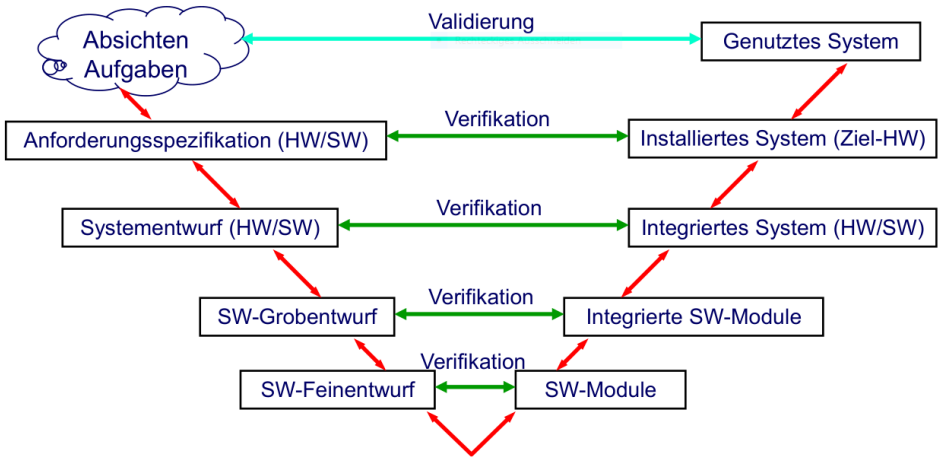
Zusammenfassung Software-Entwicklung in Großprojekten

von Marco Ammon

- Software-Fehler
  - Irrtum (mistake): Denkfalle, Unachtsamkeit (führt stets zu einem oder mehreren Fehlern)
  - Fehler (fault): Abweichung zwischen beabsichtigtem und realisiertem Produkt (kann zu fehlerhaftem Zustand oder Versagen führen)
  - fehlerhafter Zustand (error): Abweichung zwischen beabsichtigtem und realisiertem internen Zustand (kann zu Versagen führen) (zur Laufzeit)
  - Versagen (failure): Abweichung zwischen beabsichtigtem und realisiertem Verhalten (Offenbarung des Fehlers)
- Softwareprozess: Abfolge von Aktivitäten und daraus resultierenden Ergebnissen, die zur Herstellung eines Softwareprodukts führen
- Prozessmodell: vereinfachte Beschreibung eines Softwareprozesses
- Lebenszyklus:
  - Anforderungsphase (wozu?): z.B. Modell des Produkteinsatzes
  - Spezifikationsphase (was?): z.B. vertraglich verbindliche Anforderungen
  - Entwurfsphase (wie?): Grobentwurf, Feinentwurf
  - Implementationsphase
  - Integrationsphase
  - Installations-, Nutzungs-, Wartungs-, Ablösungsphase
- Wasserfall-Modell: direktes Vergleichen mit vorheriger Phase (aber Fehler fällt erst am Ende auf)
- V-Modell: Erweiterung des Wasserfall-Modells durch Verifikation (Übereinstimmung zwischen Produkt und Spezifikation) und Validierung (Übereinstimmung zwischen Produkt und Einsatzzweck):



- Anforderungsspezifikation:
  - Muss-, Soll- und Wunschanforderungen
  - Softwarespezifikation: Zusammenstellung der Anforderungen und Randbedingungen für Einsatz
  - Funktionale Anforderungen: erwünschtes Verhalten
  - Nichtfunktionale Anforderungen: z.B. einzuhaltende Normen, Hardware-Umgebung, Qualität, Effizienz

- Eigenschaften von Anforderungen:
  - Vollständigkeit (z.B. alle möglichen Eingabewerte und detaillierte Beschreibung der gewünschten Reaktion)
  - Konsistenz (Anforderungen in sich selbst und untereinander widerspruchsfrei)
  - Korrektheit (Absicht des Auftraggebers vollständig und konsistent wiedergegeben, nachprüfbar durch Validierung der Spezifikation)
  - Eindeutigkeit
  - Realisierbarkeit (in Bezug auf Randbedingungen, Hardware, Systemgrenzen, Kosten, etc)
  - Verfolgbarkeit (eindeutige Identifizierung)
  - Nachweisbarkeit (Existenz von eindeutigen Kriterien zur Überprüfung der Erfülltheit)
- Anforderungsermittlung:
  - Identifikation der Stakeholders
  - Ermitteln der Anforderungen durch z.B. Volere-Modell: (Anwendungsfallmodellierung [etwa Anwendungsfalldiagramm], Festhalten der Anforderungen, Zusammenstellen der Spezifikation)
- Spezifikationssprachen:
  - informal (Fehlen einer eindeutigen, wohldefinierten Semantik), z.B. natürliche Sprache
  - semi-formal (enthält Elemente mit eindeutiger, wohldefinierter Semantik), z.B. ER-Diagramm, funktionale Tabelle, Ablaufdiagramm
  - formal (durchgehend eindeutige, wohldefinierte Semantik), z.B. endliche Zustandsautomaten (FSMs), Petrie-Netze, OCL
  - OCL:
    - `context: Typename (::operationName(param1 : Type1, ...) : ReturnType)`
    - `inv`: Invariante
    - `pre`: Vorbedingung (etwa Voraussetzungen an Parameter)
    - `post`: Nachbedingungen (`result` bezeichnet Rückgabewert, Werte aus der Vorbedingung `wert@pre`)
    - `init`: Anfangswerte der Attribute
    - Konstrukt `let variable = ??? in !!! + variable`
    - Collections: `Set` (Menge), `Bag` (Menge mit Duplikaten), `Sequence` (geordnete `Bag`) werden mit `->` verwendet (etwa `select (v : Type | boolean expression)`, `forAll()`, `exists()`, `size()`, `allInstances()`, `includes()`, `isEmpty()`, `notEmpty()`, `implies()`)
- Software-Ergonomie (insbesondere bei GUIs):
  - Aufgabenangemessenheit, z.B. sinnvolle Vorbelegungen von Eingabefeldern
  - Selbstbeschreibungsfähigkeit: Jeder Schritt verständlich oder wird erklärt
  - Steuerbarkeit: Dialogablauf durch Benutzer start- und beeinflussbar (etwa Richtung und Geschwindigkeit)
  - Erwartungskonformität, z.B. Dialog entspricht Kenntnissen des Benutzers aus Anwendungsdomäne
  - Fehlertoleranz, z.B. Erkennen und ggf. Ausbessern von falschen Eingaben
  - Individualisierbarkeit, z.B. eigene Hotkeys, anpassbare Menüs
  - Lernförderlichkeit, z.B. Vorschau bei Formatierung

- Software-Entwurf:
  - Architektur: beschreibt Systemkomponenten und ihre Beziehungen untereinander
  - Systemkomponente: abgegrenzter Teil, Baustein für die physikalische und logische Struktur der Anwendung (etwa Klassen, abstrakte Datentypen)
  - Software-Grobentwurf: Erstellung einer Software-Architektur, die die Produkt- und Qualitätsanforderungen erfüllt und die Schnittstellen zur Umgebung versorgt
  - Prinzip: Zerlegung des Systems in einzelne Bausteine, Abstraktion (Verbergung von Implementierungsdetails)
  - Vorgehensweisen: Top-Down (Spezialisierung), Bottom-Up (Generalisierung) oder Kombination aus beiden
  - Ergebnis des Grobentwurfs: Architektur und Spezifikation der Systemkomponenten (Schnittstelle, Funktions- und Leistungsumfang)
  - Architekturmodelle: Blockdiagramm, Datenspeichermodell, Schichtenmodell, Client/Server-Modell, verteiltes System
  - Kopplung (Grad der Interaktion, möglichst gering) in aufsteigender Güte:
    - Content coupling (Inhalt): z.B. direkter Sprung in / Modifikation von Code eines anderen Moduls
    - Common coupling (Global): lesender und schreibender Zugriff auf gemeinsame Daten
    - Control coupling (Kontroll): Übermittlung eines kontrollflussbestimmenden Elements, etwa Flag
    - Stamp coupling (Datenstruktur): Übermittlung einer Datenstruktur, wenn nur Teile davon gebraucht werden
    - Data coupling (Daten): Übermittlung einfacher Daten oder komplett verwendeter Datenstrukturen
  - Kohäsion (Grad der funktionalen Bindung innerhalb einer Komponente, möglichst hoch) in aufsteigender Güte:
    - Coincidental cohesion (zufällig): vollkommen unzusammenhängend
    - Logical cohesion (logisch): Menge verwandter (teils alternativer) Funktionalitäten (oft durch Flag realisiert)
    - Temporal cohesion (zeitlich): Ausführung einer Reihe von Aktionen in zeitlichem Zusammenhang (Reihenfolge irrelevant)
    - Procedural cohesion (prozedural): Ausführung einer Reihe von Aktionen auf unterschiedlichen Daten in zeitlicher Reihenfolge (Reihenfolge relevant)
    - Communicative cohesion (kommunikativ): Ausführung einer Reihe von Aktionen auf gemeinsamen Daten (Reihenfolge irrelevant)
    - Sequential cohesion (sequentiell): Ausführung einer Reihe von Aktionen in sequentieller Abfolge (Ausgabedaten werden als Eingabedaten weiterverarbeitet)
    - Functional cohesion (funktional): Alle Elementie tragen zur Ausführung einer einzigen gemeinsamen Aufgabe bei
    - Informational cohesion (informationell): Reihe von Operationen mit jeweils eigenem Ein-/Ausgang und unabhängigem Code auf der gleichen Datenstruktur (etwa ADT)
  - Feinentwurf: Anpassung an Detailstruktur des Systems (etwa Sprachparadigma, Speicherbeschränkungen, Befehlssatz)
  - UML
    - Klassen: abstrakte Klasse *kursiv* oder {`abstract`}, Interface `<<interface>>`

        * Sichtbarkeiten:
            · `public`: +
            · `default`:
            · `protected`: #
            · `private`: -
        * Kennzeichnung von statischen Attributen und Methoden durch Unterstreichen
        * Attribute: `Sichtbarkeit name : Type`
        * Methoden: `Sichtbarkeit name(param: ParamType) : returnType`
        * Assoziationen: Bezeichner mit Pfeil für Leserichtung (etwa `uses`), Pfeilrichtung für „Referenz auf“, Multiplizität, geschlossene Spitze (Vererbung, mit gestrichelter Linie Implementierung)
    - Objektorientierte Analyse:
        * Statische Modellierung (Ziel: Klassendiagramm): Identifikation der relevanten Klassen, ihrer Attribute und Beziehungen zueinander
        * Dynamische Modellierung (Ziel: Sequenz-/Kommunikationsdiagramm); Erstellung von Szenarien (Ableitung eines Anwendungsfalles) zur Definition des Flusses von Botschaften durch das System
    - Objektorientiertes Design:
        * Architekturmodellierung:
            · Logische Systemarchitektur: Zusammenfassung von Klassen in Pakete (Paketdiagramme) und Komponenten (Komponentendiagramm)
            · Physikalische Systemarchitektur: Verteilung von Komponenten auf mehreren Rechnern und Kommunikation zwischen diesen (Einsatzdiagramm)
        * Statische Modellierung: Ergänzung der Klassen um weitere Klassen, Operationen, Attribute und Präzisierung der Assoziationen (name, Lese-/Benutzungsrichtung, Rollennamen, Multiplizität)
        * Spezialfälle der Assoziation: Aggregation (Teil-Ganzes-/Besteht-aus-Beziehung, leere Raute), Komposition (existenzabhängig vom Aggregat, ausgefüllte Raute)
        * Dynamische Modellierung: Präzisierung der bzw. Ergänzung um weitere Operationen (Modellierung oft durch Zustandsdiagramm)

- Entwurfsmuster (Design Patterns) als Wiederverwendung von Entwurfslösungen:
    - klassenbasiert: statische auf Vererbung basierende Klassenbeziehungen
    - objektbasiert: dynmische (zur Laufzeit veränderbare) Objektbeziehungen
    - Erzeugungsmuster (Objekterzeugung):
        * objektbasiert:
            · Singleton: Nur genau ein Objekt einer Klasse darf existieren, realisiert durch privaten Konstruktor und `getInstance()` (etwa Logger)
            · Abstrakte Fabrik: verwamdte Objekte sollen einheitlich erzeugt werden, aber Fabrik wird erst zur Laufzeit ausgewählt (etwa GUI je nach Plattform)
    - Strukturmuster (Zusammensetzung von Klassen und Objekten):
        * klassenbasiert:
            · Adapter (mit Vererbung): Übersetzung der Schnittstelle einer Klasse in eine vom Client erwartete, realisiert durch Implementierung der Client-Schnittstelle und Erben von Dienst-Klasse

        * objektbasiert:
            · Adapter (mit Delegation): realisiert durch Implementierung der Client-Schnittstelle und Referenz auf Dienst-Objekt
            · Brücke (Bridge): Abstraktion (Klasse mit Methode, die in Abhängigkeit von der aktuellen Klasseninstanz unterschiedlich realisiert werden kann) und Implementierer (enthält Realisierung der Abstraktion), erlaubt Implementierung zur Laufzeit auszutauschen
            · Proxy: Stellvertreterobjekt z.B. zur Zugriffskontrolle, realisiert durch Erben des Proxys und des eigentlichen Objekts von einer gemeinsamen abstrakten Klasse
            · Dekorierer (Decorator): Dynmische Erweiterung der Zuständigkeit eines Dienstes eines Objekts zur Laufzeit, realisiert durch Komponente als Teil des Dekorierers, Dekorierer erbt von abstrakter Komponente (dadurch beliebige Schachtelung möglich, im Kern eine konkrete Komponente), z.B. Scollbalken an Grafikobjekten
            · Kompositum (Composite): Zusammenfassung von Objekten zu einer Baumstruktur, ermöglicht gleichartige Behandlung von einzelnen Objekten und Kompositionen, realisiert durch Weiterleitung der Aufrufe an alle Kindobjekte, z.B. Grafik
    - Verhaltensmuster (Zusammenarbeit und Aufteilung von Verantwortlichkeiten zwischen Klassen und Objekten)
        * klassenbasiert:
            · Schablonenmethode (Template Method): Beschreibung eines Algorithmus, lässt konkrete Implementierungsdetails offen bis zur Übersetzungszeit, z.B. Framework
        * objektbasiert:
            · Befehl (Command): Auszuführender Befehl als Objekt gekapselt, Übergabe zur Laufzeit (etwa Menüpeinträgem mit Parametern)
            · Beobachter (Observer): Änderungen an Objekt werden an andere Objekte weitergeleitet (entweder nur signalisiert oder gleich Übergabe der geänderten Daten)
            · Strategie (Strategy): Variante der Lösung einer Teilaufgabe soll zur Laufzeit ausgewählt werden, realisiert durch Kontext mit Referenz auf konkrete Strategie (erbt von einer abstrakten Strategie), z. B. Formatierer für verschiedene Datentypen
            · Zuständigkeitskette (Chain of Responsibility): Objekt kann Anfrage selbst behandeln oder an übergeordnetes Objekt weiterleiten (z. B. Hilfeknopf in GUI-Dialogen)

- Implementierungsphase:
    - Wahl der Programmiersprache nach wirtschaftlichen/technischen Gesichtspunkten
    - Festsetzung von Code-Regeln zur Erhöhung der Les- und Wartbarkeit
    - Code-Generierung aus bisher erstellten UML-Konstrukten:
        * Erzeugung der Klassen
        * Typanpassung der Attribute und Operationen
        * Umsetzung der Assoziationen
        * Umsetzung der Interaktionsdiagramme in den Methoden
        * Fertigstellung der Methoden, etwa der eigentliche Algorithmus

- Testen:
    - Modultest: Prüfung auf Übereinstimmung von Programmmodulen und dem zugehörigen Software-Feinentwurf
        * Black-Box-Test (funktional): Testauswahl aufgrund der Spezifikation, Programmstruktur irrelevant, etwa Äquivalenzklassen- / Grenzwert- / Cause-Effect- / Error-Guessing- / Zufallswerttests

    - Grey-Box-Test (modellbasiert): Testauswahl aufgrund des Modells, etwa
    - White-(/Glass-)Box-Testing (strukturell): Testauswahl aufgrund der Programmstruktur (etwa Kontroll- und Datenflüsse), etwa Überdeckungstesten (Coverage) (z.B. Anweisungs-, Verzweigungs- und Pfadüberdeckung)
    - Regressionstest: Wiederholtes Testen der Testfälle nach Code-Änderungen
  - Integrationstest: Prüfung des Zusammenspiels der Module (und Übereinstimmung mit Software-Grobentwurf)
    - benötigt meist Stubbing (also Simulation der anderen Module)
    - ermöglicht inkrementelles Testen (Stubs werden nach und nach durch bereits implemtierte Module ersetzt)
  - Systemtest: etwa Vollständigkeit, Leistung, (Daten-)Volumen, Stress
  - Abnahmetest: Prüfung direkt beim Kunden durch ihn selbst (oder unabhängige Gutachter)
  - Testplanung: Definition des Testziels (etwa minimaler Grad)
  - Testerstellung: Auswahl der Testdaten
  - Testdurchführung
  - Testauswertung: Vergleich der Ausgabedaten mit Spezifikation
- Wartung:
  - Korrektur (corrective): Entfernung aktueller Fehler
  - Anpassung an neue Benutzeranforderungen (perfective): z.B. zusätzliche Funktionalität, verbesserte Performance
  - einschließlich vorbeugende Maßnahmen (preventive): z.B. Aktualisierung der Dokumentation, Refactoring
  - Anpassung an neue Umgebung (adaptive): z.B. neue Hardware, Compiler, Betriebssysteme, Änderungen an gesetzlicher Rahmenlage
  - Refactoring (nicht Performance-Optimierung):
    - auf Codeebene: Erhöhung der Lesbarkeit und des Verständnisses, Aufdecken von Fehlern
    - auf Designebene: Verbesserung des Designs (etwa nach hinzugefügter Funktionalität), Reduktion versteckter Abhängigkeiten, Erleichterung der späteren Erweiterung
    - Beispiele:
      - Methode extrahieren
      - Methode integrieren
      - Ersetzung von Methode durch Methodenobjekt (z.B. `Calculator`)
      - Klasse extrahieren (Aufspaltung der Zuständigkeiten)
      - Klasse integrieren (zuständigkeiten zusammenlegen)
      - Delegation verbergen
      - Collection kapseln (Rückgabe einer lesenden Sicht, Immutability)
      - Ersetzung von bedingtem Ausdruck durch Polymorphie
      - Verschiebung gleicher Felder aus Unterklassen in gemeinsame Oberklasse
      - Verschiebung gleicher Methoden aus Unterklassen in gemeinsame Oberklasse
      - Unterklasse extrahieren
      - Oberklasse extrahieren
      - Vererbungsstruktur entzerren

        - Entfernung unnötiger Indirektionen (möglicherweise auch durch Refactoring entstanden)

- Wiederverwendung:
  - Wiederverwendbarkeit: Komponenten eines Produkts in anderem Produkt einsetzbar
  - Portabilität: Produkt in anderer Umgebung (etwa OS, Hardware, Compiler) einsetzbar
  - Interoperabilität: Zusammenarbeit mit Code anderer Hersteller