# Softwaretechnik II

Lernzusammenfassung zur Vorlesung am KIT
Lukas Bach - lbach@outlook.de - lukasbach.com

# 1 Regeln der Softwaretechnik

- Brooks Law: Adding manpower to a late software project makes it later.
- Boehms First Law: Errors are more frequent during requirements and design.
- Dijkstra: Testing shows the presence, not the absence of bugs.
- Lehmans Law 1: A system that is used will be changed.
- Lehmans Law 2: An evolving system increases its complexity unless work is done to reduce it.
- Parnas Law: Only what is hidden can be changed without risk (Information hiding).

# 2 Clean Code

## 2.1 SOLID

- *S*ingle Responsibility Principle (SRP)
- *O*pen Closed Principle (OCP)
- *L*iskov Substitution Principle (LSP)
- *I*nterface Segregation Principle (ISP)
- *D*ependency Inversion Principle (DIP)

### 2.1.1 Single Responsibility Principle

Each responsibility deals with one core concern.

Bad smell: Big class.

Extract classes, separate commands (side-effects) from queries (pure requests).

### 2.1.2 Open Closed Principle

Software entities should be open for extension, but closed for modification.

Modify behaviour by adding new code, not changing old code.

### 2.1.3 Liskov Substitution Principle

Functions that use pointers to base classes must be able to use objects of derived classes without knowing it.

Use “Reactangle” instead of “Square”.

### 2.1.4 Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.

Interfaces should be separated if used by different clients.

Instead of (Human::Worker, Robot::Worker; use Human::Feedable, Human::Workable, Robot::Workable)

### 2.1.5 Dependency Inversion Principle

High-level- and low-level modules should not depend upon each other, but depend upon abstractions.

Instead of (Copy->Keyboard, Copy->Printer) use (Copy->Reader, Copy->Writer).

Why inversion? Copy no longer depends upon a separate module (Keyboard), but depends upon an internal module (Reader) which an external module (Keyboard) inherits.

## 2.2 More Principles

- *Law of Demeter*: A module should not know about the innards of the object it manipulates.
  - No “getClassA().getClassB()….getNeededVal()”
- *Boy scout rule*: Leave the campground cleaner than you found it.
- *Principle of Least Surprise*: Entities shoukd implemented most expected behaviour.
- *DRY*/Don’t repeat yourself: No code duplication.
- *KISS*/Keep it simple, stupid.
- *YAGNI*: You Aint Gonna Need It: Only implement required features. Features are costly.
- *SLA*/Single Level of Abstraction.
- Refactoring
  - Bad code smells: Long method, duplicated code, Feature envy (class excessively calls another classes methods), data class, god class, inappropriate intimacy.

## 2.3 Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

CI-enabled dev-process:

1. Manual
   a. Change code artifacts
   b. Test locally
   c. Commit change
2. Automated
   a. Dependency materialization
   b. Compiliation
   c. Testing
   d. Bundling
3. Automatically provide dev feedback
4. Manual: Fix broken builds

Rapid deployment via Continuous Delivery.

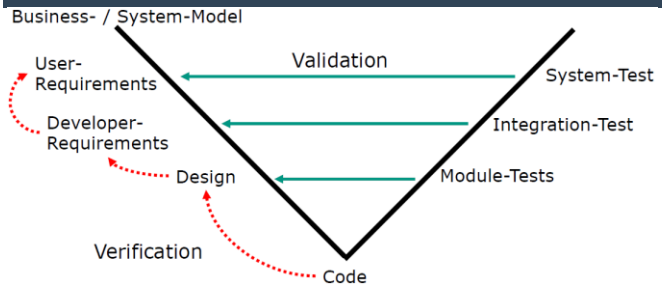
Important dev artifacts in central repo required, including: Source code, tests (unit+integration), build automation

artifacts, build job descriptions (Jenkinsfile), deployment description for testing.

Fail early!

# 3 Reviews

## 3.1 V-Model




Shows relations of different software artifacts.

- Validation: case based check on expected behaviour.
- Verification: Check whether refinement relation holds between two documents (requirements <-> code, requirements <-> architecture/design, architecture/design <-> code).

- Forms of reviews: Inspection, Team Review, Walkthrough, Pair Programming, Change-based review, Pass-around, Ad-hoc review.

- Dangers of reviews: Testing is omitted, Authors get frustrated.
- Danger for reviews: Managers cut time near deadline, unprepared people, Code obfuscation to protect authors.
- Benefits of reviews: Quality, correctness, understanding, teaching of style, better readability.
- Roles during review: Planning, Overview, Preparation, Meeting, Rework, Follow-Up, Causal Analysis.
- Roles: Author, Moderator, Reader, Recorder, Verifier

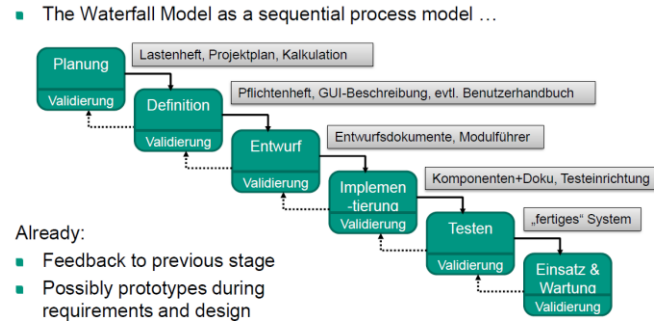
## 3.2 Psychological Interaction Patterns

Success of reviews depends on social and psychological factors.

- Alcoholic pattern
  - Roles: Addicted (bad habit), Helper, Punisher, Ashamed
  - Bad habits: Late, not prepared, etc
- Now I have got you
  - Roles: A discoveres that B made a fault
  - B has no option except to respond calmly
- See what you made to me
  - Roles: A does something, B interferes and causes A to make a mistake.
- Hurried pattern
  - Roles: A is overworked, but accepts more tasks. B increases workload of A, causing more mistakes by A.
- If it were not for you
  - Roles: A is doing something. A to B: “If it were not for you, I would have achieved …”
- Look how hard I tried
  - Roles: A works on foreseeably failed project. A increases work to show that project failure is not A’s fault.
- Schlemiel pattern
  - Roles: A (schlemiel) makes errors and affects B’s work. B detects error and complains at A, A apologizes. Endless iteration, manager wonders about slow progress of B.
- Yes-but pattern: …
- Wouldn’t it be nice if – pattern: …

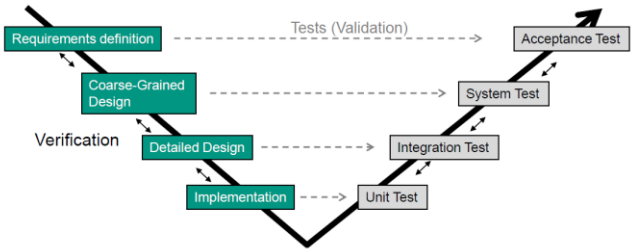
# 4 Software development processes

## 4.1 Waterfall model

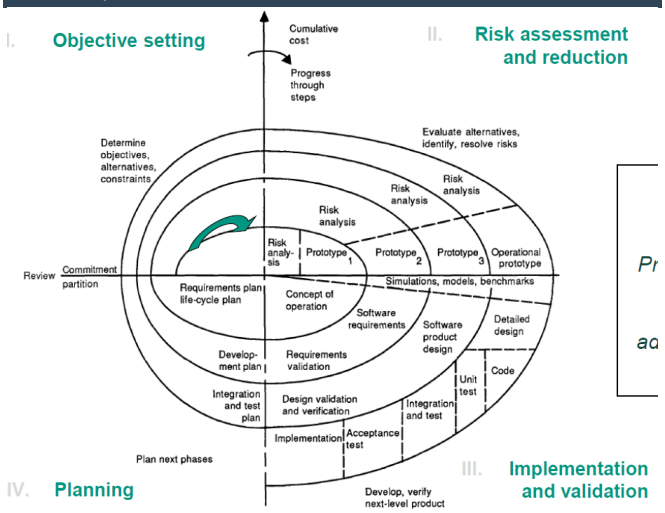



Problem: System cannot be specified in advance.

## 4.2 V-Model

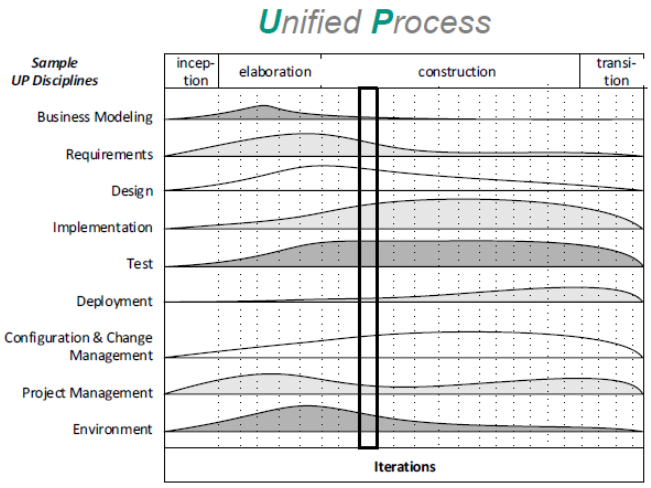


## 4.3 Spiral model




## 4.4 Unified Process (UP)

Waterfall model, original V model and spiral model are building blocks for software dev processes.




- 4 abstract **PHASES**
  - *Inception*: Vision and business case for project, feasible?, rough cost range.
  - *Elaboration*: core risky architecture is programmed and tested, most requirements are discovered and stabilized, major risks are mitigated.
  - *Construction*: Iterative implementation of remaining lower risk elements, prep for deployment
  - *Transition*: Beta tests, deployment
- 9 **DISCIPLINES**
  - 6 engineering disciplines: Business modeling, Requirements, Design, Implementation, Test, Deployment
  - 3 supporting disciplines: Configuration&Change Management, Project Management, Environment

### 4.4.1 Rational Unified Process (RUP)

Specific implementation of UP, provides disciplined approach: roles->who?, activities->how?, artifacts->what?

Best practices:

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Model software visually
- Verify software quality
- Control changes to software

# 5 Agile Development

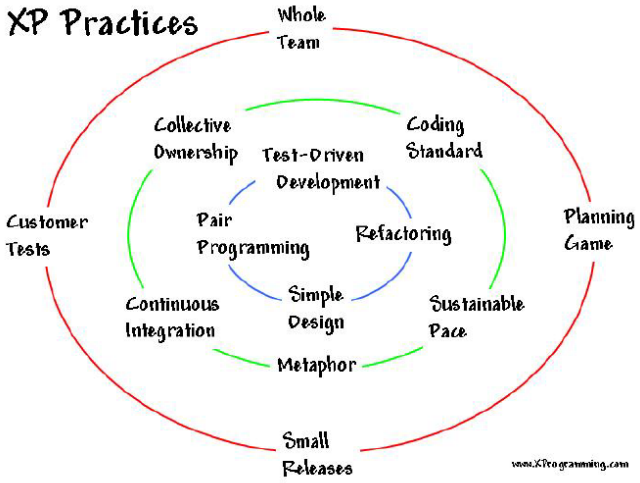
- Individuals and Interactions over Processes and tools
- Working software over Comprehensive documentation
- Customer collaboration over Contract negotiation
- Responding to change over Following a plan

Important: Cannot plan ahead, need to monitor continuously.

## 5.1 Extreme Programming (XP)

Main values: Communication, Simplicity, Feedback, Courage.

Some principles: Quick delivery, rapid feedback, keep it simple, incremental change, embrace change.




project-wide practices
team-specific practices
individual (developer-specific) practices

Criticism: Not usable for large projects, missing documentation, missing cross-project reuse, client needs to cooperate, some practices not fully validated.

## 5.2 Scrum

- Product owner is responsible for product list (collection of features/"user stories")
- Sprint Planning Meeting before each sprint. Which features have highest prio and can be delivered? -> Sprint Backlog.
- Sync during Daily Scrum, sprint progress is recorded in Burn Down Chart.
- Scrum Master trains the team, removes impediments and ensures effectiveness.
- Functionality dev during sprint
- Presentations during Sprint Review Meeting. Improvements discusss during Retrospective Meeting.

3 *Roles*: Product Owner, Scrum Master, Team

4 *Meetings*: Planning, Daily, Review, Retrospective

3 *Artifacts*: Product Backlog, Sprint backlog, Sprint Featureincrement.

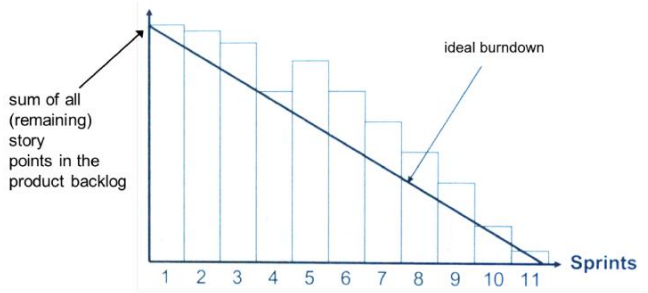
Project planning on 3 levels: release planning, sprint planning, workday planning. Entire project plan is unrealistic.

### 5.2.1 Roles

- Product Owner
  - Defines product, what needs to be delivered.
- Scrum Master
  - Responsible for process, not for staff! Moderator role.
- Team
  - Multifunctional, knowledge complete, small, fulltime members, self organized, working physically close together, common goal, respect.

### 5.2.2 Sprint velocity

Diagram showing how many story points are cleared compared to expected story point clearing.




### 5.2.3 Large projects with more than 1 team

- Start small and grow organically: Start with one team, split at the end and add new members. Add one team every 2-3 sprints.
- Minimize dependencies between teams.
- Daily Scrum of Scrums, with representatives per team.

### 5.2.4 Distributed projects

- Never separate Scrum Master and Team
- Distribute teams stepwise. Core team which gets seperated, one core team member per new team.
- Exchange team delegates from time to time.

# 6 Requirements Engineering

Errors are more expensive to fix the longer they exist. Good RE avoids req errors and saves error fixing costs later.

Software requirements are described…

- Adequate (describes customer needs)
- Complete
- Consistent (No contradictions)
- Understanable
- Unambiguous (no wrong interpretation)
- Verifiable
- Suitable for risk

Types of requirements: Functional reqs, Quality reqs, Constraints.

## 6.1 Requirements Engineering Process

- Requirements Elicitation (Gewinnung)
- Requirements Documentation (create SRS: Software Requirements Specification)
- Requirements Agreement (Übereinstimmung, resolve conflicts, acceptable for most stakeholders?)
- Cross cutting actions: Requirements Validation and Requirements Managemet.

Stakeholder: Person who (in)directly influences the requirements of a system. User, Operator, Purchaser, Devs, Architects, Tester.

Requirements Engineer: Translates between users and devs, responsible for elicitation, documentation and agreement.

Requirements Elicitation Techniques: Questioning, Creativity techniques (Brainstorming, Analogy), Retrospective techniques (reuse, competing systems), Observation techniques, others (Mind maps, Workshops, use case modelling, …)

Non-functional requirements are also important, even if hard to define.

## 6.2 Requirement Classification

Better properly faceted classification of requirements rather than non- or functional requirement.

- Concern-based (functional, quality, constraint)
- Representation (Operational, Quantitative, Qualitative, Declarative)
- Satisfaction (Hard, Soft)
- Role (Prescriptive, Normative, Assumptive)

### 6.2.1 Concern-based Classification

A concern is a functional concern if it targets the behaviour of a system to given input (Functionality and behaviour, data, stimuli, reactions).

A concern is a quality concern if it targets the quality of the software (Performance, Reliability, Usability, Security, Availability, Portability, Maintainability).

A concern is a constraint if it's a different restriction to what the system should do (Physical, Legal, Cultural, Environmental, Design&Implementation Interface)

### 6.2.2 Representation based Classification

- Operational
  - Specification of operations or data. Verified by review, test or formal verification
- Quantitative

- Specification of measurable properties, verified by Measurement
- Qualitative
  - Specification of goals, subjective verification.
- Declarative
  - Description of required feature, verified by Review.

Classification examples: F07.24

## 6.3 Modern Requirements capture

Usually user stories (agile) and use cases (model-based).

Focus on user interaction with software. Use sentence templates and glossary for templated requirements.

## 6.4 Requirements Validation

Requirements errors are expensive.

Requirements validation techniques:

- Inspection, Reviews, Walkthroughs (manual approach)
- Simulation
- Prototyping (oriented towards design model)
- System test cases
- Model checking (formal verification)

# 7 Use Cases

"Black-box user goal use case."

Goals and actors are dependent on use case scope.

- System Boundary (Scope): Border between system and environment. Multiple scopes can be relevant for a project, but only one at a time.
- System Context: Part of system environment relevant for requirements
- Context Boundary: Seperates system context from irrelevant environment

Scope can usually be white- or black-box.

See example in F08.56ff.

## 7.1 Elementary Business Process (EBP)

EBP is defined as a task performed by one person, in one place at one time, in response to a business event which adds measurable business value and leaves data in a consinstent state.

Don't define at too low a level!

Shaping heuristics for user goal use cases: Boss test (How would you describe your workday to your boss?), Coffee beak test (Finish a use case when you would intuitively make a coffee break).

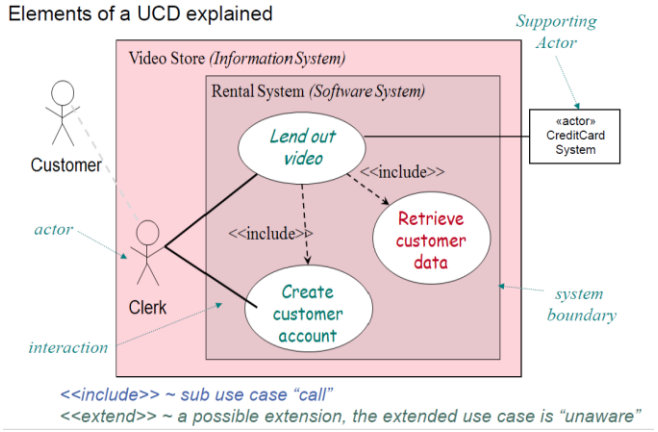
## 7.2 Use Case Goal Levels

- (High-level) *Summary*
- *User Goal* (=EBP): How is a user goal reached by interacting with the system
- *Sub(function)*: Subgoals required for each goal

Everything smaller is too low level and considered a "*System Operation*".

To move up a level: Ask why? To move down: Ask how?

## 7.3 Use Case Diagram




Recommandations F08.18.

## 7.4 Terminology

- Stakeholder: Person that (in)directly influences the systemrequirements.
- Actor: Entity with behaviour outside the system.
- Primary actor: initiates the interaction.
- Use case model: Set of all use cases.
- Scenario: Sequence of actions between actors and system, aka an use case instance, one story of using the system.
- Use case: Collection of relatd scenarios. Set of scenarios.

## 7.5 How to find use cases

- Choose system boundary (eg Point of Sale system)
- Identify primary actors (eg cashier, admin, store owner)
- For each actor, identify user goals/stories.
- Define use cases that satisfy user goals. Name them according to their goal.

Iterative use case elaboration: Work breadth-first, brainstorm for use cases first and refine them later.

Use case templates: F08.25

## 7.6 Writing Guidelines

- Each step shows a (sub)goal succeeding, captures the actors intent (not GUI interaction) and has an actor!
- Do not have UI details in use cases.
- Data descriptions: Precision level 1 (itemIdentifier), Prec level 2 (itemIdentifier: string), Prec lvl 3 (itemIdentifier: String, a 13 char barcode).

- Keep main scenario free from if-statements
- Mark <<include>> through underlining

## 7.7 Use Case Sections

- *Preface elements*: e.g. primary actor, scope, goal level
- *Stakeholders and Interest List*
- *Preconditions*: Are not tested by use case but assumed to be true
- *Post conditions*: should meet needs of all stakeholders
- *Main success scenario*: Happy path scenario. No branching.
- *Extensions*: Alternative flows. All other scenarios which succeed or fail. Alternative scenario merges back to default unless explicitly halted. Extension condition that can always happen is labeled with *.
- *Special requirements*: Associated non-functional requirement, qualiy attribute or constraint.
- *Technology and Data variations*: "How" rather than "what".

See example in F08.56ff.

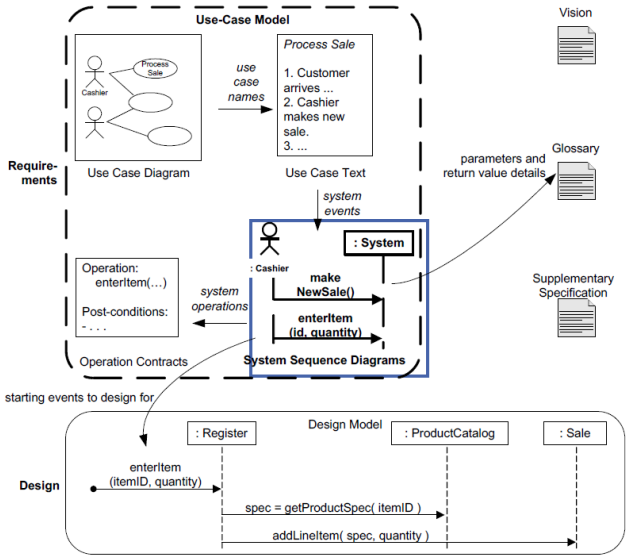
## 7.8 Software requirements Spec

Complete description of external software behaviour. Documentation of all interfaces between software and environment ("what", not "how". System remains black-box).

Requirements should be kept in separate repo and be linked with documents and dependent requirements.

# 8 Object-oriented Analysis

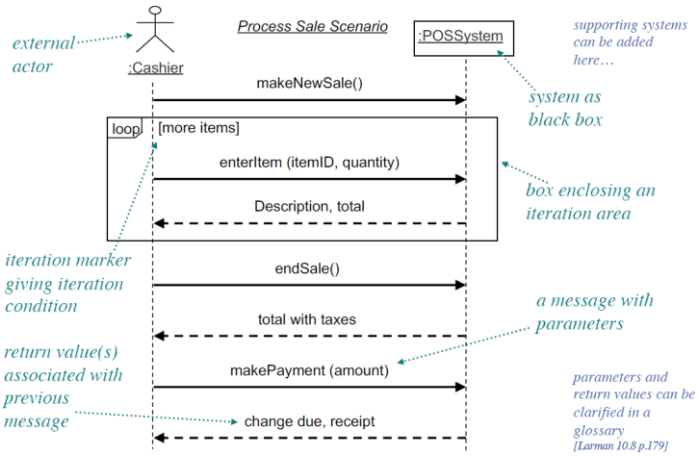
## 8.1 System Sequence Diagrams (SSD)




Use cases describe how external actors interact with blacbox-system. SSDs describe individual system operations. Emphasis is on interactions across system boundary.

For a scenario of a use case, SSD shows events by external actors, order of these events, necessary parameters and return values and inter-system events. System boundary is important because SSDs focus on events which cross that boundary.

SSDs are graphically based on UML-sequence-diagrams.




## 8.2 Operation Contracts

Focus analytically on *what* must have happened rather than *how*, helpful for systemdesign which focus on *how*. Describe system operations of use cases in detail, in terms of *state changes* to objects in *domain model* before and after the operation was executed.

Schema:

- Operation: Name and parameters
- Cross Reference: (optional) possible triggering use cases
- Preconditions: Noteworthy state assumptions. Will not be tested, but assumed to be true.
- Postconditions

Conditions are declarative and focus the domain model objects.

Postconditions *theatre stage* metaphor: Take a picture of the stage before the operation, close curtains and apply the operation, open curtains and take second picture. Compare both pictures and express the changes in postconditions.

Uses: Busines Processes, Use Cases, System Operations, Java Methods. Contracts are useful when there is complexity or inexperienced people (state changes can not always be expressed in use cases). Avoid contracts if devs can understand state changes without them.

Examples: F09.28, F09.32ff.

Contracts are an initial best guess and are not complete: Fine design details are only discovered during work.

Creation Summary:

- Identify system operations from use cases and sequence diagrams
- Construct contracts for complex or unclear system operations

- Describe postconditions in 3 categories: (instance creation/deletion), (attribute modification), (associations formed/broken)

# 9 Software Design

Design model focuses on implementation classes rather than real-world oriented domain model.

## 9.1 Responsibility-Driven Design

Types of responsibilities:

- Doing responsibilities: Objectcreation, calculations, initiating external action
- Knowing responsibilities: Knowledge on: Private encapsulated data, related objects, derivatives, calculation-possibilities.

A Responsibility is not necessarily a method.

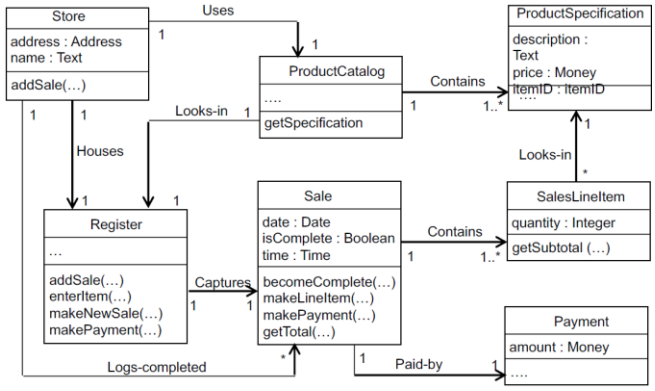
### 9.1.1 Assigning responsibilities

Start with operation contracts of system operations. Find object interactions and document them in interaction diagrams

## 9.2 Design Class Diagram (DCD)

Is done based on interaction diagrams to identify software classes, but are often created parallel to interaction diagrams.

Typical information: Classes, associations, attributes. Interfaces. Methods, attribute type information, navigability, dependencies.




- After classes are discovered, draw class diagram for these classes.
- Find methods also by interaction diagrams, name them respective to one of 3 categories: creation/getter|setter/collections, use language-independent syntax.
- Assign responsibilities based on operation contracts and use cases.
  - Information expert pattern: Is there a class which already has that responsibility? If not, are there some that can be expanded to fulfill the role? If not, Controller or Pure Fabrication?

Example: F10.19ff

## 9.3 Dynamic Design / GRAS Patterns

GRAS: General Responsibility Assignment Software Patterns.

1. Information Expert (see above)
2. Creator
   What object should be responsible for creating X? Choose object C such that C contains or aggregates X, C closely uses X (calls lots of operations on X) and has the initializing data for X.
3. Controller
   Assign responsibility for receiving/handling system events to a controller ("Façade"). One façade per use case or one for entire system.
4. Low Coupling
   Reduces impact of changes. Use classes with few dependencies.
5. High Cohesion
   Reduces impact of changes. Use classes with few responsibilities.
6. Polymorphism
   Use polymorphic method calls rather than if's. But: favor composition over inheritance (black-box) (strategy pattern).
7. Pure Fabrication
   No appropriate domain object exists for responsibilities? Artificial class with cohesive responsibilities! E.g. PersistantStorage class.
8. Indirection
   Responsibility on class would yield high coupling or low cohesion? Create adapter object between other classes.
9. Protected Variations
   Information hiding, interfaces, polymorphism…
   Hide protected data.

# 10 Software Architecture

## 10.1 Foundations

Software architecture: Set of design decisions which are hard to revert. Improve stakeholder communication, system analysis, large-project reuse and planning. Try reusing existing architectures or use as reference!

Definition of Architecture: *fundamental* concepts or properties of a system in its environment embodied in its *elements*, *relationships*, and in the *principles* of its design and *evolution*.

Our definition: Result of a set of design decisions comprising the structure of the system with components, their relationships and their mapping to execution environments.

- View: Representation of arch. elements as written and read by system stakeholders. E.g. documentation.
- Structure: Set of elements itself, as they exist in software/hardware.
- View point: groups views related to a concern.

Every system has a structure, but not always a documented architecture. Architecture docs can be inconsistent with implementation.

### 10.1.1 View Points in Palladio

- *Structural view types*: Static systemproperties. Can be differentiated into "repository view type" (reusable components and interfaces) and "assembly view type" (component instantiation and instance connectivity).
- *Behavioural view types*: Execution semantics, e.g. sequence diagrams; Userbehaviour using the usage model view types.
- *Deployment view point*: allocation view type, containing info on which components are allocated on which container; resource environment view type describing all containers and links between them.

Other architectural views: F11.17ff. Examples: F11.19ff.

Architecture and system characteristics: Performance, Security, Safety, Availability, Mainainability.

### 10.1.2 Architectural Patterns and Styles

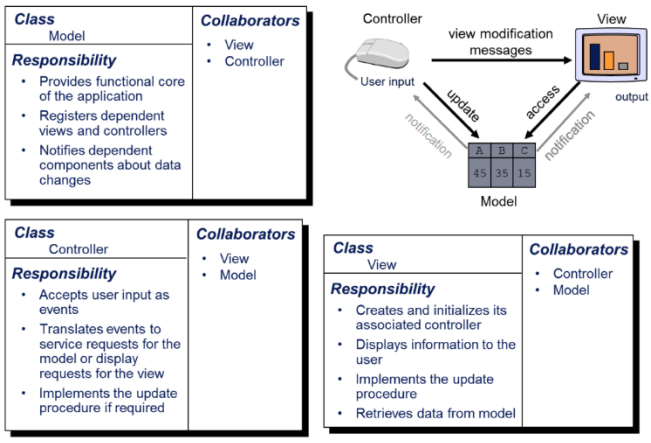
#### 10.1.2.1 Terminology

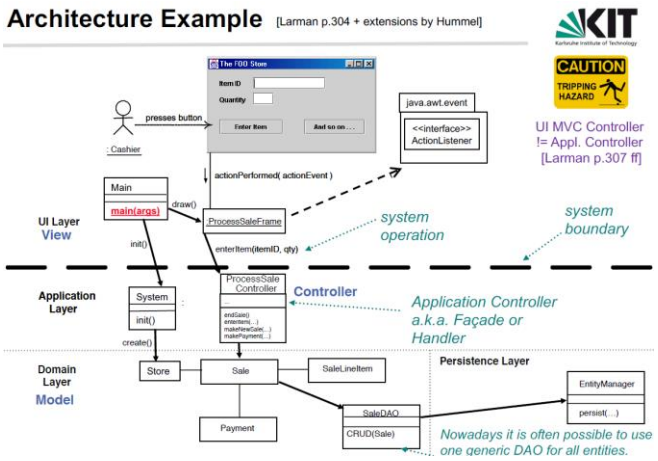
- Architectural pattern: Solution to recurring problem balanced at architectural level.
- Architectural style: Solution principles independent of application. Should be used throughout architecture.
- Reference architecture: Defines domain concepts, components and subsystems.

Examples: F11.34ff.

#### 10.1.2.2 Pattern: Model View Controller (MVC)

Trivial.




#### 10.1.2.3 Observer pattern

Trivial.

## 10.2 Patterns of Enterprise Application (EA) Architecture

Enterprise applications are mainly about data.

Properties: persistent data (long living data, existing data needs to be integrated), large amounts of data, concurrent data access, different UIs, interfaces to external systems, conceptual dissonance (overloaded terms), complex business "logic".

### 10.2.1 Layers of EA

- Frontend/Presentation
- Domain (Middle, Business)
- Data Source

### 10.2.2 Pattern Families

- Domain Logic: How to represent business rules?
- Data source architecture: How to separate domain logic and data source?
- Object-relational structural patterns: How to map objects to relational databases?

### 10.2.3 Domain Logic Patterns

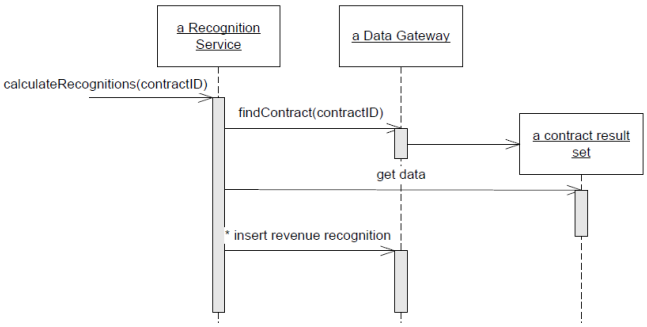
Possible challenges: Highly complex domain logic, needs to be changeable, needs connection with frontend and data source.

When to use what? See summary at F12.60.

#### 10.2.3.1 Transaction Script

> *Organizes business logic by procedures where each procedure handles a single request from the presentation.*

Put all logic for transaction in a procedure. Single procedure per transaction type, factor common behaviour into subroutines.




| Advantages | Problems |
|---|---|
| Simple understandable procedures | Doesn't scale with complex logic |
| Easy connection to data sources | Usually dublicate code |
| Transaction boundaries easy to determine | |

#### 10.2.3.2 Domain model

> *An object model of the domain that incorporates both behaviour and data.*

Design a domain layer using a domain model. High coupling between design classes and domain model. Object-oriented thinking, logic is kept with the concept.

| Advantages | Problems |
|---|---|
| Better organizes complex domain logic | Complicated if not familiar with OO |
| | Data source mapping more complex |

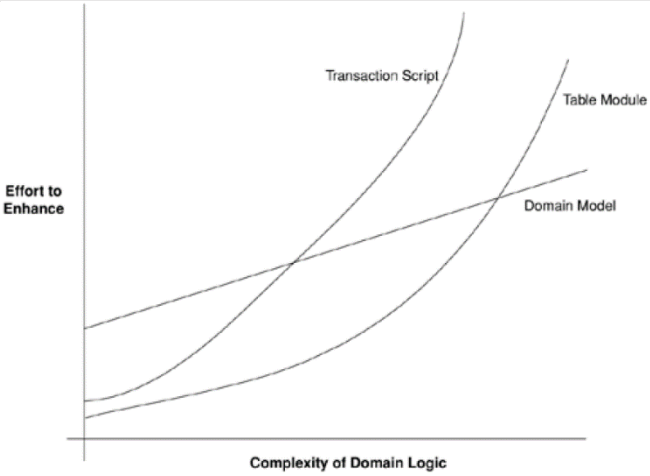
#### 10.2.3.3 Table Module

> *A single instance that handles the business logic for all rows in a database table or view.*

One class handles business logic for a table, called with data from the DB. Can use subclasses wich also are initialized with full DB data.

| Advantages | Problems |
|---|---|
| Straightforward data mapping | No object instances: Can be bad for complex logic |
| Seperates logic and concepts | |
| Useful if used technology supports it | |

#### 10.2.3.4 When to use which pattern?




How difficult to map data source? Are developers familiar with domain models? What tools are used?

### 10.2.4 Data Source Patterns

OO vs relational DB follow different approaches. Solutions: Manual storage, XML files/OO databases or OO/Rel mapping frameworks.

Patterns: Record Set, Table Data Gateway, Active Record, Row Data Gateway, Data Mapper (, Identity Map).

#### 10.2.4.1 Record Map

> *An in-memory representation of tabular data*

Object which looks like result of SQL query, but can be easily generated and manipulated. Often generated by frameworks (a.executeQuery(sql)).

#### 10.2.4.2 Data Table Gateway

> *A class acts as a gateway to the database table. One instance handles all the rows in the table.*

Simplest solution for DB access. Seperation of SQL statements and code which uses the data. Should cover all CRUD[1] methods. Each method maps parameters to SQL statement and executes that statement.

Not useful for Domain Model Pattern, but for Transaction Script.

#### 10.2.4.3 Active Record

> *An object that wraps one row in a database table or view, encapsulates the database access and adds domain logic on that data.*

OO approach, brings data and logic together. Like Data Table Gateway, but class doesn't only include DB CRUD operations, but also stores data and includes domain logic.

Database schema and active record need to be isomorphic. Doesn't work well with complex business logic, use Data Mapper instead.

#### 10.2.4.4 Row Data Gateway

> *An object that acts as a gateway to a single record in a data source. There is one instance per row.*

Object which looks like DB record structure, but is typed and implemented with code.

Details of data source access is hidden, but data structure is not. Usually with a "Finder" class for abstracting queries.

E.g.: Classes "Person (name, age, domainLogicMethods())", "Person Finder (find())", "Person Gateway (name, age, insert(), update(), delete())".

Can often be auto-generated.

#### 10.2.4.5 Identity Map

> *Ensure that each object gets loaded only once by keeping every loaded object in a map. Look up objects using the map when referring to them.*

Similar to a DB cache.

[1] Create, Read, Update, Delete

#### 10.2.4.6 Data Mapper

> *A layer of Mappers that moves data between objects and database while keeping them independent of each other and the mapper itself.*

PersonMapper (find(id): Person, insert(Person), update(Person), delete(Person)).

When to use:

- Database schema and object model evolve differently.
- Complex business logic so that Active Record does not suffice.
- Legacy systems.

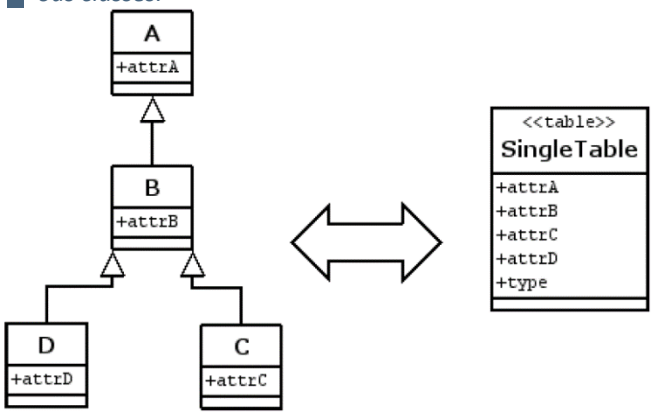
### 10.2.5 Object-relational structural Patterns

Mapping OO structures onto a relational DB.

When to use what? See summary at F12.73.

#### 10.2.5.1 Single Table Inheritance

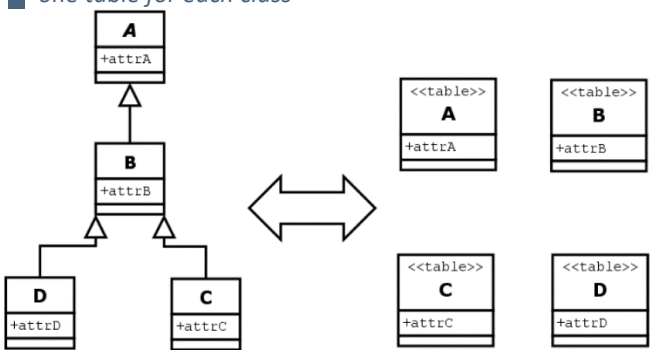
> *Represents an inheritance hierarchy of classes as a single table that has columns for all fields of the various classes.*




- Pro
  - Simple DB schema: Only one table
  - No Joins
  - Refactoring does not require DB changes
- Contra
  - Can be confusing
  - Tables can get very large
  - Only one namespace for all fields

#### 10.2.5.2 Class Table Inheritance

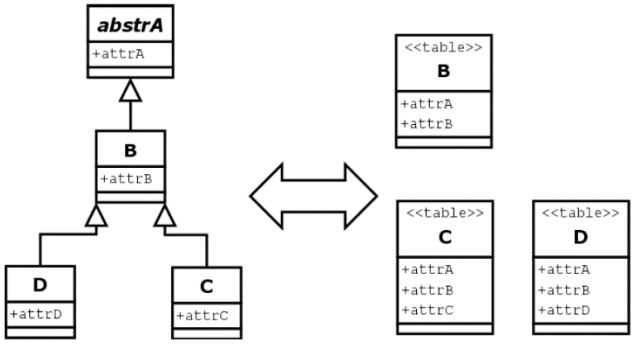
> *Represent an inheritance hierarchy of classes with one table for each class*




- Pro
  - All columns are relevant for every row. Easier to understand and more compact.
  - Easier to map legacy schemas
  - Straightforward relationship between domain model and DB schema
- Contra
  - Loading an object requires multiple table queries
  - Refactoring (moving class up/down the hierarchy) requires changing the schema
  - Supertypes can become bottlenecks.
  - High normalization makes schema harder to understand.

#### 10.2.5.3 Concrete Table Inheritance

> *Represent an inheritance hierarchy of classes with one table per concrete class in the hierarchy.*




- Pro
  - Tables are self contained.
  - No Joins required
  - Performance
- Contra
  - Refactoring requires database schema changes.
  - Changes of supertype fields influence all subtype tables
  - Finding on superclass forces check on all subtype tables or a complex join.

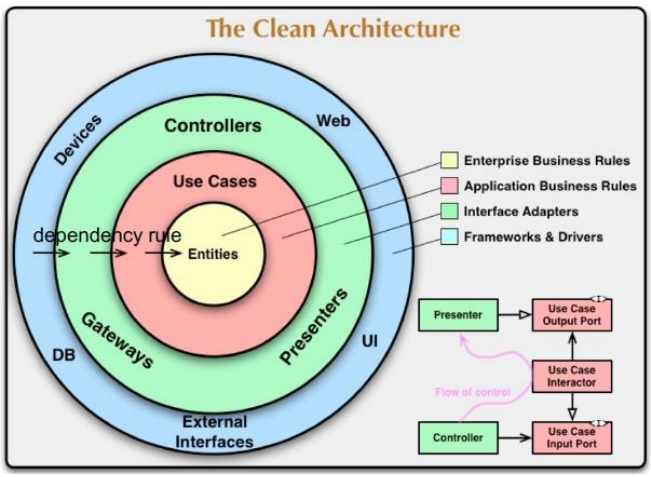
## 10.3 Clean Architecture

Two values: Behaviour (functional requirements fulfilled) vs Mainainability.

Well working but unmaintainable program is useless, but not working maintainable program can be made working.

Layers of Clean Architecture helps seperating different concerns.

- Dependencies in the “right” direction: Any volatile component should not be depended on by a component that is difficult to change; Depend in the direction of stability.
- Clean Architecture Style: Seperation of different areas of software. Inner areas = higher level.
  Dependency Rule: Source code dependencies always point inwards.




- Claimed Benefits: Independence of frameworks, testable systems (e.g. does not have to test UI), Independence of UI/DB/external agency.
- Entities: Encapsulated business rules, stable against external changes.
- Interface adapters: Convert data between layers between formats most convenient for the current layer.

## 10.4 Software Components

> *A unit of composition with contractually specified interfaces and explicit context dependencies only. Can be deployed independently and is subject to composition by third parties.*

Not necessarily black-box, low composition/deployment/adaption effort. Cannot be a Java-Object because Inheritance conflicts with the definition.

Usually comprise the application and domain layer, “business components”.

### 10.4.1 Technical Realisation

Usually late binding, encapsulation, interface inheritance.

#### 10.4.1.1 Open Service Gateway Initiative (OSGi)

OO uses extreme modularization: many fine-granular objects, but no module concept above packages.

OSGi solution: *bundles*: JAR files with public interface via manifest, must be required in consumer bundle. OSGi provides a registry for services/bundles.

#### 10.4.1.2 Service-Oriented Architecture (SOA)

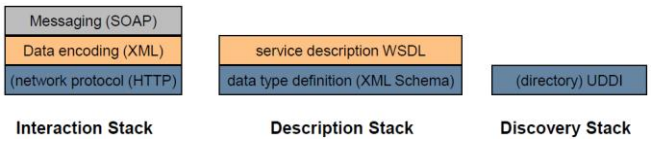
System build from web services playing one of 3 roles:

- **Service requestor**: use *find*-operation to retrieve service descriptions from service registry. Requestors *bind* to providers based on service descriptions to locate and invoke services.
- **service provider**: *publish* services by advertising service descriptions to registry.
- **service broker** (repo)

Web service: self-contained, self-describing, modular, published, located, invoked.

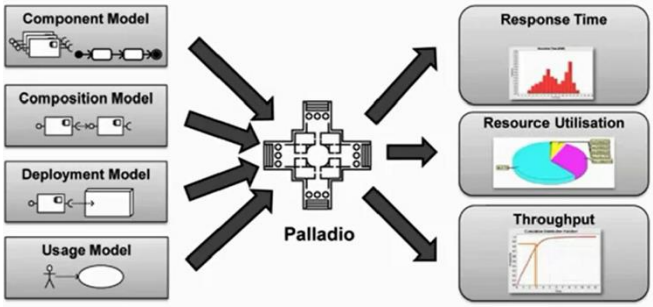
Core Web Service Technologies:

- SOAP: Simple Object Access Protocol
- WSDL: Web Service Description Language
- UDDI: Universal Description, Discovery and Integration



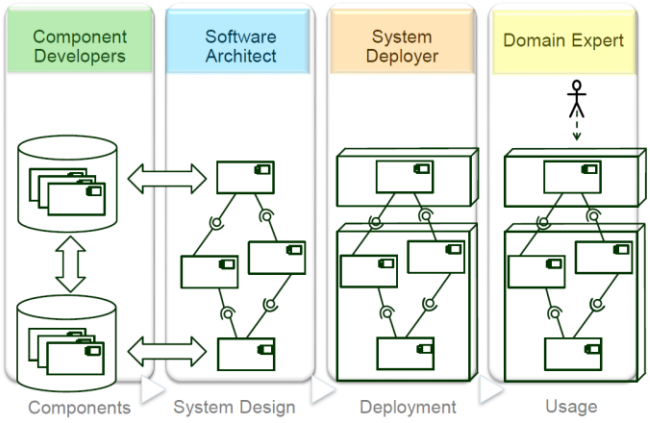

#### 10.4.1.3 Palladio Component Model

Domain specific modelling language with early performance predictions. All performance influence factors are made explicit. Usage context (e.g. changing hardware, usage, assembly) are explicit parameters.




Why modules?: Legacy extensions, performance predictions at design time, analytically solveable, simulation.

Models and their creators:




- Component developer: Create composable component specs reusable in different contexts unknown to CD (Parametrization!). Independent from external services, use abstracted interfaces only. Usage must also be parametrized (e.g. used filesizes).
  Tasks: Specify components&Interfaces&Data Types, build composite elements, create parametrization. Implement, test and maintain components.
- Software Architect: Specify architecture for system from existing components and interfaces. Specify new components. Make design decisions. Performance prediction and delegate implementation to CDs.
- System Deployer: Abstract specs of resources and deployment environment, then derive timing values.
- Domain Expert: Creates Usage Model: Models user behaviour on system, create useage scenarios.

After complete model is built, dependencies are resolved.

## 10.5 Cloud Computing & Cloud Architecture

Target is to have the illusion of infinite IT-resources, which are always available, on-demand and do not require ownership or reservation. Pay-to-use.

Separate logical services (Virtual Resource Sets, VRS) instead of physical servers (Physical Resource Sets, PRS).

Types of Resources: Computing power, storage, network, data, apps, others.

Advantages of Virtualization:

- Consolidation (Improved energy efficiency)
- Availability
- Quality of Service
- Overbooking of Resources
- Logical view on resource pools improves management
- Almost no performance penalty with current technology

### 10.5.1 Conceptual elements

- Five *characteristics of cloud services*:
  - Elastic Scalability
  - On-demand Self-Service
  - Ubiquitous Network Access
  - Resource Pooling
  - Measured Service
- Three *Delivery models*
  - SaaS: Webapps, E.g. Salesforce
  - PaaS: Dev/execution envs, E.g. Google App Engine
    - Deploy consumer-created apps, consumer does not manage OS/network/storage, but can configurate hosting environment.
  - IaaS: E.g. AWS, Azure VMs
    - User can deploy containers and has access to most controls including OS/network/storage, but does not manage underlying infrastructure.
- Three *Deployment models*
  - Public (customer and external provider), Private (customer and provider in the same org), Hybrid

Characteristics of SaaS: Network-based access to software, activities are managed centrally, app-delivery closer to a one-to-many model, Centralized feature updates.

*Single-tenant vs Multi-tenant architecture*: Single tenant, every customer has apps, data, OS and hardware on its own system. Multi tenant, all customers share a subset of the same stack.

GrepTheWeb example: F15.33ff.

Ensure that the application is scalable by designing each component to be scalable on its own. Loosely couple components, implement parallelization for better infrastructure usage, ask “what if it fails?” (resilience), consider costs.

### 10.5.2 Architecture Principles

- *Decentralization*: Removes bottlenecks/Single point of failure
- *Asynchrony*: Progress is always done.
- *Autonomy*: Components can make local decisions.
- *Local reponsibility*
- *Controlled concurrency*: Only limited concurrency control required.
- *Failure tolerant*: Failure is expected, the system can handle failed components.
- *Controlled parallelism*: So that new nodes can be inserted easily or recovery is easy.
- Decomposition in small well-understood *building blocks*
- *Symmetry*: Nodes have identical features and require minimal configuration.
- *Simplicity*: System is as easy as possible.

## 10.6 Microservices

Microservices: Seperate components into processes to maximize decoupling and independent deployability.

Services are independently deployable/scalable, use different stacks and are managed by different teams. Each Microservices handles its own database. Because services can fail anytime, fault tolerance is required.

Conform to CONWAYS LAW:

> *Any organization that designs a system will produce a design whose structure is a copy of the organization’s communication structure.*

- Tolerant Reader: Ignore unknown elements and make minimum assumptions in order to increase robustness.
- Consumer-Driven Contracts: Represents the expectations of all consumers the provider has to satisfy.

- Pro:
  - Strong module boundaries
  - Independent Deployment
  - Technology Diversity
- Contra:
  - Distribution (harder to program, slow remote calls, risk of failure)
  - Eventual Consistency: hard to achieve, every microservice has to achieve it on its own
  - Operational Complexity

How to begin? Start with clean monolith and separate later, or start with microservices.

# 11 Model Driven Development

> *There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, reliability and simplicity.*

Model Features: What is a Model?

- *Representational Feature* (Abbildungsmerkmal): Models always depict something, representations of natural or artificial originals.
- *Reduction Feature* (Verkürzungsmerkmal): Models don't capture all attributs of the originals, but only those which are relevant for the model.
- *Pragmatic Feature* (Pragmatisches Merkmal): Models aren't directly associatable to originals. They replace them for
  - specific subjects
  - during specific time intervals
  - and under restriction to particular mental or actual operations.

## 11.1 Metamodels

Metamodel: Model which describes the structure of models. E.g. UML diagram describing possible instances. A complete metamodel must cover:

- *Abstract Syntax*
  - Describes contructs of which the model consists, as well as their properties and relations between them. Independent of concrete depiction of these.
  - E.g. UML class diagram
- *Concrete Syntax*
  - Constructs, properties and relations specified in the abstract syntax. At least 1, arbitrarily many concrete syntaxes are required.
  - E.g. Data instanziation
- *Static Semantics*
  - Modelling rules and restrictions that are hard to express in abstract syntax. Using specialized languages for restriction-definitions as constraints.
  - E.g. OCL contraint
- *Dynamic Semantics*
  - Describes the meaning of the constructs. Natural language specs.

Models represent Originals, Originals instantiate models.

## 11.2 Object Constraint Language (OCL)

"Design-by-contract" approach, which can specify invariants, pre/postconditions, initial values, derivation, body definition and guards.

## 11.3 Model-Driven Software Development

- Model-driven Engineering: Reduce platform complexity by using domain-specific modelling languages, declarative descriptions and transformation engines.
- Model-driven Software Development: Application of MDE for software dev.

### 11.3.1 Model-driven vs Model-based

- Model-based Development: "Secondary" models are used for documentation and communication. Manual analysis.
- Model-driven Development: Models are primary artifacts and cannot be omitted. Models are explicitly specific and developed. Analysis trough model transformations.

Goals and Benefits of MDSD: F17.32.

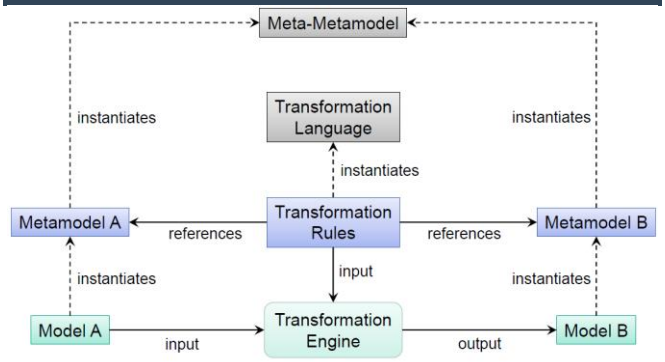
## 11.4 Model-Driven Architecture (MDA)

MDA allows specifying a system independently of the supporting platform, specifying platforms, choosing particular platforms for the system and transforming the system specs into one for a particular platform.

- *Computation-Independent Model* (**CIM**): describes system requirements and environment.
  - E.g. UML Use Case diagram
- *Platform-Independent Model* (**PIM**): describes operation of a system while hiding platform details.
  - E.g. Component Model
- *Platform-Specific Model* (**PSM**): Combines PIM with focus on on the use of a specific platform by a system.
  - E.g. UML Class Diagram

F17.38-39.

## 11.5 Model Transformations




- Transformation: Automatic generation of a target model from a source model, according to a transformation def.
- Transformation definition: Set of transformation rules.
- Transformation rule: Description of how one/multiple constructs in source language can be transformed to target language.

Declarative vs Imperative approach: F17.50.

# 12 Real-time Development and Patterns

- The result of hard real-time data processing, if *logically* and *temporally* correct.
- Real-time system: Correct functioning depends on the produced result and the time of production.
- Soft rt-system: Operation is degraded if results are not produced according to requirements.
- Hard rt-system: Operation is incorrect if results are not…

- Stimuli may be periodic (predictable intervals) or aperiodic (can occur anytime).
- Periodic system processes must be selected carefully. Use the process with the shorter deadline.
- Monitor systems take action when exceptional sensor values are detected
- Control systems continuously control hardware actuators depending on sensor values.
- Data acquisition system: Alternative to other RT systems. Collect sensordata for analysis.

## 12.1 Real Time OS (RTOS)

- RT OS: Specialized OS for RT systems with little management overhead.
- Usual components: Real-time clock, interrupt handler, scheduler, resource manager, dispatcher.
- RTOS scheduler uses real-time clock.
- Must support at least the priority levels “Interrupt level prio” and “Clock level prio”.
- Process Management
  - Scheduler chooses next process
  - Resource manager allocates memory and CPU
  - Dispatcher loads program and starts execution

## 12.2 RT System Design

Design Process:

- Choose execution platform: hardware and RTOS
- Identify stimuli and associated responses
- Define timing contraints for stimuli/responses
- Aggregate stimuli/response processing into concurrent processes
- Design algorithms for stimuli processing and response generation. Finite state machines can be used fore modelling RT systems.
- Specify data to be exchanged.
- Design scheduling system with quarantees.

Dimensions of Dependability: F18.30.

*Safety vs Reliability*:

- Safety: Absence of danger for humans/environment
- Reliability: Probability/duration of failure-free operation

Fault-Error-Failure Chain

- *Fault*: Defect in system. May lead to failure. (“Bug”)
- *Error*: Discrepancy between intended and actual behaviour. Occurs in activation of a fault.
- *Failur*: System displays behaviour contrary to its spec.

*Random fault*: Error which can be resolved by just trying again. *Systematic fault*: An actual solution has to be found.

## 12.3 RT Patterns

Summary: See F18.45.

### 12.3.1 Channel

Pipe which sequentially transforms data. Each internal elements has simple operation. Performance can be increased by using multiple identical homogeneous channels. Reliability is achieved by multiple channels with fault tolerance. Safety can also be achieved when using multiple channels by adding fault identification and safety measures.

Homogeneous redudancy (backup channel): Protects against random faults, not assuming a fail-safe state.

Triple Modular Redundancy (3 channels): Protects against random faults without a failsafe state.

Heterogeneous redundancy: Independently designed channels. Protects against random and systematic faults, given that the same mistake is not made in both versions. Does not require a fail-safe state.

### 12.3.2 Monitor-Actuator Pattern

Monitoring- and Actuation-Channel. Protects against random and systematic faults with a fail-safe state.

### 12.3.3 Sanity Check Pattern

Lightweight protection against random and systematic faults with a fail-safe state. Actuation- and Sanity-Channels, Sanity-Channel monitors data and approximates, maybe gives shutdown signal to data integrity check.

### 12.3.4 Watchdog Pattern

Lightweight protection against time-based faults and detection of deadlocks with a fail-safe state.

The Watchdog-component receives periodic “Heart-Beats” from data processor and actuator controller and shuts down if too much time passed without Heart-Beat.

### 12.3.5 Safety Executive Pattern

Safety for complex systems with non-trivial mechanisms to achiveve fail-safe state.

Actuation Channel, Heart Beat signals Watchdog which may trigger Fail-safe processing channel (backup).

# 13 Software Security

Main Goals: Confidentiality (data privacy), Availability and Integrity.

Supporting Goals: User Authentication, Traceability & Auditing, Monitoring, Anonymity.

How to find Security Requirements: Cosider goals, try to find potential “Misuse Cases”, formulate precise requirements.

- Secure the weakest link
- Use defence in depth (multiple defence layers)
- Fail securely
- Secure by default

- Principle of the least privilege: Only grant minimal privileges, only for as long as necessary.
- No security through obscurity (Kerkhoffs principle)
- Minimize attack surface (less code, few interfaces)
- Privileged Core (isolated with security privileges)
- Input validation and Output encoding
- Don’t mix data and code

## Inhalt