SOFTWARE ENGINEERING

1. Einführung in das Software Engineering

1.1. Ein Beispiel aus dem Alltag:

Jemand erhält eine Rechnung über 0.00CHF. Er amüsiert sich darüber und schmeisst die Rechnung in den Papierkorb.
Nach vier Wochen erhält er eine erste Mahnung. Er ist leicht verärgert, unternimmt aber erneut nichts. Sie können sich das weitere Szenario leicht vorstellen.
Nach zwei Monaten und einer sehr unfreundlichen Mahnung, mit der Aufforderung, umgehend den Betrag zu überweisen, da sonst rechtliche Schritte eingeleitet werden, diskutiert der betroffene das Problem mit einem Software Ingenieur. Dieser rät ihm, einen Check über 0.00 CHF an die Firma zu überweisen.
Nach der Überweisung hören die Beschwerden auf. Das Problem ist also scheinbar gelöst.
Als der Betroffene das nächste Mal mit seiner Bank telefoniert, fragt in der zuständige Sachbearbeiter der Bank, nicht des Rechnungstellers, ob er sich eigentlich bewusst sei, was sein Check bei ihnen und in ihren Informationssystemen zur Folge gehabt habe.

1.1.1. Ein (wahres) Beispiel aus dem Verteidigungsbereich

Sie alle kennen den Film „War Games“. Ungefähr fünf Jahre vor dem Film ereignete sich Folgendes:
Plötzlich schlugen die automatischen Überwachungsanlagen des „Startegic Air Command“, das „World-Wide Military Monitoring Command and Control System“ (WWMCCS) Alarm:
Gemäss dem WWMCCS hatte die Sowjetunion Raketen gestartet, mit Ziel USA.
In Wahrheit hatte die Überwachungssoftware aus was für Gründen auch immer, Simulationsdaten als echte Daten interpretiert und entsprechende Aktionen gestartet!

1.1.1. Von Spitälern und Golf Krieg

Jedes Jahr sterben Patienten, weil die medizinischen Geräte in Spitälern versagen. In der Literatur wird von mehreren Fällen berichtet, bei denen Patienten an einer Strahlen-Überdosis geschädigt wurden, weil die entsprechende Steuerungssoftware fehlerhaft war.

Im Golfkrieg musste die Software der „Patriot“ Raketen im Felde auf einen neuen Stand gebracht werden, weil die Tests der Software über einen zu kurzen Zeitraum liefen, und im Feld die Raketen über einen längeren Zeitraum im Einsatz waren als gemäss Software-Tests erwartet wurde. Dies führte zu dazu, dass ein „Scud“ Angriff der Saudis nicht bezw nicht rechtzeitig abgewehrt werden konnte. Bilanz: mehr als 20 Tote!

1.2. Geschichte des Software Engineerings

1967 hat die NATO einen Konferenz einberufen zum Thema „Software Engineering“. Der Auslöser waren grosse Probleme beim Entwickeln strategischer Software-Systeme.
Ziel dieser Konferenz (in Garmisch) war es, die Qualität der Software zu verbessern.
Ausgangspunkt war, dass andere Engineering Disziplinen in der Lage waren, zuverlässige Produkte zu liefern. Als Beispiel sei der Brückenbau erwähnt: es ist sicher undenkbar, eine Brücke „neu zu booten“, wenn sie zusammen gekracht ist, weil der Planungs- oder der Berechnungs-Ingenieur versagt hat.

© J.M.Joller

Es ist allerdings kein besonders gutes Beispiel! Brücken werden seit vielen Jahrhunderten gebaut und vermutlich sind früher auch viele Brücken zusammen gebrochen. Aber das Wissen, wie eine Brücke gebaut werden muss, ist in der Zwischenzeit gefestigt. Falls eine Brücke Risse oder ein anderes ungewöhnliches Verhalten zeigt, wird sofort intensiv untersucht, was der Grund dafür ist. Die Erkenntnisse fliessen ein in neue Standards und alte Brücken müssen gegebenenfalls nachgearbeitet werden.

Software beschreibt diskrete Systeme, mit Zuständen, die in der Regel binär beschrieben werden können. Brücken werden klassisch berechnet, mit Hilfe analytischer Methoden. Software-Probleme entstehen oft, weil das Software-System Zustände annimmt, die der Software-Entwickler nicht eingeplant hatte und auch nicht getestet hat.

Die Wartung einer Brücke und eines Software-Systems unterscheiden sich ebenfalls fundamental. Während die Wartung einer Brücke in der Regel mit einem Neuanstrich erledigt ist, umfasst die Wartung der Software in der Regel auch Erweiterungen, Portierungen und die Behebung von Fehlern. Es ist nicht unüblich, dass nach 5 Jahren 50% eines Software Systems ersetzt worden sind, wegen zusätzlicher, geänderten Anforderungen oder wegen Fehler - Behebung, oder einem teilweise Redesigns (Benutzeroberflächen müssen an neue Anforderungen und neue Entwicklungen angepasst werden, Files werden durch Datenbanken ersetzt, ...).

1.3. Wirtschaftliche Aspekte

Einer der Gründe, für neue und neuartige Entwicklungen im Software Bereich, sind neue Erkenntnisse aus der Informatik-Forschung, die nebenbei bemerkt fast zu 100% in der Industrie, also zu fast 0% an den Hochschulen geschieht (oder kennen Sie einen Mikroprozessor, ein Betriebsystem, eine Datenbank oder eine betriebswirtschaftliche der technische Anwendung, die an einer Hochschule entstand?).
Solche Erkenntnisse werden in der Regel von der Software-Gemeinde mit Begeisterung aufgenommen, wenn zum Teil auch nicht verstanden.

Betrachten wir nun die Auswirkung einer neuen Entwicklungsmethodik für die Erstellung von Software, EM2. Diese erlaubt es, die Software um 10% schneller als mit der bestehenden Methodik EM1 zu erstellen.

In Manager, der diese Daten in einem Artikel der „Business Week“ auf seinem Flug aus den Ferien zurück an seinen Arbeitsplatz liest, reagiert sofort:
In seiner Firma werden in Zukunft alle Software-Systeme nach der EM2 entwickelt. Alle Software Entwickler werden geschult, die neue Methodik wird systematisch eingeführt und siehe da:
Die Produktivität des Entwicklungsteams steigt um 10%, nach einer Einführungs- und Schulungsphase von 15 Monaten.

Später zeigt es sich aber, dass die Wartung der Software, die mit der neuen Methodik EM2 entwickelt wurde, wesentlich komplexer ist und zu einem höheren Wartungsaufwand führt.

War der Management-Entscheid für die neue Methodik EM2 ein Fehlentscheid?

© J.M.Joller

1.4. Wartungsaspekte

Bevor wir uns mit dem Thema Wartung befassen, müssen wir uns klar werden, welche Phasen eine Software durchläuft!

Die klassischen Phasen eines Software-Systems sind die Folgenden:

1. Requirements phase (Anforderungs-Phase)
2. Specification phase (Spezifikations-Phase)
3. Planing phase (Planungs-Phase)
4. Design phase (Entwurfs-Phase)
5. Implementation phase (Implementierungs-Phase)
6. Integration phase (Integrations-Phase)
7. Maintenance phase (Wartungs-Phase)
8. Retirement (Ablösungs-Phase)

Jede Entwicklungs-Methodik verwendet in der Regel eine leicht modifizierte Version dieser Phasen. Phasen können auch weiter unterteilt werden, es können zusätzliche Phasen eingefügt werden (z.B. Modultest-Phase, Systemtest-Phase). Aber in Grossen und Ganzen orientieren sich alle gängigen Modelle an diesen Phasen.

Bis Ende der 70er Jahre verwendeten die meisten Firmen das sogn. Waterfall / Wasserfall-Modell, auf das wir noch eingehen werden.

1.4.1. Was geschieht (grob) in den einzelnen Phasen?

Requirements Phase	Die Wünsche und Vorstellungen des Kunden werden untersucht und falls möglich formalisiert.
Specification (Analysis) Phase	Die Kundenbedürfnisse werden analysiert und in Form eines *Spezifikations-Dokument* festgehalten. Die Spezifikation beschreibt das „WAS“, nicht das „WIE“!
Planning Phase	Der *„Software Projekt Management Plan“* wird erstellt. Er beschreibt den Ablauf der Software Erstellung im Detail.
Design Phase	In der Regel unterscheidet man zwei Ebenen des Designs: den *„Architekur Design“* und den *„Module Design“*. Im Architektur Design wird das System in (möglichst unabhängige) Module zerlegt, die dann im Modul-Design weiter verfeinert werden.
Implementation Phase	In dieser Phase findet die eigentliche Programmierung statt.
Integration Phase	Die einzeln entwickelten und möglichst unabhängigen Module müssen zu einem Gesamtsystem zusammen gefügt und getestet werden. Das Testen geschieht in der Regel in mehreren Phasen: im Alpha Test wird die Grundfunktionalität getestet, in der Regel nicht durch den Benutzer; im Beta Test wird dem zukünftigen Benutzer ein System zur

© J.M.Joller

	Verfügung gestellt, welches in etwa dem endgültigen Produkt entspricht: „die Software reift beim Kunden“ (Bananen-Software). Der Kunde muss in der Regel Abnahmetests („acceptance tests“) durchführen und dem Ersteller bestätigen, dass er mit dem Ergebnis zufrieden ist.
Maintenance Phase	Nachdem der Kunde das Produkt abgenommen hat, werden verschiedene Aktivitäten nötig: *„corrective maintenance“* (korrektive Wartung) dient dem Eliminieren der allzu offensichtlichen Fehler; *„enhancements“* dient der Erweiterung des bestehenden Systems; *„perfective maintenance“* führt zu Änderungen, die entweder der Kunde wünscht, oder die aus Sicht des Kunden zu einer Verbesserung führen *„adaptive maintenance“* umfasst Änderungen, die durch ein geändertes Umfeld nötig werden (neue Steuerbedingungen,...). Statistisch gesehen verteilt sich die Wartung etwa wie folgt: 17.5% Korrekturen, 60.5% Perfektion, 18% Adaption.
Retirement Phase	Das (Software) Produkt wird aus dem Betrieb genommen.

1.4.1.1. Achtung:

in der Regel ist die Wartung nicht ein Kennzeichen für schlechte Software; im Gegenteil: wenn die Software unbrauchbar wäre, würde sie einfach zur „Shelfware“ dh niemand würde sie mehr einsetzen. Gewartete Software ist in der Regel also durchaus Software, die auch praxisrelevant ist!

Der Wartungsaufwand kann beträchtlich variieren, je nachdem wie gut oder wie schlecht die Software geschrieben ist.

© J.M.Joller

1.4.2. Betrachten wir dazu ein Beispiel:

Die Software „Abaka“ ist eine Software für die Betriebsbuchhaltung. Sie ist zu 100% in Java geschrieben und enthält in einem Modul den aktuellen Steuerprozentsatz fest:

```
public static final float verkaufsSteuer = (float)6.0;
```

Wenn nun der Prozentsatz von 6% auf 7% erhöht wird, muss der Programmierer in diesen Modul eingreifen und die Konstante ändern!

Dumm wäre der Programmierer, der einfach mit „ersetzen“ alle 6.0 durch 7.0 ersetzt.

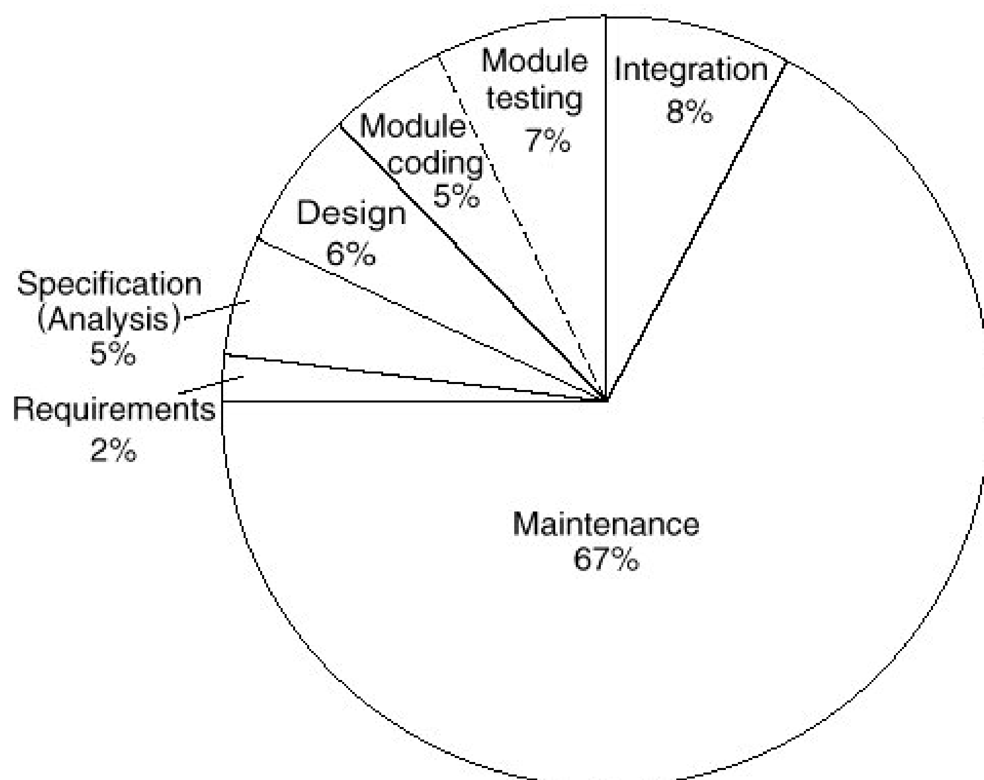
Was wären die Auswirkungen?
Im schlimmsten Fall würde irgend eine Division durch 6.0 in Zukunft fälschlicherweise durch 7.0 erfolgen!

Wir müssen in unserer Software also Wege finden, wie wir evtl. „variable Konstanten“ besser behandeln könnten. Ein Weg wäre sicher die Auslagerung solcher „Konstanten“ in eine Initialisierungsdatei.

1.4.3. Wie gross ist nun aber der Wartungsaufwand?

Diese Frage ist für Sie überlebenswichtig, wenn Sie Ihre Software verkaufen und „am Leben“ erhalten möchten. Der Wartungsaufwand legt die Wartungsgebühren fest. Typischerweise betragen diese heute für kommerzielle Software zwischen 11% und 19% des Lizenzpreises.

Die Frage wurde sehr systematisch untersucht, unter anderen von IBM, von TRW (speziell im Space Sektor) und wurden systematisch ausgewertet.



© J.M.Joller

Betrachten wir nochmals unser Beispiel mit den unterschiedlichen Entwicklungsmethodiken EM1 und EM2. EM2 führt zu einem um 10% reduzierten Entwicklungsaufwand in der Codierungsphase.

Unter der Voraussetzung, dass dadurch keine andere Phase betroffen ist, wird der Manager kaum auf die Methodik 2 umschwenken, da der Schulungsaufwand schlicht zu gross ist und mit dem Risiko behaftet ist, dass die Produktivität in der Umstellungsphase unter eine kritische Grösse absinken könnte!

Da die Entwicklungsphase lediglich 5% des gesamten „Lebensaufwandes" ausmacht, ist die Verbesserung unerheblich!

Betrachten wir nun die andersartige Entwicklungsmethodik EM3, die zu einer Reduktion des **Wartungsaufwandes** um 10% führt.
Diese Methodik müsste sofort eingesetzt werden, da der Nutzen über den gesamten Lebenszyklus signifikant ist, in Zahlen 6.7% (10% von 76% Wartungsaufwand)!

Wir erkennen aus dem Beispiel :

Die Wartungsphase spielt eine zentrale Rolle beim Bewerten der Praxisrelevanz einer Methodik!

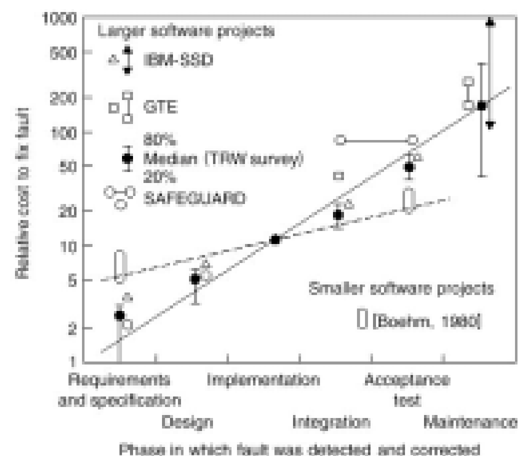
1.5. Spezifikation und Design Aspekte

Dass ein Entwickler nicht fehlerfrei arbeitet ist normal und kaum erstaunlich, wenn man bedenkt, wie komplex heutige Software Systeme sind (wir sprechen von realen Software Systemen mit mehreren 100'000 Zeilen Code, nicht von Demo - Beispielen mit einigen Duzend Zeilen Programmcode).

Da die Software Entwicklung aus unterschiedlichen Phasen zusammen gesetzt ist, müssen wir uns überlegen, in welchen Phasen wir besonders intensiv Fehler vermeiden sollten. Ein Tippfehler in einer Programmzeile kann eventuell mit Hilfe des Compilers gefunden werden; aber ein Designfehler kann auch vom besten Test - Tool nicht erkannt werden!

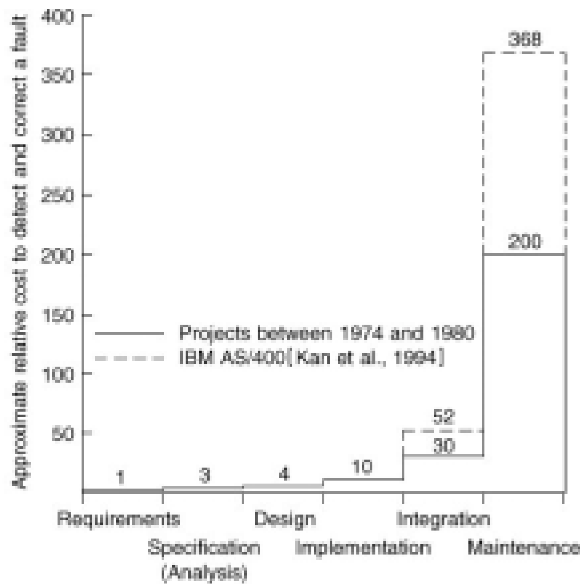
Auch diese Frage wurde in der Literatur ausführlich diskutiert:

Die nebenstehende Grafik zeigt, eine ältere Statistik.



© J.M.Joller

In der nächsten Grafik sehen Sie entsprechende Zahlen, die bei der Entwicklung des Betriebssystems der IBM AS/400 (dem OS/400) erfasst wurden.



Interpretationsbeispiel:
Ein Fehler in der Spezifikationsphase, der mit minimalem Aufwand (20 USD) behoben werden kann, kostet in der Wartungsphase 2'000 USD.

1.6. Team Aspekte

Betrachten wir zuerst ein negatives Beispiel:

1.6.1. Modulare Programmierung

Zwei Programmierer beschliessen, zusammen eine neuartige Software zu entwickeln. Damit die Realisierung schneller voran geht, legen sie bestimmte Schnittstellen fest und starten dann unabhängig voneinander mit der Implementierung der Module.

Nun hat sich aber ein Missverständnis eingeschlichen! Programmierer 2 vertauscht die Reihenfolge der Parameter ein einem seiner Module. Da die Parameter von gleichem Typus sind, entdeckt der Compiler den Fehler nicht.

Sie sehen: wenn dieses Team eine entsprechende Programmiersprache gewählt hätte, die den Teamaspekt berücksichtigt, dann wäre der Fehler nicht passiert! Natürlich kann das Problem auch mit andern Hilfsmitteln, die wir z.T. noch kennen lernen werden, vermieden werden!

1.6.2. Teamgrösse

Eine Faustregel für Projektleiter besagt: „wenn Du in einem Projekt verspätest bist, dann kannst Du die Verspätung erhöhen, indem Du noch mehr Programmierer einsetzt!“.

Gründe dafür:
Durch die zusätzlichen Programmierer wird die Kommunikation komplexer, die neuen Mitarbeiter müssen sich einarbeiten, informelle Vereinbarungen müssen durch formal saubere ersetzt werden, ...

© J.M.Joller

1.7. Das Objekt Orientierte Paradigma

Seit der NATO „Software Engineering“ Konferenz 1967 hat man signifikante Fortschritte gemacht:
In einer ersten Phase wurden, etwa zwischen 1975 und 1985, die strukturierten Methoden, das sogenannte „Strukturierte Programmieren Paradigma“ vorangetrieben und systematisch untersucht.

Die Resultate sind unter anderem in den Programmiersprachen PASCAL und MODULA zu finden.

Diese Methoden zeigten einen beachtlichen Erfolgt; aber konnten viele Probleme auch nicht lösen. Insbesondere bei sehr komplexen Software-Systemen, wie sie heute üblich sind, mit mehreren 100'000 bis zu mehreren 1000000 Programmzeilen, zeigen sich Schwächen der Methode (wahrscheinlich haben andere Methoden die selbe und / oder andere Schwächen!).

Das Paradigma der Strukturierten Programmierung führte zu wesentlichen Verbesserungen in der Software-Industrie, sofern diese konsequent angewandt wurden.

Strukturierte Methoden wurden / werden normalerweise in „datenorientierte“ und „prozessorierte“ Methoden eingeteilt. Beide beschränken sich nicht auf den einen oder den andern Aspekt; aber die Daten oder die Prozesse stehen im Vordergrund.

Die objektorientierte Sicht, das objektorientierte Paradigma, versucht, beide Aspekte (Daten und Prozesse) zu verbinden.

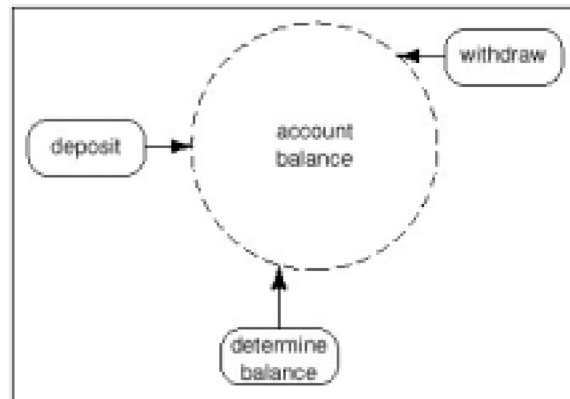
Betrachten wir ein Beispiel, aus dem die unterschiedlichen Betrachtungsweisen deutlich werden:

© J.M.Joller

1.7.1. Problemstellung:

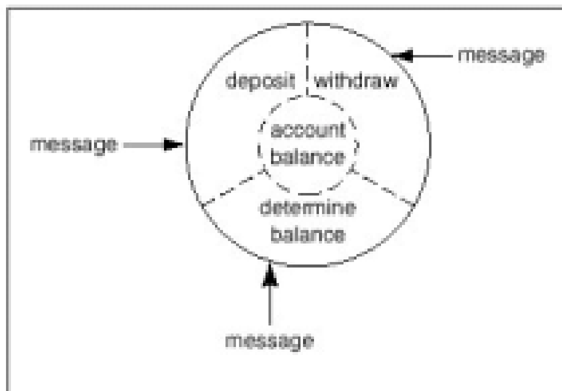
Wir betrachten ein Bankkonto. Typischerweise können wir einem Bankkonto folgende Operationen oder Prozesse zuordnen:

1. Einzahlen
2. Abheben
3. Kontostand abfragen



Betrachten wir nun eine Implementierung nach den Grundsätzen der strukturierten Programmierung:

Wir haben einen Datensatz und „unabhängige“ Funktionen, in der Abbildung mit „deposit“ (einzahlen), „withdraw“ (abheben) und „determine balance“ (Kontostand ermitteln).



Eine Implementierung mit Hilfe objektorientierter Techniken (im „Objektorientierten Paradigma“) sieht wesentlich anders aus, aber für viele Benutzer eher unklarer!
An Stelle von Funktionsaufrufen („deposit“) treten Meldungen an das Objekt „Bankkonto“. Der Benutzer weiss nicht und muss auch nicht wissen, wie die „Methode“ („deposit“, „withdraw“ und determineBalance“) implementiert sind. Die Implementierung kann sich im Verlaufe der Zeit ändern!

Für die Wartungsphase impliziert diese Änderung (verschieben des Programmcodes zu den Daten; Dezentralisierung der Prozesslogik), dass der Wartungsprogrammierer nicht mehr das System als Ganzes beherrschen muss; er sollte sich auf seine lokale Umgebung konzentrieren können!

Die Konsequenzen des OO Paradigmas sind sehr vielfältig:

1. Objekte werden unabhängiger voneinander, ein gutes Design vorausgesetzt!
2. Alles was zu einem Objekt gehört wird gewissermassen „gekappselt“(„encapsulated“), in das Objekt eingeschlossen.
3. Implementationsdetails bleiben verborgen. Andere Objekte kommunizieren mit dem Objekt mit Hilfe wohldefinierter Meldungen. Dies führt zum sogn. „Responsibility Driven

© J.M.Joller

Design“ oder einem „Design by Contract“. Wie etwas gemacht wird, wird an das Objekt bezw. dessen Methoden „delegiert“

4. Das OO Paradigma ermöglicht eine verstärkte Wiederverwendung von bestehenden Objekten, da diese relativ unabhängig voneinander sind.
5. Der OO Lebenszyklus muss den neuen Möglichkeiten angepasst werden. Objekte entstehen sehr früh in der Spezifikationsphase und werden laufend verfeinert

1.7.2. Vergleich der Entwicklungsphasen „Strukturiertes versus OO Paradigma“

	Strukturiertes Paradigma	Objekt Orientiertes Paradigma
1.	Requirement Phase	Requirement Phase
2.	Specification (Analysis) Phase : WAS	Object Oriented Analysis Phase
3.	Planning Phase	Planning Phase
4.	Design Phase : WIE	Object Oriented Design Phase
5.	Implementation Phase	Object Oriented Programming Phase
6.	Integration Phase	Integration Phase
7.	Maintenance Phase	Maintenance Phase
8.	Retirement	Retirement

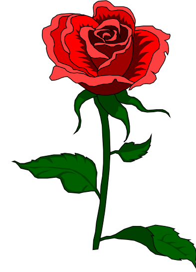
Die Änderungen im Phasenmodell sind sehr wesentlich und dürfen auf keinen Fall unterschätzt werden:
Beim Vorgehen nach dem Strukturierten Paradigma war der Übergang von der Spezifikation und der Design Phase in der Regel recht abrupt. Fehler in der Analyse führten somit in der Regel zu gravierenden Konsequenzen in den späteren Phasen.

Im Rahmen des OO Paradigmas wird bereits sehr früh die Objektstruktur des Systems erarbeitet. In späteren Phasen können weitere Objekte dazu kommen, ohne dass der Systemdesign völlig überarbeitet werden muss!

	Strukturiertes Paradigma	Objekt Orientiertes Paradigma
2	Spezifikations (Analyse) Phase	Objekt Orientierte Analyse Phase
	- bestimme, was das Produkt tun muss (WAS)	- bestimme, was das Produkt tun muss (WAS)
		- Extrahiere (bestimme) mögliche Objekte
4	Design Phase	- Objekt-Orientierte Design Phase
	- Architektur Design : bestimme die Module	- Detaillierter Design
5	Implementations Phase	Objekt-Orientierte Programmier-Phase
	- Implementiere die Module in einer angepassten Programmiersprache	- Implementiere die Objekte in einer angepassten Objekt - orientierten Programmiersprache

© J.M.Joller

1.7.3. Selbsttestaufgabe:



Sie möchten Ihrer Grossmutter in einem etwas weiter entfernten Ort zu ihrem Geburtstag einen Blumenstrauss schicken. Da Sie davon ausgehen, dass Blumen schlecht per Post versendet werden können, beschliessen Sie, einen Blumenladen damit zu beauftragen, die Blumen Ihrer Grossmutter zuzustellen.
Sie haben jetzt zwei Möglichkeiten

- Sie versuchen einen Blumenladen in der Name des Wohnortes Ihrer Grossmutter zu finden und beauftragen diesen Laden
- Sie gehen zum Blumenladen in Ihrer Wohngegend und beauftragen diesen, Ihrer Grossmutter einen Blumenstrauss zukommen zu lassen.

Beschreiben Sie für beide Fälle Ihr Vorgehen in Form eines Algorithmus.
Welche der beiden Möglichkeiten ist „Objekt - orientierter"?

© J.M.Joller

1.8. Terminologie

1.8.1. Was ist ein System und was ist ein Programm?

Die Begriffe sind eigentlich klar:
Ein System kann aus Hardware und Software (und evtl. weiteren Teilsystemen) bestehen.

1.8.2. Der Lebenszyklus der Software:

Er besteht primär aus einer Entstehungs(makro)phase und einer Wartungsphase, wobei die Entstehungsphase wie bereits besprochen weiter unterteilt werden kann.

1.8.3. Was ist der Unterschied zwischen Methode und Paradigma?

Eine Methode beschreibt eine Vorgehensweise.
Ein Paradigma beschreibt ein Muster oder ein Modell.

1.8.4. Sprachenvielfalt und Terminologievielfalt

In der Literatur ist, speziell im OO Bereich, die Terminologie sehr vielfältig und uneinheitlich.
Je nach Programmiersprache verwendet man unterschiedliche Begriffe für ein und dasselbe.

Datenkomponenten:
Diese werden als Attribute, Data Members, Felder, Instanzenvariablen .... bezeichnet.

Methodenkomponenten:
Diese werden als Messages, Methoden, Function Members, ... bezeichnet.

© J.M.Joller

1.9. Übungen

- Sie sollen für eine grössere Firma ein Software- Paket entwickeln, welches alle Buchhaltungsfragen beantworten kann. Die Firma ist bereit Ihnen dafür 350'000 CHF zu bezahlen, pauschal. Aber der Auftrag ist an mehrere Bedingungen gebunden: Sie müssen das Paket zeitlich fristgerecht und im Rahmen des Budgets realisieren und implementieren. Beschreiben Sie, wieviel Geld Sie jeder Entwicklungsphase (also ohne Wartung!) zuordnen!
- Sie haben gerade Ihre eigene Firma gestartet und erhalten von der mittelständischen Firma (KMU) „Kartoffelstock“ den Auftrag, die Firma auf den Stand der Technik zu bringen. Der Kunde möchte die aktuellen Aufträge verfolgen können, die Buchhaltung integrieren und Lohn und Gehalt damit abwickeln können.
  Der Geschäftsleiter stellt sich das Ganze etwa so vor:
  die 12 Mitarbeiter der Buchhaltung
  die 35 Mitarbeiter im Verkaufsinnendienst
  die 40 Mitarbeiter im zentralen Warenlager
  sowie 15 Gruppenleiter, Abteilungsleiter und Mitglieder der Geschäftsleitung benötigen Zugriff auf die Daten.
  Hardware und Software dürfen zusammen nicht mehr als 40'000 CHF kosten.
  Das Projekt muss innerhalb von drei Wochen abgeschlossen werden.
  Was würden Sie dem Geschäftsleiter raten, unter der Voraussetzung, dass Sie auf den Auftrag angewiesen sind!
- Sie haben sich nach Ihrem anstrengenden Studium nach Andorra zurück gezogen. Dort werden Sie verantwortlich für das Luftüberwachungssystem. Ihr Auftrag lautet: entwickeln Sie ein neues, modernes, ausbaufähiges Luftüberwachungssystem.
  Welche Klausel dürfen Sie in Ihrem Vertrag mit Andorra bezw. Ihrem Subkontractor, der für Sie die Software entwickeln wird, nicht vergessen?
- Beschreiben Sie aus der Sicht des Softwarehauses, welche Gründe zu einem Scheitern des obigen Projektes führen könnten.
- Sie haben Software für eine automatische Raffinerie entwickelt. In der Betriebsphase tritt ein kritischer Fehler auf, der unbedingt korrigiert werden muss! Die Korrektur kostet 24‘800CHF. Wieviel hätte die Entdeckung des Fehlers in der Designphase gekostet?
- Wieviel hätte die Behebung des der Fehlers in der Implementierungsphase gekostet?
- Schauen Sie in einem Lexikon nach, was man unter dem Begriff „System“ versteht.
- Sie müssen ein Finanz - und Rechnungswesen entwickeln. Welches Paradigma wenden Sie an? Objekt - orientiertes oder Strukturiertes Paradigma?

© J.M.Joller

1.10. Projekt

Falls Sie kein anderes Projekt haben:
Lesen Sie folgende Projektbeschreibung (Case Study) durch!
Wie würden Sie auf die Anforderung nach einer neuen Berechnungsformel reagieren:

- Alles von Grund auf neu entwickeln?
- Das alte System der neuen Formel anpassen?

© J.M.Joller

1.11. Literaturstudium

Beschaffen Sie sich folgenden Artikel:
A. Snyder: „The Essence of Objects : Concepts and Terms“
in IEEE Software Vol 7 Seiten 31-42, November 1990

Lesen Sie diesen Artikel und bereiten Sie eine kurze Zusammenfassung vor, die Sie in maximal 15 Minuten vortragen könnten!

© J.M.Joller

1. Einführung in das Software Engineering 1
1.1. Ein Beispiel aus dem Alltag: 1
1.1.1. Ein (wahres) Beispiel aus dem Verteidigungsbereich 1
1.1.1. Von Spitälern und Golf Krieg 1
1.2. Geschichte des Software Engineerings 1
1.3. Wirtschaftliche Aspekte 2
1.4. Wartungsaspekte 3
1.4.1. Was geschieht (grob) in den einzelnen Phasen? 3
1.4.2. Betrachten wir dazu ein Beispiel: 5
1.4.3. Wie gross ist nun aber der Wartungsaufwand? 5
1.5. Spezifikation und Design Aspekte 6
1.6. Team Aspekte 7
1.6.1. Modulare Programmierung 7
1.6.2. Teamgrösse 7
1.7. Das Objekt Orientierte Paradigma 8
1.7.1. Problemstellung: 9
1.7.2. Vergleich der Entwicklungsphasen „Strukturiertes versus OO Paradigma“ 10
1.7.3. Selbsttestaufgabe: 11
1.8. Terminologie 12
1.8.1. Was ist ein System und was ist ein Programm? 12
1.8.2. Der Lebenszyklus der Software: 12
1.8.3. Was ist der Unterschied zwischen Methode und Paradigma? 12
1.8.4. Sprachenvielfalt und Terminologievielfalt 12
1.9. Übungen 13
1.10. Projekt 14
1.11. Literaturstudium 15

© J.M.Joller