu finden. Es hat sich beim Polytool Framework als Unschätzbar erwiesen.

Entgegen dem reinen Verlassen auf Konvention bringt reference counting zweifellos mehr Sicherheit. Das grösste Problem damit tritt jedoch bei komplexen Datenstrukturen wie Trees und DAGs auf, die jedoch meist gerade den Kern komplexer Frameworks darstellen, da sie beim composite object Pattern entstehen. **Reference Counting verschlechtert die Performance** bei Tree Traversal enorm (20-30%) und zwingt teilweise zu obskuren Techniken wie z.B. dass das Root Objekte eines DAGs im Konstruktor seinen eigenen Ref Count erhöhen muss, damit das Freigeben seiner Kinder funktionieren kann. Diese dekrementieren nämlich ihrerseits den Refcount des Root objects und wenn das letzte Child dies getan hat, kehrt der Flow of Control zum Root Object zurück. Nur dass dieses mittlerweile ref count gleich Null hatte und bereits freigegeben wurde. Generell treten mit Reference Counting **Probleme mit zirkulären Datenstrukturen** auf.

Zu Garbage Collection Techniken: [WILSON], speziell C++:[JELLIS]

Letztlich hilft bei komplexen Datenstrukturen nur ein Garbage Collector, der es bezüglich der Performance durchaus mit Reference Counting aufnehmen kann.(Siehe: Memory Allocation Costs in Large C and C++ Programs, [ZORN]. Beim Reto Team ist ein kommerzielles Produkt für C++ erfolgreich im Einsatz.

Coding Standards und Dokumentation

Automatisch generierte Class Reference

Heutzutage wird die Class Reference automatisch aus dem Source Code erzeugt, z.B. im HTML Format inclusive automatisch eingefügter Links auf Baseklassen bzw. verwendete Klassen. Jede Klasse sollte im Source Code beschreiben wozu sie dient und wie sie funktioniert unter Hinweis auf eingesetzte patterns. Dafür ist ein tag Schema zu entwickeln das automatisch verarbeitet werden kann. Bei der Beschreibung der Funktionalität der Constraints kann gar nicht genug dokumentiert werden. Vor allem in Zeiten stürmischer Entwicklung steht nämlich im allgemeinen NUR die Dokumentation im Source selbst zur Verfügung.

Es darf nur das beschreiben werden, was nicht durch ein Tool aus dem Source selbst entnommen werden kann:

z.B. Return Value: NONE ist m.E. ein sinnloser Komment. Die Signatur der Klasse kann vom Tool komplett herausgezogen und untersucht werden, d.h. diese Aussage kann und sollte automatisch generiert werden. Sonst ändert jemand die Signatur aber nicht den (ohnehin bloss Information verdoppelnden Kommentar und es entsteht eine Inkonsistenz) Der Entwickler sieht an der Signatur sowieso, dass sie Return type void hat.

Anders ist es mit der Verwendung der gelieferten Parameter, die beschreiben werden sollte. Dafür genügt ein einfacher Tag mit Namen des Parameters.

Eine automatisch generierte Class Reference enthält pro Klasse:

- Source Code Version
- Interface Version

- ClassName und Derivation
- Comment mit Zweck der Klasse
- Class Methods
- Comment mit Zweck der Methode
- Typedefs
- Defines (Nur so werden „harte" Stellen sichtbar)
- Globale und statische Funktionen

Dazu genügt ein einfaches Tagging Schema wie es z.B. bei Fresco oder Java eingesetzt wird.

Wenn die verwendeten Namen der Klassen und Methoden einer Konvention folgen und die Struktur des Source Codes geregelt ist, dann kann das Dokumentationstool auch automatisch Hyperlinks zu verwandten oder verwendeten Klassen generieren.

Tags benutzen um Strukturen bzw. Auswirkungen auf Strukturen sichtbar zu machen

Runtime Structure

JEDE ABWEICHUNG von der generellen memory management policy muss mit einem besonderen Tag versehen werden durch das das Dokumentationstool eine besondere Hervorhebung generieren kann bzw. sogar in eine Liste kritischer Objekte einfügen kann.

Multithreading:

Dieser Punkt wird in C++ gerne „nach hinten" verschoben. Leider besitzt C++ kein keyword „synchronized" wie es z.B. Java hat. Aus dem Sourcecode ist somit nicht ersichtlich, ob einer Methode/Class von mehreren Threads verwendet werden kann. Hier sind unbedingt klare Regeln zu treffen, z.B. durch Einführen eines „dummy" keywords und tags damit synchrone Methoden gekennzeichnet werden können.

Ein weiteres Problem beim multithreading sind sog. suspend() methoden, mit denen ein thread suspendiert werden kann. Was passiert dabei üblicherweise mit resourcen die dieser Thread allokiert hat? Sie bleiben gesperrt und es entstehen üble deadlocks!!

Extension Structure

Wenn eine Methode IMMER überschrieben werden muss, sollte sie pure virtual (abstract) sein. Ein tag „Override" soll dann die Verpflichtungen beschreiben die dem Überschreiber daraus erwachsen.

Template und Hook patterns sollten durch entsprechende Tags gekennzeichnet werden, die das dahintersteckende Pattern sichtbar machen. Z.B. Template Tag, Hook Tag.

C++ spezifisch:

Jede Verwendung von „inline" Methoden, Default Parametern und Template Klassen oder Funktionen betrifft die Extension Struktur fundamental, da diese Konstukte Teil des Client Codes werden.

Was ist allen dreien gemeinsam?

Sie sind ein Alptraum für die Wartung und Kapselung von Produkten, da sie alle Teil des Client Codes werden. Änderungen bei diesen 3 Typen zwingen alle Clients die sie benutzen zum Recompilieren. Der Coding Standard bzw. das automatische Source Tool sollten unbedingt jede Verwendung dieser Techniken besonders her-

vorheben. Service und Wartung sind davon total abhängig.

Unter IBM Cset kam noch hinzu, dass die Entwicklungsumgebung Abhängigkeiten von Source oder Header Änderungen bei Templates NICHT erkannte und deshalb inkonsistente Module erzeugt wurden.

Bei Templates gibt es Techniken wie hoisting und Einschliessen der Templates in factories hinter einem abstrakten Interface die diese Gefahren verringern.

Warum sollten nicht alle Methoden von Interface Klassen inline sein? Laut Lacos wird die Vtable dann mangels eindeutigem Platz bei JEDEM Client dieser Klasse nochmals angelegt!

Namen für Klassen, Variablen und Packages, Components

Je mehr Denkarbeit in die Zusammensetzung von Klassennamen investiert wird, um so leichter ist die nachfolgende automatische Generierung von Paketen, DLLs und Dokumentation.

Strukturelemente sind hier:

Produkt oder Firma Kürzel

Früher wurde ein eindeutiges Kürzel zur Vermeidung von Name Collisions eingesetzt. Dazu gibt es heute andere Möglichkeiten wie z.B. Name spaces oder module declarations. Ein einheitliches Kürzel hat dennoch seine Bedeutung. Es zeigt dem Entwickler an, welche Erwartungen er an eine Klasse oder eine Funktion stellen kann – ob sie z.B. der Memory Management Policy des Frameworks unterliegt oder nicht. Ob sicher ein dynamische Laden des jeweiligen Packages erfolgt oder nicht. D.H. Erwartungen an ALLE Strukturen drücken sich im Anfangskürzel aus.

Ein einheitliches Anfangskürzel drückt nicht nur einen Namespace aus, sondern den Geltungsbereich eines Regelwerkes wie es die Strukturen eines Frameworks darstellen.

Domain Kennung

Ein Buchstabe im Klassen oder Funktionsnamen sollte reserviert sein für die Kennzeichnung der logischen Domain zu der Klasse oder Funktion gehören. Je nach Domain können unterschiedliche Regeln bezügliche physischer oder extension Struktur gelten.

Klassentyp

Ein weiterer Buchstabe sollte die physiche Art der Klasse kennzeichnen. Es gibt folgende Kriterien:

- Interface
- Default Implementation
- Normal Implementation of Interface Class
- Template Class
- Concrete Class (do not derive from)
- Function

Diese Kennzeichnung gibt dem Programmierer Hinweise auf die physische Struktur. Z.B. lässt ihn ein „N" für normal class sofort erkennen, dass compile und linktime Abhängigkeiten zu der Klasse N bestehen. Ein Tool kann eine Componente daraufhin untersuchen, ob Abhängigkeiten zu Implementationen ausserhalb der Componente selbst bestehen.

Je nach Klassentyp sind die vom Compiler auch automatisch erzeugbaren Methoden IMMER mit defensiven Defaults vom Source Tool zu generieren. (CTOR, DTOR, assignment operator, copy constructor). Darf/kann ein Client eine Klasse mit "new" erzeugen oder zwingen wir ihn, über eine Factory/Broker zu gehen? Ein versehentliches "new" kann im Fehlerfall dazu führen, dass man nicht eine DLL mit Implementationen austauschen kann sondern den ganzen Client ersetzen muss.

VariablenTyp

Die Praxis hat gezeigt, dass das Kennzeichnen des Variablentyps (Member oder local, parameter) enormen Einfluss auf die Verständlichkeit des Codes besitzt, speziell wenn ein neuer Mitarbeiter sich einarbeitet oder ein Fehler in fremdem Code gesucht wird.

Sinn

Nur lange und klare Klassennamen sind wartbar und verständlich. Lediglich bei Namen von DLLs sollte die 8.3 Konvention beachtet werden, da z.B. OS/2 bis einschliesslich Warp 3.0 nur DLLs laden konnte, die dieser Syntax entsprachen (auch von HPFS File Systemen)

Package Zugehörigkeit

Hier gehen die Meinungen auseinander. Wenn die kleinste Einheit der Isolation ein Package ist, dann scheint die Kodierung der Package Zugehörigkeit im Namen der Klasse oder Funktion sinnvoll.

Wenn dagegen ein flexibleres Broker Schema benutzt wird, wird aus der phyischen Kennzeichnung eines Packages bloss wieder eine logische Gliederung, da die physische Location der Klasse ohnehin verborgen bleibt. In diesem Fall ist ein Package Tag mehr eine weitere Untergliederung der logischen Domains.

Zwingend für die Codegenerierung mit Microsoft C++ ist eine Package Kennung in Bezug auf das „declspec" keyword und zwar auf Client wie auch auf Implementationsseite.

Keine Verwendung von * und & (Pointer und Referenz)

Möglichst die automatisch generierten Typen XYZ_var und XYZ_ref verwenden, damit der eventuelle **Umstieg auf CORBA leichter fällt**. (Deshalb ist das automatische Generieren der Typedefs so wichtig, sonst können keine Forward Declarations mit Pointern oder Referenzen gemacht werden).

Dies ist ein weiteres Beispiel für „vorausschauende Typedefs".

Programmierregeln

An dieser Stelle kann nur ein kleiner Ausschnitt aus den Programmierregeln eines Frameworks erwähnt werden. Ausgewählt wurden sie im Hinblick auf ihre Konsequenzen bezüglich Source Code Änderungen und Runtime Fehlern.

const Problematik: bitwise vs. logical constness

Das C++ Konzept der bitwise constness stösst immer dann auf Probleme wenn Reference Counting, Caching, Lazy Evaluation und Proxies eingesetzt werden. Diese Techniken haben nichts mit der logical constness zu tun, d.h. das Objekt wird in seinem logischen Status nicht verändert. Dennoch müssen Member variablen wie recount oder chaching variables gesetzt werden auch wenn nur gelesen werden soll. Da hilft oft nur der cast von this auf ein non const MyThis.

Der Coding Standard sollte hier eine verbindliche Guideline schaffen.

Erfahrung zeigt, dass nachträgliche Änderungen der „const" Eigenschaft eines Typs (egal ob hinzugefügt oder entfernt) zu umfangreichen Anpassungen durch viele Module hindurch führt.

Das „final" Konzept von Java unterscheidet sich hier drastisch von der C++ Implementation. So konnten in Versuchen mit dem JDK1.1 beta als „final private" deklarierte Variablen mühelos über Methoden herausgegeben

und modifiziert werden.

Exceptions

Hier sei im wesentlichen auf die Artikel von Jim Reeves in diversen C++ reports aus 1996 verwiesen [REEVES]. Nur eines: Die Frage ob exceptions oder nicht stellt sich nicht mehr da viele Fremdprodukte oder Systemkomponenten einfach Exceptions werfen. Sie müssen auf jeden Fall ordentlich behandelt werden. Ein übler Nebeneffekt des Fangens von Exceptions ist, dass unter Umständen Fehler unterer Schichten maskiert werden können.

Im coding standard müssen die Regeln für das Fangen von Exceptions genau festgehalten werden.

Ein Nebeneffekt von Exceptions auf die physische Struktur: Ohne Interface Klassen auch für Exceptions entstehen Link – time Abhängigkeiten.

Ein weiterer Nebeneffekt auf die Source Code Struktur liegt darin, dass das nachträgliche Einfügen von Exception handling (vielleicht noch punktuell begrenzt) umfangreiche Source changes auslöst, ähnlich der "const" Problematik.

Vorausschauende Typedefs

Häufig werden als Parameter einer Methode Strings übergeben, die jedoch **semantisch e**twas ganz anderes ausdrücken. Typische Fälle sind „Name", „ID" etc. Ein Hinweis darauf ist meist im Variablennamen zu finden.

Wenn sich semantisch ein neuer Typ ergibt, die Implementation aber vielleicht aus Zeitgründen den Typ als String implementiert (die UML 1.0 unterscheidet bezeichnenderweise Typ und Class), dann sollte ein **vorausschauender Typedef** verwendet werden, z.B:

typedef String Name;

Dies lässt die Möglichkeit zu, nachträglich aus Name einen eigenen Typ zu bilden ohne massive Änderung der Interfaces. Sogar völlig transparente Proxies sind nachträglich möglich. Ausserdem macht der Typ den Sinn viel klarer.

Ein weiteres Beispiel ist die Verwendung eines symbolischen Tokens bei der Anforderung von Objekten aus Factories:

Statt *factory->getForm(String SpecialForm)*

Besser *factory->getForm(RequestPattern SpecialForm)*

Dies lässt die nachträgliche Implementation eines Trader Patterns offen.

Minimale Interfaces

Die Forderung, dass Interfaces von Klassen minimal sein sollen, bekommt im Framework Design eine neue Bedeutung. Da in einem Framework viele Klassen miteinander kollaborieren, dies aber nicht explizit programmatisch sondern nur implizit im Source Code ausgedrückt werden kann, sollten die Interfaces so minimal wie möglich sein um die Benutzung zu vereinfachen und semantische Klarheit zu ermöglichen.

Häufig wird durch „convenience" Methoden das interne Interface zur Basis Klasse korrumpiert mit dem Effekt von Deadlocks.

Dr. Bernhard Scheffold, einem Freund und ehemaligen Kollegen von mir verdanke ich das folgende Beispiel mit dem herrlichen Titel: **Foot.shoot_in_yourself()**

Eine Falle im Zusammenhang mit polymorphen Methoden

(Wie man sich in der OOP in den Fuss schiessen kann, oder Rekursion ueber die vtable)

Wir haben eine Klasse, die zwei virtuelle Methoden implementiert, die in einer abgeleiteten Klasse etwas modifiziert werden muessen. Im wesentlichen

Sollen die Methoden der abgeleiteten Klasse das gleiche bewirken, wie die Methoden der Elternklasse, so dass sie im wesentlichen einfach diese aufrufen.

Konkret ist die Elternklasse wie folgt implementiert:

```
class parent
{
virtual enable (boolean doEnable) { /* code to perform enable or disable */
}
virtual disable () {enable (false;}
};
```

Die Kindklasse wird (besonders, wenn obige Implementation nicht bekannt ist) naiv wie folgt implementiert:

```
class child: parent
{
virtual enable (boolean doEnable)
{
if (doEnable)
{
parent::enable ();
// additional code
}
else
{
parent::disable ();
// additional code
}
}
virtual disable ()
{
parent::disable ();
// additional code
}
};
```

Was passiert nun bei folgendem Code?

parent p = new child ();*

p->disable;

Da p vom dynamischen Typ child ist, wird die Methode child::disable aufgerufen.

Diese ruft parent::disable, welche nun nicht, wie urspruenglich geplant parent::enable (false) aufruft, sondern, da p vom dynamischen Typ child ist, wieder child::disable.

Wir haben also eine **Endlosschleife**!

Diese Endlosschleife lässt sich beheben (wobei man allerdings auch wieder In die gleiche Falle tappen kann!).

Eine Loesung sieht wie folgt aus (hierbei ist die Kenntnis der Implementation von parent hilfreich!):

```
class child: parent
{
virtual enable (boolean doEnable)
{
if (doEnable)
{
parent::enable ();
// additional code
}
else
{
parent::enable (false);
// additional code
}
}
virtual disable ()
{
parent::enable (false);
// additional code
}
};
```

Regel: Wer Convenience Funktionen zum Gebrauch einer Klasse braucht, hat sie AUSSERHALB dieser Klasse in Helper Klassen zu implementieren.

Dies ergänzt die von John Lacos aufgestellten physischen Prinzipien um einen logischen Aspekt.

Bernhard hat mir überdies mitgeteilt, dass er nach dieser Erfahrung sowohl eigenen Framework Code als auch

Visual Age C++ Class libraries auf solche Konstrukte durchsucht hat und in beiden fündig geworden ist!

Im Fall des eigenen Framework hatte die Einführung einer zusätzlichen firstChild() Methode zusätzlich zur ohnehin vorhandenen nextChild(Child_ref) Methode eine subtile Endlosschleife verursacht.

Manuelle Dokumentation und Metadokumentation

Bei jedem Stück Dokumentation sind folgende Fragen zu stellen:

1. ist sie nötig oder kann sie automatisch generiert werden?
2. gibt es eine Trennung von Dokument und Source Code?
3. Welchen Mangel dokumentiert die Tatsache eines extra Dokumentes?

Ad 1)

So weit wie irgend möglich sollte Dokumentation automatisch erzeugt werden. Dazu müssen extra Tags sowie strenge Konventionen der Source Code Struktur eingeführt werden.

Ad 2)

Jede Trennung erzeugt ein Wartungsproblem. Das Argument, dass extra Dokumentation übersichtlicher als Source Code Studium ist, gilt keinesfalls, da Tools beliebige Sichtweisen auf Source Code erzeugen können.

Ad 3) Vom Source getrennte Dokumente sowie Meta- Dokumente zeigen an, dass sich ein Sachverhalt nicht programmatisch ausdrücken lässt und mithin einen Mangel in der Informatik. In vielen Fällen liegt jedoch einfach eine verkehrte Sicht des Verhältnisses von Tools und Dokumentation vor wie im Beispiel des Buildprocesses:

Es kann nicht sein, dass die Parameter die bei einer IDE eingestellt wurden um ein Projekt zu erzeugen anschliessend dokumentiert werden und jeder Entwickler muss die Parameter neu eingeben bei einem neuen Projekt. Stattdessen müssen die Parameter der Generierung an einer Stelle beschrieben werden und aus dieser Beschreibung wird ein Makefile bzw. ein Projektfile erzeugt:

Beschreibung IST Source Code

Chapter 11. VISUALISIERUNG MULTI-DIMENSIONALER DECOMPOSITION UND COORDINATION

Dieses Kapitel ist noch in Bearbeitung. Themen sind:

VRML

ASPEKT ORIENTED COMPUTING

HYTIME/SGML Tools

Chapter 12. LITERATUR

[LAC96] John Lacos, Large Scale C++ Software Designs, Addison Wesley 1996

[WKR96] Zur Bedeutung von C++ Coding Standards, Systor intern 1996

[WKR96/1] Workshop zu SGML, Systor intern 1996

[WKR97/1] Organizational and Software Frameworks, Systor intern 1997

[UML1.0] http://www.rational.com (Systor intern: ARS NewsDB)

KUHN] Thomas Kuhn, die Struktur wissenschaftlicher Revolutionen, in: Wolfgang Steegmüller, Grundfragen der Gegenwartsphilosophie

[FRESCO], http://www.faslab.com

[REEVES] Jim Reeves, Exception Handling, C++ Report Vol 1996

[JDK11GUIDE] Guide to the API, Java Development Kit 1.1

[BAI/WKR] Frank Baier, Walter Kriha, Workshop zu JDK 1.1, Systor intern, 1997

[MAHLER] Eve Mahler, Jeanne El Analoussi, Designing Document Type Descriptions,

[KICZ96] Gregor Kiczales u.a., Aspect – Oriented Programming. A position paper from the Xerox parc Aspect – Oriented Programming Project. http://www.parc.xerox.com/aop/

[KICZ/PAEPKE] Gregor Kiczales, Andreas Paepke, Open Implementations and Metaobject Protocols, Xerox Corp. 1996 http://www.parc.xerox.com/oi/

[NATAN] Ron Ben Natan, Introduction to CORBA,

[REDLICH] Jens – Peter Redlich, CORBA 2.0, Praktische Einführung für C++ und Java, Addison Wesley 1996

[MELLIS] Margarete Ellis, ..

[JELLIS] John Ellis, David L. Detlefs, Safe, Efficient Garbage Collection for C++,Xerox 1993

[MAFF97], Sylvano Maffeis, reliability in large CORBA designs

[MEYERS1] Scott Meyers, 50 ways to improve your C++ designs,

[MEYERS2] Scott Meyers, 35 more ways to improve your C++ designs,

[MOWB97] Corba Design Patterns,

[TALIGEN1] Taligent Guide to Programming,

[SEITERS] Linda Seiters, Design Patterns for Evolving Systems

[LIEBERHERR] Karl J. Lieberherr, Adaptive Object Oriented Software, The Demeter Method with Propagation Patterns, PWS Publishing Company, 1995

[SIEMENS] E.Buschmann u.a. Design Patterns for Systems???

[SIMONSEN/KENSING] Jesper Simonsen, Finn Kensing, Using Ethonography in Contextual Design. Communications of the ACM 7/1997

[ACKERMANN] Philipp Ackermann, Developing Object-Oriented Multimedia Software. Dpunkt Verlag, 1996

[GOLDBERG], Adele Goldberg, Smalltalk-80,??

[FUCHS] Mathew Fuchs, The User Interface as Document: SGML and Distributed Applications, 1996

[SUTTER], Christoph Sutter, FISH – a Flexible Information System for Hypermedia, Systor intern, 1997

[FOWLER97], Martin Fowler, Analysis Patterns – Reusable Object Models. Addison Wesley 1997

[WILSON], Paul R. Wilson, Uniprocessor Garbage Collection Techniques, ftp://ftp.cs.utexas.edu/pub/garbage/

[WHORF], Benjamin Lee Whorf, Sprache, Denken, Wirklichkeit.

[WITTGENSTEIN], Wittgenstein,

[ZORN], Benjamin Zorn, University of Colorada at Boulder, Technical Report CU-CS-665-93, 1993