Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Stephan Heidinger

Software Engineering
Wintersemester 2011/2012
21. April 2012

Find any errors? Please send them back, I want to keep them!

Definitions for Software Engineering
"Multi-person construction of multi-version software" (D. Parnas, 1987)
"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

Inhaltsverzeichnis

1 Software Crisis

Term from NATO study group in 1967.

fault mistake/error made by a human during a software activity (erroneous design, requirements, coding)

failure observed departure of a system from the desired state

1.1 symptoms

- products are delivered late ⇒high cost
- projects exceed budgeds ⇒high cost ⇒waste of resources
- product doesn't do, whats its supposed to do ⇒inefficient ⇒hight cost
- products are defective ⇒high cost (failures, maintenance,...), ethical considerations
- projects get abandonded before delivery ⇒waste of resources

1.2 Characteristics of software

- Software is *engineered*, not manufactured (custom built, little to no (mechanical) assembly, human intensive process)
- Software does not wear out (but: change of requirements, changes in environment)
- software is *complex*
- software is *detemining system factor* (up to 80% of development effort)

1.3 Why is software difficult to produce?

- no similiar system built so far (problems unknown, assumptions about environment may be wrong)
- *Requirements* are not well understood/phrased
- Requirements *change* during development
- complex *interaction*
- nature of systems:
  - concurrent systems (races, deadlocks, ...)
  - embedded systems (hardware interaction, timing, ...)
  - information systems (complexity, legacy, ...)
- software is *easily changeble* ⇒"code and fix"
- software is *discreet* (either fails, or doesn't)

1.4 Software Debvelopment Myths

- Management
  - standard books, software, tools, ... ⇒software hard to standardize
  - behind schedule? add programmers! ⇒effort to integrate new people
- Customer
  - general statement of objectives sufficient
  - change (as in requirements) can easily be implemented
- Practioner
  - once programm is running, job is done
  - until programm is running, no way to check quality
  - only deliverable is working programm

1.5 Software Engineering

Communication between a huge number of parties (customer, end user, sw designer, developer, ...) must be maintained.
Highly complex systems: SE has to provide solutions.

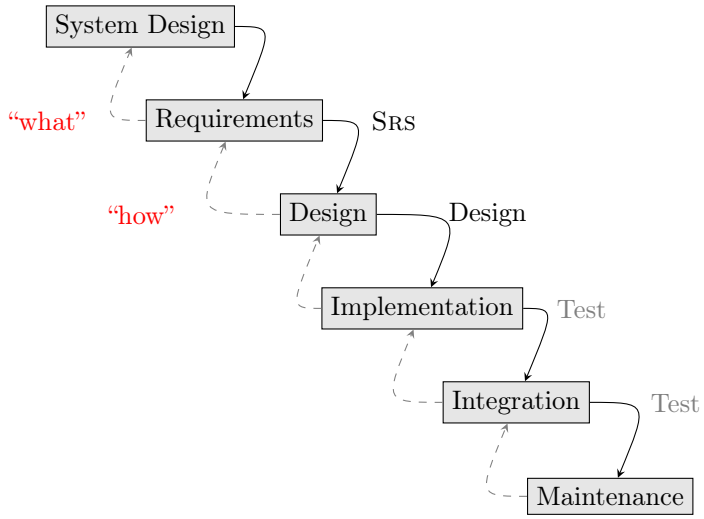
⇒ base sw production on engineering approach

⇒ design sw as a *process* that supports

- correctness & dependability
- cost-effectivenes
- complexity of project
- longevity of product, life cycle, changes
- communication amongst parties

⇒ software engineering process model

2 waterfall model



System Design problem & System requirements, fesability study, define main subsystems (allocate to hw/sw), system design document (SDD, informal, with customer, somtimes: user manuals, user interfaces, test plans)

Requirements requirement analysis ("what" the system has to do (not how), software requirements describing observable external behaviour (functional, nonfunctional)), software requirement specification (SRS)

Design "how" the system is working, architectural (high level) design (decompose problem into components, global data structures, internal interfaces), detailed design (algorithmic design, internal data structures, programming language(s))

Implementation (and Testing) ⇒translate design modules into code, test modules in isolation

Integration (and Testing) integrate tested modules to form system, *integration testing*, validation, customer accptance tests

maintenance deploy product, perform maintenance (corrective, adaptive, perfective), retire product

2.0.1 Benefits

- Definition of seperate Tasks (Seperation of concerns)
  dividing complex design problems into smaller units ⇒facilitates team work/reproducibility
- specification and documentation ⇒enforces documentation, facilitates testing (against requirements specification)

- further concepts
  - verification / validation (compare intermediate results with requirements and design)
  - prototyping (mock-up available early, reduces risk)
  - evolutionary process model (accommodate change)

2.0.2 Defects

- no feedback loops
- document driven, inflexible
- large time gap between inception and completion

2.0.3 Alternatives

- modified waterfall model
- V-Model, V-Model XT
- Evolutionary Process models
- Spiral model
- Rational Unified Process
- Agile Processes
- ...

2.1 Software Qualities

Correctness: the software behaves according to the *requirement specification*

Reliability: the software guarantees a level of quality service (<Correctness)

Robustness: the software behaves "reasonably" even in unexprected circumstances (e.g. input, power failure, ...)

Maintainability: software is easy to maintain/extend

Performance: system is usable (e.g. input response)

Reusability: re-use of previously used, tested and verified code

Interoperability: standardisation, interfaces, ...

2.2 Software Engineering Principles

Rigor: use a method and apply it rigorously to every step

Formality: use (mathematical) formalizable methods and notations

Speration of Concerns: deal with different aspects in seperate steps

Abstraction: seperate the concern of the important aspects from the concern of unimportant ones

Modularity: decompose problem into independent modules ⇒separation of concern

Generality: focus on the discovery of more general problems

3 Requiremnt & Analysis

3.1 Requirement Elicitation

"A **requirement** is a condition or capability that must be met or professed by a system component to satisfy a contract, standard, specification or other formally inposed document."
(ANSI/IEEE Standard 729-1983)

requirements ⇒description of the *externally visible (interface) behaviour*, the "what"
Interaction between the system and the system-relevant portion of the envrionment

User (Client) requirements: statements in natural language plus diagramms

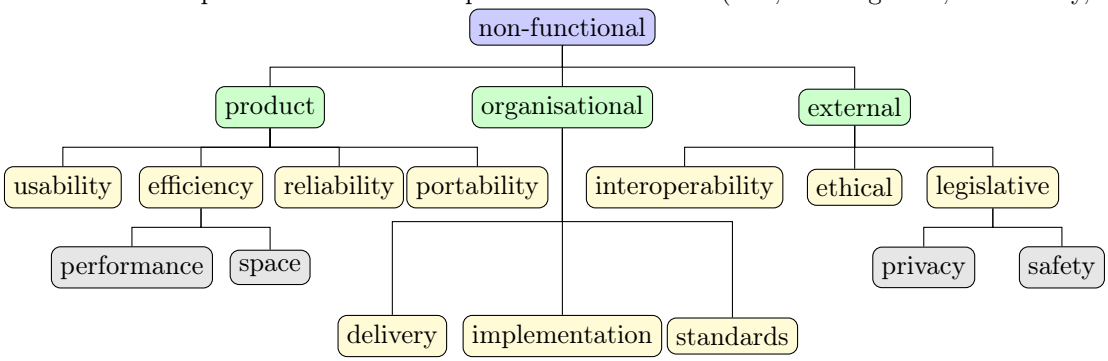
System requirements: structured document, detailed description of system's functions, services and operational constraints; may be part of contract between client and contractor

3.1.1 Types of requirements

functional requirements: high-level-"what" the system should do

- statement which services the system should provide
- interaction between system and environment (system states, i/o)
- how the system should behave in particular situations
- describe the system services in detail
- often included with "shall/should"

nonfunctional requirements: (if not met, system may be useless (not usable))
nonfunctional requirements often have quantitative measures (MB, training time, availability, . . .)



- not primarly related to sytem's funcions/services, but to quality and additinal (quantitative) characteristics
- constraints on the services/functions:
  - reliability (avalability, integrity, security, safety)
  - accuracy of results
  - performance / timing
  - human-computer interface issues
  - operating and physical constraints
  - portability and interoperatbility
  - reliability
  - response time
  - storage requirements
  - standards . . .
  - also: particular system, programming language or development method
- product requirements (product must behave in particular way (execution time, speed, reliability, . . .))
- organisational requirements (standards used, impelementation requirements, . . .)
- external requirements (interoperatbility/legislative requirements, . . .)

domain requirements: (if not met, system may be unworkable in that domain)

- derived from application domain
- characteristics/features of domain
- constraints on existing requirements
- define specific computations
- may be expressed in domain specific language ⇒often hard to understand
- often implicit ⇒hard to aquire

3.1.2 Requirement Specification Properties

correctness: facts in requirement specification ⇒required properties of the system

unambiguity: all specifications have a single interpretation

completeness:

1. every property required of the system is expressed in the specification
   does this include things that are *not permitted?*
2. the responses of the software system on all types of possible input values are specified
3. ... (more possible)

verifiability: there exists an effective (manual/automatic) process for checking wether a software product satisfies the required properties (formal verification (mathematical proof) or validation (model checking, testing, simulation)), often not possible for every requirement

consistency: no two requirements contradict each other: never $A \wedge \neg A$

traced: origin of every requirement is clear

traceable: requirement specification is edited so it is easy to reference every single requirement (e.g. numbering)

design independent: the requirement specification does not require specific software/architecture/algorithms

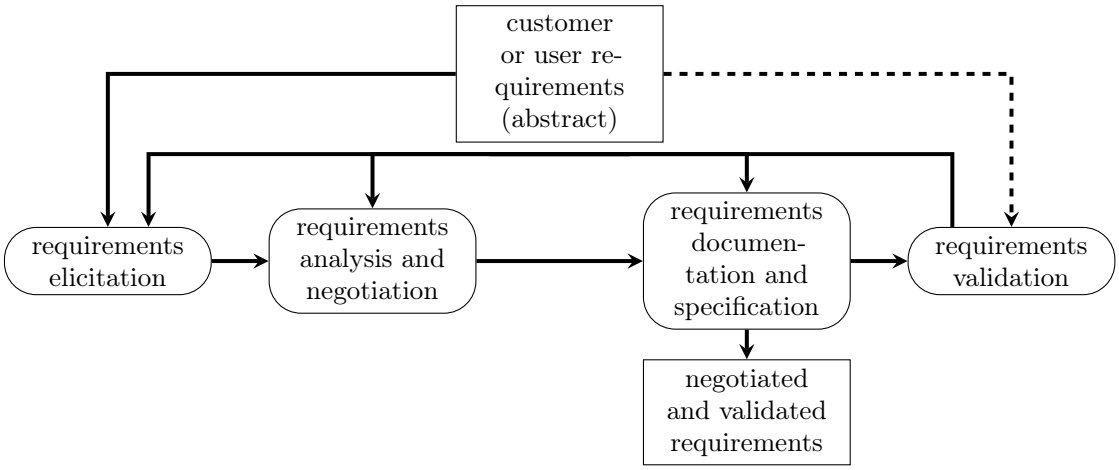
3.1.3 Activities during the requirement Stage

starting point

- customer requirements (abstract)
- system specification document (hw & sw)

Activities (Kotonya+Sommerville)

- requirement elicitation (interviews, scenarios, market observation, ...)
  determine, which of possibly contradictory requirements are important
- requirement documentation and specification (comprehensible requirements document)
- requirement validation (consistency, completeness, correspondence of documented requirements and abstract customer or user requirements)



social activity: Not a single person knows everything about the system ⇒communication is needed ⇒difficult (technical language, ambiguities, customers not aware of needs, personalities.)

3.1.4 Interviewing / Structured Questioning Techniques

1. context-of-system (why are we building this system/who are the users/critical functionality/needs)
2. open-ended questions (produce large amount of info/use, when not much known yet
3. close-ended questions (specific questions))
4. rephrase questions (make sure answers understood right/inconsistencies/ambiguities)
5. whom else to interview

3.1.5 Learning from Existing Systems

- analyse user manuals
- use/play with existing systems
- market analysis (competing systems, market research, which features to be included)
- reverse engineering

3.1.6 requirement uncertainty

When the customer does not know, what he really needs/wants ⇒prototype. Built prototype early (only if prototype build is faster than system build), use prototype for requirement elicitation. Prototyping also part of some life cycle and process models (spiral model)

3.1.7 Requirements Gathering - Brainstorming

- generate ideas (free of criticism/judgement) ⇒as many ideas as possible
- discuss, revise, organize ideas
- evaluate, prioritize

3.1.8 Fast (Facilitated Application Specification Technique)

- overcome we/them attitude (developer/user/customer)
- team oriented
- guidelines:
  - must participate entire meeting
  - participants are equal

- preparation as important as meeting
- pre-meeting documents are only "proposed"
- off-site location preferred
- set agenda, maintain it
- no technical details

3.1.9 JAD (Joint Application Design, 1977)

- technique to get all participants to aggree on sw requirements and design
- 20% reduction of life cylce cost, $\tilde{1}5\%$ functionality less missed
- JAD/Plan ⇒software requirement elicitation
- JAD/Design ⇒software design
- Main Objective:
  1. group dynamics ⇒enhace capcabilites of individuals
  2. communication and understanding ⇒use of visual aids
  3. rational, organized process ⇒repeatability
  4. standardized documentations ⇒use of standard forms
- customisation (prepare tasks for session)
- session (customized, facilitated meetings (developers and user))
- wrap-up (finalize results from session)
- participants

  session leader: manages sessions, management/communications skills, problem domain competence

  analyst: ⇒session output, technical understanding req.

  executive sponsor: high-level strategic insight into system, executive-level decissions (resources, ...)

  user representative: "user"

  information systems representative: feasibility assessment

  specialist: knowledge from application domain
- Customization phase
  - orientation (sponsor endorses project, session leaders (+analyst) gains understanding of system/environment)
  - organize team (select participants, think of questions ahead)
  - tailor the process (how much time/resources, customize documents)
  - prepare materials (slides, presentations, ..., white boards, flip chats, ..., agenda)
- Session phase
  - orientation (welcome, overview)
  - define high-level requirements (objectives, benefits, strategic/future considerations, constraints and assumptions, security/audit/control) ⇒recorded by analyst, discussion (refine, assess requirements)
  - defince scope of system (organize and prioritize requirements, scopes, s.t. system meets objectives, but not too costly or complex)
  - prepare JAD/Design (estimate resources, identify participants, schedule meetings)
  - document issues and considerations (problems that affect requirements ⇒assign a person for resolution)

  - conclusion (review information and decissions, voice remaining concerns, conclude s.t. particip. have sense of ownership/commitment)

- Wrap-Up Phase
  - transform session documents into formal planning documents (analyst)
  - edit JAD/Plan
  - review JAD/Plan (all participants ⇒changes possible)
  - sponsor approval

3.2 Object Oriented Analysis

mapping onto waterfall process

- requirements ⇒object-oriented *analysis*
- architectural design ⇒system design
- detailed design ⇒object design

oo swe processes are *more continuous*

- which are *objects of interest*
- what are they doing
- UML use case, sequence, class diagrams

Object
"A discrete entity with a well defined boundary and identity that encapsulates state and behaviour; ..."
"A concept, abstraction or thing with crisp boundaries and meaning for the problem at hand..."
"Objects serve two purposes: The promote understanding of the real world and provide a practical basis for computer implementation"
"All objects have identity and are distinguishable."

J.
Ru
et

3.2.1 Obejct-Oriented Software Engineering

- software is defined with

  objects: entities, correspondence in real world (identity, attributes, methods)

  classes: abstact of objects

- objects encapsulate data (interface to data)
- interface is relevant, not implementation
- reuse through inheritance
- abstraction through polymorphism

3.2.2 Software Models

- software systems complex ⇒break down into subproblems
- software models ⇒abstraction of real world
- usage

  eary life-cycle stages: evaluate properties of real-world system (elicitate, document, verify, validate requirements, simulation)

  design stage: document architecture, assess performance/behaviour

  implementation/coding stage: automatically synthesize code (class skeletons, state machine behaviour)

  testing stage what to test (requirements models)

  maintenance stage: document existing system for evolution

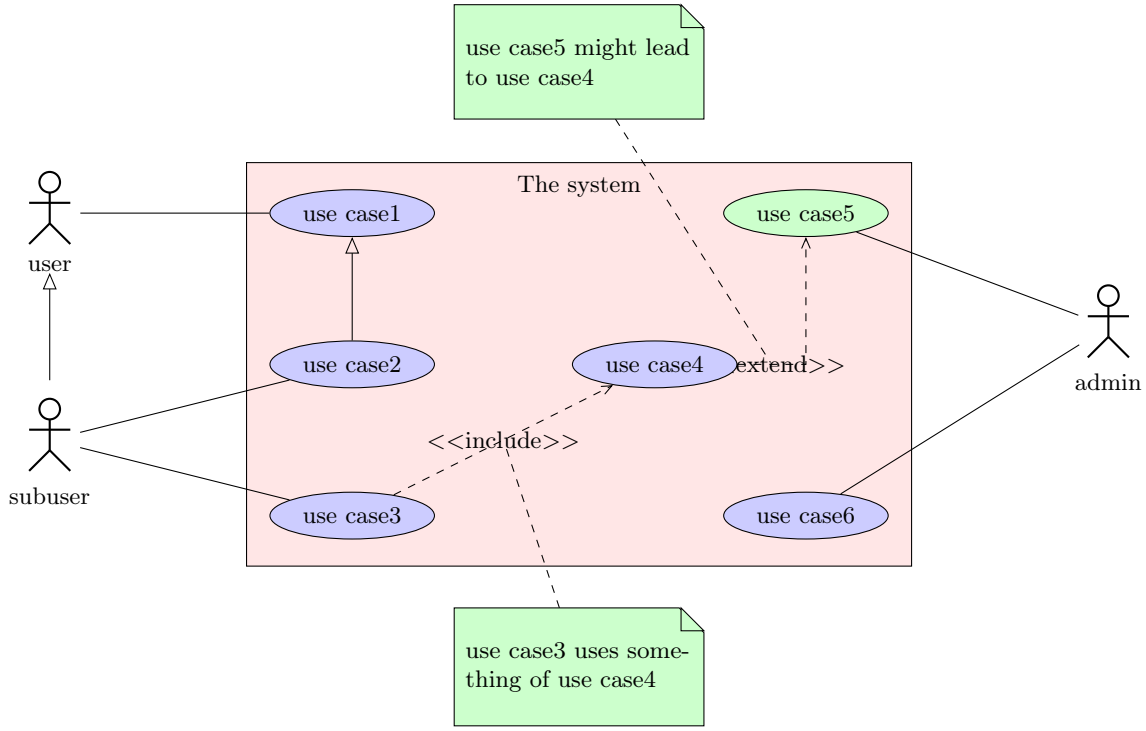
3.3 Analysis with the UML

- visual notation for analysis and design
- standardized by *Object Management Group* `http://www.omg.org`

3.3.1 Static

Use Case Diagrams

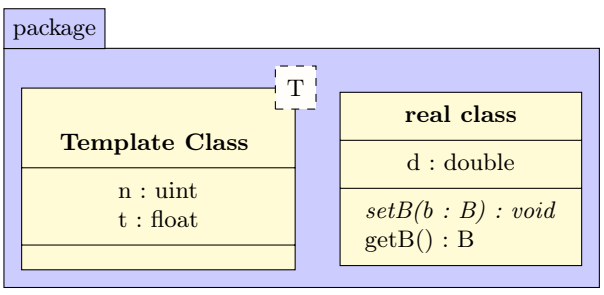
- often first step in OO-dev-process
- a use case represents a usage scenario (atomic)
- documents the functionalities the system provides to user ("what" does the system do)
- actors (anything or anyone interacting with the system, provide input/output) and principal use cases (interaction between actor and system)



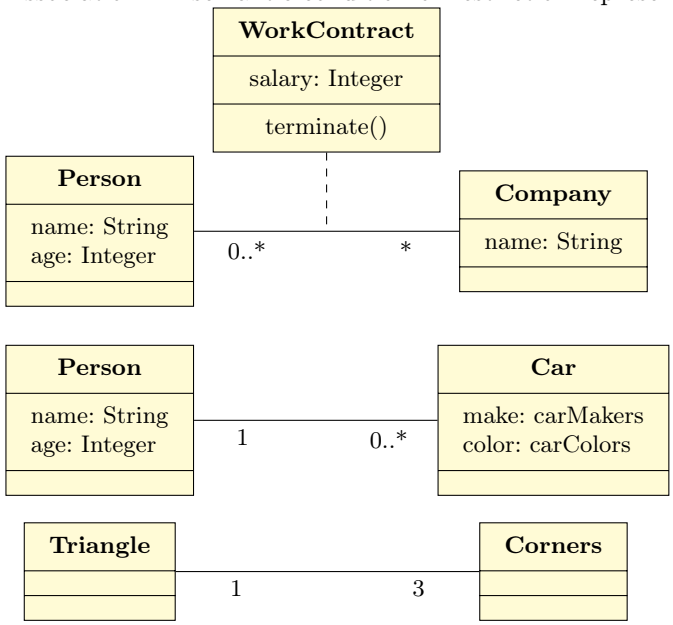
- «include» ⇒a use case uses functionality from another use case
- «extend» ⇒exceptional or optional behaviour

Class Diagrams:

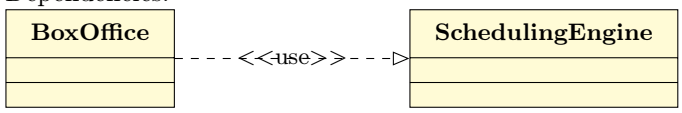
- Phenomen (object in the world of a domain as perceived
- Concept (properties iof a phenomen) is a 3-tuple: name (distringuishes it from others), purpose (properties that determine, if phenomen is member of concept), members (phenomena, that are part of concept)
- Type (abstraction in context of programming (name: int)
  `SimpleWatch` or `Firefighter`
- Class (abstraction in the context of oo-languages, encapsulates state(variables) and behaviour(methods))
- Instance (existing instance of a class) `myWatch:SimpleWatch` or `Joe:Firefighter`



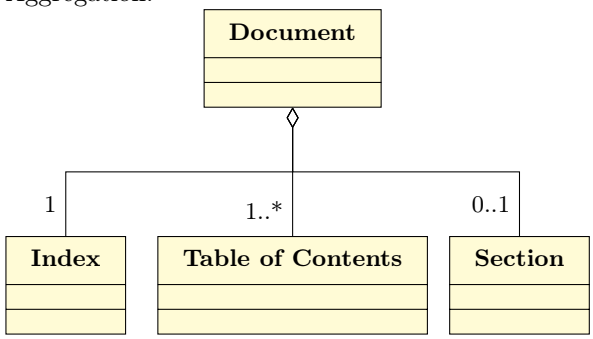
- Generality (most general aspect of problem)
- Abstraction (not individual objects, what they have in common)
- "The descriptor for a set of objects that share the same attributes, relationships and behavior. A class represents a concept within the system being modeled."
- used during requirement analysis (model problem domain concepts), system design (model subsystems and interfaces), object design (model prog language classes)
- Association: "A semantic condition or restriction represented as an expression."



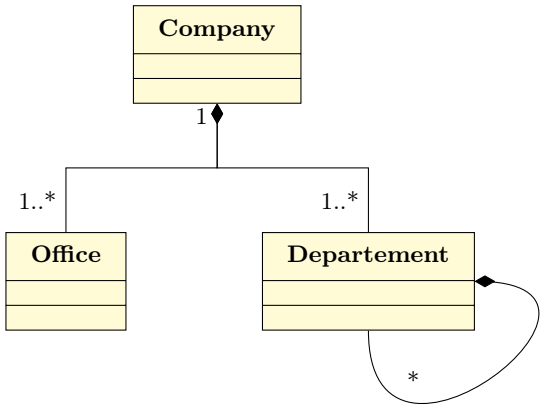
- Dependencies:



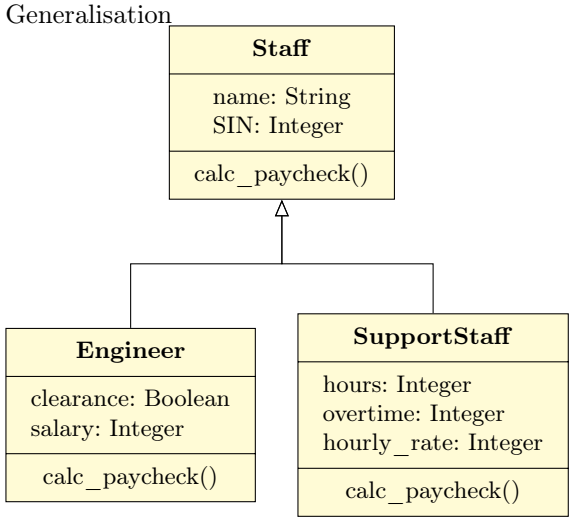
- Aggregation:



- Composition



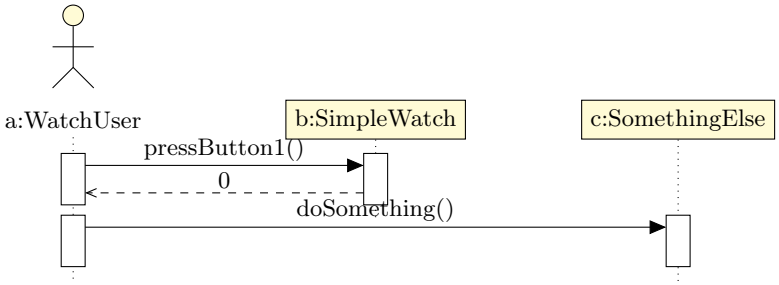
- Generalisation



- Visibility (+ public, - private, # protected)

3.3.2 Dynamic

Sequence Diagrams used during requirement analysis (refinde dynamic behaviour), design (document subsystem interfaces), good for real-time specifications



- classes ⇒columns
- messages ⇒arrows
- return values ⇒dashed arrows
- activations ⇒narrow rectangles
- lifelines ⇒dashed lines

State Chart Diagram state machine description (only for dynamically interesting objects)

Dynamic Modeling: Collection of State Chart Diagrams

Example Use Case Format

use case name	ReportEmergency
entry condition	FieldOfficer activates "ReportEmergency" function
flow of events	FRIEND responds by presenting form FielOfficer fills out form Dispatcher reviews information
Exit condition	FieldOfficer receives ack and response

3.3.3 Object Oriented Analysis

- use-case driven approach (followed by most UML-based approaches)
- linguistic analysis method (requirements in natural language ⇒cast into OO analysis model, more traditional OO approach)
- identify Entity Objects (persistent information tracked by system)
- indentify Boundary Objects (interactions betwen actor/system)
- identify Control Objects (responsible for realizing use cases)
- identify Associations, Aggregates, Attributes
- Model State-Independent Behaviour of objects
- Mapp Use Cases with Sequence diagrams
- Model Inheritance Relationships
- Review Analysios Model

3.4 Document Requirements / SRS

Requirement Specification: "A specification that sets forth the requirements for a system or system component; . . . typically included are functional requirements, performance requirements, interface requirements, design requirements and development standards."
"A *software requirements specification* is a document containing acomplete specification of **what** the system will do without saying how it will do that."

- contractual agreement (⇒litigation)
- shall ⇒mandatory
- should ⇒desired
- ambiguity ⇒to be prevented
- IEEE 830-1998
- DIN 69905 Lastenheft/Pflichtenheft
- SRS by Sommerville

Chapter	Description
Preface	expected readership, version history, rationale for creation of new versions, summary of changes
Introduction	need for the system, systems functions, how system works with other systems, how system fits into buisness
Glossary	define technical terms, assume no knowledge
User requirements definition	services provided for user, nonfunctional also described (natural language, diagrams, understandable notations, process and product standards
System architecture	high-level overview of anticipated system architecture, distribution of functions across system modules, reused components highlighted
System requirements specification	functional & nonfunctional requirements in detail, define interfaces
System models	graphical system models, relationships between system components, systen and environment, data-flow models, semantic data models
System evolution	fundamental assumptions on which system is based, anticipated changes due to hw evolution, changing user needs, ...
Appendices	detailed, specific information related to application, hw/database descriptions
Index	alphabetic index, index of diagrams, index of functions, index of tables, ...

- IEEE 830-1998 SRS
  1. Introduction
     (a) Purpose
     (b) scope
     (c) definitions, acronyms, abbreviations
     (d) references
     (e) overview
  2. overall description
     (a) product perspective
     (b) product functions
     (c) user characteristics
     (d) constraints
     (e) assumptions and dependencies
     (f) apportioning of requirements
  3. specific requirements
     (a) external interfaces
     (b) functions
     (c) performance requirements
     (d) logical database Requirements
     (e) design constraints
     (f) software system attributes
     (g) organizing the specific requirements
     (h) additional comments
- Lastenheft (DIN 69905), vom Auftraggeber erstellt
  1. Ausgangssituation/Zielsetzung
  2. Produkteinsatz
  3. Produktübersicht

4. Funtionale Anforderungen
5. Nichtfunktionale Anforderungen
   Benutzbarkeit, Zuverlässigkeit, Effizienz, Änderbarkeit, Übertragbarkeit
6. Risikoakzeptanz
7. Skizze des Entwicklungszyklus und der Systemarchitektur
8. Lieferumfang
9. Abnahmekriterien

- Pflichtenheft (DIN 69905)
  vertraglich bindend, präzise, detailiert, vollständige Beschreibung der zu erfülenden Leistung, Akzeptanztests
  vom Aufgragnehmer erstellt, vom Auftraggeber bestätigt

1. Zielbestimmung (Musskriterien, Wunschkriterien, Abgrenzungskriterien (sollen nicht erreicht werden))
2. Produkteinsatz (Anwendungsbereiche, Zielgruppen, Betriebsbedingungen)
3. Produktübersicht
4. Produktfunktionen (genau, detailiert)
5. Produktdaten (langfristig zu speichernde Daten aus Benutzersicht)
6. Produktleistungen (Anforderungen Zeit, Genauigkeit)
7. Qualitätsanforderungen
8. Benutzeroberfläche (grundlegende Anforderungen, Zugriffsrechte)
9. Nichtfunktionale Anforderungen (Gesetze, Normen, Sicherheitsanforderungen, Plattformabhängigkeiten)
10. technische Produktumgebung (SW und HW, organisatorische Rahmenbedingungen, Schnittstellen)

3.4.1 Funktionale Anforderungen

"funktionale Anforderungen legen fest, welche Dienste (aus Sicht des Benutzers) das System anbieten soll/welche Aufgaben es erfüllen soll"

- Aufbau: Anforderungsnummer: Bezeichnung, kurze Beschreibung
- was, nicht wie
- eindeutig/widerspruchsfrei
- zentrale Vorgaben für Systementwicklung
- Beschreibung funktionale Anforderungen ⇒use-cases

3.4.2 Nichtfunktionale Anforderungen

„nichtfunktionale Anforderungen sind Anforderungen, welche die durch das System zu leistenden speziellen Funktionen nicht direkt betreffen"

- können sich auf wichtige System-Eigenschaften beziehen
- können Beschränkungen definieren
- sind selten an einzelne Systemfunktionen gebunden
- „sichtbare" Eigenschaften der Software (Geschwindigkeit, Verfügbarkeit, ...)
- sind of relevanter als funktionale Anforderungen (Unbedienbarkeit, ...)

- Typen von funktionalen Anforderungen:

  Produktanforderungen Geschwindigkeit, Speicherbedarf, akzeptable Fehlerquoten , portierbarkeits-Anforderungen, Anforderungen bezüglich Benutzbarkeit

  Unternehmensanforderungen Vorschriften für die Entwicklung (bestehende Standards, spezielle Anwendungen), Umsetzungsanforderungen (Programmiersprache, Entwurfsmethode, Lieferanforderungen)

  externe Anforderungen Kompatibilität (ausführbar auf div. Geräten/Betriebssystemen, language-packs), rechtliche Anforderungen (Datenschutz, Speicherung), ethische Anforderungen (Langzeitspeicherung, keine fremden Kundendaten anzeigen, unfreundliches Design)

- oft allgemein formuliert ⇒kann später zu Problemen führen
- direktes Überprüfen oft schwer ⇒Tests und Metriken (festlegen)

4 Design

4.1 Design Objectives

What ⇒How
Manage Complexity
Design for Change (anticipate change at all stages, reduce cost of changes)

- design process should not suffer from "tunnel vision"
- design should be traceable to analysis model
- don't reinvent the wheel
- minimize intelectual distance between software and real-world problem
- uniformity and integration
- acommodate change
- degrade gently, when aberrant data/events/...encountered
- Design is not coding
- assess design for quality while it is created
- review design

4.2 Classical and Object-Oriented Design

4.2.1 Procedural Design

- decomposition of system into modules
- describe module interactions
- structure of procedure definitions/invokations
- no abstract data types/inheritance

Architectural Design: determine major elemts (modules) and relationships

Interface Design: how to design interrelation/communiccation/mutural reliance (contracts) of modules

Data Design: data structures

Procedural Design: determine procedural description for elements

Module

- well defined component of system (provides set of services to other modules, consists of (name,

interface, body), information hiding)

- master complexity, facilitate documentation, enable teamwork
- interface descrition ("contract server/client") languages (CORBA IDL, Eiffel)
- *Cohesion* (intra module bindings, "degree to which a module performs one and only one function")
- *coupling* (inter module bindings, "degree to which a module is connected to other modules" (direct access, data transfer, ...))
- desired: high cohesion, low coupling

4.2.2 Object-Oriented Design

- model system als collection of classes + objects
- description of object interfaces
- encapsulation, abstract data types
- reuse through inheritance
- polymorphism

Activities

- partition model into subsystems
- identify concurrency
- allocate subsystems to processor tasks
- develop ui design
- choose strategy for data management
- identify global resources (and control mechaninsm, task management)
- consider boundary conditions (and how to be handled)
  - Sonderfälle (Systemstart, Initialisierung, Herunterfahren, Schwere Fehler und Ausnahmen, beschädigte Daten, Netzwerkfehler, ...)
  - ⇒**Boundary Use Case** (für alle Subsysteme und Dauerhafte Objekte: Konfiguration, Start/Herunterfahren, Fehlerbehandlung)
  - Für Komponentenfehlertyp (Netzerkausfall, ...) wird entschieden, wie System reagieren soll ⇒Anlegen eines **Exceptional Use Case**
- review, consider trade-offs

Booch Method: macro development process (architectural planning, patritioning, layering) + micrp development process (rules for implementing specifics)

Rumbaugh Method: system design (analysis models represent system, layout of components, partitioning into subsystems, taking exec environment into account) + object design (design of algos/data structures)

Jacboson Method: traceability from analysis model, adaptrion of analysis model to real-world env, primary design objects categorized (interface, entity, control), determine communication ⇒organize into subsystems

Wirfs-Brock Method: analysis ⇒design, def of protocols through inter-object contracts, specification of classes and subsystems

- maintenance (objects work standalone), reuse (objects can be used again), reduce semantic gaps (real-world ↔ software)

Generalisation & Inheritance

- objects are members of classes (define attributes, operations)
- class hirarchy ⇒Generalisation
- generalistaion in UML ⇒inheritance in OO
- abstraction ⇒classify entities
- inheritance graph ⇒organistational knowledge about domain/system
- *Cohesion* how well fits the functionality of a class together
- *Coupling* B subclass of A ⇒B inherits from A ⇒coupling between A/B, reflects real-world structure (not design decision)

4.3 Design with the UML

4.3.1 types of models

design models: show objects, object classes, relationships

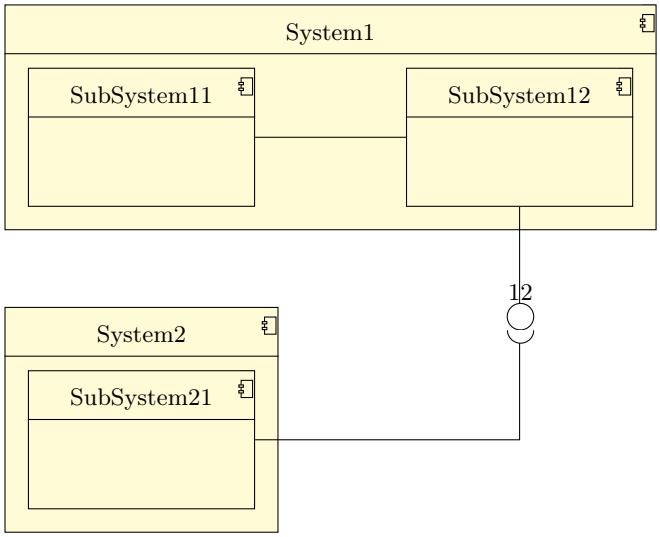
static models: describe static structure, object classes and relationships

dynamic models: describe dynamic interactions between objects

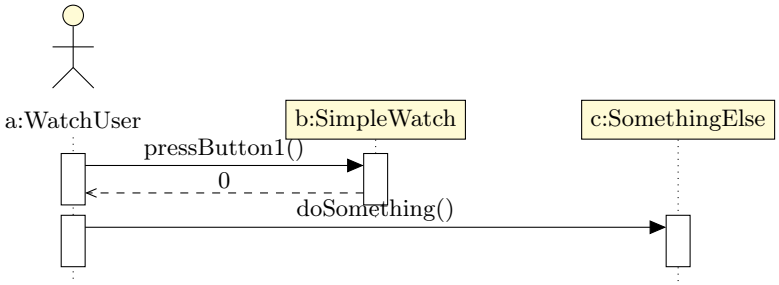
4.3.2 Typically used models

subsystem/component diagrams: structural refinement of objects

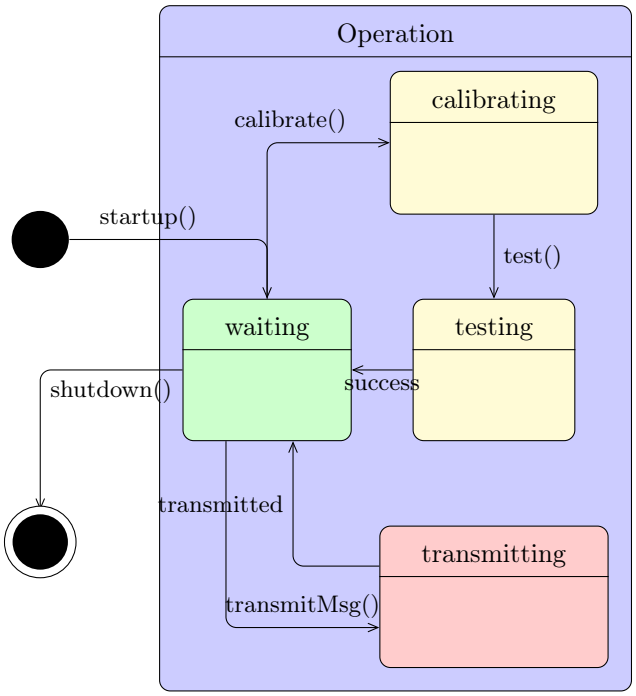
A **UML component** "represents a modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces " ⇒subsystem



sequence diagrams: model order of interactions among objects



statechart diagrams: model state behaviour of one object, show how objects respond to and state transitions triggered by service requests, concurrency possible (seperate by dashed line)



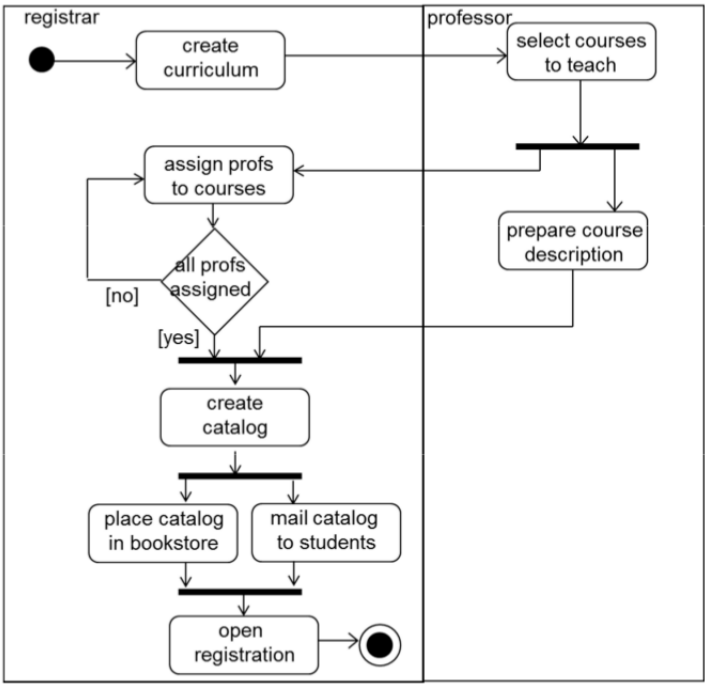
acitivity diagram: model concurrent behabiour of object

Deployment Diagram: represents alloc of different UML Nodes, HW/SW allocation ("like component diagram in 3D")

use-case diagram: model interactions (include description)

4.4 Interfaces and Contracts

- interfaces must be specified, s.t. other objects can be developed in parallel
- several interfaces per object/module possible
- use UML class diagrams
- operations A,B concurrent, if ...A...B... and ...B...A... is possible (logical parallelisation, execution on different hw, A/B not data-flow interdependent)
- use inherent concurrency to distribute system
- use **Activity Diagrams** to model concurrent workflows, like workflows, thick bars are sync-points, "swim-lanes" are separation denoting different systems



- in distributed systems: remote object invocation, middlware (hide the fact, that object is remote, CORBA, COM/DCOM, .net, Java RMi)
  - **CORBA** (Common Object Request Broker Architecture)
  - object request broker ORB (locate object, activate object, communicate request, return reply) server/servants don't need to be objects (C,Cobol)
  - **static** remote interface of CORBA object is known at compile time (code used to generate client stub)
  - **dynamic** remote interface not known at compile time
  - interface definition language IDL: interfaces need to be exactly specified (direct access to variables in remote classes, access only through interface, desired language-independent IDL, that compiles into access methods)
  - CORBA IDL interface specifies names + methods, extended use of C++ syntax

    IDL modules logical grouping of interface/type definitions, defines naming scope

    IDL interface methods that are available in CORBA objects implementing interface

    IDL methods specify signatures, parameters are labeled `in` (from client to server), `out` (from server to client) and `inout`, `oneway` (client will not be blocked on invoking), `raises` (exceptions)

    IDL attributes like public fields, may be readonly, get/put methods automatically generated

    IDL interface inheritance interface may be refinement of other interface

    IDL type names generated by compilter IDL prefix, type name, prefix number
  - CORBA object model (implements IDL interface, has remote object reference, capable of responding to remote invocations)
  - CORBA has no objects/classes ⇒use structs, classes cannot be arguments/results
  - CORBA can be documented with UML
  - CORBA architecture

    skeltons generated by IDL compiler, marshaling/unmarshaling of arguments/exceptions

    client stub/proxy in application language, generated by IDL interface definition, marshaling/unmarshaling of arguments/exceptions
- Design-by-Contract™⇒Eiffel language
  - server must *ensure postcondition*, can *assume precondition*
  - client must *ensure precondition*, can *assume postcondition*

– UML Object Constraints Language (OCL)

HashTable
numElements:int
put(key,entry;Object) get(key):Object remove(key:Object) containsKey(key:Object):boolean size():int

put precondition `!containsKey(key)`, postcondition `get(key==entry)`
get precondition `containsKey(key)`, postcondition `!containsKey(key))`
remove precondition `containsKey(key)`, postcondition `!containinsKey(key)`

- *subcontracts* (inheritance), subconstructor may
  - keep or weaken precondition (more may be let in)
  - keep or strengthen postcondition (less may come out)

4.5 Data Dictionaries

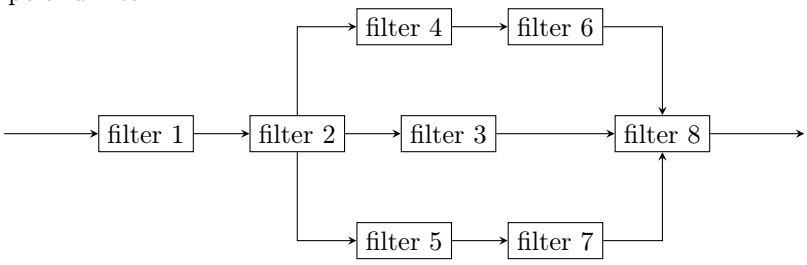
- low level data decisions ⇒late in design process
- representation of data structures only known to those with direct data interaction (information hiding)
- **Data Dictionary** list of all names, entities, realtionships, attributes used (name management, avoid duplication, store organisational knowledge)

4.6 Software Architecture and Patterns

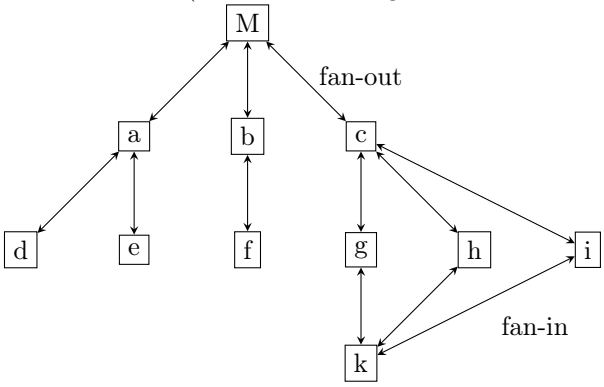
"The *architecture of a software system* defines that system in terms of computational components and interactions among thos components."
"...involves the description of elements from which systems are built, interactions among there elements, patterns that guide their composition, and constraints on these patterns."

Architectural Styles:

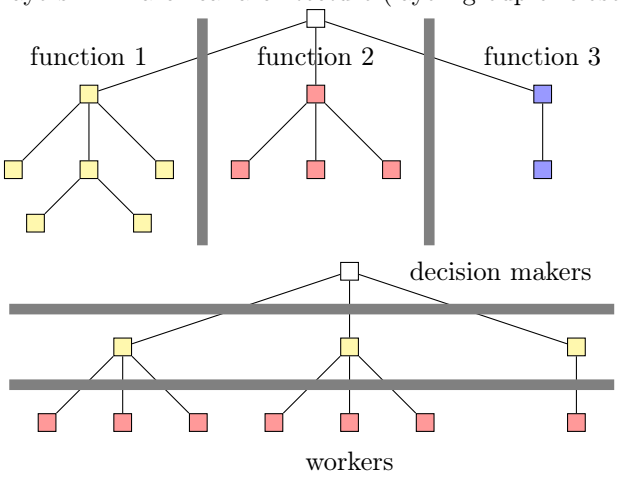
- pipe and filter



- call and return (recommended high fan-in, low fan-out)



- layers in hirarchical architecture (layer: group of closely related and highly coherent functionalities)



- opaque layering: vm can only callfrom layer below ⇒separation of concerns, maintainability, flexibility
- transparent layering: vm can call from any layer ⇒runtime efficiency (no parameter/message passing, data conversion, format changes)
- clients/servers (communication by recipients)

  client process wishing to access data/use resources/perform operations

  server: process managing data and shared resources

4.7 Design Patterns

"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." Christopher Alexander

- improved documentation, more abstract level of programming, improved communication
- patterns as components, classes

4.7.1 Adapter - Wrapper

Pattern and Name Classification Adapter Wrapper

Intent Anbieten eines alternativen Interfaces für eine Klasse

Motivation fremde Klasse mit unpassendem Interface in passendes Interface verwandeln

4.7.2 Brücke

Pattern and Name Classification Brücke

Intent Decouple an abstraction from its implementation so that the two can vary independently.

Motivation Eine Brücke kann eine dauerhafte Verbindung zwischen Abstraktion und Implementierung verhindern.

4.7.3 Decorator

Pattern and Name Classification Decorator

Intent Fügt individuellen Objekten Zuständigkeiten hinzu, aber nicht der Klasse

Motivation Bspw TextArea einen Rahmen und eine ScrollBar hinzufügen

4.7.4 Facade

Pattern and Name Classification Facade

Intent baut eine Klasse als Fassade vor viele andere Klassen, ermöglicht Methoden, die wieder Methoden in hinteren Klassen aufrufen, vereinfacht Schnittstellen

Motivation einfache Klasse für Zugriff von Außen, voller Zugriff auf Methoden des Basissystems

related patterns Wrapper, Mediator

4.7.5 Observer

Pattern and Name Classification Observer / publish-subscribe

Intent Abstrakte Kommunikationsmöglichkeit

Motivation Das beobachtete Objekt bietet Anmeldung für Observer an, informiert diese dann über interne Änderungen. Muss Struktur des Observers nicht kennen, meldet einfach jede Änderung weiter ⇒Observer muss darauf reagieren können.

4.7.6 Proxy/Stellvertreter

Pattern and Name Classification Proxy, Stellvertreter, Surrogat

Intent Kontrollierter Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreters

Motivation Proxy hält eine Referenz auf das reale Objekt, stellt gleich Schnittstelle zur Verfügung

4.7.7 Singelton

Pattern and Name Classification Singelton

Intent Klasse mit genau einer Instanz, normalerweise globaler Zugriff

Motivation Klasse hat komplette Zugriffskontrolle,

Applicability Bpsw Druckerbuffer, Filesystem, Window Manager

4.8 From Design to Code

4.8.1 Mapping Design Models to Code

optimization meet performance requirements (reduce multiplicity of associationsm redundant associations)

realisation of associations map asociations to source code

mapping of contracts to exceptions raise (and handle) exceptions, when contract broken

mapping of class models to storage schema select persistend storage strategy (database, flat files), define relational database schema

implementation of visibility attributes private, protected, public

4.8.2 Implementing Interface Contracts

- most OO languages no built-in support for contracts (except Eiffel)
- some OO languages have built-in exception handling support (throw-catch in Java)
- check each *precondition* before beginning the method
- check each *postcondition* after the method
- check *invariants* before and after method

4.8.3 Maping to Relational Databases

- map UML-constructs to tables (not everything can be mapped)
- class → table
- class attribute → column in table
- class instance → row in table
- many-to-many associations → own table
- one-to-many → foreign key
- no direct mapping → do something in sql,...

4.9 Documenting Design

4.9.1 System Design Document - SDD

1. Introduction
   (a) Purpose of system
   (b) Design goals
   (c) Definitions, acronyms, abbreviations (glossary)
   (d) References
   (e) Overview
2. Current Architecture (only if there is current system available)
3. Proposed Software Architecture
   (a) Overview (funtionality of each subsystem)
   (b) Subsystem Decomposition (responsibilities of indiv. subsystems, main contrib of SDD)
   (c) Hardware/Software mapping
   (d) Persistent Data Management (database usage and schemes)
   (e) Access control and security (access rights, encryption/key management)
   (f) Global software control (initiation of requests, subsystem synchronization, concurrency)
   (g) Boundary Conditions (startup, shutdown, error behaviour)
4.

4.9.2 Object Design Document - ODD

1. Introduction
   (a) Object design trade-offs (buy vs build, memory vs response time, ...)
   (b) Interface documentation guidelines, coding conventions
   (c) Definitions, acronyms, abbreviations
   (d) References
2. Packages
   (a) package decomposition and file organistaion of code
3. Class Interfaces
   (a) public interfaces (incl operations, attributes) of each class
   (b) dependencies with other classes

part can be done automatically with e.g. JML/Javadoc

5 Testing and Software Quality Assurance (SQA)

- about 1 faults per 1000 lines of code
- verification and validation techniques (code analysis, review, testing, formal verification), the earlier, the better (cost)

  verification *prove* that a product meets its specification (formal verification, correctness proofs)

  validation *experiment* with system, to show iot meets requirements (simulating, testing, model cheking)
- code quality assessment (software metrics)

5.1 Software Testing and Strategies

5.2 Reviews and Inspections

5.2.1 Reviews

- technical meetings (walk-through, inspections)
- objectives: software quality assurance, training
- non-objectives: progess review, budget, trouble-shooting, reprisals, political intrigue
- review leader, standards bearer (SQA), mainenance oracle (devils advocate), recorder, user representative, producer, reviewer
- evaluate before review
- review product, not producer (ask questions instead of accusing, avoid criticising style ⇒technical correctness)
- raise issues, don't resolve them

5.2.2 Code walk-through

- "playing the computer" ⇒"interpret code"
- 3-5 participants
- discover, don't fix errors
- moderator, secretary, auditors, designer (explain own design)

5.2.3 Code Inspections

- similar to code-walkthrough, different goal: look for commonly made mistakes, no simulation of computers behaviour
- unitialized variables, jumps into loops, incompatible assignements, nonterminating loops, array boundaries, storage allocation/deallocation, parameter mismatches
- analysis tools can be helpfull

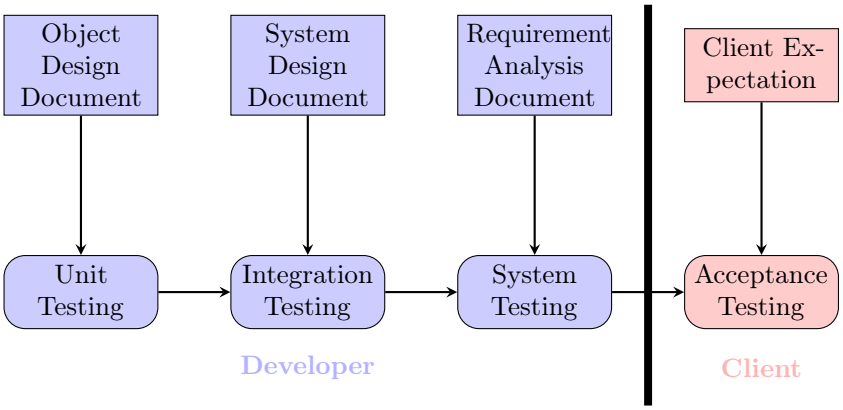
5.3 Test Coverage

white box testing source is revealed ((code) structural testing)

black box testing no source available (test against requirements specification)

grey box testing some internal state variables are revealed

5.3.1 Types of Testing



unit/module Testing check that one module meets design specifications (often during programming)

- to be tested: methods within object, classes with several attributes/methods, components consisting of multiple classes + internal interfaces
- it's not testing the system as a whole
- complete coverage: testing all operations with all possible combinations of parameters + setting and testing all possible combinations of object attribute values + testing all possible states ⇒number blowup
- two types of tests:
  normal ⇒unit does, what it is supposed to do
  robustness test ⇒when subjected to "hostile environment" unit is robust

integration testing done during assembly of complete system, test that modules work together, test that system meets requirements (done by integrators, test specialists)

system testing testing the system as a whole

acceptance testing test that customer performs when software is delivered (may be fixed in requirement specification or contract)

regression testing performed during maintenance/when component has changed (ensure that what worked in the past, will work in the future)

stress testing test system under extreme conditions

5.3.2 formalized testing

D**:** input domain

R**:** arbitrary set, results

P**:** patial function, describing programm input/output behaviour: $P : D \to R$

OR**:** the output value requirements (from SRS)

testing: determine where for given $d \in D, P(d)$ meets requirements stated in OR, if ⇒sucess, else fail

$d \in D$**:** is test case

$T \subseteq D$**:** is test set/suite

T**:** is sucessfull, iff all $d \in T$ sucessfull

$C \subseteq 2_f^D$**:** is selection criterion, 2_f^D is set of all finite subsets of D

T**:** satisfies C iff $T \in C$

for all i, k**:** let T_i, T_k satisfy C, C is consistent if (T_i is sucessfull iff T_k is sucessfull)

C**:** is complete if, when P is incorrect, there is a T satisfying C which is unsucessfull

if C **complete and consistent:** any test T that satisfies C decides P's correctness

5.3.3 selective testing - how many paths to test?

Statement Coverage select T s.t. by executing $P(d)$ for every $d \in T$, **each statement is at least executed once** statement ⇒square

edge coverage select T s.t. **each edge in control flow graph is traversed at least once**

condition coverage select T s.t. **edge coverage is satisfied and all possible values for consituent proposition inside conditions are exercised at least once** each check must be executed

multiple condition coverage each boolean combination of the consitutent predicates of every condition needs to be exercised at least once each combination (of sucess/failure) at least once

path coverage select T s.t. all paths from an inital ndoe to a final node in the control flow graph are traversed at least once

5.3.4 Basis Path Testing

- assess how many tests are needed to achieve statement & path coverage (prog is single entry, single exit, control flow graph available, all decisions binary)
- calculate cyclomatic complexity $V(G)$
  - $V(G) = \#$ of simple decissions $+1$
  - $V(G) = \#$ of enclosed areas $+1$
- loop testing (n is number of allowable passes)
  1. skip the loop
  2. one pass
  3. two passes
  4. m passes through loop, $m < n$
  5. $(n-1), n, (n+1)$ passes through the loop

5.3.5 testing based on contracts

- possible test selection criteria
  - input conform with the precondition
  - inputs with a violated precondition
  - inputs where key element is of correct type
  - inputs where key element is of wrong type

5.3.6 Acceptance Testing

- demonstrate system is ready for operational use
- choice of tests by client, is conducted by client
- may be part of contract
- alpha test: tests in devs environment, beta test: tests in clients environment

5.3.7 integration testing

- driver: component that calls the `testedUnit`, controls test cases
- stub: component that the `testedUnit` depends on ("fake")
- big bang - just put together and see, if it works
- top-down - tests the top layer first, then downward ⇒stubs are needed, test can be designed by the functionality of system

- bottom-up - subsystem in lowest layer testet first ⇒drivers are needed (test ui (most important part) last)
- sandwich - combination of bottom-up and top-down, converges at target layer ⇒does not test individual layers thouroghly
- modified sandwich - combination of all other tests

5.3.8 Testplan

1. Einführung (Beschreibung der Testobjekte)
2. Bezug zu anderen Dokumenten (RAD, SDD, ...)
3. Systemübersicht (strukturelle Aspekte des Tests, Übersicht auf die zu testenden Objekte)
4. Features zum Testen und Nicht-Testen (funktionale Aspekte, Beschreibung der zu testenden Features)
5. Pass/Fail-Kriterien
6. Approach (Herangehensweise an Testprozess, Grund für Wahl einer Teststrategie)
7. Suspension & Resumption (Kriterien zum Unterbrechen eines Tests, Spezifizierung der Testabläufe nach Wiederaufnahme)
8. Testmaterial (benötigte Resourcen)
9. Testfälle (Test Case Specification, Test Incident Report)
10. Testing Schedule (Verantwortlichkeiten, Personalbedarf, Risiken, Kosten, Ablaufplan)

Test Case Specification

1. Testfallbezeichner (Name)
2. Test items (Komponenten und Features)
3. Eingabespezifikationen
4. Ausgabespezifikationen
5. Umgebungsanforderungen (HW/SW)
6. Besondere prozedurale Anforderungen (Zeitlimits, ...)
7. Abhängigkeit zwischen Fällen

5.4 Software Metrics

5.4.1 McCabe's Cyclomatic Complexity

- determine control flow graph
- Cyclomatic Complexity $C = e - n + 2p$ (e - edges, n - nodes, p - number of connected components (1 for totally connected))
- C determines number of paths
- no GoTos, single entry/single exit ⇒$C - 1$ number of decission nodes
- $3 \leq C \leq 7$ ⇒good values, $C = 10$ ⇒maximum

5.4.2 Software Product Metrics

fan-in/fan-out number of functions or methods, that call other functions or methods (X). Fan-out is the number of functions, thar are called by X. High Fan-in shows tight coupling (changes to X not good), high Fan-out suggests high complexity

length of code larger programm ⇒more errors

Cyclomatic Complexity

Length of Identifiers longer identifiers are simpler to understand

depth of conditional nesting deep nesting is hard to understand

fog index average length of words and sentences in documents, higher fog is harder to understand

5.4.3 Object-Oriented Software Metrics

depth of inheritance tree deeper inheritance tree ⇒more complex design

Method fan-in/fan-out distringuish between internal (good) and external (bad) methods, see above

weighted methods per class number of methods in a class weighted by its complexity ⇒high value is bad

number of overriding operations high value indicates, superclass not good model for subclass

5.5 Formal verification

das lass ich mal weg, das seh ich als nicht wichtig an