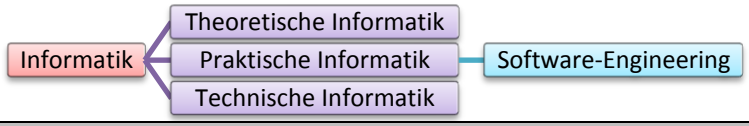
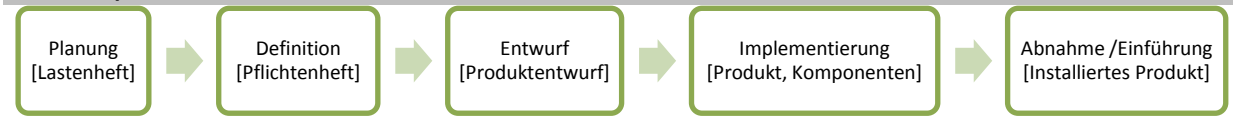
SOFTWARE ENGINEERING (SWE)

Übersicht

Softwarekrise von 1968	Ursache: Programmsysteme der 60er Jahre wurden komplexer. Aber es gab keine geeigneten Sprachen, Methoden und Werkzeuge. Folge: Software-Kosten stiegen, während Hardware-Kosten fallen viele Projekte scheiterten. Auslöser für die Einführung von Softwaretechnik	
Schwierigkeiten von Softwareherstellung	Kommunikationsprobleme mit dem Anwender; immateriell (=unbegrenzt) leichter modifizierbar als Hardware (Änderungen während Entwicklung) kein Verschleiss, aber altert (Wartungsprobleme); viele Varianten Portabilität (versch. Plattformen); Akzeptanzprobleme (Arbeitslosigkeit)	
Software Engineering = Softwaretechnik	Entwicklung, Pflege und Einsatz von Methoden, Prinzipien und Werkzeugen	
Softwareingenieur =Softwaretechniker	Guter Programmierer mit Kenntnissen in Datenstrukturen und Algorithmen, in Programmiersprachen, in Anwendung von Modellen und Vorstellungen	
Softwarequalitätsmerkmale	korrekt, zuverlässig, robust, effizient, benutzerfreundlich, wartbar	
	externe Qualitätsfaktoren:	sichtbar für Benutzer des Systems
	interne Qualitätsfaktoren:	sichtbar für Entwickler des Systems
Korrektheit von Software	Verhalten nach Anforderungsdefinition	
Zuverlässigkeit von Software	Wahrscheinlichkeit, des erwarteten Verhaltens	
(ROFOC)	rate of failure occurence	Häufigkeit von unerwartetem Verhalten (2/100)
MTTF	mean time to failure	mittlerer Zeitabstand zweier Fehler
-	availability	mittlere Verfügbarkeit (998/1000)
Robuste Software	Funktion unter unvorhergesehenen Umständen	
Effiziente Software	ökonomische Verwendung von Hardware-Ressourcen O-von-Rechnungen, Simulationsmodelle, Messung am realen System (profiling)	
Benutzerfreundliche Softw.	Leichtes erlernen und benützen	
Kategorien	Anfänger	Menüs, Tutorial, Online-Hilfe
	Gelegenheitsbenutzer	Online-Hilfe, häufige Control-Keys
	Experte	immer Control-Keys, Konfigurationsmöglichkeiten
Wartbarkeit von Software	Möglichkeit von Änderung aufgrund von Fehlern oder geänd. Anforderungen Durch: gute Systemstruktur, Dokumentation, kontroll. Änderungsprozeduren	
Vertrauenswürdige Softw.	Verursacht im Fehlerfall keine Katastrophen	
Portierbare Software	Falls es in verschienen Umgebungen (=Hardware, Betriebssystemen) läuft Durch: Trennung kleiner umgebungsabhängige Teile, Standardisierung	
Kompatible Software	Kann leicht mit anderer Software kooperieren oder ersetzt werden	
Effizient des Softwareerstellungsprozess	Produktive Softwareherstellung in keinem Fall durch **LOC** (Lines of Code) messen	
Wiederverwendbarkeit	leicht für neue Anwendungen verwendbar (Zusammenstecken von Software)	
Qualität	Werkzeuge (Case-Tools), Methoden (Pragmatik), Sprachen (Semantik, Syntax), Konuepte	
Objektorient. Paradigma	Denkmuster, das das wissenschaftliche Weltbild einer Zeit prägt	
Einteilung	Strukturiertes Paradigma	hierarchisch strukturierte Funktionen
	Objektorientiertes Paradigma	gekapselte Objekte (Klassen)
4 P's des SWE	People (who is doing), Product (what is the result), Project (the doing of it), Process (how is it done)	
Einteilung		

Prozesse / Phasen



Projektplanung

Durchführbarkeitsuntersuchung

- Lastenheft
- Projektkalkulation
- Projektplan
- evtl. Glossar

Lastenheft (=Grobes Pflichtenheft)

1. Zielbestimmung	Die Firma ... möchte ein Programm für ...
2. Produkteinsatz	Das Produkt dient ...
3. Produktfunktionen	(LF=Lastenheft Funktionen) /LF10/ Ersterfassung und Löschung von Kunden
4. Produktdaten	(Lastenheft Daten = LD) /LD10/ Es sind alle Daten über die Kunden zu speichern
5. Produktleistungen	(Lastenheft Leistungen= LL) /LL10/ Die Funktion /LF10/ darf nicht länger als 5 Sekunden dauern
6. Qualitätsanforderungen	sehr gut gut normal nicht relevant Funktionalität Zuverlässigkeit Benutzbarkeit Effizienz Änderbarkeit Übertragbarkeit

Aufwandsabschätzung (Kosten und Termin)

geschätzter Aufwand	$= \frac{ges.LOC}{LOC/Monat}$		**Einflussfaktoren** Quantität (LOC, Umfang, Komplexität) Qualität (Qualitätsmerkmale) Produktivität (Brooksches Gesetz) Entwicklungsdauer (Mitarbeiteranzahl) Kosten (Function-Point-Methode)
optimale Entwicklungsdauer	$= 2.5 * Aufwand\hat{}s$	$s(Online) = 0.35$ $s(Echtzeit) = 0.32$	
optimale Mitarbeiter	$= \frac{Aufwand}{Entwicklungsdauer}$		

Teufelsquadrat: Qualität-Kosten Quantität-Entwicklungsdauer stehen in einem Konflikt

Brooksches Gesetz: Adding manpower to a late software-project makes it later

Function-Point-Methode

Idee	Schätzverfahren für den **Zeitaufwand** eines Software-Projekts --> Kosten
Voraussetzungen	Produktanforderungen / Lastenheft vorhanden Anwendungsprojekt aus sicht des Benutzers
Vorgehen	1. Kategorisierung jeder Produktanforderung 2. Klassifizierung jeder Produktanforderung (einfach, mittel, komplex) 3. Eintrag in Berechnungsformular 4. Bewerung der Einflussfaktoren 5. Berechnung der bewerteten Function Points (FP) 6. Ermitteln des Personalaufwandes aus einer FP-MM Kurve oder Tabelle 7. Aktualisierung der empirischen Daten als Schätzgrundlage für Folgeprojekte
Vorteile	Ausganspunkt sind Produktanforderungen, nicht LOC Anpassbar an Anwendungsbereiche, Techniken, Unternehmenspezifikationen frühe Schätzung, leicht erlernbar, geringer Zeitaufwand, gute Genauigkeit / Transparenz
Nachteile	nur Gesamtaufwand schätzbar, personalintensiv und nicht automatisierbar stark funktionsbezogen, Qualitätsanforderungen werden nicht berücksichtigt

Objektorientierte Analyse (OOA)

Idee	Brücke von Anwender zur Softwaresicht
Aufgaben	Beschreibe das Systemverhalten -> SSD Identifiziere Domänenobjekte -> Domänen-Modell
POS	Point-Of-Sale

Entwicklungsprozess

Inception
Ungefähre Vision
Untersuchung der Machbarkeit

→

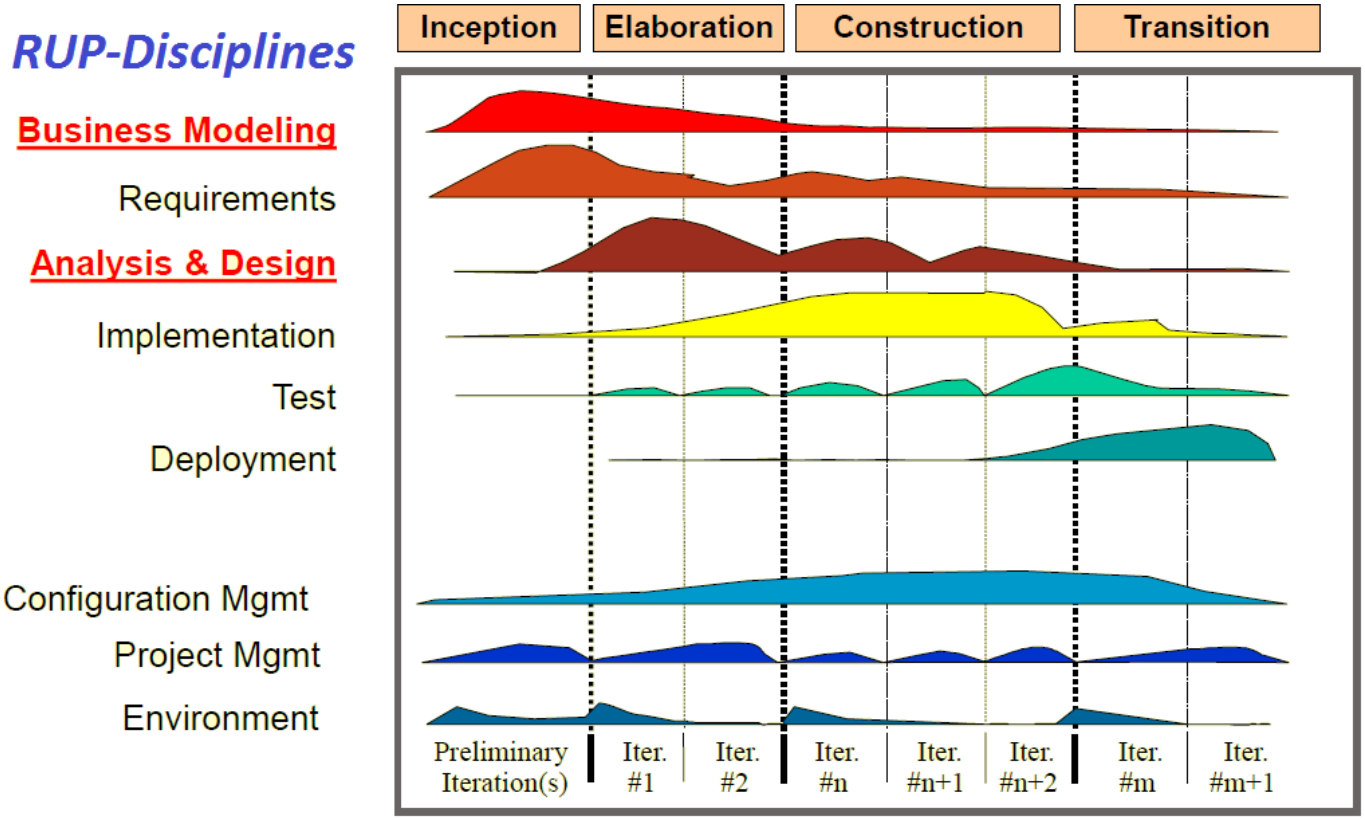
Elaboration
Entwurf, Ausarbeitung iterative Entwicklung der Kernarchitektur

→

Construction
Iterative Entwicklung der restl. risikoärmeren & leichteren Elemente

→

Transition
Übergang, Umsetzung Betatests, Deployment



RUP-Disziplinen: Zuerst schwierige Sachen implementieren

SSD Systemsequenzdiagramme [Verhaltensmodell]

Idee	Haupteingabe und Ausgabesystemereignisse identifizieren
Voraussetzungen	Use-Cases (Text mit ins Diagramm nehmen)
Tipps	Systemevents mit Verb anfangen

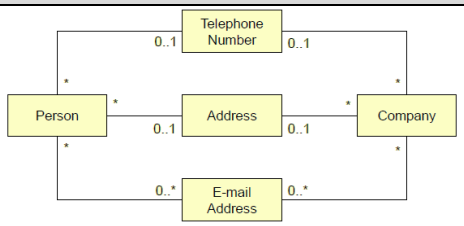
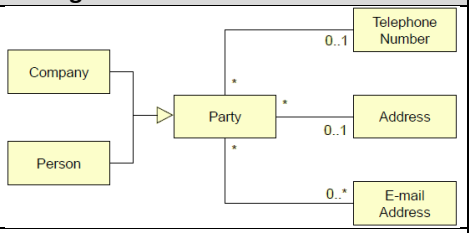
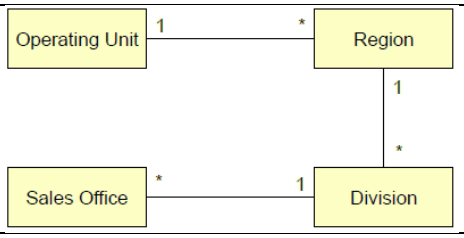
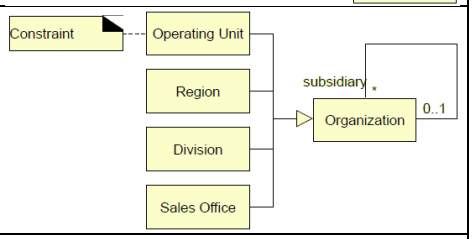
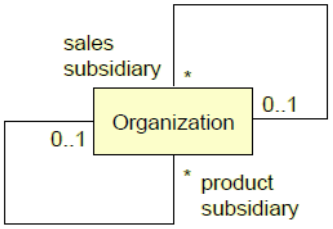
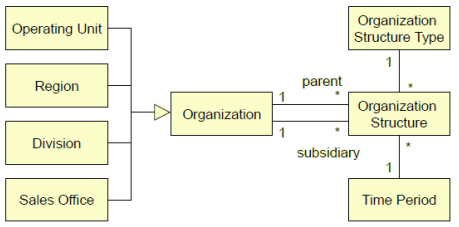
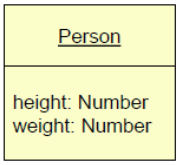
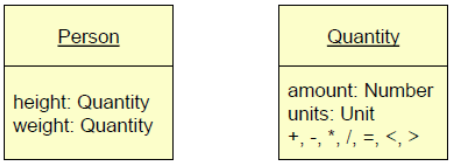
Klassendiagramm [Domänenmodellierung]

Anforderungen	Richtig, Vollständig, Konsisten, Nachweisbar

OOA: Domänenmodellierung / Pflichtenheft

Analysemuster

Zweck: Hilfe für Softwareingenieure bei der Umwandlung von konzeptuellen Modellen in Software

Muster	Problem	Lösung
Party Modelliere ein Adressbuch von Personen und Organisationen -> Generalisierung einführen		
Organization hierarchies Modelliere Organisationen mit ihren Tochtergesellschaften -> Generalisierung einführen		
Organization structure Modelliere die Organisation mit mehreren Hierarchien Typisierte Bezeichnung		
Accountability		
Quantity Einheiten Problem -> Abstrakte Einheiten		
Conversion Ratio		

Grundsätze

- Behandle das System als Blackbox
- Domänen-Objekte sind nicht Softwareklassen, können aber welche werden
- Benutze Muster für Domänenmodellierung

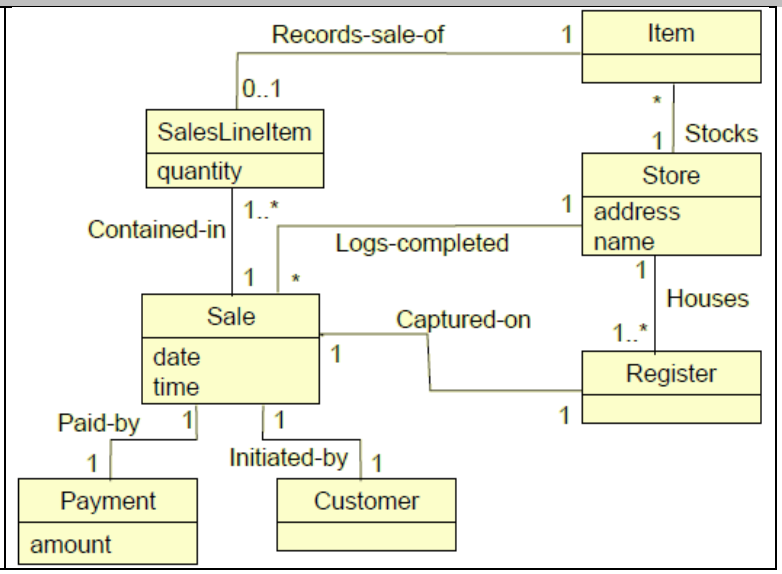
Domänenmodell

Vorgehen

1. Identifiziere Kandidaten für Klassen
2. Zeichne sie in einem Domänenmodell
3. Füge die Assoziationen dazu
4. Füge den Objekten Attribute hinzu

Finden von Konzeptuellen Klassen

- Kategorienliste
  (Auflistung üblicher Kategorien)
- Noun Extraction
  (Hauptwörter aus Problembeschreibung)
- Analyse-Muster



Pflichtenheft

Use-Case Beschreibungen
Use-Case Domänenmodell
Domänenmodell
Systemsequenzdiagramme

Objektorientiertes Design – OOD

Unterschiede zu OOA

OOA: Problemanalyse
OOD: Lösungsentwurf

Information Hiding

Innere Details verstecken
Genau definierte Schnittstellen
innere Details ändern -> Kunden sollten nicht betroffen sein

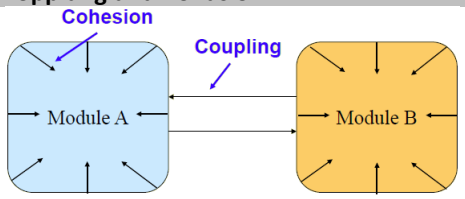
Ziel

Inhalt der Klasse: hohe Kohäsion, möglichst lose Kopplung
Klassen mit klarer Verantwortung
Design Patterns benutzen

GRASP (General Responsibility Assignment Patterns)

Information Expert
Creator
High Cohesion
Low Coupling
Controller

Kopplung und Kohäsion



Kohäsion

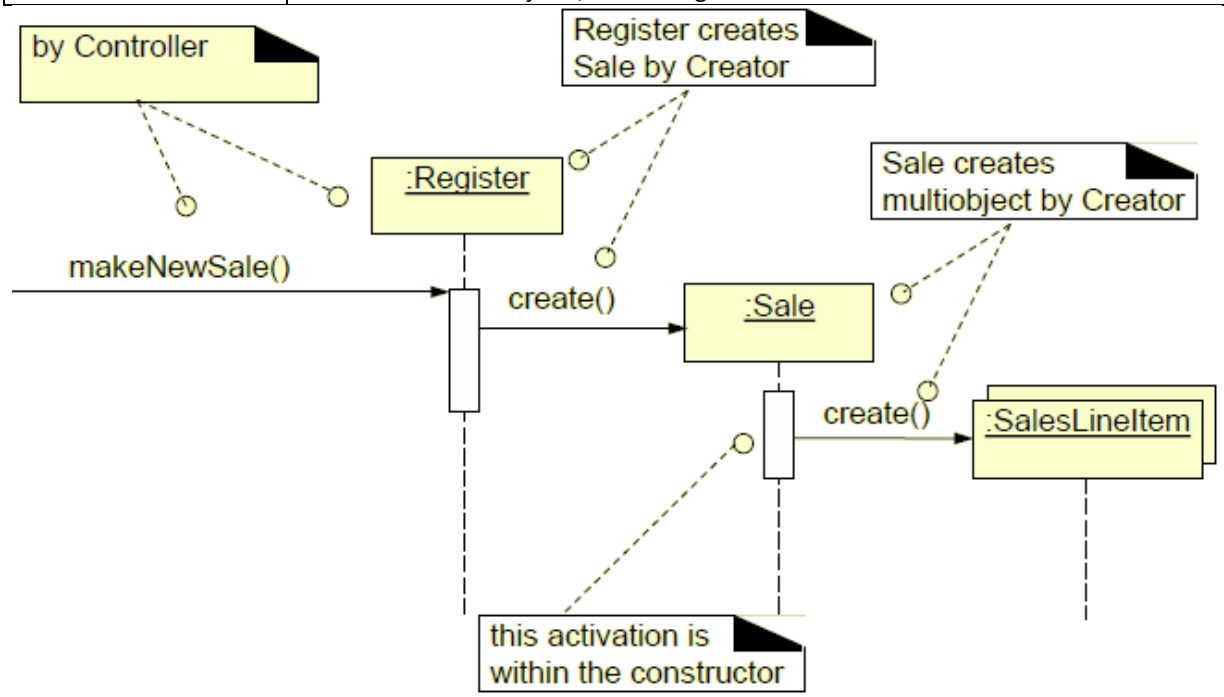
Funktionsaufrufe
Datenzugriffe (Instanzvariable, Parameter)
Menge von Methoden die eine Aufgabe erfüllen
->Verstärkung, wenn auf gemeinsame Daten zugegriffen wird

Kopplung

Methodenaufrufe
Transfer von Daten zur Methode
gemeinsamer Zugriff auf globale Daten
Zugriff auf Instanzdaten

Objektdesign:

Use Case	Process Sale
Systemoperation	makeNewSale()
Preconditions	keine
Postconditions	Instanz s wurde erzeugt; s wurden Register zugeordnet; Attirbute von s sind initialisiert
Designentscheidung	Wer erhält die Meldung?
GRASP anwenden	Controller Pattern
Entscheidungen	Controller für das ganz System, oder Use case spezifischer Controller
Design Entscheidung	Wer erzeugt die neue Sale Instanz?
GRASP anwenden	Creator Pattern
Wahl	Der Container des Objekts, das erzeugt werden soll.



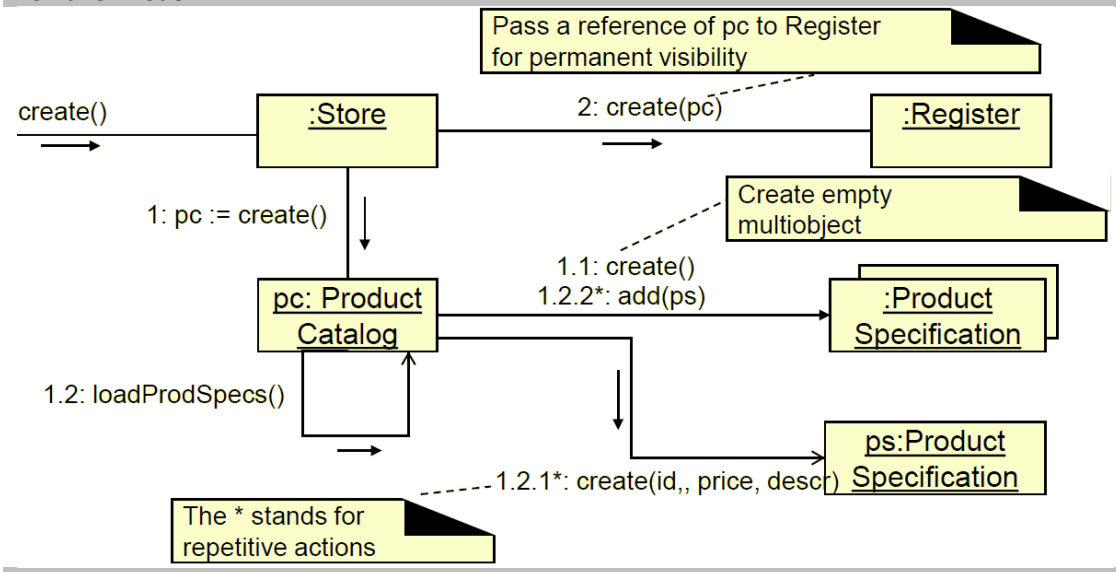
OOD

Startdomänenobjekt = Wurzel der Aggregationshierarchie, hier Store

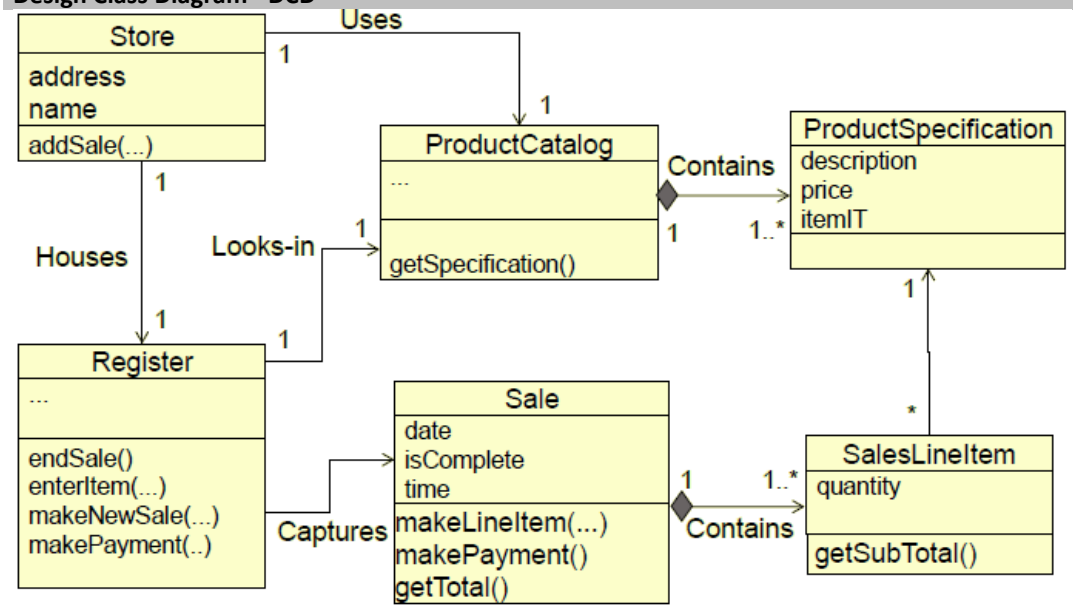
Verantwortung der UI-Schicht	**Verantwortung der Domänenschicht**
Anfragen an das System aufnehmen Ansichten auffrischen	Behandlung der Anfragen (Funktionen) (Macht die Arbeit)

Sichtbarkeit erzeugt Kopplung

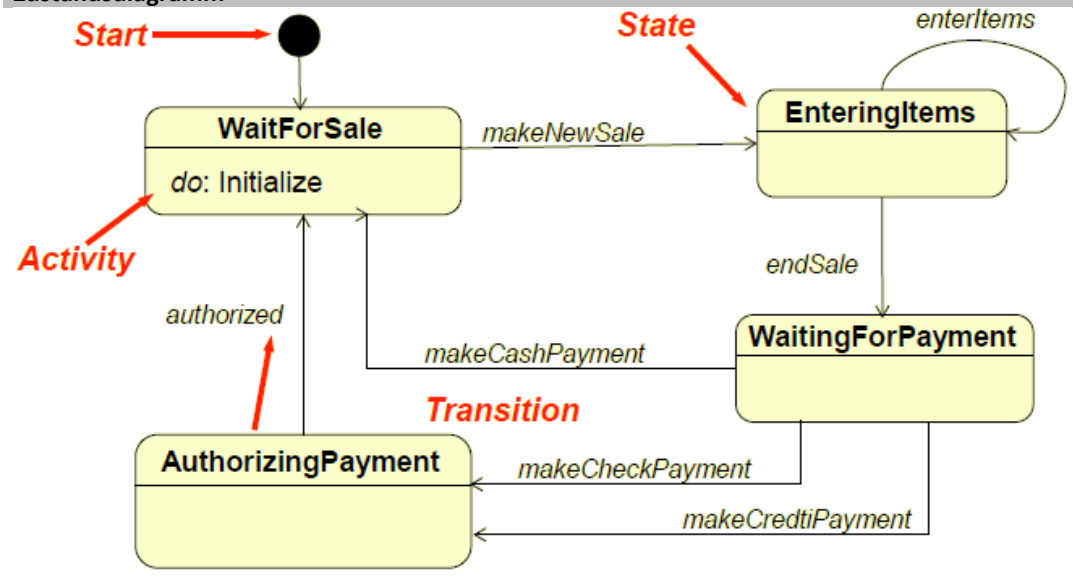
Domänenmodell



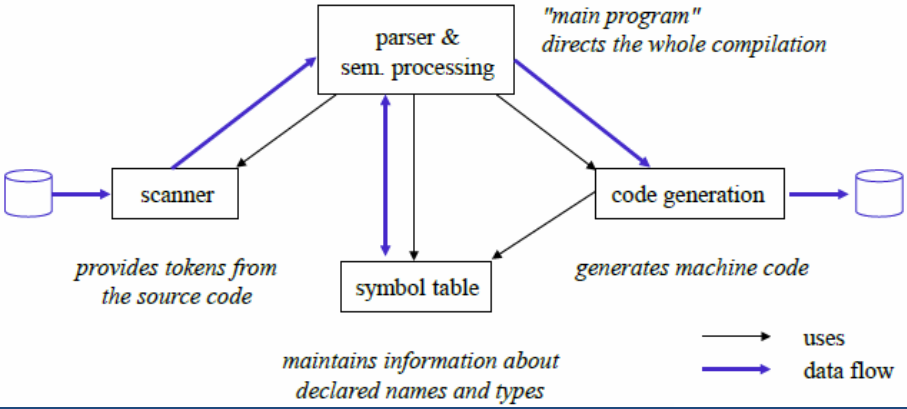
Design Class Diagram - DCD



Zustandsdiagramm



CocoR



CodeReview

V&V (Verifikation und Validierung)	ständige Kontrolle, um Fehler frühzeitig zu entdecken für Qualität im Prozess der Softwareentwicklung
Software Inspektion	statisch, früh anwendbar
Software Testen	Teilsystem vorhanden, mit Testdaten
Teilnehmer	Moderator (Durchführung, Planung) Autor (Hat Dokument erstellt) Gutachter (~3, decken Fehler auf) Protokollführer (erstellt Reviewbericht)

Metriken

Zyklimatische Zahl
ZZ sagt etwas über die Komplexität aus.

$$ZZ = E - N + 2k < 11$$

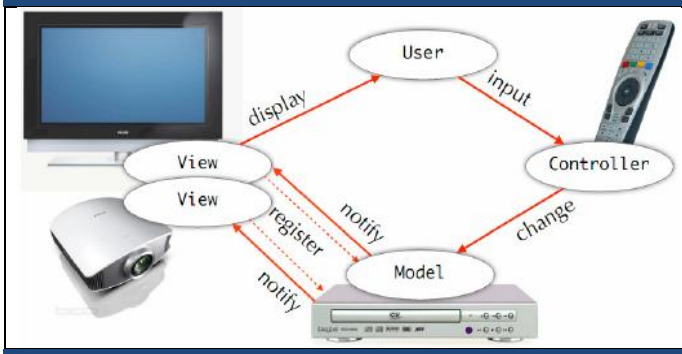
E	Kanten
N	Knoten
k	Verbundene Komponenten

Number of Methods (NOM)
NOM = NIM+NCM=NEM+NHM
NIM = Number of Instance Methods
NCM = Number of Class Methods
NEM = Number of External (public) Methods
NHM = Number of Hidden (private/protected) Methods

Number of Attributes (NOA)
NOA = NIV + NCV
NIV = Number of Instance Variables
NCV = Number of Class Variables

Depth in Inheritance Tree (DIT)
Tiefe im Vererbungsbaum
Number of Children (NOC)
Coupling Between Objects (CBO)
Response for a Class (RFC)
Henderson-Sellers (LOCOM) - Lack of Cohesions of Methods

MVC



Model, View und Controller trennen

Model: Enthält die Daten
View: Zeigt die Daten an
Controller: Steuert

Verifikation & SCM (Software Configuration Management)

Contract = wer ist für die Prüfung zuständig (Kooperativ oder Defensiv)
Assertions (Vor- und Nachbedingungen)
Verifikation = Mathematischer Beweis der Korrektheit
SCM ist für die Integrität eines Produkts in einem Softwareprojekt verantworlich (z.B. Treiber)
Versions und Freigabe-management (durch Repository -> check out -> Sandbox -> check in -> Repository)
X.Y (X=Level der Funktionalität (starts with 1), Y=Update Level (starts with 0))