

# Software Engineering

## Summary



**Author:** Ruben Deisenroth  
**Helper:** Hendrik Rüthers

**Semester:** winter term  
**2020/21 Version:** v1.0  
(SNAPSHOT)

**Status:** 1 March 2022  
Science

**Department:** Computer

## Table of contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Software.....	4
1.1.1	Types of software .....	4
1.1.2	Software characteristics.....	4
1.2	Engineering .....	5
1.2.1	Characteristics of Engineering Approaches .....	5
1.2.2	Problems of software development .....	5
<b>2</b>	<b>Requirements Engineering</b>	<b>6</b>
2.1	Requirements analysis .....	6
2.1.1	User requirements.....	6
2.1.2	System requirements (System Requirements) .....	6
2.1.3	Domain Requirements.....	7
2.1.4	Functional requirements.....	7
2.1.5	Non-Functional Requirements.....	7
2.1.6	RE process.....	8
<b>3</b>	<b>Use cases</b>	<b>8</b>
3.1	Use Case Analysis .....	8
3.1.1	Requirements vs Use Cases .....	8
3.1.2	Use Case Formats .....	9
3.1.3	Guidelines for the development of use cases.....	9
3.2	UML Use Case Diagrams (UML Use Case Diagrams).....	10
<b>4</b>	<b>Domain modelling</b>	<b>11</b>
4.1	Diagram: Domain Model (UML).....	12
4.1.1	Class vs attribute .....	13
4.1.2	Attribute vs connection.....	13
4.1.3	UML State Machine Diagram.....	13
<b>5</b>	<b>Software Architecture</b>	<b>14</b>
5.1	Architectural characteristics .....	15
5.2	Architectural styles .....	15
5.2.1	Monolithic architectural styles (Monolithic Architecture Styles) .....	16
5.2.2	Distributed architectural styles (Distributed Architectural Styles).....	17
<b>6</b>	<b>Design Principles</b>	<b>18</b>
6.1	Software quality .....	18
6.1.1	Factors .....	18

6.2	Measuring software quality .....	19
6.2.1	Overview of metrics for measuring software quality .....	19
6.2.2	Cyclomatic complexity (Cyclomatic Complexity) .....	19
6.2.3	Control flow graph (CFG) .....	19
6.2.4	Heuristics .....	20
6.2.5	Linking/coupling .....	21
6.2.6	Responsibilities .....	22
6.2.7	Cohesion .....	22
6.3	Design principles .....	23
6.3.1	Single Responsibility Principle (SRP) .....	23
6.3.2	Inheritance vs. delegation .....	23
6.4	Encapsulation .....	23
6.4.1	Access rights (Field Access) .....	24
6.5	Problems .....	24
6.5.1	God Classes .....	24
6.5.2	Class Proliferation .....	24
<b>7</b>	<b>Design techniques</b> .....	<b>25</b>
7.1	Documentation .....	25
7.1.1	Readability .....	25
7.1.2	Types of comments .....	25
7.2	Refactoring .....	26
7.2.1	Reasons for revising .....	26
7.2.2	Extract method (Extract Method) .....	26
7.2.3	Move method .....	27
7.2.4	Extract class (Extract Class) .....	27
7.3	Idioms .....	27
<b>8</b>	<b>Design concepts (design patterns)</b> .....	<b>28</b>
8.1	Introduction .....	29
8.1.1	Template Method .....	29
8.1.2	Strategy .....	30
8.1.3	Observer .....	31
8.1.4	Factory method .....	33
8.1.5	Abstract factory .....	34
<b>9</b>	<b>Verification</b> .....	<b>35</b>
9.1	Introduction .....	35
9.2	Verification and validation .....	35
9.2.1	Techniques .....	35
9.2.2	Code reviews .....	36
9.3	Testing .....	37
9.3.1	Test types .....	37
9.4	Test coverage .....	38
9.4.1	Structural .....	38
9.4.2	Logical .....	38
9.5	Test automation .....	39
9.5.1	Automated Test Case Generation (ATCG) .....	39
<b>10</b>	<b>Maintenance and further development (maintenance &amp; evolution)</b> .....	<b>40</b>
10.1	Maintenance (Maintenance) .....	40
10.1.1	Trigger .....	40
10.1.2	Parallelisation as a maintenance task .....	40
10.1.3	Software evolution .....	41
10.2	Software Variability Engineering .....	41
10.2.1	Challenges in variability .....	41

10.2.2 Software Product Line Engineering.....	41
10.2.3 Feature diagrams.....	42
<b>11 Software processes</b>	<b>44</b>
11.1 Basic terms .....	44
11.1.1 Basic steps in software development.....	44
11.1.2 Process models .....	44
11.1.3 Activities.....	44
11.2 Process model: waterfall.....	45
11.3 Agile development.....	46
11.4 Process model: eXtreme Programming (XP).....	46
11.4.1 Development.....	47
11.4.2 Planning .....	47

## 1 Introduction

---

### 1.1 Software

---

**Definition - Software** The term *software* refers to a programme and all the associated data, information and materials. This includes, for example

- An (installable), executable (operational) programme and its data
- Configuration files (configuration files)
- System documentation, e.g. architecture model, design concepts, ...
- User documentation, e.g. instructions, ...
- Support (e.g. maintenance, website, telephone, ...)

- W.S. Humphrey, SCM SIGSOFT, 1989

---

#### 1.1.1 Types of software

---

Type	Type Description
Application Software	- interacts directly with the user - Can be both general purpose (e.g. word processing, ... ) and application-specific (e.g. cash register system, ... )
System Level Software	- usually does not interact directly with users - Ensures a functional system (e.g. drivers, ... )
Software as a Service (SaaS)	- Runs on a server - Access usually only indirectly via client (e.g. via browser, SSH, ... )

TabZILL 1: Types of software

---

#### 1.1.2 Software characteristics

---

- Software does not wear out, but must constantly adapt (e.g. to new hardware, ... )
- Lifetime usually longer than expected (difficult to determine as future requirements are uncertain)
- Difficult to measure: software quality, development progress and reliability

---

## 1.2 Engineering

---

**Definition - Engineering** is the process of using scientific principles to design machines, structures and many other things.  
- Cambridge Dictionary

**Definition - Software engineering** is a branch of computer science and refers to the application of a systematic, disciplined and qualified approach to the development, execution and maintenance of software, as well as the research and development of such approaches.

---

### 1.2.1 Characteristics of Engineering Approaches

---

- Fixed/defined process
- Separate development phases, typically: Analysis, development (design), evaluation of the development, construction, quality control (e.g. TÜV, or mathematical proof of correctness)
- Clear separation of individual subsystems

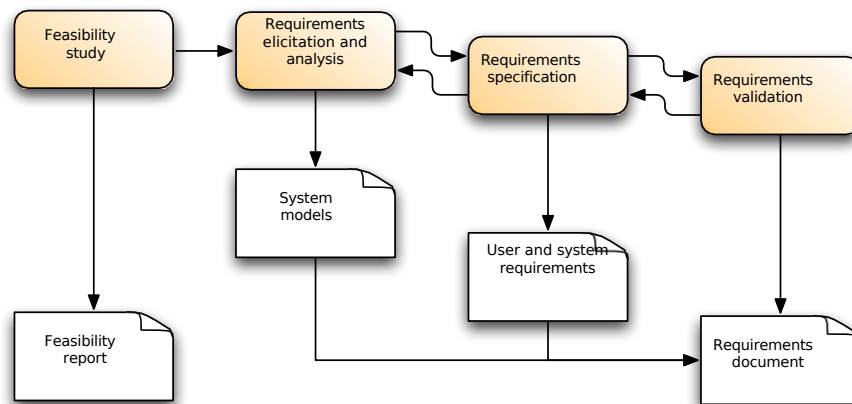
---

### 1.2.2 Problems of software development

---

- Customer/user little/not at all involved in development
- Constantly changing software requirements
- Software developers often not sufficiently trained
- Management problems
- Inappropriate methods, programming languages, tools

## 2 Requirements Engineering



(from: I. Sommerville, Software Engineering, Pearson)

### 2.1 Requirements analysis

**Definition** - Requirements are the descriptions of the tasks to be fulfilled by the designed system and its limitations

The requirements analysis deals with recognising, analysing, documenting and validating the requirements.

- Requirements are recorded in so-called System Requirements Specification.
- use cases, state diagrams, etc. are recorded in the product backlog.
- Requirements are **not** solutions/implementations.

#### 2.1.1 User requirements

- describe tasks and restrictions in natural language or with diagrams (usually written by customers)

**Example:** The system should save all bookings as by law.

#### 2.1.2 System requirements (System Requirements)

- Precise and detailed description of tasks and limitations of the programme (usually written by developers)
- Refinement of user requirements

**Example:** The bookings must be stored for 10 years from the time of booking.

---

### 2.1.3 Domain requirements

---

- Usually not specified by the customer or developer, but by the domain (e.g. by the legislator, . . . )

**Example:** In Germany, every transaction must be temporarily stored for 2 weeks for police investigations.

---

### 2.1.4 Functional requirements

---

- The services that the system should be able to provide
- the reaction of the system to certain inputs and
- the behaviour of the system in certain situations.

**Example:** If the user presses the "New document" button, a new text document is created.

---

### 2.1.5 Non-Functional Requirements

---

Specify limitations of the system's services/functions, which often cannot be fully covered by tests:

- Product requirements
  - Portability (runs on different platforms/can be easily adapted to them)
  - Reliability/robustness (also reacts sensibly to unforeseen/illegal inputs and situations)
  - Efficiency (performance, storage space)
  - Usability (comprehensibility, . . . )
- Organisational requirements
  - Delivery requirements
  - Implementation
  - Utilisation of standards (ISO, IEEE, . . . )
- External requirements (External requirements)
  - Interoperability requirements, i.e. interaction with other systems
  - Ethical requirements
  - Legal requirements (data protection, security, . . . )

Non-functional requirements are often large-scale requirements that affect the entire system. However, such requirements are usually more critical than functional requirements (e.g. "The system should be secure against attacks.").

In order to make non-functional requirements verifiable, it is often necessary to reformulate or modify the actual requirement. This can also reveal functional requirements.

**Example:** The user interface should be appealing and easy to use.

### 2.1.6 RE process Viewpoint-

---

#### Oriented approach

- Interactor viewpoints: direct interest
- Indirect viewpoints: indirect interest
- Domain viewpoints: indirect interest

**Definition - FURPS+ Model** Functionality, Usability, Reliability, Performance, Supportability Plus Implementation (Interface, Operations, Packaging, Legal)

#### Requirements validation checks

- Correctness (Validity): Do the requirements contain all the necessary functions or are others needed?
- Consistency: Are there conflicts in the requirements?
- Completeness: Are all functions and restrictions specified as required?
- Realisability (realism): Are the requirements realistically achievable?
- Testability (verifiability): Can the fulfilment of the requirements be tested?
- Traceability: Is it possible to understand why the requirement exists?

---

## 3 Use cases

---

### 3.1 Use Case Analysis

---

**Definition** - Use cases describe how a member of a role (*actor*) uses the system in a specific *scenario*.

---

#### 3.1.1 Requirements vs Use Cases

---

Requirements	Use Cases
- Focus on desired <i>functionality</i>	- Focus on possible <i>scenarios</i>
- Are usually simply declared (declaratively)	- Are described on the basis of scenarios (operationally)
- Perspective of the client	- Perspective of the user

TabZllZ 2: Requirements vs Use Cases

- Almost impossible to write good/complete use cases without user involvement
- Use cases supplement the requirements analysis, but do not replace it (cannot capture non-functional requirements)



### 3.1.2 Use Case Formats

- short (brief): Short summary, usually the main success scenario
- informal (casual): Informal format, multiple lines covering multiple scenarios
- Fully dressed: All intermediate steps and variations are written down in detail, there are help sections, e.g. preconditions and success guarantees.

A fully developed use case looks like this:

Section	Description/Restriction
Use Case Name	The name of the use case / Starts with a verb
Scope	Affected area of the system
Level	Level of abstraction/ User target, summary or sub-function
Primary actor	Initiator of the use case
Stakeholders and Interests	Persons affected by this use case
Preconditions	What must apply at the start of the programme? Is it worth telling the reader?
Acceptance criteria (minimum guarantee)	Minimum promise to stakeholders
Success criteria (success guarantee)	What should the programme be able to do if it is successful?
Main Success Scenario	Typical sequence of the scenario / Numbered step list to achieve the goal
Extensions	Alternative success and failure scenarios / failure points of the main scenario
Special requirements	Related, non-functional requirements
Technologies	Technologies to be used
Frequency of occurrence	Frequency of occurrence of the use case
Other (Misc)	E.g. open tickets

not  
always  
necess  
ary

TabZllZ 3: Fully Dressed Use Case schema

### 3.1.3 Guidelines for the development of use cases


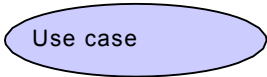
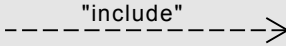
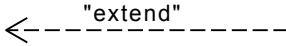
1. List stakeholders and their interests=> First level of precision
2. Stakeholders, Trigger (first step of the main success scenario), Validate=> Second level of precision
3. Identify and list all failure scenarios
4. Write error handling

### 3.2 UML Use Case Diagrams (UML Use Case Diagrams)

**Definition - Unified Modelling Language (UML)** Visual but precise design notation for software development

- Main objective: Standardise object modelling by agreeing on a fixed syntax and semantics

**Definition - Use case diagram** requirement cases and their relationships to the system and its actors.

Element	Item description
 Actor	Represents an actor within the system.
 Use case	Represents a use case in the system
	Adds a functionality to a use case. If the extended use case is "executed", this use case is also "executed".
	Extends a use case with a functionality. If the condition in the description is true, the extended use case is "executed".

TabZILZ 4: Use case diagram elements

#### Example

In a car dealership, it is possible to pay both in cash and by credit card. It is also possible for customers to hire cars. As the dealership wants to attract new customers, it is now possible to collect loyalty points when renting a car. It is possible for the shop owner to new cars to the range. Once a showroom car has been hired, the purchase price of the car is reduced.

*These requirements are shown in Figure 1.*

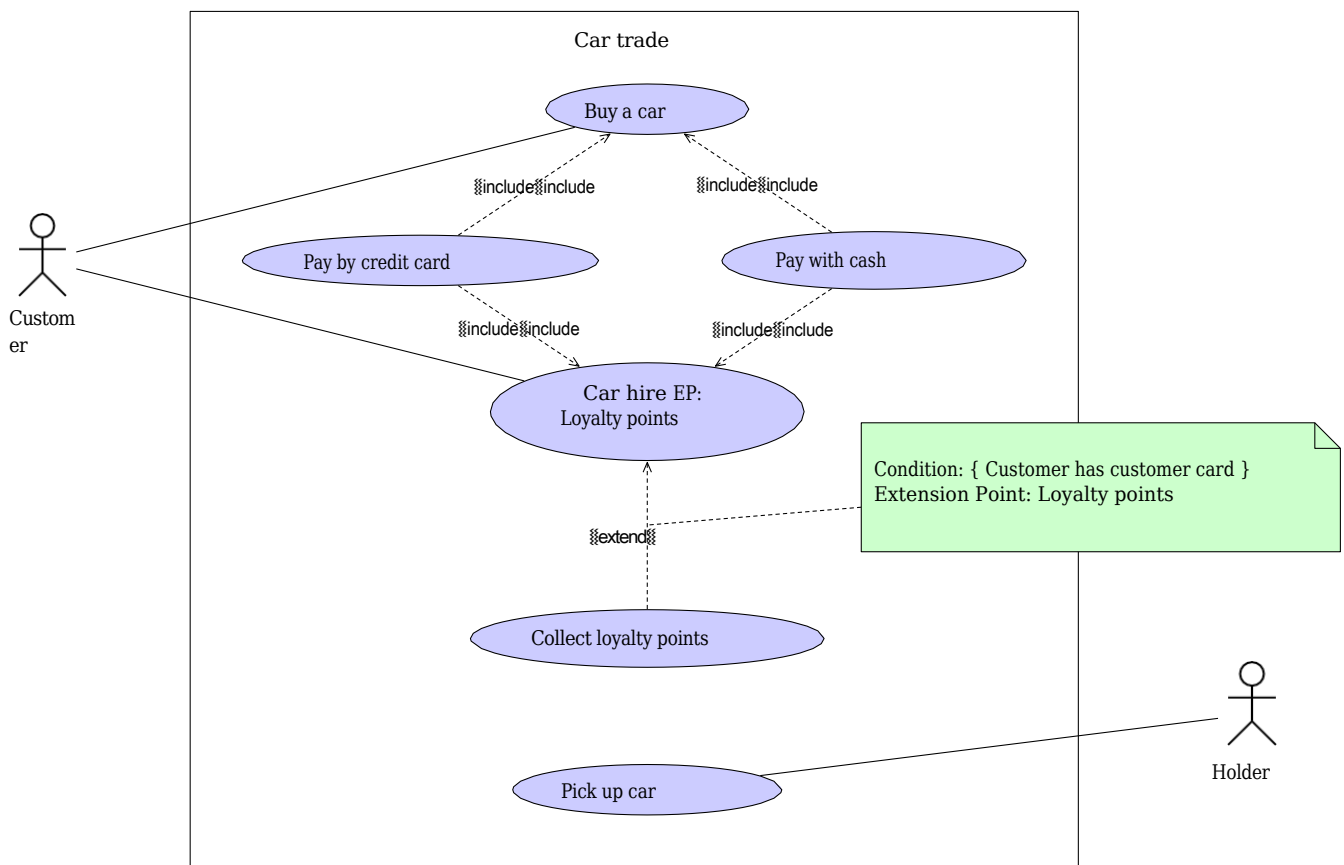


FIG. 1: Example: Use case diagram

## 4 Domain modelling

**Definition - Domain Modelling** Repairing terminology and fundamental activities in the target space (solution space)

**Definition - Domain model** The domain model consists of the objects (including their **attributes**) of the domain and their **relationship** to each other. It is modelled by identifying the relevant concepts that are currently required during the object-oriented analysis. A profound understanding of the domain (the area of application) is required for a good software design (Curtis law).

A domain model (analysis model, concept model)

- the domain into concept objects,
- should develop the concept classes and
- is completed iteratively and forms the basis of software development. Domain

concepts/concept classes are *not* software objects!

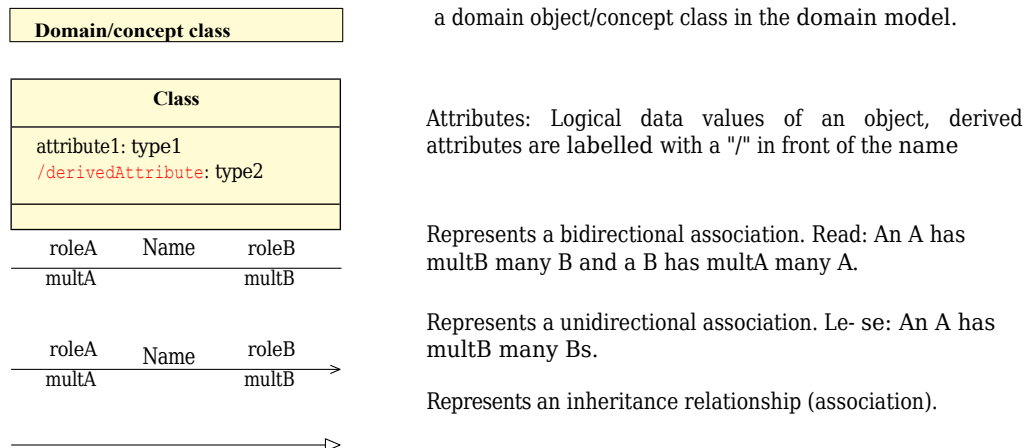
#### 4.1 Diagram: Domain Model (UML)

##### Description

Domain models are visualised with the help of simple UML class diagrams, but only apply individual parts of the class diagram:

- Only domain objects and concept classes
- Associations only (no aggregations or compositions)
- Attributes on concept classes (but should be avoided)

Diagram 2 shows an example of a domain model. The individual components are explained below.



##### Example

At a university, every lecture is read by at least one lecturer. As part of the lectures, students have to complete assignments in learning groups of up to 3 people. Each student can be supervised by exactly one lecturer if the student requests this. Students also attend lectures. If no students attend a lecture, it does not take place. *This textual description is shown in diagram 2.*

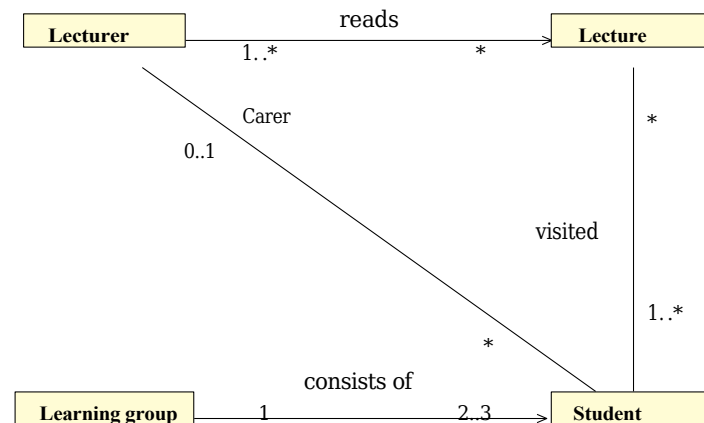


FIG. 2: Example: Domain model

#### 4.1.1 Class vs attribute

**Definition - Description Classes** Contains information/attributes that describe an object Necessary if:

- Information about an object or function is required
- Deletion of the described object leads to data loss
- duplication of information can be avoided

**Heuristic: class or attribute** - If we do not see a concept class C as a number, a text, or a real-world datum, then C is most likely a concept class, not an attribute.

**Heuristic: Insert connection?** If several pieces of information are to be available through the connection over a longer period of time

**Definition - Class name verb phrase-Class name - Format** Separate words with "-" instead of spaces, class names in CamelCase

#### Connection names

- in class name-verb phrase-class name format
- Precise

#### 4.1.2 Attribute vs connection

Attribute	Connection
• <u>Primitive data types are always attributes - relations between classes</u>	

TabZILZ 5: Attribute vs. connection

#### 4.1.3 UML State Machine Diagram Description

State machine diagrams use simplified finite automata to represent event-driven behaviour of the system (behaviour) and interaction sequences (protocol).

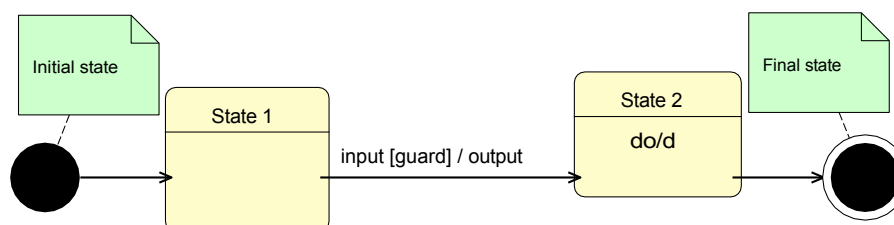


FIG. 3: Example of UML State Machine Diagram

5 Software Architecture

Software architecture includes:

- Architectural characteristics
- Architecture styles (architecture styles, software system structure)
- Architectural decisions
- Design principles

Architects	Development team
<ul style="list-style-type: none"><li>• The characteristics of the architecture from the dependency analysis</li></ul>	<ul style="list-style-type: none"><li>• Create class structure for each component</li></ul>
<ul style="list-style-type: none"><li>• Select an architectural style for the system</li></ul>	<ul style="list-style-type: none"><li>- Design a UI (user interface)</li></ul>
<ul style="list-style-type: none"><li>• Create component structure (via class level) - Write and test source code</li></ul>	

TabZILZ 6: Allocation of responsibilities in software architecture

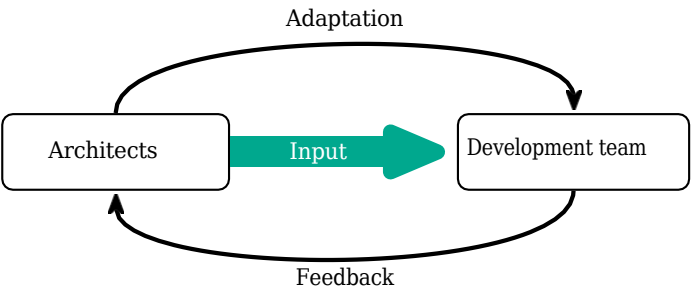


FIG. 4: Relation architect-developer

## 5.1 Architectural characteristics

---

### Definition - Architectural characteristics

- Specifies operational criteria to implement a certain requirement
- Influences the structural design, requires special architectural elements (not standard ones)
- It is necessary that the application works as desired (functional and non-functional dependencies)

### Operation characteristics (Operational):

- Availability: Period of time during which the system is online/available
- Performance: Includes peak utilisation analyses, response times, stress tests
- Scalability: (Scalability): Ability to cope with an increasing number of requests

### Structural:

- Extensibility: How easy it is to add new functionality
- Maintainability: How easy it is to improve or replace (i.e. maintain) the system
- Reusability (Leveragability): Reuse common components in multiple products
- Localisability (localisation): Support for multiple languages, currencies, units, . . .
- Configurability (Configuration): Possibility for the user to configure the system according to his preferences through a usable interface.

### Cross-Cutting (Divide and Conquer)

- Accessibility: Must also be usable by people with disabilities (visual, hearing impairments, . . . )
- Data protection (privacy): Possibility to make certain data inaccessible to other users (including privileged users)
- Security: Encryption, authentication, . . .

---

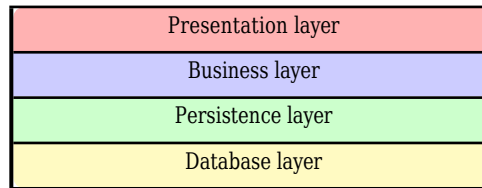
## 5.2 Architectural styles

---

- Help to specify the fundamental structure of a system
- Have a major influence on how the finished architecture looks
- Define the global properties of the system (e.g. how data can be exchanged, what restrictions the subsystems have)

Software systems are usually composed of several architectural styles

### 5.2.1 Monolithic Architecture Styles Layers (Layered)



- Number of levels not fixed, some architectures combine levels or add new ones

pro	against
• Simplicity	- Scalability
• Costs	- Performance (parallelisation <b>not</b> supported)
• Architectural style can be changed later - Availability (long start times, . . . )	
• Good for small-medium applications	

TabZllZ 7: Layered architecture style Pros Cons

#### Model View Controller (MVC)

The MVC pattern splits the software into the fundamental parts for interactive software:

- Model: Contains the core functionality and data
  - Regardless of the output format and input behaviour
- View: Presents the data to the user
  - The data is loaded from the model
    - Controller: Processes the user's input
  - A controller is assigned to each view
  - Receives inputs (e.g. through events) and translates them for the model or the views
    - Every interaction goes through the controller

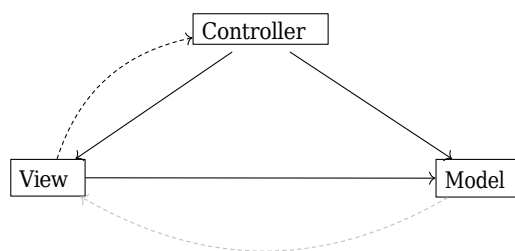


FIG. 5: Model view controller

Controller and view are directly linked to the model, the model is not directly linked to the controller or the view (see 5).



Disadvantages:

- Increased complexity: Splitting into view and controller can increase complexity without gaining more flexibility.
- Update proliferation: Possibly many updates; not all views are always interested in all changes.
- View/controller coupling: View and controller are strongly coupled.

---

## 5.2.2 Distributed architecture styles Distributed Architectural Styles)

---

### Service-based

- Main variant:
  - Only one user interface (UI) for all services
  - Services consist of several components
  - All services access a common database
- Non-monolithic variant (Non-Monolithic)
  - Service-based UIs (one UI per service)
- Service-local database variant
  - Services can have their own or shared databases

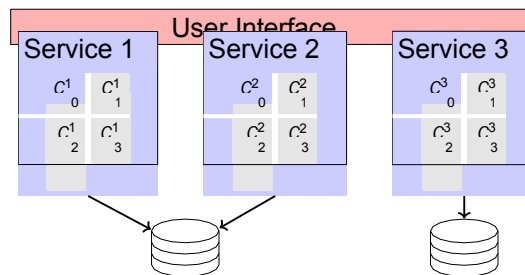


FIG. 6: Example of a server-local database variant of a service-based architecture

## 6 Design principles

### 6.1 Software quality

#### 6.1.1 Factors

The factors of good software are divided into *internal* and *external factors*:

- Internal factors: View of the developers (code quality). Represents a "white box".
- External factors: User perspective (internal quality factors are not known). Represents a "black box".

##### Internal factors

- Modularity
- Comprehensibility
  - Naming (methods, parameters, variables, . . . )
- Cohesion
  - Conciseness (no/few duplicates, clear (short) code)
- . . .

##### External factors

- Correctness
- Reliability
- Expandability
- Reusability
- Compatibility
- Portability
- Efficiency
- Usability
- Functionality
- Maintainability
- . . .

##### Characteristics of good software:

- Maintainability: Can be customised to the customer's requirements
- Efficiency: No waste of resources
- Usability: Must be understandable and usable for the user
- Dependability: Causes no economic or physical damage if the system fails

## 6.2 Measuring software quality

### 6.2.1 Overview of metrics for measuring software quality

Fan In	Number of methods that call $m$
Fan Out	Number of methods that $m$ calls
Code length	Number of lines
Cyclomatic linear independent paths through the code (control flow graph)	Complexity
Nesting depth	Deep nesting of if/else, switch..case, etc. are difficult to understand
Weighted Methodological complexity	Weighted sum of method complexities per class
Inheritance depth	Deep inheritance trees are highly complex (subclasses)

### 6.2.2 Cyclomatic complexity (Cyclomatic Complexity)

The cyclomatic complexity  $C$  is calculated by  $C = E - N + 2P$ , where  $E$  is the number of edges,  $N$  the number of nodes and  $P$  the number of possible connection components (in most cases 1) of the CFG.

### 6.2.3 Control flow graph (CFG)

A control flow graph represents code without syntax, enables better analyses.

#### Example

The code shown in Figure 7 is represented as a control flow graph in Figure 8.

```

1 public static int fibonacci(final int num) {
2     if (num <= 0) {
3         throw new IllegalArgumentException();
4     }
5     int current = 1;
6     int previous = 0;
7     for (int i = 0; i < num - 1; i++) {
8         int next = current + previous;
9         previous = current;
10        current = next;
11    }
12    return current;
13 }
```

FIG. 7: Example: Control flow graph / code

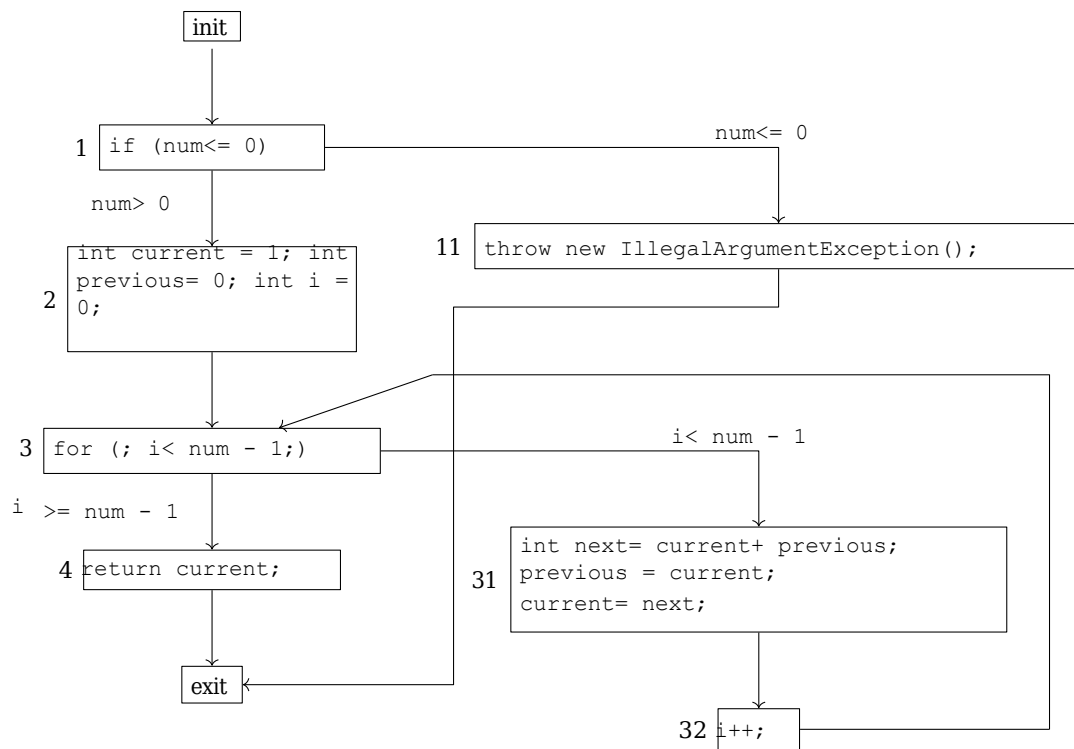


FIG. 8: Example: Control flow graph

#### 6.2.4 Heuristics

Design heuristics to answer the question of whether the design of a software is good, bad or somewhere in between.

- Insights into OO design improvements
- Language-independent and allow the integrity of a software to be categorised
- Not difficult but quick: Should produce warnings which, among other things, allow you to ignore them if necessary

**Example:** All data in a base class should be private; the use of non-private data is prohibited, access methods should be created which are protected.

## 6.2.5 Linking/coupling

**Definition - Class Coupling** is a guideline for measuring the dependencies between classes and between packages:

- A class C is *coupled* to class D if C depends directly or indirectly on d
- A class that is based on 2 other classes has a looser coupling than a class that is based on 8 other classes.

### Coupling in Java

The class

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 public class QuitAction implements ActionListener {
5     @Override
6     public void actionPerformed(ActionEvent event) {
7         System.exit(0);
8     }
9 }
```

is coupled with the following other classes: ActionEvent, ActionListener, Override, System, Object

### Loose coupling vs. fixed coupling

- Too many links are bad, because:
  - Changes in linked class can lead to domino effect of changes
  - Closely linked classes are difficult to understand when viewed in isolation
  - Closely linked classes are difficult to reuse (as all dependencies would also have to be copied)
- However, too few or no links are also undesirable, because:
  - Goes against the principle of object-oriented programming: "A system of linked objects that communicate via messages"
  - Leads to the formation of god classes
  - Leads to high complexity
- Generic classes must have very few links
- Many links of stable modules (libraries, e.g. from the Java standard library) are no problem

**Warning:** The requirement for loose coupling for reusability in (mystical) future projects harbours the risk of unnecessary complexity and high project costs!

## 6.2.6 Responsibilities

---

### Description

**Class** The name of the class/actor.

**Responsibilities** The responsibilities of the class; identifies the problems to be solved.

**Collaborations** Other classes/actors with which the class/actor cooperates in order to a task.

### Important conclusions

Class	- The name should be descriptive and unique.
Responsibilities	- Long list of responsibilities=> Should the class be split up? <ul style="list-style-type: none"><li>• Responsibilities should be interrelated.</li></ul>
Collaborations	- Many collaborators=> Should the class be split? <ul style="list-style-type: none"><li>• <b>Avoid cyclical collaboration!</b> =&gt; Higher levels of abstraction should be introduced.</li></ul>

---

## 6.2.7 Cohesion

---

**Definition - Cohesion** is a benchmark for measuring the relationship between elements of a class. All operations and data within a class should "naturally" belong to the concept modelled by the class.

Types of cohesion (ranked from very poor to very good):

1. **Random** (coincidental, no cohesion present): No meaningful connection between the elements of a class (e.g. for utility classes, undesirable).
2. **Temporal cohesion** (Temporal): All elements of a class are executed "together".
3. **Sequential cohesion** (sequential): The result of one method is passed to the next.
4. **Communicative cohesion** (communicational): All functions/methods of a class read/write to the same input/output.
5. **Functional cohesion**: All elements of a class work together to perform a single, well-defined task. (**ideal case**)

Classes with low cohesion should be avoided, as:

- Difficult to understand
  - Difficult to reuse
  - Difficult to maintain (easy)
  - Symptomatic of very coarse-grained abstraction
  - Tasks that should be delegated to other classes have been taken on
- Classes with high cohesion can often be described with a simple sentence.

### Lack of Cohesion of Methods (LCOM)

**Definition - LCOM value** The *Lack of Cohesion of Methods value* (LCOM value for short) is a guideline value for assessing the cohesion of a class.

Let  $C$  be a class,  $F$  its instance variables and  $M$  its methods (constructors excluded). Let  $G = (M, E)$  be an undirected graph with the nodes  $V = M$  and the edges

$$E = \{ \langle m, n \rangle \mid m \in M \times M \mid \exists f \in F : (m \text{ uses } f) \wedge (n \text{ uses } f) \}$$

then  $\text{LCOM}(X)$  is defined as the number of connected components of the graph  $G$ . If  $n = |M|$  is the number of methods, then  $\text{LCOM}(X) \in [0, n]$  applies.

A high LCOM value is an indicator of insufficient cohesion in the class.

## 6.3 Design principles

### 6.3.1 Single Responsibility Principle (SRP)

**Definition - Single Responsibility Principle** "A class should have only one reason to change", i.e. as few dependencies per class as possible (ideally only one), because dependencies are the main reason for changes

### 6.3.2 Inheritance vs. delegation

Inheritance (take over/inherit)	Delegation
<ul style="list-style-type: none"> <li>(Also) irrelevant functionality is inherited -</li> <li>Difficult to maintain</li> </ul>	<ul style="list-style-type: none"> <li>An object is passed that only contains the desired Functionality implemented</li> <li>- Dependencies unchanged</li> </ul>

TabZILZ 8: Inheritance vs Delegation

## 6.4 Encapsulation

**Definition - Interface concept** An interface declares the method signatures and public constants of its subclasses

- Advantages:
  - Interfaces make it possible to separate the functionality from the implementation
  - to avoid unjustified assumptions about implementation
  - Interfaces are more stable than implementations
  - To change the implementation, it is sufficient to change the constructor
  - Easier to avoid clutter

---

#### 6.4.1 Access rights (Field Access)

---

Assuming there were no access rights and everything was public, then:

- Violates the information hiding principle:
  - No distinction between implementation-specific and public data
  - Implementation-specific details are revealed to the client
- Unstable code if the implementation of the field changes
- for fields that are defined as an argument or under an alias, it is very difficult to recognise the dependencies

So better: Getter and Setter, so that accesses can be specifically controlled Problems if too little is released:

- Functionality may need to be implemented twice
- you to set the dependencies incorrectly

**Warning:** Instance fields (set by the constructor) should never be Public

---

### 6.5 Problems

---

---

#### 6.5.1 God Classes

---

A class encapsulates most or all of the (sub-) system logic. Indicator of:

- Poorly distributed system intelligence
- Poor OO design, which is based on the idea of objects working together



#### Solution approaches

- Even distribution of system intelligence.  
Upper classes should distribute the work as equally as possible.
- Avoidance of non-communicative classes (with low cohesion).  
Classes with low cohesion often work on a limited part of their own data and have great potential to become god classes.
- Careful declaration and utilisation of access methods.  
Classes with many (public) access methods pass a lot of data to the outside and therefore do not keep the behaviour at one point.

---

#### 6.5.2 Class Proliferation

---

Too many classes in relation to the size of the problem. Often triggered by enabling (mystical) future extensions too early.



---

## 7 Design techniques

---

---

### 7.1 Documentation

---

---

#### 7.1.1 Readability

---

- Good readability includes:
  - Documentation
  - Comments in the source code
  - the source code itself:
    - \* Code style conventions
    - \* Restrictions on the use of certain language constructs (e.g. var in Java)
    - \* Good naming of classes, methods and variables
    - \* meaningful comments

---

#### 7.1.2 Types of comments

---

- API comments:
  - Detailed class and method level comments
  - Target group: Other developers who use the code as a library/framework
  - Must contain:
    - \* Restrictions of method parameters beyond the type (e.g. not null or not negative)
    - \* Effects on the condition of the property
    - \* When and which exceptions are thrown
- Statement-Level-Comments:
  - Comments on why which commands were used within a method
  - Purpose: Describe implementation, structure code
  - Target group: Developers working on the same project
  - Should only be used if really necessary
  - Can be an indicator of very complex code⇒ Refactor

---

## 7.2 Revise (refactoring)

---

**Definition - Refactoring** is a disciplined technique to restructure an existing body of code, changing its internal structure without changing its external behaviour.

### Goals:

- Prepare the addition of new functions
- Improve design, e.g. improve cohesion
- Improve comprehensibility
- Improve maintainability

---

### 7.2.1 Reasons for the revision

---

- Makes it easier to add new functions
- Less redundant code
- Avoiding nested conditions
- Makes code simpler/more understandable

---

### 7.2.2 Extract method (Extract Method)

---

- When:
  - when one method does too many things
  - if a functionality is used several times within a method or by several methods
- Procedure:
  1. Create new method (target)
  2. Copy extracted code to the new method
  3. Identify local variables that were used in the extracted code and passed as parameters
  4. Replace code in original method with method call

---

### 7.2.3 Move Method (Move Method)

---

- When:
  - method does not use any other functions of its class,
  - method does not overwrite any other method or is overwritten **and**
  - Method calculates something for another object
- Procedure:
  1. Declare method in target class
  2. Copy method to target class and adapt
  3. Find out how the target class can best be referenced
  4. Convert source method to delegation method (logic place in target class, but call in source class)
  5. Decide whether the source method is still needed or whether the call in the target class makes more sense

---

### 7.2.4 Extract class (Extract Class)

---

- When:
  - Class has two or more independent dependencies
  - No other class is suitable for Move Method
- Procedure
  1. Create new class
  2. New class as an attribute of the old class
  3. Move Method of the method
  4. Check dependencies again

---

## 7.3 Idioms

---

Idioms are *not* design patterns, but small snippets of code:

- Limited in size and
- often specific to a programming language.

## 8 Design concepts (design patterns)

### Definition - Design Pattern

- describes a problem that often occurs within the domain
- provides a blueprint for solving the problem, which can often be reused, but never the blueprint itself but a customised form

- Documented expert knowledge
- Use of generic solutions
- Increasing the degree of abstraction
- A pattern has a name
- The solution can easily be applied to other variants of the problem

### Structure of a pattern

Pattern Name Short, descriptive name

Problem description	- When the pattern should be applied <ul style="list-style-type: none"> <li>• Cases for which the pattern offers a solution</li> </ul>
Solution	- Elements that describe the design <ul style="list-style-type: none"> <li>• Relationships, dependencies, connections</li> </ul>
Consequences	- Side effects, restrictions, effects on flexibility, expandability and portability

### Template Design Pattern

1.	- <b>Name:</b> Name of the pattern <ul style="list-style-type: none"> <li>• <b>Intention:</b> Goals and reasons why you should use the pattern</li> </ul>
2.	- <b>Motivation:</b> Describe the problem situation (scenario) <ul style="list-style-type: none"> <li>• <b>Applicability:</b> Conditions when it can be used             <ul style="list-style-type: none"> <li>• <b>Structure:</b> static structure of the pattern (e.g. UML class diagram)</li> </ul> </li> </ul>
3.	- <b>Participants:</b> Which classes are involved <ul style="list-style-type: none"> <li>• <b>Collaboration:</b> Interaction between the participants</li> <li>• <b>Implementation:</b> How the pattern can be customised/implemented</li> </ul>
4.	- <b>Consequences:</b> Side effects, restrictions, Effects on flexibility, expandability and portability
5.	- <b>Known use cases:</b> Examples of when the pattern should be used <ul style="list-style-type: none"> <li>• <b>Related patterns:</b> references and comparisons with other patterns</li> </ul>

TabZILZ 9: Template Design Pattern

---

## 8.1 Introduction

---

---

### 8.1.1 Template Method

---

**Definition - Template Method** The template method defines the algorithm using abstract (and concrete) operations.

#### Abstract

**Goal:** Implementation of an algorithm that allows it to be applied to several specific problems.

**Idea:** An algorithm is implemented which passes on concrete actions to abstract methods and thus subclasses.

#### Consequences:

- Division of variable and static parts
- Prevention of code duplication in subclasses
- Control over subclass extensions

#### Generic class diagram

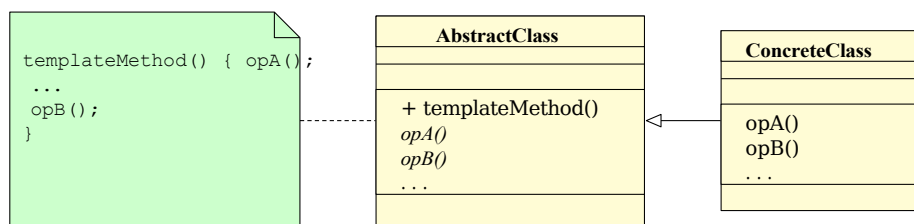


FIG. 9: UML: Template Method Pattern

#### Variants/extensions

Instead of abstract operations, which *have to* be implemented, hooks can be used, which *can* be implemented.

### 8.1.2 Strategy

#### summary

##### Motivation:

- Many related classes differ only in their behaviour instead of implementing differently related abstractions

##### Target:

- Allows a class to be configured with one of many behavioural variants
- Implementation of different algorithm variants can be built into the class hierarchy **Idea:** Definition of a family of algorithms, encapsulation of each and creation of interchangeability. **Consequences:**

- Users must be aware of how the implementations differ and decide in favour of one or the other.
- Users are exposed to implementation errors
- Strategy should only be used if the specific behaviour is relevant for the user

#### Generic class diagram

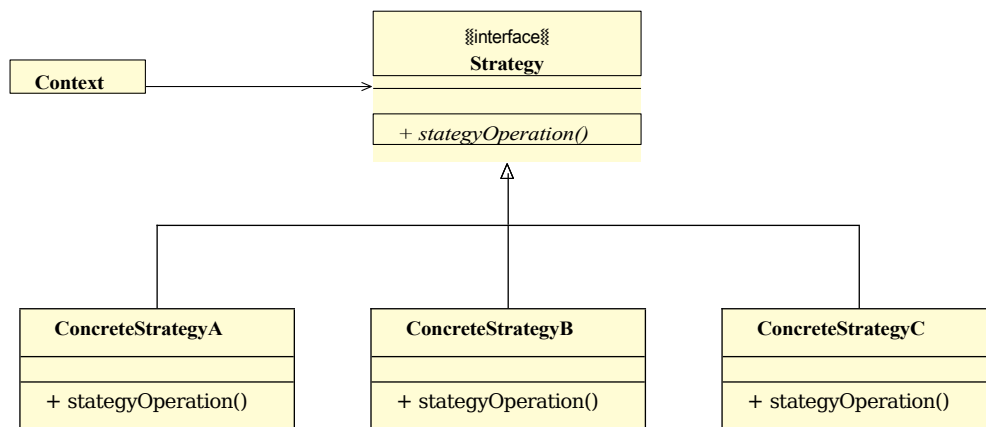


FIG. 10: UML: Strategy Pattern

#### Description of the

The context (user) creates instances of specific strategies that define the algorithm in an interface.

#### Variants/extensions

- Optional strategy object
  - Context checks whether the Strategy object has been set and uses it accordingly
  - Advantage: Users are only exposed to the strategy object if the standard behaviour is not to be used

### 8.1.3 Observer

#### short version

**Motivation:** OOP (object-orientated programming) simplifies the implementation of individual objects, but the wiring of these can be difficult unless you want to hard-couple the objects.

**Objective:** Decoupling the data model (subject) from the entities that are interested in changes to the state. Prerequisites:

- The subject should not know anything about the Observer.
- The identity and number of observers is not known in advance.
- New observers can be added to the system at a later date.
- Polling should be avoided (as it is inperformant).

**Idea:** Creation of observers (generalised by means of an interface), which be added to a subject and called up.

#### Generic class diagram

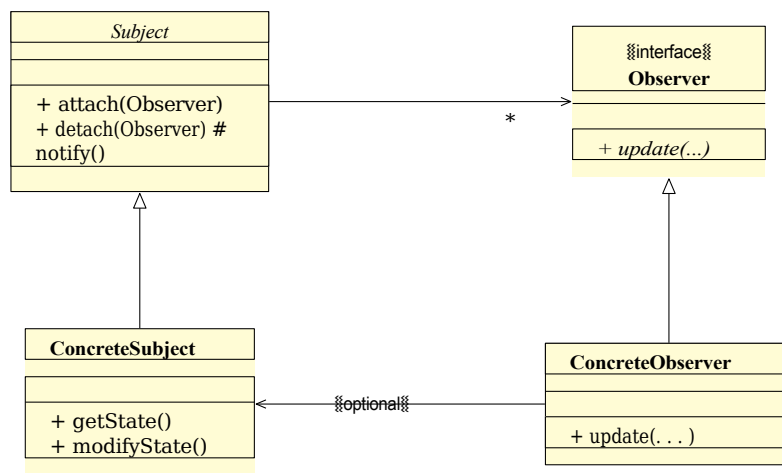


FIG. 11: UML: Observer pattern

#### Description of the

**Subject** Abstract class, offers methods for implementing the pattern.

**Observer** Interface for receiving signals from a subject.

**ConcreteSubject** The concrete subject, sends notifications to the observers

**ConcreteObserver** A concrete observer (implements observer interface), registers with the subject, receives messages from the subject

**Consequences:**

- Advantages
  - Abstract coupling between subject and observer
  - Support for broadcast communication
- Disadvantages
  - Risk of update cascades between subject, observer and its observers
  - Updates are to everyone, but are only relevant for some
  - Lack of details about the changes (Observer must find this out for themselves)
  - General interface for Observer severely restricts the parameters

**Variants/extensions**

- Pull Mode:
  - Subject is asked for change
  - Often results in more data having to be retrieved than just the change
- Push Mode:
  - The change is passed as a parameter to the update() method
  - Only the area that really needs to be changed is changed⇒ better response times
  - Requires precise knowledge of what needs to be updated, making it more error-prone



### 8.1.4 Factory Method

**Definition - Factory Method** Declares an interface for object creation, but lets the subclasses themselves decide which class should be instantiated (e.g. `public abstract Document createDocument()` and class can use e.g. text document or other derivation for this)

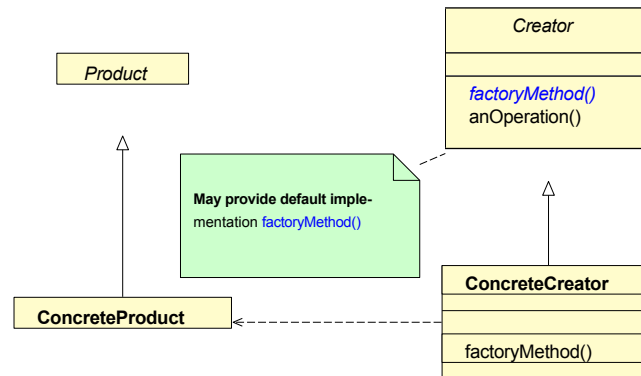


FIG. 12: Pattern Structure Factory Method

Product interface for objects created by the factory method

Concrete product Implements the product interface

Generator (Creator) - Declares the factory method that creates an object of type Product

- Generator can contain default implementation of the factory method that creates a concrete product

Concrete generator Overwrites the factory method so that it creates a different concrete product

#### Consequences

- The client code only recognises the product interface
- Provides a hook for superclasses

#### Variants

- Generator as abstract class
- Generator as a concrete class, with a meaningful default implementation

## 8.1.5 Abstract Factory (Abstract Factory)

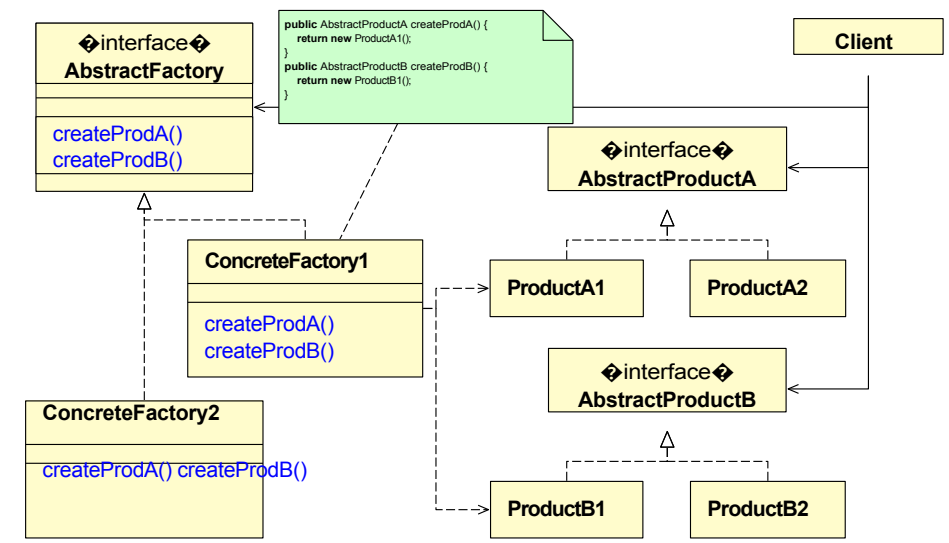


FIG. 13: Abstract factory pattern structure

Abstract factory Provides interface for the product creation of a family

Concrete factory Implements operations to create concrete products

Abstract product Declares an interface to create the concrete products

Concrete product Provides implementation for the product, which was created by the Konkoreten factory.

Client Creates product by using the Concrete Factory, implementing the Abstract Product Interface

- **Advantages:**

- Code does not need to know about all concrete products
- Changing one line of code is enough to support other products
- Can support multiple products
- Forces the creation of consistent product families

- **Disadvantages:**

- Code must follow a new convention for creating product families (instead of using the standard constructor)

---

## 9 Verification

---

---

### 9.1 Introduction

---

---

### 9.2 Verification and validation

---

**Definition - Verification** Is the system created correctly?

**Definition - Validation** Is the right system being created?

---

#### 9.2.1 Techniques

---

##### Static techniques

Static techniques do not require the programme to be executed.

Software reviews Manual/manual review

Automated software analysis, e.g. type tester

Static analysis

Formal verification Formal proof that a programme fulfils a certain property See 6.2.1.

##### Dynamic techniques

Dynamic techniques require the programme to be executed.

Testing Executes the program and tests it for certain properties (behaviour)

Runtime check Analysis tools that test programmes for compliance with certain restrictions (e.g. memory restrictions)

---

### 9.2.2 Code reviews

---

**Definition - Code review** the code review is a structured inspection process of software, usually carried out in a team, with the aim of finding possible errors in the code.

The aim of the code investigation is

- Programme error,
- Standard error and
- to find design

errors.

This is usually worked out in (external) teams that systematically analyse the code. One possible checklist is, for example

Documentation Is the code meaningful, understandable and sufficiently documented?

Comprehensibility Is the code understandable and are the variables meaningfully named?

Structure Are all programming standards adhered to and does the code fulfil the agreed design?

Error handling Are the most common cases covered?

Defensive Are all files and devices left in a valid status in the event of an error? Are the

Programming

Visibilities for the classes, methods and fields as restrictive as possible?

- Advantages:
  - Studies show that it works and can save costs
  - to distribute knowledge about the code to the team members
  - Errors can be found before they appear in tests
  - Can increase code quality
  - No code execution necessary
- Disadvantages
  - Requires an "adult" approach: Team members may feel criticised
  - They feel unproductive under time pressure
  - Only work well if managed correctly

### 9.3 Testing

---

"Programme testing can reveal the existence of errors, but never prove their absence!"

- E. W. Dijkstra, EWD 249

**Definition - Test plan** A *test plan* is intended for execution by humans. It documents the steps of the test and the respective expected result. During test execution, the actual result can then be compared with the expected result.

**Definition - Test case** consists of

- Start state (Pretest state)
- Test logic
- Expected results

**Definition - Test Suite** A collection of test cases is called a test suite (test collection)

**Definition - Test Run** Execution of a test suite on the IUT, if the test passes it gets the "Verdict Pass", otherwise it is labelled as failed.

**Definition - Test Driver** Class or programme that applies the test to the IUT

**Definition - Test Harness** System of test drivers and other tools to support test execution

---

#### 9.3.1 Test types

---

##### Unit tests

Very small, automated tests that test a functionality. Typical software projects include  $\leq 1000$  Unit tests.

##### Integration tests

Testing of a complete (sub-) system to test the co-operation of the components.

##### System tests

Testing of a completely integrated application (function, performance, stress test, . . . ).

## 9.4 Test coverage

---

**Definition - Condition** A *condition* is a Boolean expression.

**Definition - Decision** A *decision* is a composition of conditions which, for example, represents the test of an `if` expression.

**Definition - Basic block** A basic block *B* is a maximum sequence of instructions without jumps.

---

### 9.4.1 Structural

---

Structural test coverage is based on the control flow graph (CFG) of a programme

- Statement Coverage (SC): All expressions were executed at least once.
- Basic Block Coverage (BBC): All basic blocks were executed at least once.
- Branch Coverage (BC): Each side of each node has been executed at least once (i.e. each edge of a control flow graph).
- Path Coverage (PC): All paths have been executed at least once (see CFG).

---

### 9.4.2 Logical

---

- Condition Coverage (CC): Each condition was evaluated at least once for true and false.
- Decision Coverage (DC): Each decision was evaluated at least once as true-false.
- Modified Condition Decision Coverage (MCDC): Combines aspects of condition coverage, decision coverage and independence tests.  
A test for a condition *c* in decision *d* (duplicates of *c* are not counted) fulfils MCDC gdw.
  - he evaluates *d* at least twice,
  - of which *c* evaluates once to true and once to false,
  - *d* is analysed differently in both cases and
  - the other conditions in *d* are evaluated identically in both or not evaluated in at least one. For 100% MCDC coverage, this must apply to every condition in the programme.
- Multiple Condition Coverage (MCC): All possible combinations within a decision were executed at least once.

---

## 9.5 Test automation

---

A test automation system

- starts the "implementation under test" (IUT),
- sets up the environment,
- returns the system to the expected initial state,
- applies the test data and
- evaluates the results and the status of the system.

---

### 9.5.1 Automated Test Case Generation (ATCG) White Box

---

- Syntactic approach: Search for conditions, evaluation enables logic-based coverage
- Unreachable code can be found
- Symbolic execution: try to run CFG with different values, evaluation enables structural coverage

#### Black Box

- Analysing the input and output data of the IUT
- Problems: library calls, unnecessary test cases, recursion, ...

#### Test Coverage Recording

1. Initialise IUT
2. Execute Test Suite and collect information during the test run
3. Analyse and display test coverage

#### Test Oracle Synthesis

- Human oracle: Time-consuming and error-prone
- Testing through code: Needs expert knowledge, difficult to maintain

**Static Checking** Based on CFG, helps with:

- Runtime exceptions, information flow
- Fully automated
- Danger: False positives
- Little time required

#### Dynamic Checking

- Runtime monitoring

#### Formal Checking

- Design-By-Contract (like racket contracts with preconditions and so on)

---

## 10 Maintenance and further development (maintenance & evolution)

---

---

### 10.1 Maintenance (Maintenance)

---

- Software does not age
- However, the software environment changes faster and more often than in any other engineering discipline

---

#### 10.1.1 Trigger

---

1. Fix bugs
2. Dependence is changing
3. New function is required
4. Hardware has evolved
5. Software architecture has evolved
6. Software stack (libraries and such) has evolved

---

#### 10.1.2 Parallelisation as a maintenance task

---

**Observation:** Multi-core processors and systems are becoming increasingly widespread  
⇒ Great speed improvements are possible, but most existing software is still written sequentially.

**Consequence:** Parallelisation of software becomes an important part of maintenance (can be regarded as complex refactoring)

- Correct parallelisation requires a good knowledge of:
  - The Space problem
  - Parallelisable algorithms and data structures (AuD vibes and such)
  - The underlying hardware architecture must support parallelisation

#### The Pipeline Parallelisation Pattern

1. Receive data from the previous processing step (stage) (exit if all data has already been processed)
  2. Process data
  3. Send data to next processing step
- Provides for:
    - Stable, synchronised processing sequence
    - Specific processing steps can be transferred to optimised hardware
    - Allows easy modification, addition and rearrangement of the individual processing steps
    - But cannot compensate for input faults



---

### 10.1.3 Software evolution (Software Evolution)

---

**Definition - Software evolution** The series of changes, new versions and customisation that occur during maintenance between the start of development and the last version.

#### Problems:

- Can destroy already existing functionality (deterioration)
  - Solution: test a lot and distribute beta versions to experienced users if necessary
- Can separate specification, documentation and implementation
  - Solution approach: New functions trigger a "full development cycle".

---

## 10.2 Software Variability Engineering

---

**Definition - Software Variability** The need to maintain a large number of closely related product variants with different dependencies and functions.

---

### 10.2.1 Challenges in variability

---

- Possibly very many variants (millions, . . . )
- Dependency conflicts very likely

---

### 10.2.2 Software Product Line Engineering

---

**Definition - Software Product Line** (also Product Family) A collection of software systems that share common building blocks and production patterns.

- Typical similarities:
  - Code base of the core functionality
  - Architecture
  - Implementation language(s)

**Goal:** Not having to maintain each variant individually

**Definition - Feature (software feature)** A unique selling point of the software (e.g. performance, portability or functionality)

**Definition - Product** A collection of features and parameter instantiations that are sufficient to produce executable code with the SPL.






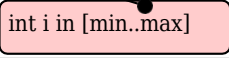
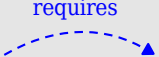
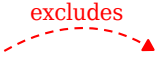
**Definition - Variant** Executable code that implements a product of an SPL.

### SPLE principles

1. Design of the "feature space" is separate from the software design and is carried out **beforehand**.
2. A feature should never be implemented more than once
3. It must be possible to localise the code that implements a particular feature.

### 10.2.3 Feature diagrams Structure

- Above is the "Root Feature"
- No subfeature possible without parent

Symbol	Meaning
	And (And)
	Or (Or)
	Exclusive Or (Xor)
	Necessary feature
	Optional feature
	Set the number of a feature
	Specifies that a feature requires another feature to function
	Specifies that a feature excludes another feature

TabZILZ 10: Feature Diagram Symbols

### Problems

- Code with many macros difficult to understand
- Redundancy is difficult to avoid (code is used by several features)
- Analysis, testing and type checking heavy at family level

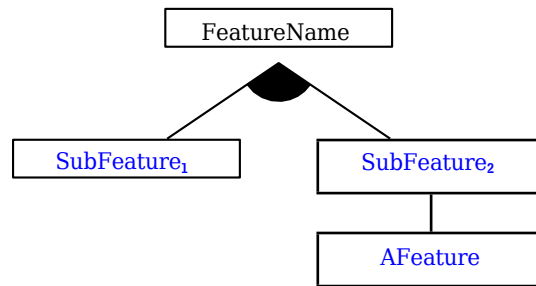


FIG. 14: Basic structure feature diagram

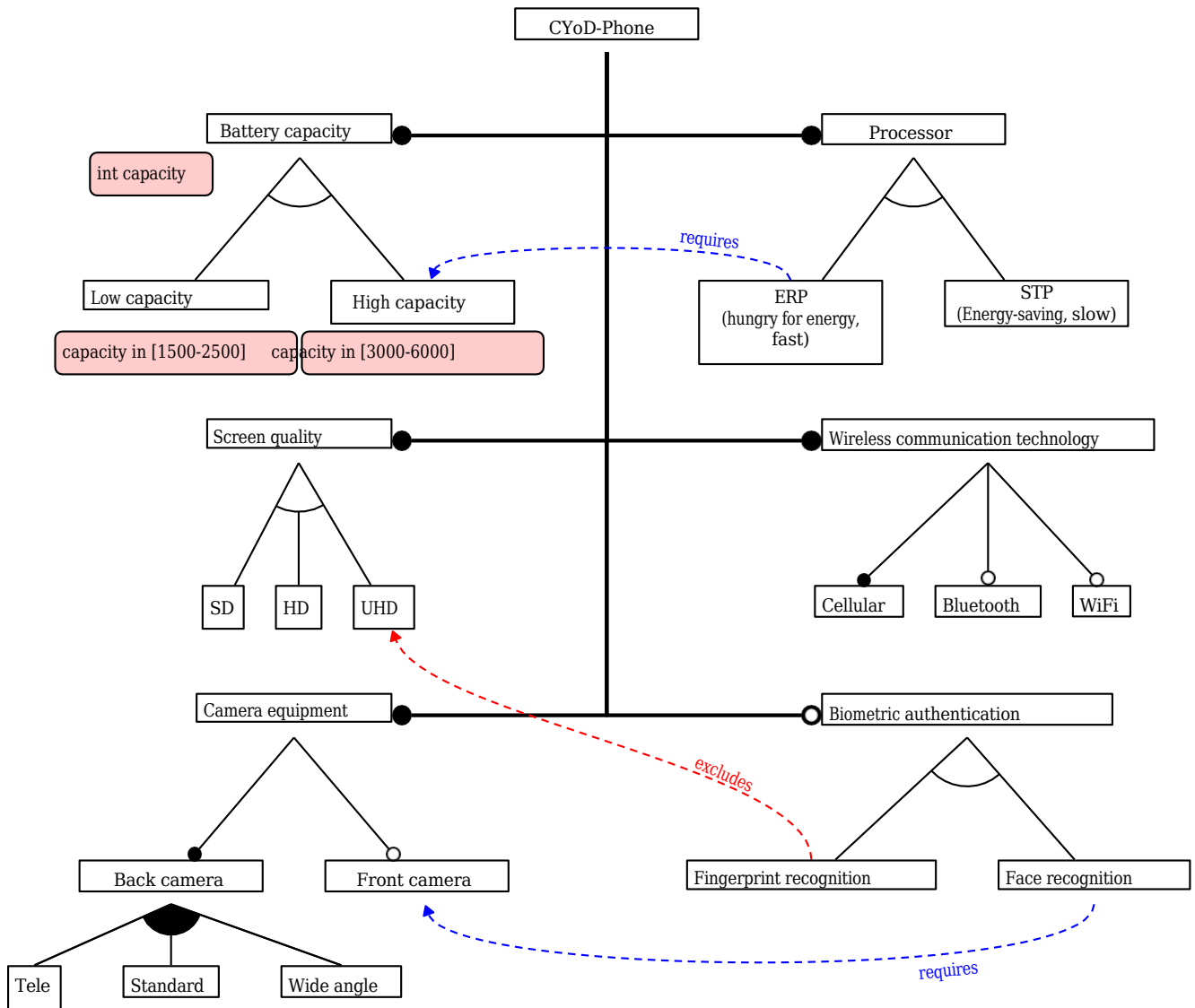


FIG. 15: Exemplary feature diagram

---

## 11 Software processes

---

---

### 11.1 Basic terms

---

---

#### 11.1.1 Basic steps in software development

---

**Definition - Software specification** Definition of the software to be developed and the constraints of this operation.

**Definition - Software development** Design and implementation of the software.

**Definition - Software validation** Ensuring that the software fulfils what the customer requires.

**Definition - Software evolution** Adaptation and modification of software to deal with changes in customer or market conditions.

---

#### 11.1.2 Process models

---

**Definition - Software process models** are simplified and abstract descriptions of a development process that represent a view of the development. They contain activities within the development, software products (e.g. architecture descriptions), source code, user documentation and the roles of the people involved in the development.

Examples of process models are

- Waterfall
- Spiral
- V-Model (XT)
- eXtreme Programming

---

#### 11.1.3 Activities

---

- Project planning
- Cost estimation
- Project monitoring and supervision
- Personnel selection and evaluation
- Presentation

## 11.2 Process model: waterfall

**Description** The following activities are processed strictly in the following order and the next step only starts when the previous one has been completed:

1. Requirements analysis and definition:
  - The requirements are developed in collaboration with the system users
2. System and software design:
  - Definition of the underlying software architecture
  - Identification of abstractions and relations
3. Implementation and unit testing:
  - The software design is represented as a collection of "programme units"
  - Testing is used to check whether each unit fulfils its specification
4. Integration and system testing:
  - Programme units are integrated and tested as a complete system
5. Commissioning and maintenance (Operation and maintenance)

If errors occur in a phase, it is stopped and the previous phase must be repeated. This means that the results of all phases are incorporated into the previous ones, as shown in 16.

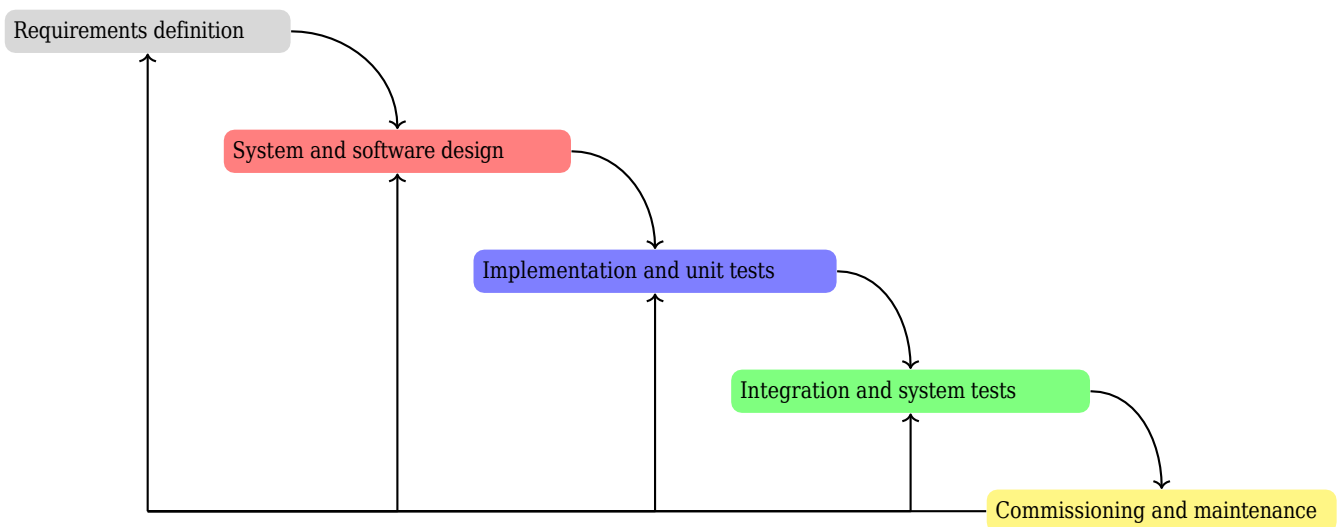


FIG. 16: Process model: Waterfall

---

### 11.3 Agile development

---

**Goal:** Develop software as quickly as possible and incorporate changes to requirements

- Satisfying the customer has top priority
- Regular delivery of interim results (e.g. every fortnight)
- Functioning software is the primary goal (30% implemented- 30% of the project completed)
- Simplicity is essential
- Changes to requirements during development are problematic
- The team reflects at regular intervals in order to become more effective
- Managers and developers must work together

---

### 11.4 Process model: eXtreme Programming (XP)

---

In XP, work is carried out in short iterations, with specific user stories being processed in each iteration. The user stories to be processed are defined in the iteration planning.

#### User stories

**Definition - User stories** The essential part of development. A user story describes a requirement. A user story is then broken down into individual tasks.

- User stories must be understandable for the customer
- Every user story must be of value to the customer
- User stories should be extensive enough that you can work through a few per iteration
- User stories should be independent
- Every user story must be testable
- Each user story is assigned *story points*, which represent an abstract measure of time for estimating the processing time
- There should be an acceptance criterion in every user story

Long template: As a <role> I want <destination>, so that <benefits>. Short template: As a <role> I would like <destination>.

---

### 11.4.1 Development

---

**Test-driven development** Every implementation starts with the development of tests so that the implementation is correct as soon as all tests work.

**Continuous integration** Developers regularly check in their code (several times a day) so that a CI server can regularly run tests (automated tests and build system).

**Pair programming** Code is written by exactly two developers together

- One focusses on the best way of implementation
- The other focusses on the code (from a strategic point of view)

---

### 11.4.2 Planning

---

#### Initial planning (start of the project)

- Developers and customers work together to identify the relevant user stories
- Developers estimate the story points for each user story  
A user story with twice as many story points as another should take twice as long.
- The *velocity* (story points per time) is required for an accurate time estimation; this becomes more accurate over time (a prototype implementation for measuring the velocity is called *Spike*)

#### Release planning

- Developers and customers agree on a date for the first release (2-4 months)
- Customers decide on the rough composition of the user stories (taking velocity into account)
- If the velocity becomes more precise, the release schedule is changed

#### Iteration planning

- The customer searches for the user stories for the iteration
- The order of processing is a technical decision
- The iteration ends on a specific day, even if not all stories have been processed
- The estimate for all *successfully processed* stories is totalled and the velocity is calculated
- The planned velocity of the next iteration is based on the velocity of the previous iteration

$$\text{Velocity} = \frac{\text{Sum of Story Points}}{\text{Total time spent per user story in an iteration}}$$

or

$$\text{Velocity} = \frac{\text{Sum of story points}}{\text{(usually 1)}} \times \text{Sum of iterations}$$

#### Planning tasks

- Stories are broken down tasks that require 4h to 16h
- Every developer can choose tasks