7. Wiederverwenbarkeit, Portabilität und Interoperabilität

7.1. Konzepte

Ein Produkt ist *portabel*, falls es ohne grössere Änderungen auch auf einer anderen Hardware und Betriebssystem-Software lauffähig ist.

Ein Produkt ist *wiederverwendbar,* falls Komponenten in einem anderen (Software) Produkt ebenfalls eingesetzt werden können.

Wiederverwendbare Komponenten können Design Spezifikationen, Handbücher, ... sein.

Man unterscheidet zwei Arten der Wiederverwendung:
- Zufällige Wiederverwendung
- Gezielte Wiederverwendung

Bei der gezielten Wiederverwendung von Komponenten wird von Anfang an daran gearbeitet, Module, Designs, ... so auszulegen, dass sie möglichst universell einsetzbar sind.

Solche Komponenten müssen in der Regel:
- Sehr gut strukturiert
- Dokumentiert
- Erweiterbar, modular und damit leicht wartbar

sein.

Auf der anderen Seite sind Komponenten, die wiederverwendbar ausgelegt werden müssen, in der Regel teurer als normale Komponenten. Es muss schlicht wesentlich besser designed und implementiert werden. Aber in der Regel lohnt sich der Extra-Aufwand wegen der mehrfachen Nutzung.

Bei den Rechnern der ersten Generation wurde so gut wie nichts mehr als einmal eingesetzt. Jeder Rechner war in diesem Sinne ein Unikat.

Mit der Zeit wurden zunehmend standardisierte Hardware Komponenten eingesetzt. Zudem wurden Standard-Bibliotheken verfügbar, speziell für FORTRAN Programme, zur numerischen Berechnung (NAG Bibliothek)

Bei Java spielen Class-Libraries eine entscheidende Rolle. Viel Funktionalität wird in Packages speziell geliefert (zum Beispiel : RS232 Package für Java, Security und Krypographie Packages, Netzwerk Packages, spezielle Packages für die Anbindung der Telefone, für eingebettete Systeme,...).

Es wird also zunehmend wichtig, sich auf bereits vorhandene Komponenten stützen zu können. Gemäss einer empirischen Studie werden lediglich 15% aller Software Pakete ausschliesslich für ihren ursprünglich vorgesehenen Zweck eingesetzt.

Eine obere Limite für die Wiederverwendung von Komponenten ist: 85% eines komponentenbasierten Systems lässt sich aus Standard-Komponenten zusammen bauen. Der

Rest ist Customizing oder Spezialentwicklung. Dieser Wert ist eher hoch angesetzt. In vielen praktischen Fällen ist der Abdeckungsgrad durch Standard Komponenten lediglich 60-65%. Falls der Prozentsatz noch tiefer liegt, wird der Einsatz der Komponenten allmählich fraglich!

7.2. Hindernisse für die Wiederverwendung

Das erste und wichtigste Hindernis für die Wiederverwendung von Komponenten ist das Ego. Die meisten Software Entwickler haben den Eindruck, ausser ihnen können niemand programmieren.

Dieses Phänomen kann durch das Management abgeschwächt werden, also top down, durch brutale Gewalt!
Ein anderer etwas cleverer Approach ist:

- Der Entwickler erhält einen Bonus, sofern seine Komponenten wieder verwendet werden
- Der Entwickler erhält auch einen Bonus, wenn er Komponenten anderer Entwickler einsetzt

Die Wiederverwendbarkeit der Module ist die eine Seite, das Finden der relevanten Module ist die andere. Auf diesem Gebiet ist noch sehr viel zu tun. In einer Diplomarbeit (Bärtschiger 1998) wird ein Komponenten-Management System für eine Firma entwickelt, die eine eigene Komponenten Bibliothek entwickelt hat und bestehende Module und Applikationen mit Hilfe von Wrappern CORBA fähig gemacht hat. Das heisst, die bestehenden Module erhalten eine Schale, einen Wrapper, umgelegt, und dieser Wrapper kommuniziert über eine Standard-Schnittstelle (zum Beispiel IDL : Interface Definition Language) mit einem Software Bus (IIOP oder CORBA = Common Object Request Broker).

7.3. Wiederverwendbare Software - Case Studies

Betrachten wir einige Fälle aus der Praxis:

7.3.1. Raytheon Missile Systems Division

Raytheon ist einer der grössten Arbeitgeben an der Ostküste Amerikas, speziell zwischen Boston und New York. Insbesondere werden bei Raytheon Atom-U-Boote entwickelt und gebaut. Raytheon ist primär im Militärsektor tätig.

Bei unserem Beispiel geht es aber um die kommerzielle Software, die Inhouse entwickelt wurde. Insgesamt wurden über 5'000 COBOL Programme analysiert. Es wurde festgestellt, dass sich die Programme auf 6 Grundfunktionen zurück führen lassen.

Zwischen 40 und 60% des Software Designs und der Software Module wurden standardisiert und wiederverwendet. Als Gundfunktionen wurden definiert:

- Daten sortieren
- Daten editieren oder manipulieren
- Daten kombinieren
- Daten auswerten
- Daten updaten
- Daten analysieren

Über sechs Jahre wurden die Module systematisch standardisiert und vereinheitlicht.

Kriterien für die Wiederverwendung waren:

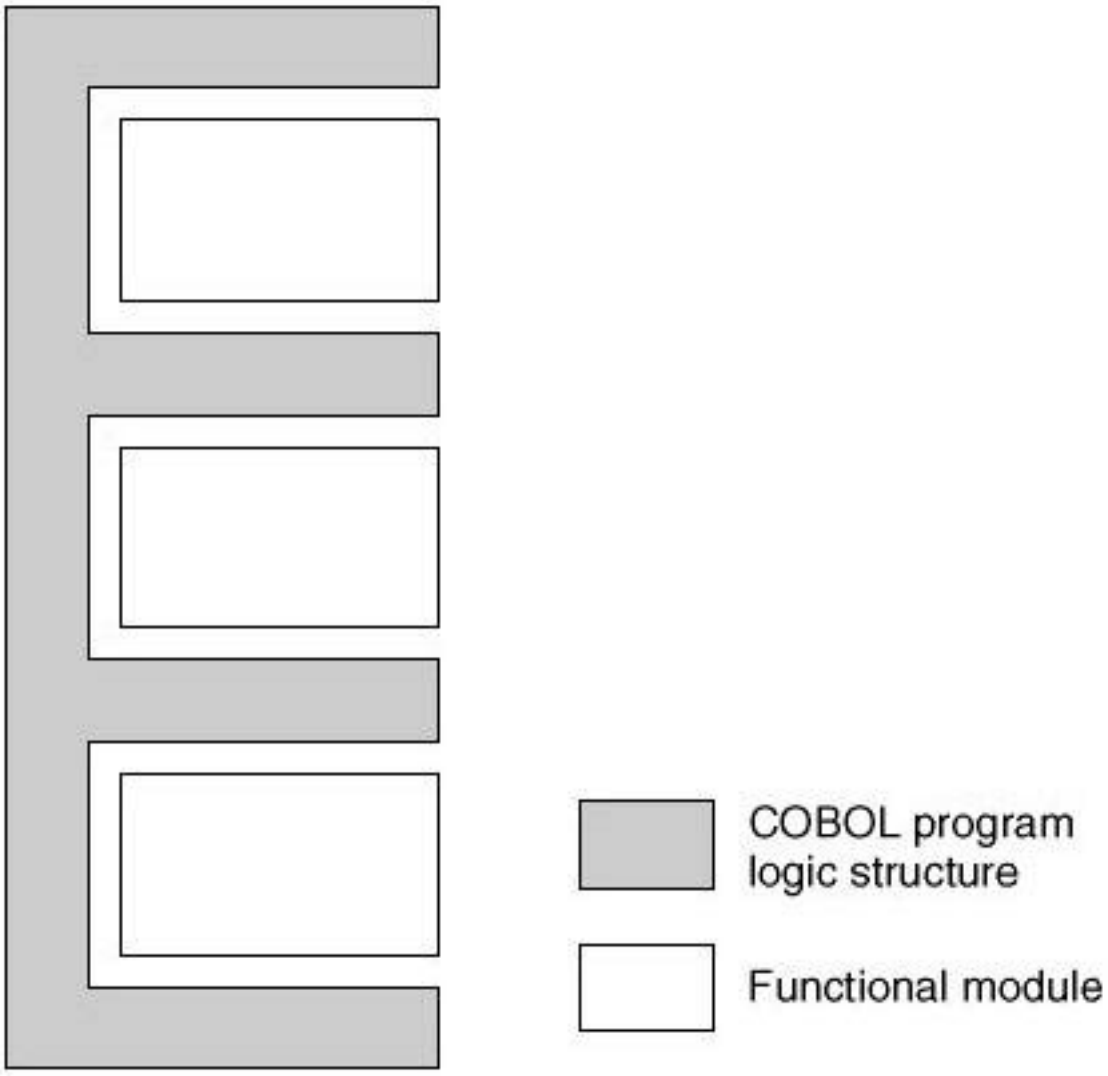
- Funktionale Module welche eine der Grundfunktionen von der obigen Liste implementieren
- Möglichst tiefer Wartungsaufwand

Raytheon reduzierte die Anzahl Module von 5'000 auf 3'200, so dass eine Wiederverwendbarkeit von ca. 60% resultiert.

Die Standard Komponenten wurden in Bibliotheken abgelegt und jeweils mit **copy** in die entsprechenden Modul-Rahmen kopiert.

Gemäss Raytheon ergab sich der Hauptnutzen aus der schnellen Reaktion auf Benutzerwünsche.

Über ungefähr zehn Jahre wurde die Wiederverwendbarkeit der Module erhöht, zudem die Lesbarkeit der Module, wodurch sich der Wartungsaufwand drastisch reduzierte. Die Wartungskosten reduzierten sich gemäss Raytheon um bis zu 80 Prozent.



7.3.2. Toshiba Software Factory

1977 startete Toshiba in Fuchu eine Software "Fabrik", um Prozesssteuerungssoftware herzustellen.

1985 umfasste die Software Fabrik 2300 Techniker und umfangreiche Software, überwiegend in FORTRAN geschrieben.

Toshiba setze sum Teil auch noch Assembler ein, allerdings mit schlechter Produktivität: pro FORTRAN Programmzeile rechnet Toshiba mit 4 Zeilen Assembler Programmcode.

60 Prozent der Software ist in FORTRAN, 20% in Assembler und der Rest in speziellen Sprachen, speziell für die Echtzeit-Applikationen bezw. Maschinen.

Die Software Produktivität wird in EASL = *equivalent assembler source lines* gemessen.

Als Standard wurde das Wasserfall Modell für die Software Entwicklung eingesetzt. Als "Börsenkurs" wird EASL verwendet, sowohl auf der Stufe der einzelnen Programmierer als auch insgesamt. Typisch für Japanische Betriebe, wird EASL täglich angezeigt und statistisch sehr genau verfolgt.

Es hat sich gezeigt, dass die Produktivität über die Jahre um etwa 8-9% gesteigert werden konnte.

Es wurde zudem versucht , die Wiederverwendung der Software zu fördern und die Fehlerrate pro 1000 EASL zu reduzieren.

Als wichtiger Schritt wurde eine *software reuse database* installiert und definiert. Das Ergebnis zeigt folgendes:

- 33% Wiederverwendung der Designs
- 32% Wiederverwendung der Dokumentation
- 48% Wiederverwendung in der Implementations-Phase

Zusätzlich wurde die Grösse der Komponenten gemessen:

- 55% zwischen 1 und 10 EASL
- 36% im Bereich 10K - 100K EASL

7.3.3. NASA Software

In einer Studie wurde untersucht, in welchem Grade die Software bei NASA wieder verwendet wird.

Insgesamt wurden 25 Produkte untersucht, im Bereich zwischen 3000 und 112'000 Zeilen Code pro Produkt.

Die Software wurde klassifiziert. Es resultierten vier Gruppen:

- Komponenten, die ohne Änderung wiederverwendet wurden
- Komponenten, die mit leichten Änderungen übernommen werden konnten (weniger als 25% Änderungen)

- Komponenten, die nur mit grösseren Änderungen übernommen wurden (mehr als 25% Änderungen)
- Komponenten, die neu entwickelt wurden

Insgesamt wurden 7188 Komponenten untersucht.

Im Detail wurden 2954 FORTRAN Module genauer untersucht und folgende Ergebnisse berichtet:
- 45% der Module werden direkt oder modifiziert wieder verwendet
- 28% fielen in Kategorie 1
- 10 % in Kategorie 2
- 7% in Kategorie 3

Wiederverwendete Module waren in der Regel eher klein, klar strukturiert und mit klar definierten Schnittstellen.

Das ist nicht sonderlich erstaunlich: komplexere Module sind eher wenig wiederverwendbar. Es wurden keinerlei Hilfsmittel zur Unterstützung der Wiederverwendung eingesetzt.

7.3.4. GTE Data Services

Bei GTE wurde eine kommerzielle Applikation systematisch mit Management Support auf Wiederverwendung ausgerichtet.

Jeder Entwickler, der eine wieder verwendbare Komponente entwickeln konnte, erhielt zwischen 50 und 100 USD pro akzeptierten Modul. Zudem wurde eine Wiederverwendung in Form von Royalities dem Entwickler gut geschrieben.

Die Ergebnisse:
- 14% Wiederverwendung
- Savings ca. 1'500'000 USD im ersten Jahr
- In den Folgejahren zunehmende Savings durch Wiederverwendung im Bereich 20-50%

GTE baute eine umfangreiche Komponenten-Datenbank. Der Fokus lag dabei auf grossen Modulen. Man wollte also analog zu Toshiba möglichst grosse Komponenten, mit viel Funktionalität, realisieren.

GTE erachtete den Management Support (und den finanziellen Anreiz) als wesentlich für die erfolgreiche Implementierung des Wiederverwendungs-Programmes.

7.3.5. Hewlett Packard

HP implementierte in verschiedenen Divisions ein Wiederverwendungs-Programm mit dem Ziel modularere Software und Firmware zu entwickeln.

Die Ergebnisse(Start 1983):
- 4.1 Fehler pro 1000 Zeilen Code bei neu entwickelter Software
- 0.9 Fehler bei wieder verwendeter Software
- Insgesamt eine Reduktion der Fehlerrate auf 2 Fehler pro KLOC oder anders ausgedrückt, eine 50% Reduktion der Fehlerrate

Das Programm kostete 1 Mio USD; die Savings lagen in der Grössenordnung von 4.1 Mio USD über 10 Jahre gemessen. Der Break Even Point (investierte Summe plus laufende Kosten neu = laufende Kosten alt) lag bei zwei Jahren.

In der Plotter und Laser Printer Division wurde ein analoges Programm für die Entwicklung der Firmware gestartet. Die Ergebnisse:

- Ca 20'000 LOC in C
- Kosten über drei Jahre : 2.6 Mio USD
- Wiederverwendung über drei Jahre brachte folgende Savings:
  - 5.6 Mio USD
  - 24 % weniger Fehler
  - 1.3 Fehler pro KLOC
  - Produktivität stieg um 24%
- Die Entwicklungskosten der wieder verwendbaren Firmware kostete 11% mehr als die konventionelle Entwicklung
- Die Integration bestehender Komponenten kostete nur 19% der typischen Entwicklungskosten

7.3.6. Zusammenfassendes Ergebnis der Case Studies:

Wiederverwendung von Software muss eines der Ziele sein. Das Ziel wird (hoffentlich) auch durch die Bequemlichkeit der Programmierer begünstigt.

7.4. Objekte und Wiederverwendbarkeit

Zum Zeitpunkt als die Konzepte des Composite/Structured Designs (C/S D) aufkamen, war es das Ziel Module funktional kohäsiv zu gestalten. Ein solcher Modul übernimmt eine und nur eine Funktion.
Diese Module waren klare Kandidaten für die Wiederverwendung. Allerdings zeigte es sich schnell, dass dies nur bedingt richtig ist. Was fehlt sind die dazu gehörigen Daten.

Die nächste Stufe in der Cohesion Hierarchie sind die informational cohesion Module, also im wesentlichen Objekte.
In der Praxis hat es sich gezeigt, dass in der Regel die Produktivität mit OO Methoden gesteigert werden kann *und* die Wiederverwendbarkeit erst noch erhöht wird.

Allerdings muss man zu solchen Studien zwei Punkte anmerken:

1. in der Regel sind OO Programmierer höher qualifiziert als die andern
2. in den ersten Projekten mit einer neuen Technologie werden in der Regel keine wesentlichen Verbesserungen erzielt, im Gegenteil!

Das grösste Problem bei OO Systemen ist die korrekte Definition der Klassenstruktur.
Falls dies nicht gelingt, dann muss in der Regel das Klassensystem von Grund auf neu entwickelt werden.

7.5. Wiederverwendung in der Design und der Implementationsphase

In der Design und der Entwurfs Phase kann die Wiederverwendung bestehender Designs oder ganzer Software Pakete den Projektverlauf signifikant beeinflussen. Im Extremfall geht es um einen Make or Buy Entscheid.

7.5.1. Module Wiederverwendung

Im Verlaufe des Designs kann einem Entwickler Module definieren, die er bereits in früheren Projekten entwickelt oder gekauft hat. Speziell wenn das neue Projekt sich mit den selben Anwendungsgebieten wie das frühere Projekt befasst: man spricht dann auch von branchenspezifischen Referenzmodellen. Diese existieren auf unterschiedlichen Ebenen:

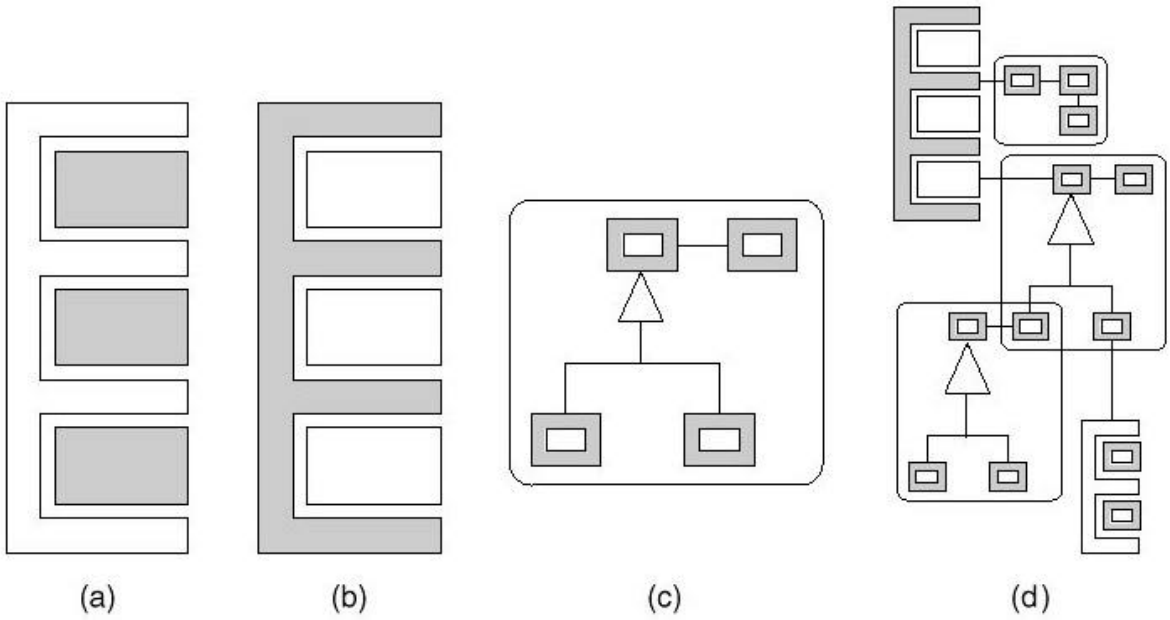
1. Datenmodelle
2. Objektmodelle
3. Referenzmodelle (Rahmenwerke)

Typische Anwendungsgebiete einer solchen Wiederverwendung sind:

1. Klassen / Modul Bibliotheken für spezielle Anwendungen
   Mathematische Berechnungen
2. GUI Design

Speziell im GUI Bereich lohnt es sich für Firmen spezifische Bibliotheken zu entwickeln, die immer wieder eingesetzt werden können:

1. dadurch wird der Programmieraufwand drastisch reduziert
2. dadurch sehen die Oberflächen einigermassen einheitlich aus.



Design Wiederverwendung : schematische Darstellung

(a) Bibliotheken oder Toolkits
(b) Frameworks
(c) Design Patterns
(d) Software Architekturen mit (a) - (c) kombiniert

Schauen wir uns die einzelnen Möglichkeiten genauer an:

7.5.2. Anwendungs Frameworks

Ein Applikations-Framework wie oben schematisch dargestellt, beschreibt eine einheitliche Kontroll-Logik für Programme.
Gemäss der "Gang of Four" (GoF) (Gamma, Helm, Johnson, Vlissides : Design Patterns - Elements of Reusable Object-Oriented Software [1995]) ist ein Framework:
"a set of cooperating classes that make up a reusable design for a specific class of software".

Beispiele für Frameworks sind:
(a) Klassenbibliothek für die Konstruktion von Compiler
(b) Klassenbibliotheken für die Implementierung von Kommunikationssoftware

Warum reduziert ein Framework den Programmieraufwand?
(a) es wird weniger Design Arbeit nötig sein, da bestimmte Kontrollstrukturen bereits definiert sind
(b) der in der Regel schwierige Teil des Designs der Kontrollstruktur ist bereits vorweg genommen

Neben Applikations-Frameworks existieren auch diverse Programm Frameworks, wie etwa
(a) MacApp von Apple
(b) MFC : Microsoft Foundation Class Library für die Entwicklung von C++ Programmen.
(c) graphische Bibliotheken wie etwa der Java Swing oder AWT Library, die Motif Library...

7.5.3. Design Patterns

Christopher Alexander definiert Design Patterns [1977]:
"Each pattern describes a problem, which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way, that you can use this solution a million times over, without ever doing it the same way twice".

Alexander beschrieb Design Patterns / Entwurfsmuster für die Lösung von architektonischen Problemen. Das witzige daran ist, dass seine Bücher einen Boom in der Informatik auslösten, allerdings erst fast 20 Jahre später!

Ein Entwurfsmuster ist also eine Lösung für ein allgemeines Problem wechselwirkender Klassen beziehungsweise Objekten.

In der obigen Abbildung (Abbildung c) symbolisieren wir dies durch Beziehungen von Klassen. Die weissen Boxen müssen situativ implementiert werden.

Randbemerkung
Eine der einflussreichsten Personen auf dem Gebiet des Objekt orientierten Software Engineering ist Christopher Alexander, ein weltbekannter Architekt, der gerne zugibt, nicht von Objekten zu verstehen. In einem seiner Bücher beschreibt er eine Pattern Sprache für die Architektur. Seine Ideen wurden von der "Gang of Four" aufgenommen. Patterns finden sich überall, zum Beispiel in der Mathematik, der Physik und eben der Informatik.

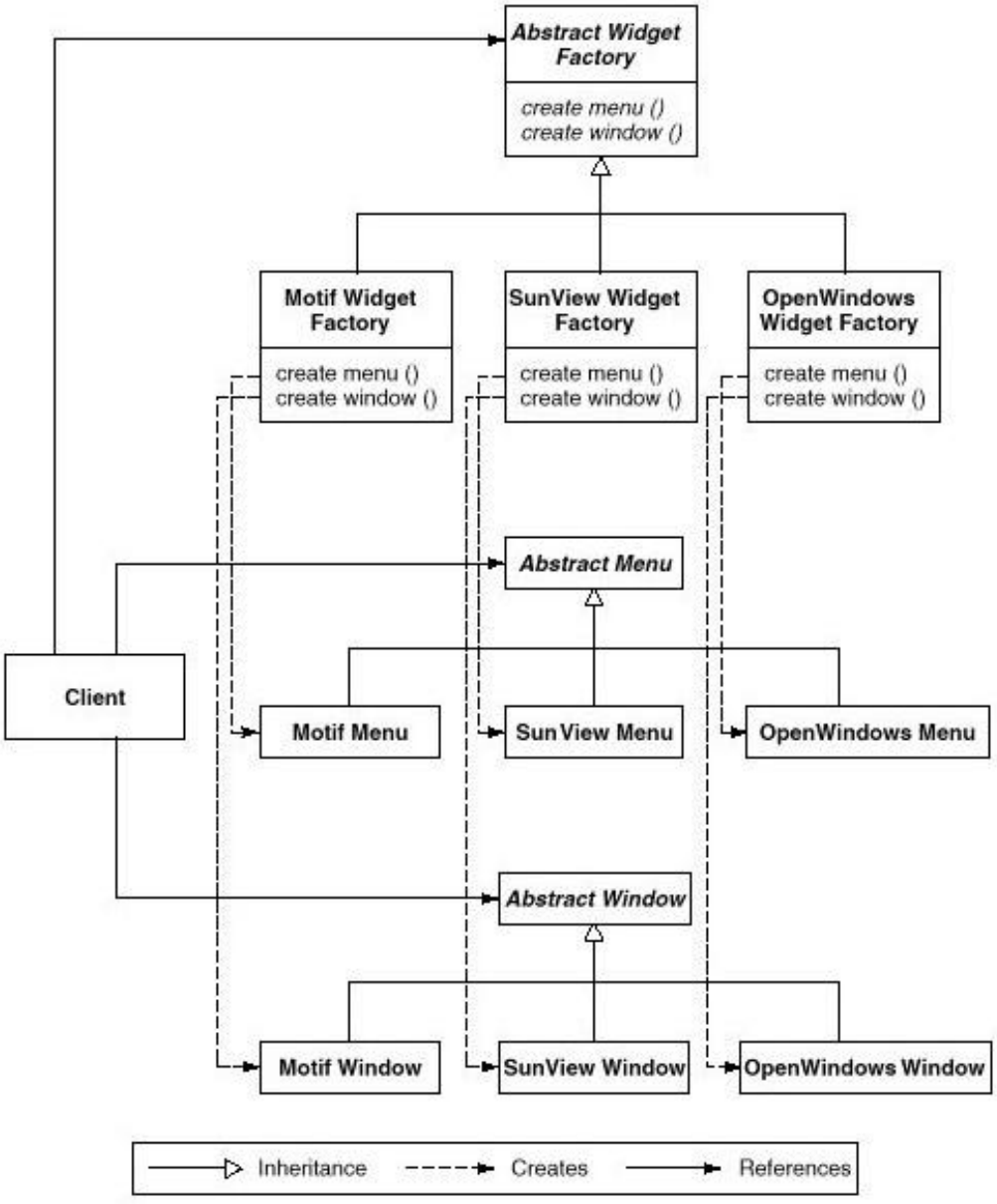
Schauen wir einmal an, wie Design Patterns für die Entwicklung einer GUI Bibliothek eingesetzt werden können:
im GUI Bereich sind verschiedene Standards definiert worden

1. Motif, der Idee nach ein Hersteller unabhängiger Standard
2. SunView, ein GUI Paket, welches sich zwar an Motif orientierte, aber nicht kompatibel ist
3. OpenWindows, noch ein weiteres GUI Standard Paket

Falls wir nun portable Software entwickeln möchten, haben wir verschiedene Möglichkeiten:

1. wir können die Software für jeden GUI Standard separat implementieren
2. wir können versuchen von den konkreten Bibliotheken zu abstrahieren und auf einer abstrakteren Ebene "generische" Funktionen zu implementieren. Diese sind dann dafür verantwortlich, dass die gewünschten Standards unterstützt werden.

Die Variante 2 entspricht grob dem *Abstract Factory* Entwurfsmuster:



Die abstrakte Klasse (kursiv geschrieben) enthält mehrere abstrakte Methoden.
Zum Beispiel:
1. *create menu*
2. *create window*

Die (konkreten) Klassen **Motif Widget Factory SunView Widget Factory OpenWindows Widget Factory** sind Subklassen der ***Abstract Widget Factory***.
Die einzelnen abstrakten Methoden werden in den obigen Factories konkretisiert.

Für jedes Widget , jedes Element der graphischen Oberfläche, existiert eine abstrakte Beschreibung. In der Abbildung haben wir exemplarisch ***Abstract Menu*** und ***Abstract Window*** aufgeführt.

Der Ablauf sieht konkret wie folgt aus:
falls das **Client** Objekt ein Fenster benötigt, sendet es eine Meldung an die Methode *create Window* der ***Abstract Widget Factory*** und es ist dann Aufgabe des Polymorphismus die korrekte Version aufzurufen.

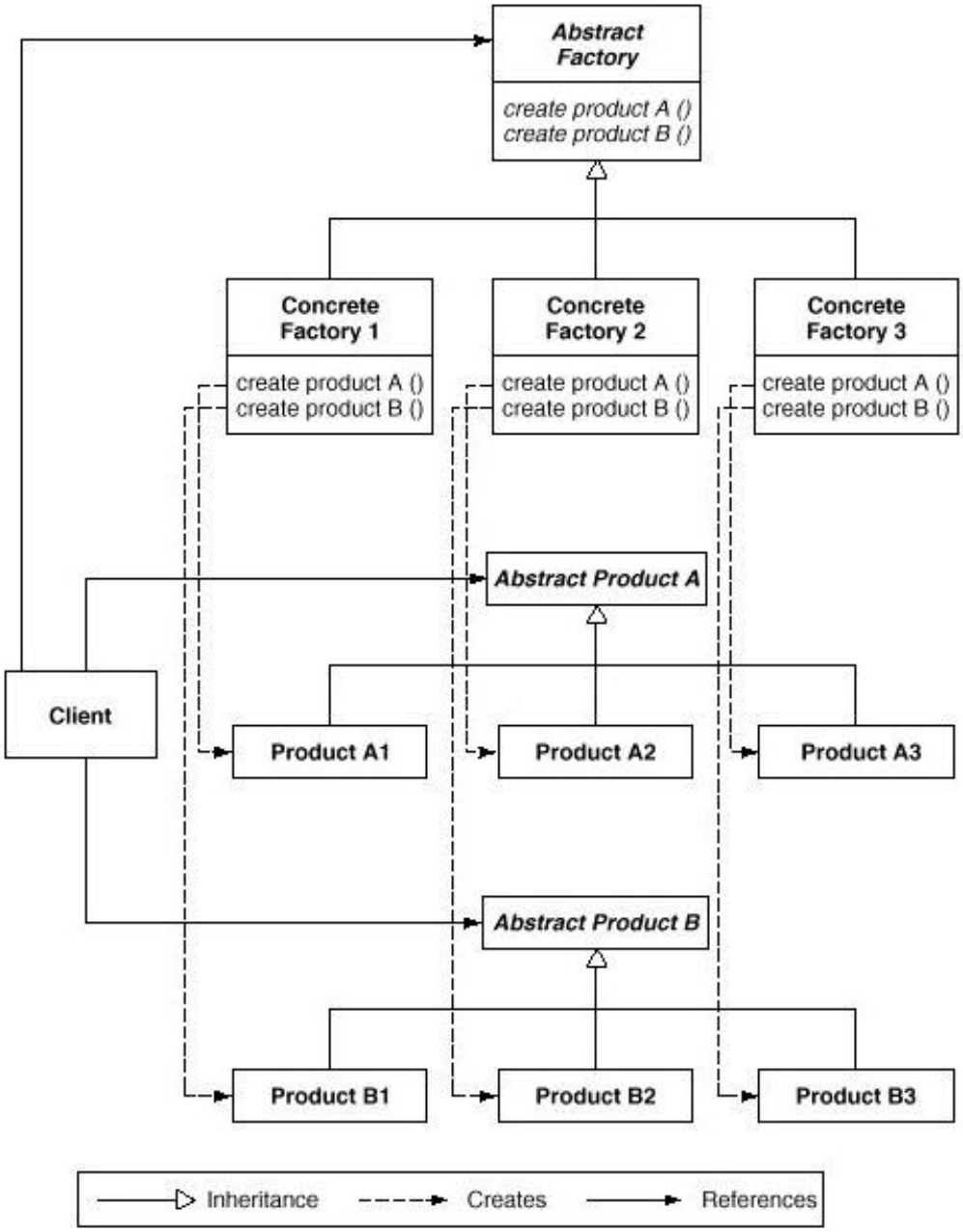
Zentral für die Lösung sind also die abstrakten Klassen, ***Abstract Widget Factory, Abstract Menu,*** und ***Abstract Window***.
Dadurch erhalten wir also eine Entkopplung des GUIs von der Platform.

Da bei einem Design Pattern die Details der Klassen je nach Anwendung spezifiziert werden müssen, haben wir in der Abbildung oben (Framework, Design Pattern,..) das innere der Klassen weiss gelassen.

Schauen wir uns nun ein konkretes Beispiel unserer abstrakten Widget Factory an. In der Abbildung unten sehen Sie, wie konkrete Objekte in konkreten Factories "gebaut" werden.

Pattern treten oft nicht isoliert, sondern zusammen mit anderen Patterns auf. Deswegen haben wir in der Übersicht weiter oben mehrere Design Patterns mit einander verknüpft.

Patterns treten auch in der Anforderungs- und der Analyse- Phase auf.



7.5.4. Software Architektur

Software Architektur beschreibt den Aufbau komplexer Software Systeme. In einer Untersuchung haben Siemens Mitarbeiter (Buschmann, Stahl,...) gängige Architektur Muster zusammengefasst.

Diese werden wir noch genauer kennen lernen. Unter anderem geht es darum, dass oft ein System in Form von (mehreren) Ebenen zerlegt wird (Layer Pattern).

Sie kennen dieses aus der Datenkommunikation : das ISO OSI Modell.

7.6. Wartung und Wiederverwendung

Traditionell wird die Wiederverwendung mit den kürzeren Entwicklungszeiten begründet. Ein anderer eigentlich wichtigerer Gesichtspunkt ist die Wartung!

Das lässt sich leicht abschätzen:

Annahme:
40% der Spezifikation wird wieder verwendet
40% der Designs
40% der Manuals
40% der Komponenten
...

Die Konsequenz ist eine verkürzte Entwicklungszeit (allerdings nicht nur 40%, da mit einem Integrations-Overhead und Customizing gerechnet werden muss).

Zusätzlich reduzieren sich die Testzeiten und die Wartungskosten Im besten Fall erreicht man eine um 30% reduzierte Entwicklungszeit. Da die Entwicklungskosten etwa 30% der lebenszyklus-Kosten sind, ergibt sich eine Reduktion der Gesamtkosten um 30% * 30% also insgesamt etwa 9% in der Entwicklungsphase und etwa 18% in der Wartungsphase.

Objekte und Produktivität
Als Constantine, Myers et al die Konzepte des Composite Designs aufzeichneten, dachte man, dass funtional cohesion das Ziel jedes Designs sei. Es zeigte sich aber, dass dies nicht so ist, da Methoden auf Daten ausgeführt werden und die Abstrakten Datentypen deswegen besser geeignet sind, oder in der Sprache des Composite Designs: informational cohesion, mit andern Worten Objekte.

Deswegen haben viele Organisationen damit gerechnet mit Hilfe der OO Methoden die Wartung besser in den Griff zu bekommen und die Produktivität der Software Entwicklung steigern zu können.

Ein wesentlicher Effekt der OO Methoden war und ist, dass grössere und komplexere Software implementiert werden kann.

Dieser Nutzen zeigt sich jedoch nicht gleich beim ersten Projekt einer Organisation. Man muss also mit Umstellungsaufwand rechnen. Speziell zeigt sich das im Bereich der GUIs, welche gemäss Sun Microsystems bis zu 60% der Software ausmachen.

Die Vererbung hat sich als sehr zweischneidig erwiesen:
Falls die Basisklassen nicht absolut korrekt definiert wurden, entsteht genau so Spagetti Code wie bei der traditionellen Programmierung.

Im schlimmsten Fall entstehen sogar Speicherplatzprobleme durch ein Übermass von Attributen, die stufenweise vererbt wurden. Vererbung von Attributen kann ja auch nicht unterdrückt werden. Die Klasse erbt also unter Umständen viel zu viel.

Die Probleme mit Polymorphismus und Dynamic Binding haben wir bereits besprochen.

Trotzdem zeigen Case Studies in vielen Unternehmen, dass OO Techniken nicht nur eine intellektuelle Herausforderung für Programmierer sind.

7.7. Portabilität

Die laufend steigenden Kosten für die Wartung der Software müssen irgend wie in den Griff bekommen werden. Eine Variante, die Wartungskosten zu reduzieren, besteht darin, die Software für möglichst viele Hardware und Betriebssysteme verfügbar zu machen.

Beispiel:
eine Programmbibliothek zur numerischen Berechnungen in der Hydrodynamik wird aus welchen Gründen auch immer, in FORTRAN geschrieben. Leider zeigt es sich sehr schnell, dass DEC VAX FORTRAN und Sun SOLARIS FORTRAN nicht voll kompatibel sind.

In diesem Falle müssen die Hardware oder Systemsoftware spezifischen Anteile "gekapselt" werden, der Einfluss der Hardware und der Systemsoftware also möglichst in einem oder mehreren Modulen zusammengefasst werden, Dadurch kann bei eventuell weiteren Portierungen lediglich dieser Modul angepasst werden.

7.7.1. Hardware Inkompatibilitäten

Die bei weitem erfolgreichste Hardware Platform ist das IBM System /360 , welches bereits seit über 30 Jahren läuft.
Software, die für die ersten S/360 entwickelt wurde, läuft im Prinzip auch noch auf den modernsten S/360. Für IBM wurde dies zu einer nicht kopierten Erfolgsstory.

Ähnliches kann man über die DEC VAX sagen, Rechner, die sehr häufig für numerische Berechnungen eingesetzt wird und wurde.

7.7.2. Betriebssystem Inkompatibilitäten

Das wohl am längsten lebende Betriebssystem ist Unix, welches sich aber im Laufe der Jahre wesentlich weiter entwickelt hat.

Unix steht in der Regel synonym für offene Systeme und Hardware Unabhängigkeit.
In Wahrheit ist dies nur eine Scheinunabhängigkeit: die einzelnen Versionen von Unix (HP-UX, DEC ULTRIX, Sun Solaris, ...) sind nicht kompatibel! Eine Anwendung, die auf einer Variante läuft, kann durchaus auf einem anderen Unix nicht laufen.

7.7.3. Numerische Software Inkompatibilität

Wichtig bei numerischer Software ist die Wortlänge : 16 Bit oder 32 Bit oder ...
Einige Programmiersprachen verwenden interne Darstellungen der Zielplatform. Das hat zur Folge, dass die Software nicht mehr portabel ist.

In Java wurde dieses Beispiel mit Hilfe einer strikten Definition der primitiven Datentypen geregelt. Eine int Variable muss immer als 32 Bit Zahl implementiert werden.

7.7.4. Compiler Inkompatibilitäten

Das Beispiel für inkompatible Compiler ist die Programmiersprache C++. Die Sprache ist eher schlecht als recht definiert und erlaubt (bekannte) Seiteneffekte:

i+ = i++ + ++i; liefert je nachdem ob von links oder von rechts aufgelöst wird, je ein anderes Ergebnis.

7.8. Warum Portabilität?

Die Grundfrage : Warum soll Software überhaupt portabel sein?
Ist es nicht Zeitverschwendung und Geldverschwendung, die Portabilität in ein Produkt hinein zu entwickeln?

Die Antwort ist ein klares : NEIN!

Warum?
Komplexe Software lebt oft länger als die Hardware. Wir müssen also damit leben, dass unsere Software im Extremfall 30 Jahre im Einsatz bleibt, wie dies mit der IBM S/360 Software geschieht.

7.9. Techniken zum Erreichen der Portabilität

Eine Variante, Portabilität zu erreichen, besteht darin, dass man den Programmieren vorschreibt, welche Elemente einer Programmiersprache eingesetzt werden dürfen.

7.9.1. Portable System Software

Analog kann man bei den Betriebssystemen vorgehen : POSIX ist eine standardisierte Version von Unix. Wenn wir also POSIX konform entwickeln, dann sollte die Software auf allen UNIX ähnlichen Betriebsystemen lauffähig sein, sofern es sich nur um Systemkompatibilität handelt.

7.9.2. Portable Anwendungssoftware

Anwendungssoftware hat im Rahmen des Layer Architektur Patterns zwei benachbarte Layer:
1. der System Software Layer
2. der GUI Layer

POSIX kann als standardisierte Schnittstelle zum Layer System Software benutzt werden; für den GUI Layer wird man sich eine Abstraktion wie im Beispiel weiter vorne bauen müssen (abstrakte Factories).

7.9.3. Portable Daten

Da viele Programmiersprachen die Daten unterschiedlich speichern, muss man bei Anwendungen, in denen verschiedene Programmiersprachen eingesetzt werden, auf sinnvolle Datenformate einigen.

In Java ist dies der Unicode, allgemeiner würde man wohl auf Character Daten setzen.

7.10. Interoperabilität

Betrachten wir als erstes ein einfaches Beispiel:
In vielen Dokumenten finden Sie komplexe Berechnungen, die in der Regel mit Spreadsheets gemacht werden. Diese werden dann mit "cut and paste" in ein Word Dokument übertragen.

Mögliche Optimierungen könnten wie folgt aussehen:

1. immer wenn eine Änderung im Spreadsheet gemacht wird, wird diese Änderung auch im Word Dokument automatisch nach geführt. Dabei muss noch erwähnt werden, dass als Spreadsheet Lotus 1-2-3, also nicht Excel, eingesetzt wird.

2. dieser Update müsste automatisch geschehen, ohne dass der Word Nutzer das Dokument neu öffnen und drucken muss.

Das Beispiel zeigt, wie wichtig Interoperabilität ist!

Verschiedene Techniken wurden vorgestellt, mit deren Hilfe interoperable Systeme geschaffen werden können:
1. OLE/COM/DCOM/ActiveX
2. CORBA
3. RMI (nur Java)
4. Jini (nur Java)

Sie werden diese Techniken im nächsten Studienjahr kennen lernen. Hier geben wir lediglich eine kurze Übersicht.

7.10.1. OLE, COM und ActiveX

Die Entwicklung von OLE geht zurück auf die Anfänge der Windows basierten Anwendungen, speziell den MacIntosh. Plötzlich wurde es möglich, Daten aus einer Applikation in Daten einer anderen Applikation einzubetten. Dazu wurde zuerst ein spezielles Datenformat verwendet, das Clipboard Format. Später ging man dazu über die Objektreferenz mit zu geben, also anzugeben mit welcher Applikation die Daten bearbeitet werden können.
Microsoft veröffentlichte 1990 eine erste Version von OLE = "Object Linking and Embedding".
Als Weiterentwicklung veröffentlichte Microsoft das Component Object Model (COM), um eine bessere Interoperabilität erreichen zu können.
1996 startete Microsoft die ActiveX Initiative, um auch Internet basierte Technologien berücksichtigen zu können. Mit der Zeit wurde ActiveX ein Synonym für OLE und COM.
Seit 1996 unterstützt Microsoft auf verteilte Komponenten, DCOM = distributed COM.

COM ist eigentlich nicht Objekt orientiert. Ein COM Objekt kann man aber als Instanz einer Klasse auffassen. Diese Klasse kennt allerdings kein typisches OO Verhalten : Vererbung etc ist nicht möglich.

Man könnte von COM als einer Objekt basierten, nicht Objekt orientierten Technologie sprechen.

7.10.2. CORBA

1989 wurde im Rahmen der OMG (Object Management Group), einem Konsortium von etwa 500 Mitgliedern (Software und Hardware Hersteller, Grosskonzerne) eine einheitliche Architektur, CORBA = Common Object Request Broker definiert und veröffentlicht.

CORBA wird auch als Software Bus bezeichnet:
CORBA erlaubt es dem Programmierer, Applikationen mit Hilfe einer Schnittstellen Beschreibungssprache IDL = Interface Definition Language zu definieren und einem Broker (das B in CORBA) bekannt zu geben.
Sucht nun eine andere Applikation oder ein anderes Objekt einen Dienst, dann kann es beim Broker nachfragen, ob der ein Objekt (oder eine gewrappte Applikation [gewrapped bedeutet hier : CORBA fähig gemacht]) kennt, welche die Aufgabe übernehmen kann.

Es gibt kommerzielle ORBs zum Beispiel von Orbix, einer Irischen Software Firma. Aber verschiedene ORBs sind als Shareware erhältlich. Java wird bereits mit einem ORB geliefert (JDK1.2 oder jünger).

CORBA ist eine Middleware, die es erlaubt Anwendungen, die in verschiedenen Programmiersprachen geschrieben sind, miteinander zu kommunizieren. CORBA kennt dafür ein sogenanntes Language Mapping, eine Abbildung der Programmiersprach- Konzepte auf die ORB Notation.

Jeder Client muss einen sogenannten Stub implementieren; jeder Server muss den entsprechenden Skeleton implementieren. Stub und Skeleton sind Programmklassen, die vom IDL Compiler generiert werden.

Das praktische Vorgehen zur Implementation einer CORBA basierten Applikation sieht etwa folgendermassen aus (Details werden Sie im Fach Parallele und Verteilte Systeme kennen lernen):

1. Definition der Schnittstellen mit Hilfe von IDL
   compilieren der IDL Beschreibung
   Ergebnis : Stub und Skeleton Beschreibung in der Ziel-Programmiersprache
   Im Falle von C / C++ : je ein h File
2. Entwickeln der Applikation unter Einbezug der Stub und Skeleton Beschribung

CORBA Applikationen wirft man vor, langsam und "overkilled" zu sein. Das stimmt aber nur bedingt: die Billet Automaten von ASCOM verfügen über einen CORBA Software Bus, um ältere Applikationen mit neuen kombinieren zu können (zumindest im ASCOM Forschungslabor: mir ist nicht bekannt, ab diese Systeme auch in den Bahnhöfen zum Einsatz kommen).

7.10.3. Vergleich von CORBA und OLE/COM

CORBA ist eine universelle Technologie :

1. es werden unterschiedliche Programmiersprachen

2. es werden unterschiedliche Hardware und System Software Systeme unterstützt.

OLE /COM ist auf Microsoft beschränkt. Allerdings wurde die Technologie von der deutschen Software AG auch auf andere Betriebsysteme portiert.

Der Nachteil einer proprietären Technologie wie OLE/COM versus einer öffentlichen Technologie wie CORBA dürfte klar sein:
im Falle von OLE/COM kann Microsoft jederzeit die Spezifikation, die heute noch unvollständig ist, ergänzen oder abändern.

Führende Hersteller, wie ORBIX schafften jedoch sogenannte Bridges zwischen OLE/COM und CORBA.

Sie finden zu OLE/COM und speziell CORBA im Internet jede Menge Unterlagen. Die Programmierung mit OLE / COM oder CORBA ist jedoch nicht einfach!

7.11. Aufgaben

1 Welchen Einfluss hat cohesion auf Wiederverwendung?
2 Welchen Einfluss hat coupling auf die Wiederverwendung von Komponenten?
3 Ihre Aufgabe ist es, eine Software Komponente zu entwickeln, die möglichst universell einsetzbar sein sollte und ein Bankkonto prüfen soll. Ihnen stehen folgende Daten zur Verfügung:
Kontostand am Monatsanfang
die Anzahl Checks, inkl. Betrag und Datum der Gutschrift
jede Auszahlung
jede Einzahlung
den Kontostand am Ende des Monates
Erläutern Sie, wie Sie erreichen möchten, dass so viele Module wie möglich Ihr Modul einsetzen.
4 Sie wurden Mitglied eines grossen Software Entwicklungs-Teams. Die bisherigen Applikationen umfassen 95'000 COBOL Module. Was schlagen Sie vor, um die Wiederverwendung zu maximieren?
5 Welche Teile der Case Study können so entwickelt werden, dass eine Wiederverwendung in anderen Projekten möglich wird?
6 Lesen Sie den Artikel. Betrand Meyer: "Lessons from the Design of the Eiffel Libraries" *Communications of the ACM* **33**, (Sept. 1990) pp 68 - 88
7 Beschaffen Sie sich den Artikel
"Using Design Patterns to Develop Reusable Object Oriented Communications Software" von D.C.Schmidt, veröffentlicht in Communications of the ACM 38 (October 1995) pp 65-74
Stimmen Sie mit der Ansicht von Schmidt überein, dass Pattern die Wiederverwendung von Software fürdert?

7. WIEDERVERWENBARKEIT 1

7.1. KONZEPTE 1
7.2. HINDERNISSE FÜR DIE WIEDERVERWENDUNG 2
7.3. WIEDERVERWENDBARE SOFTWARE - CASE STUDIES 2
7.3.1. Raytheon Missile Systems Division 2
7.3.2. Toshiba Software Factory 4
7.3.3. NASA Software 4
7.3.4. GTE Data Services 5
7.3.5. Hewlett Packard 5
7.3.6. Zusammenfassendes Ergebnis der Case Studies: 6
7.4. OBJEKTE UND WIEDERVERWENDBARKEIT 6
7.5. WIEDERVERWENDUNG IN DER DESIGN UND DER IMPLEMENTATIONSPHASE 7
7.5.1. Module Wiederverwendung 7
7.5.2. Anwendungs Frameworks 8
7.5.3. Design Patterns 8
7.5.4. Software Architektur 11
7.6. WARTUNG UND WIEDERVERWENDUNG 12
7.7. PORTABILITÄT 13
7.7.1. Hardware Inkompatibilitäten 13
7.7.2. Betriebssystem Inkompatibilitäten 13
7.7.3. Numerische Software Inkompatibilität 13
7.7.4. Compiler Inkompatibilitäten 14
7.8. WARUM PORTABILITÄT? 14
7.9. TECHNIKEN ZUM ERREICHEN DER PORTABILITÄT 14
7.9.1. Portable System Software 14
7.9.2. Portable Anwendungssoftware 14
7.9.3. Portable Daten 14
7.10. INTEROPERABILITÄT 15
7.10.1. OLE, COM und ActiveX 15
7.10.2. CORBA 16
7.10.3. Vergleich von CORBA und OLE/COM 16
7.11. AUFGABEN 18