Software Engineering Summary

Moritz Gerhardt

moricetamol

Table of Contents

1 General

1.1 What is Software?

Software can describe a lot of things, some examples include:

- Executable programs
- Configuration files
- System documentation
- User documentation
- Support environment
- etc.

In general Software can be divided into three categories:

- Application Software
  - Interacts directly with the end user
  - General purpose software (To be used in other applications: Word processing, image processing, etc.)
  - Customized Software (Software specifically for a specific purpose: CAD, IDE, BIM, etc.)
- System Level Software
  - Does not interact directly with the end user
  - Responsible for keeping systems running (Operating System, firmware, drivers, etc.)
- Software as a Service (SaaS)
  - Runs on a server
  - Indirectly accessed via client (browser, remote shell, etc.)

Furthermore, some characteristics of Software include:

- Software does not wear out, its environment does
  - Software is subject to continous change in hardware, needs to be able to adapt
  - Software should be able to support new requirements, use cases etc.
- Software often lives longer than anticipanted
  - Almost impossible to know use cases in advance as it can be in use for years or even decades (Excel used in biology geneology → lead to unexpected behaviour)
- Software properties are hard to measure
  - How does the code relate to software quality?
  - How do we measure progress?
  - How do we measure resilience?

1.2 Software Engineering

Typically a software is designed to solve different needs of different groups involved in the development of the software.

- Customer / Client
  - Often the person / organisation that'll pay for the development
  - Sets a budget, timeframe, requirements etc.
  - $\rightarrow$ Requirements analysis
- User
  - Usually the person / organisation that'll use the software
  - Defines what the software is used for and subsequently what requirements this sets
  - $\rightarrow$ Use Case Analysis
- Manufacturer
  - Usually the person / organisation that'll design and develop the software
  - Is concerned with how to build the software in a way that satisfies the customers and users
  - $\rightarrow$ Domain Modelling, Architecture, Quality Assurance, Design Practice, Verification
- Maintainer
  - Usually the person / organisation that'll maintain the software during its lifetime
  - Responsible for maintenance of the software and updates to make it usuable for new demands and requirements
  - $\rightarrow$ Maintenance and Evolution

After all these aspects are consideres the software system is the built with the specific requirements in budget and time.

There are quite a few problems that can happen with Software:

- Unexpected Errors:
  - Few errors are obvious
  - Most of them are near impossible to test for and detect (Algorithmic error, arithmetic overflow)
  - Often go undetected for a long time as they're usually the result of very specifc inputs for complex computations
- Altough errors can occur, as long as they do no violate the requirements they are not considered errors:
  - INABIAF: It's not a bug, it's a feature
- Most errors are caused by missing verification, validation or documentation.
  - This usually indicates an insufficient match between requirements and implementation

Errors can also occur as a result of social aspects:

- Insufficient validation
- Inadequete Specification
- Constantly changing requirements
- Insuffciently trained software engineers
- Managment with lacking grasp on software development
- Unsuitable methods, languages, tools etc.

2 Requirements Engineering

In the following we are gonna look at requirements engineering using the case study of a car sharing service. The main roles and functionalities of a car sharing service are:

Main Roles & Functionalities

- Role-Independent
  - Authentication
- Administrator
  - Add / change new cars, rental locations
  - Biling
- User
  - Check availability
  - Request booking
  - Change booking
- Service Staff
  - Take out vehicle for service

Requirements Analysis is concerned with building a system of what the product *needs* to fulfill in terms of budget, time and surrounding criteria.

So the objectives are akin to:

What has to be developed?

- Need to understand the problems that arise in the requirement elicitation phase
- The different kinds of requirements
- The requirement engineering workflow
- Modelling requirements
  - Scenarios & Use cases
  - Notations: Textual and Graphical

Although the objectives seem pretty straightforward, requirements analysis can be tricky due to how ambigous language can be. Thorough communication is important to understand fully understand what the client wants.

What is Requirements Engineering?

The process of

- finding
- analysing
- documenting
- validating

software requirements.

2.1 What are Requirements?

Definition

- Requirements as descriptions of the services provided by the system
  - Car booking
  - Service booking
  - Location tracking
  - etc.
- Requirements as the operational constraints of the system
  - Database throughput
  - System memory
  - Navigation systems
  - etc.

These requirements are usually handled in the form of **System Requirements Specification (SRS)** Documents (Ger: Pflichtenheft) or **User stories**, structured natural language of use cases, state diagrams etc. stored in the product backlog (ordered list of requirements)

2.2 Different Types of Requirements

Overall the requirements can be divided in the following:

- User Requirements
- System Requirements
- Functional Requirements
- Non-Functional Requirements
- Domain Requirements

2.2.1 User Requirements

State in language or diagrams:
- What services the system should provide
- What the operational constraints are

The descriptions are often high-level and abstract.

For example "According to german law, a car sharing service must keep track of all bookings"

2.2.2 System Requirements

Precise and detailed specification of the systems
- functions
- services
- operational constraints

For example: "After a successful booking the user must be shown an overview of their booking"

"Booking details must be stored for 10 years"

Characteristics

- Refinement of user Requirements
- Determine system interface (functional)
- Recorded as part of the SRS and part of the contract with the client
- Authored by software developer or business analyst with the client

2.2.3 Functional Requirements

Functionality that is clearly identifiable and localized in the code

- Services providede by the system
- System reactions to inputs or events
- System behaviour in specific situations like Network distruption

2.2.4 Non-Functional Requirements (NFR)

Constraints of the services or functions

- Service Level Agreement (SLA)
- Constraints from development process
- Alignment to standards (e.g. Protocols)

NFRs often apply to the whole system as they cannot be handled by simply adding a peice of code.

For example: "The database must be able to process 1000 queries a second" "User data must only be accessible to authorized persons"

Examples of Non-Functional Requirements

- Product requirements
  - Reliability (crashes, use cases)
  - Efficiency (performance, memory)
  - Portability (Not confined to one device or service)
- Organisational Requirements
  - Delivery mode (beta, continous)
  - Implementation (Programming language, framework)
  - Standardization (ISO standards or similar)
- External requirements
  - Interoperability (TUCaN $\Leftrightarrow$ Moodle)
  - Ethical aspects
  - Legal aspects (safety, security, privacy)

NFRs may often result in the identification of functional requirements and are often more important to adhere to strictly than individual functional requirements.

A problem with NFRs come from how subjective they are: What is ethical, what is ease of use, what is good performance etc.?

2.2.5 Domain Requirements

Are derived from the application domain rather than the needs of the user

- Often expressed in domain specific language → Hard to understand for software engineers.
- For example: Software engineers usually do not have profound knowledge of chemistry, however the client might be a chemist and needs software that can be used for very specific applications.
- Often implicitly assumed as obvious to domain experts
- Can be functional or non-functional

2.3 Feasibility Study

The objective of the Feasibility Study is to obtain a justified understanding of whether the requirements engineering and system development phases should be **started**. This is usually base on:

- Business requirements
- Outline description of the system
- Description of how the system should support the business

The resulting **Feasibility Report** then covers

- Whether the system contributes to the objective of the organization
- Whether the system can be implemented within technical, financial and schedule constraints
- Whether the system can be implemented using other systems used by the company

2.4 Requirements Elicitation and Analysis

2.4.1 Requirements Discovery

Systematic Requirement Discovery
Viewpoint-Oriented Approach

- Interactor Viewpoint
  - People or systems who interact directly with the system
  - End Users, Administrators, Service Personal, etc.
  - **Direct Stakeholders**
- Indirect Viewpoint
  - Stakeholders who influence the requirements, but won't use the system directly
  - CFO, Data protection personal, etc.
  - **Indirect Stakeholders**
- Domain Viewpoint
  - Domain characteristics & constraints that influence the requirements
  - Legal, Ethical, etc.

The goal of the requirement elicitation process is to develop more specific viewpoints and use them to discover more specific requirements.

The elicitation can be done in an interview which are usually structured as follows:

Systematic Requirement Discovery
Interviews

- Closed Interviews:
  - Predefined questions
- Open Interviews:
  - No predefined agenda
- Interviews should only be used as a supplement:
  - Interviewee can be biased
  - Interviewee can assume domain knowledge

Some further elicitation techniques are:

Systematic Requirement Discovery
Other Techniques

- Scenario Analysis
  - Analyses the sequence of interactions with the system
- Use Case Analysis
  - Analyses the use cases of the system

2.4.2 Requirements Classification & Organisation

For further structured workflow the requirements should be categorized, this can be done using the **FURPS+** Model:

- **F**unction
- **U**se
- **R**equirements
- **P**riority
- **S**cope
- +
  - Implementation
  - interface
  - Operations
  - Packaging
  - Legal

2.4.3 Requirements prioritisation & Negotiation

Another problem in the elicitation process are conflicts. Different stakeholders might have different requirements. These conflicts need to be resolves through negotiation.

2.4.4 Requirements Documentation

The produced requirements are then documented and used as a basis for further elicitation and analysis. These documents (SRS) can be formal or informal.

SRS Target Groups

- Client, users
- Managers: Client and Manufacturer
- System Engineers, system testers, system maintainers
- Anyone concerned with ordering, using, manufacturing or maintaining

The level of detail of the SRS depends on the system, development process, whether the product is developed in-house or external etc.

The usual format of an SRS is:

System Requirement Specification (SRS) Document Format

1. Introduction
   (a) Purpose of the SRS
   (b) Scope of the product (Also what isn't in the scope)
   (c) Glossary
   (d) References
   (e) Overview
2. General Description
   (a) Product perspective
   (b) Product functions
   (c) User characteristics
   (d) Limitations
   (e) Assumptions and dependencies
3. Specific Requirements
4. Appendices, Index, etc.

2.4.5 Requirements Validation

Requirement Validation Checklist

- Validity
  - Do the requirements capture the needed features?
  - Is additional functionality needed?
- Consistency
  - Are the requirments conflicting?
- Completeness
  - Do the requirements cover all the features and constraints?
- Realism
  - Can the requirements be implemented feasably?
- Verifiability
  - Is there criteria to check whether the requirements are met?
- Traceability
  - Is each requirement tracable to the source of the requirement?

3 Use Case Analysis

3.1 Client Side Involvement

To identify all and good use cases, it's imperative to involve the users. This is usually very expensive: Around 30-50% of development costs are allocated towards requirements and use case analysis and validation.

Use cases usually are text stories used to discover and record requirements. These use cases complement requirments analysis and provide operational requirements as a basis for system design. They do not replace requirement analysis as they do not capture non-functional requirements.

Definitions of Constituents of Use Cases

- Actor
  - Someone or something with behaviour (person, computer system, organisation, etc.)
  - **Primary Actor:** The person who initiates the use case (requests a service)
- Scenario (Use Case Instance)
  - Specific sequence of actions and interactions between actors and system
  - One particular story using a system
- Use Case
  - Collection of related success and failure scenarios
  - Describe an actors usage of a system to achieve a goal

Different Kinds of Use Cases

- White Box vs. Black Box: With whom does interaction occur?
  - White Box (Transparent): Use cases provide details on internal interaction with the system
  - Black Box: Use cases describe only interactions with external actors
- Corporate vs. System
  - Corporate: Use cases describe business process
    (Usually white box)
  - System: Use cases are described with respect to the system
    (Usually black box)

Use Case Formats

- Brief: Short, one paragraph summary. Usually outlines main succes scenario
- Casual: Informal, Multiple paragraphs that cover multiple scencarios
- Fully Dressed: All steps and variations in detail. Includes supporting sections on preconditions, success guarantees etc.

Should be precise (detailed) and accurate (correct).

Fully Dressed Use Case Template

- Use Case Name
  - Start with a verb ("Accomplish this task")
- Scope
  - Corporate, system (name), subsystem
  - Design Scope: Boundaries of the system of the use case (whole corporation, (sub-)system name)
  - Function Scope: Limits functionality to be realized. Managed by a list of functions in and out of scope
- Level
  - User goal, summary goal, subfunction
  - User goal: Most important goal of the user
  - Summary goal
    - Multiple User Goals: Describe context of system
    - Life cycle sequence of related goals
    - Table of content for lower-level use cases
  - Subfunction: Use case that is part of user goal. Singled out on a by-need basis, reusable in multiple goals
- Primary Actor: Initiates use case
- Stakeholders and Interests: Who is interested in this and what do they want?
- Preconditions
  - What must be true or worth telling
  - Enforced by system and known to be true
  - Will not be checked again during execution
- Minimal Guarantee
  - Fewest promises the system makes to Stakeholders
  - Especially if the primary actors goal cannot be achieved
  - (MVP) Minimal Viable Product
- Success Guarantees
  - What must be true on successfull completion
  - States the satisfied interests of the stakeholders after successful completion
- Main Success Scenario
  - Representative Scenario of successful execution
  - Numbered list of steps executed
  - Each step may reference a sub use case
  - First step specifies trigger of use case
- Extensions
  - Alternative scenarios of success or failure
  - Refer to main success scenarios step, by explaining alternative scenario for each step as well as the condition or failure needed for the alternative
- Special Requirements: Related non-functional requirements
- Technology and Data Variation: Needed / Used Technology and Data formats
- Frequency of Occurence: How often does the use case occur?
- Miscellaneous: For example: open issues

For developing use cases one should proceed incrementally. Meaning that first relevant use cases should be accurately identified as a high level and then add precision gradually.

Recommended Workflow and Tips

1. List supported actors and goals
   Review list for accuracy and completeness
2. Write stakeholders, triggers and main success scenario for each use case
   Validate that the system delivers to important stakeholders
3. Identify and list failure conditions
4. Write Failure handeling

- Start simple and focus on intent
- Write black box use cases
- Focus on actors and users of a system and their goals

A well defined task in general should fulfill the following requirements:

- performed by one stakeholder in one place at one time
- model a business event
- add measurable business value
- leaves data in a consistent state
- be more than a single step

This is called the **Elementary Business Process (EBP)**.

3.2 UML Use Case Diagrams

The Unified Modeling Language is a visual, precise design notation for software development.

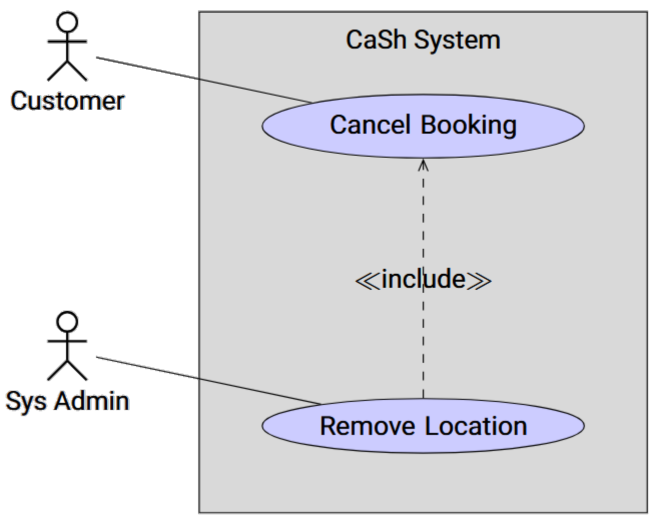
UML Use Case Diagrams Remarks

- UMLUCD is intentionally minimalist
- UMLUCD are an organizational method to improve communication and comprehension of use cases and to reduce text duplication
- UMLUCD provide a black-box view on system software
- Are only useful for early phases of use case analysis →not suitable for fully dressed use cases

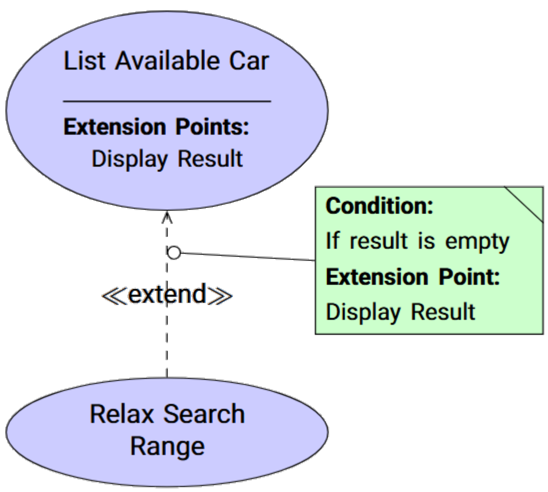
These diagrams essentially consist of system boundary (scope of the system), actors and use cases, as well as their relations.

UML Use Case Diagrams Relations

- «include»:
  - Factors out common behaviour between use cases into sub-function
  - Facilitates decomposition of large use cases and enables reuse
  - Included use cases are *always* executed
  - (Arrow goes from sub-function to base use case)
- «extend»:
  - Describes where and under what condition an extending use case extends the behaviour of the base use case
  - Most extensions do not qualify as seperate use cases →Should only be used when really justified



UML Include



UML Extend

4 Domain Modeling

Domain modeling is a method for identifying the relative concepts and tasks of a domain. It is used to fix the terminology and the fundamental activities of the domain.

Domain Model

- Goal:
  - Decompose domain into concepts or objects
  - Represent the real word (as defined by requirements specifications)
- Creation:
  - Identify a set of conceptual classes and fundamental actions
  - Completed iteratively, forms basis or software design
- Synonyms:
  - Conceptual Model, domain object model, analysis object model

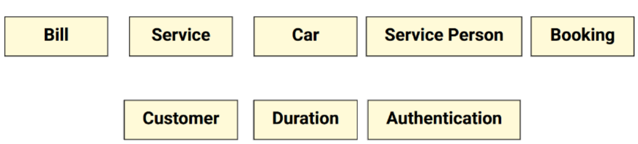
Conceptual Classes

- Represents ideas, things or objects in the domain
- Attributes:
  - Name or Symbol representing the class
  - Intention
  - Extension (contains domain elements)

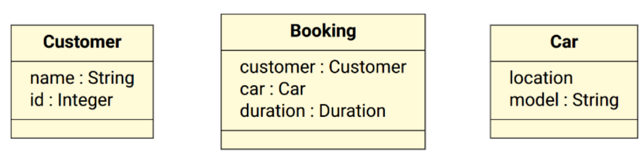
4.1 Visualization of Domain Models

Domain Models are visualized using **UML Class Diagrams** with suitable restrictions to emphasize domain modeling:

- Only domain objects and conceptual classes
- Only associations, no aggregation, no composition
- Classes may have attributes but no operations



Example of Conceptional class (Car sharing)



Example of conceptual classes with attributes (domain objects)

Hereby an objext is defined as an individual thing with a state and relations to other objects.

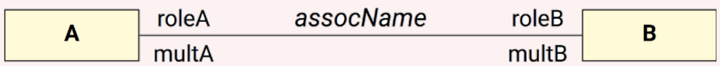
UML Class Elements and Conventions

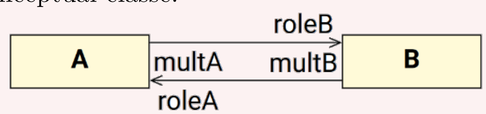
- Class Name: Always starts with an upper case letter
- Attributes:
  - Name: starts with a lower case letter
  - Type: Pre-defined type or other domain model class (can be omitted)
- Derived Attribute:
  - Name: prefixed by a slash, followed by a lower case letter
  - Describes a value computable from existing information

UML Associations / Relations

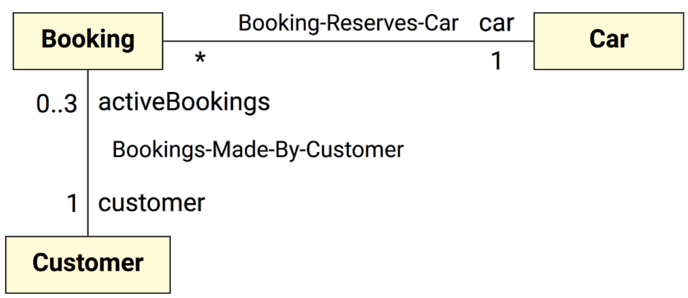
An association is a relation among classes. It consists of the following:

- Name: (optional) Should be done according to the **Class name-verb phrase-Class name** format:
  - Player-Stands-on-Square
  - Sale-Paid-by-CashPayment
  - Customer-Traveled-by-Vehicle
- Two Roles (associations):
  - Name: Defaults to class name in lowercase
  - Multiplicity: Defaults to 1.
    Possible: * (arbitrary/all) and a..b (Range, upper bound inclusive)
  - Navigability: Defaults to bidirectional, not used for conceptual classe.





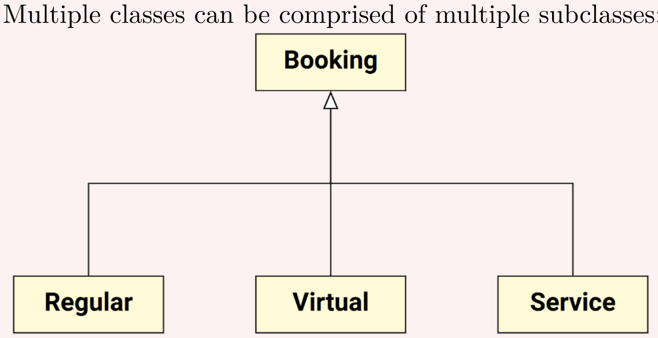
Associations should be included in the domain model if the knowledge of the relation needs to be preserved. For example: The relation between a bill and its entries needs to be preserved. However the relation between a user and their recent searches is not necessarily important.



Example of UML Class Association

Class Generalizations

Multiple classes can be comprised of multiple subclasses:

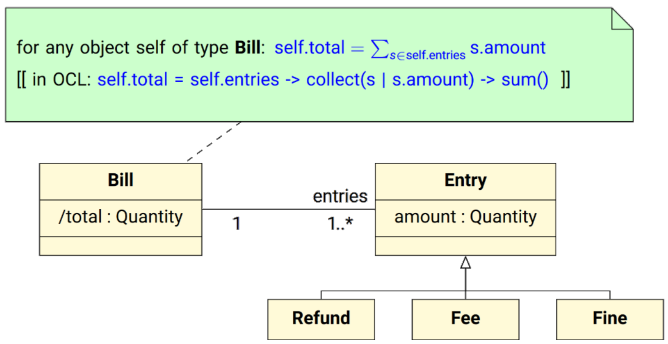


4.2 Eliciation of Domain Models

Workflow

1. Find the conceptual classes
   Strategies:
   - Re-Use or modify existing model
   - Use a category list
   - Identify noun phrases
2. Draw elicited concepts as classes in a UML class diagram
3. Add attributes
4. Add associations

4.2.1 Re-Using Existing Models



4.2.2 List of Conceptual Classes Categories

Some Categories:

- Physical objects
- Specifications of things
- Locations
- Events
- Transactions
- etc.

Conceptual Class Category	Conceptual Classes (in CaSh)
Business transaction	Bill, Payment
Transaction line item	Entry
Product or service related to a transaction or to a transaction line item	Refund, Rent, Fine
Place where transaction is recorded	Registry
Roles of people or organizations related to a transaction (actors in use cases)	Customer, Accountant
Location where transaction executed	Website
Noteworthy events, with a time or place that needs to be remembered	Bill, Booking

4.2.3 Noun Identification

Workflow:

1. Identify nouns and noun phrases in textual description (Use Cases for example) of domain
2. Consider them as a candidate for a conceptual class or attribute

Criteria for inclusion of conceptual classes:

- Must carry information not available/computable from other sources
- Must have specific semantic in relation to the business

Can only be partially automated due to the ambiguity of natural language

4.3 Description Classes

A description class contains information that describes an entity. For example, a description class for a car would contain information about the car's make, model, color, etc.

A description class should be added to the model if:

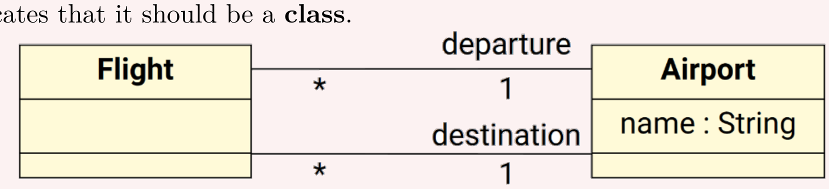
- Information about the entity is required, regardless of whether an instance of the entity even exists.
- Deleting an instance of the entity would result in loss of information.
- Redundant or duplicate information is reduced

Class or Attribute

When deciding if a piece of information should be included as an attribute or a class: If notion C is not considered:

- Number
- Text
- Date

that usually indicates that it should be a **class**.



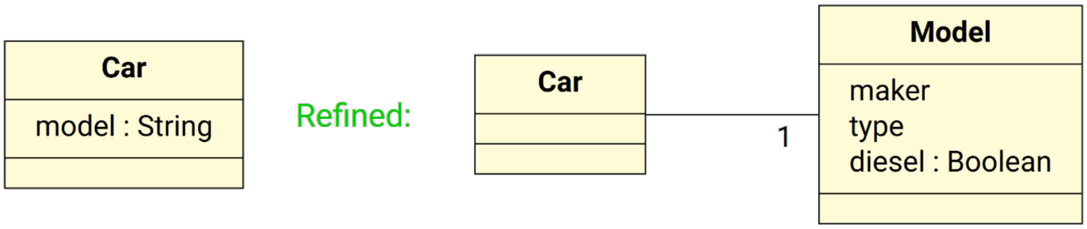
When considering the destination of a flight, it makes more sense to put that information into a class Airport, instead of making it an attribute of flight. Also allows for better alotment of additional info.

When to use Attributes or Associations

- Attributes should always describe primitive datatypes:
  - Boolean, Integer, Character, String...
  - Dates, Address, Colors, Phone Numbers...
- Quantities may be modelled as classes to attach units:
  - currency: EUR, USD, CAD...
  - distance: meters, miles, millimeters...
- Relations between conceptual classes are always associations

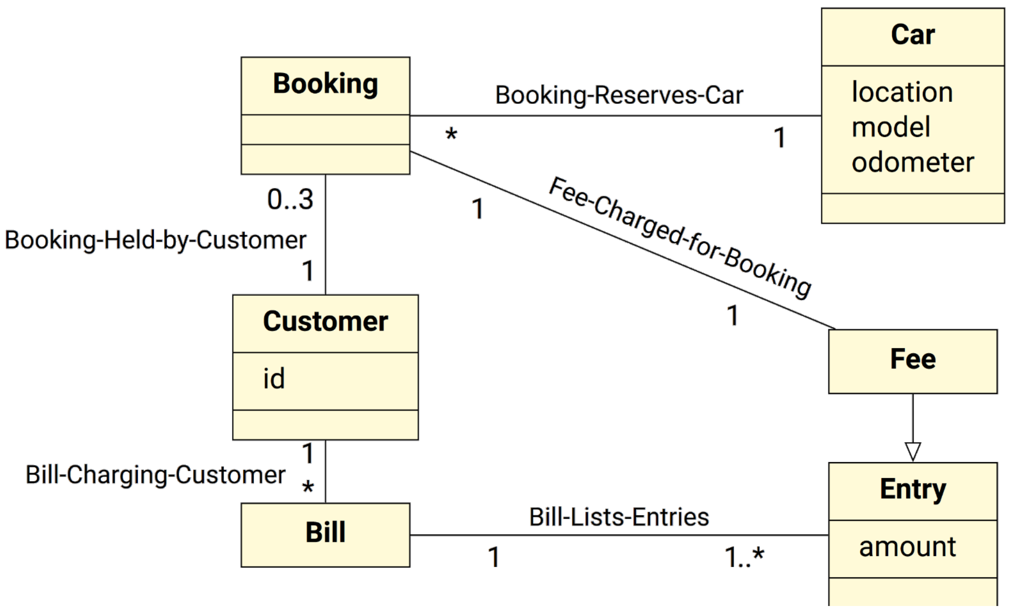
4.3.1 Refining the model

Initially it is very convenient to type attributes with Strings. It is a generic type that avoids premature decision. Later on it can be refined into a description class:



Obviously a string can only be refined if it actually contains more information that can be shown differently.

In general, the domain model serves as an inspiration for the design model later on.



An example Domain Model

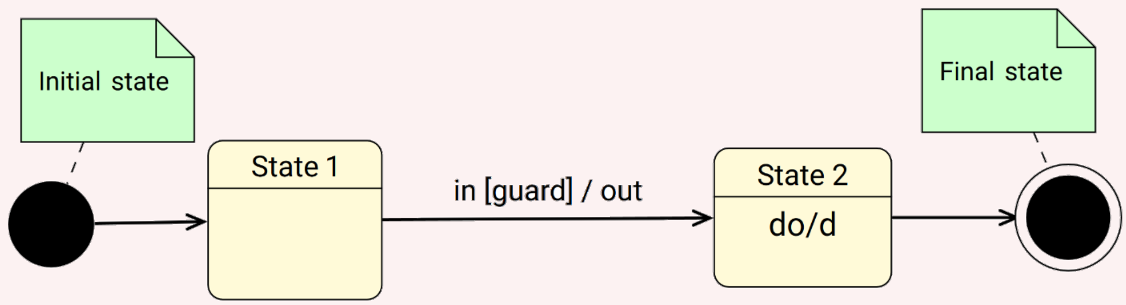
4.4 Behavioural Domain Modelling

Class Diagrams only model static aspects such as classes, objects, attributes, associations and functions.

What we are missing are the behaviours, the sequence of actions and how they change the state of the system and under which condition.

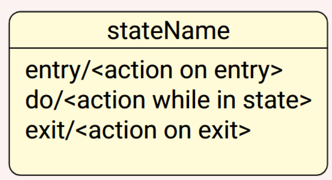
These behaviours can be displayed as UML State Machines Diagrams.

UML State Machine Diagrams



This can be read as: *When in State 1, if input in is observed and guard is true, then output out happens and current state becomes state 2. In state 2 perform the (interruptible) action d.*
So in General UML State Machine Diagrams show States and their conditions for transitioning as in- and outputs and guards.

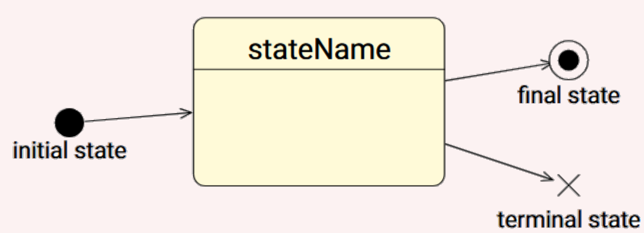
Basic States



States consist of the following:

- Name: Short description of what the state represents
- Actions: Executed Operations
  - Entry: Action performed on state entry
  - Do: Action performed while in state (until terminated or state is left)
  - Exit: Action performed on state exit

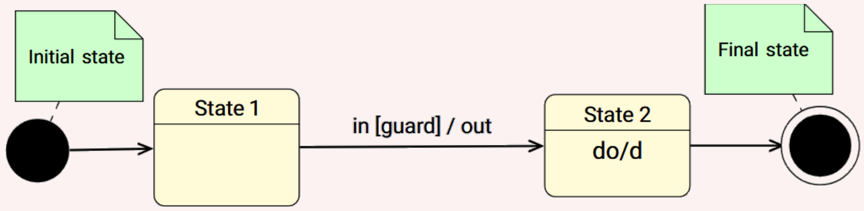
Special States



There are some special states:

- Initial State: Has a single transition to first entered state
  May be labeled by object creation event
- Final State: Indicates completion of scenario
- Terminal State: Completion and executing object destroyed

Transitions

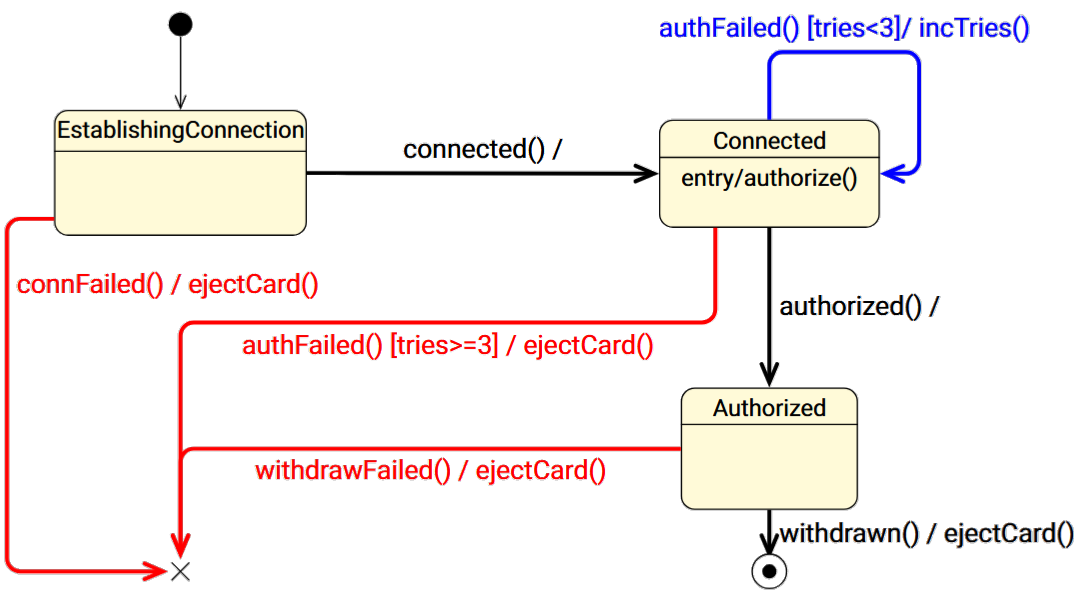


A transition label is usually in the format **input? [guard]? / output?**. Hereby all components are optional (therfore ?).

- Input (Trigger) events are observations:
  - call event (start of operation)
  - time event (e.g. time spent in a state)
  - change event (value of attribute has changed)
- Guard is a boolean expression (for example: if input == value)
- Output (action) is an operation

In general, the purpose of state machine diagrams are:

- Capturing action sequences of a use case
- Combine related use cases
- Clarify the states of an object
- Clarify protocols
- Validate the domain model
- Complete the domain model (properties, actions)
- Model **only** non-trivial behaviour



An example State Machine Diagram

5 Software Architecture

Software Architecture is concerned with how the software is designed and implemented. It is often a combination of design and implementation and is often defined in terms of dimensions.

Software Architecture is a fast evolving and ever changing field. Some practices of the past are nowadays discouraged, while some others, which were once discouraged, are now encouraged.

5.1 Architects and Developers Interoperability

Architects and Developers work together closely. They give each other feedback and adapt to each others works. This is done iteratively, which promotes the evolution of the software.

Architects

- Extract the architecture characteristcs from requirements analysis
- Choose set of styles for software system
- Create the component structure

Developers

- Design class structure for each component
- Design user interface
- Write and test source code

5.2 Dimensions

Architectural Characteristics

Concerned with how different characteristcs the software should fulfill.

Operational	Structural	Cross-Cutting
Availability	Extensibility	Accessibility
Scalability	Maintainability	Privacy
Performance	Leveragibility	Security
...	...	...

Architectural Style (Software System Structure)

Concerned with different layers of a software system are connected.

Presentation Layer
Business Layer
Persistence Layer
Database Layer

Decisions Concerning Architecture

Clarifies some questions before the architecture is implemented. For example:
- Which web framework?
- What are each layers responsibilities?
- How do layers communicate with each other?
- Which data formats should be used?
- ...

Design Principles

Concerned with how the architecture should be implemented on a technical level. For example:
- NoSQL Databases are preferred
- Immutable data structures are preferred
- Asynchronous messaging between services when possible
- Avoid usage of caching in clients
- ...

5.3 Architecture Characteristics

- Specifies non-domain design consideration: How to implement a given requirement
- Influences the structural design aspects: Requirements of specific architectural elements
- Critical that application performs as intended: Meets functional and non-functional requirements

Operational Characteristics

- **Availability:** When the system must be operational (specific times, continous, etc.)
- **Performance:** Response time, peak analysis, stress test, etc.
- **Scalability:** Functions with increased numbers of requests, users, etc.

Structural Characteristics

- Extensibility: Ability to add new features
- Maintainability: Ability to modify existing features
- Leveragibility: Ability to reuse existing features
- Localization: Support for different languages, currencies, units, etc.
- Configuration: Ability for user to configure the system to their needs via configuration interface

Cross-Cutting Characteristics

- Accessibility: Usability for a lot of people, especially people with disabilities
- Privacy: User data inaccessible to unauthorized parties
- Security: Encryption of database, network traffic, authentication, authorization, resilience to attacks, etc.

5.4 Architectural Styles

- Help specify fundamental structure of a software system
- Impacts appearance of concrete software architectures
- Defines global properties:
  - Interoperability of components
  - Boundaries of subsystems
  - etc.

A software system can have multiple architectural styles.

An architectural style does almost always bring trade-offs. Being aware of them is important to choose the right one for your needs.

5.5 Monolithic Architectural Styles

5.5.1 Layered Architectural Styles

Operational	Structural	Cross-Cutting
Availability	Extensibility	Accessibility
Scalability	Maintainability	Privacy
Performance	Leveragibility	Security
...	...	...

Trade-Offs

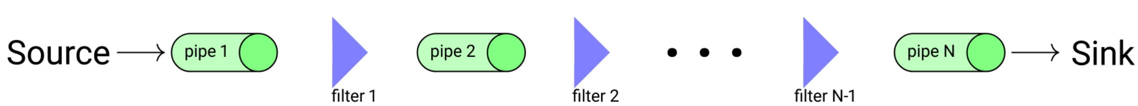
+ Simplicity
+ Cost
+ Reliability

- Elasticity and scalability
- Performance (No parallelization)
- Availability (Long startup time)

Layered Architectures in general:

- Technologically partitioned, not domain partitioned
- Works well for small to medium sized systems
- Serves as a starting point for larger systems, can be changed later
- Problem: Often created unconciously as it reflects the organisational structure of the company

5.5.2 Pipes and Filters



Pipes and Filters

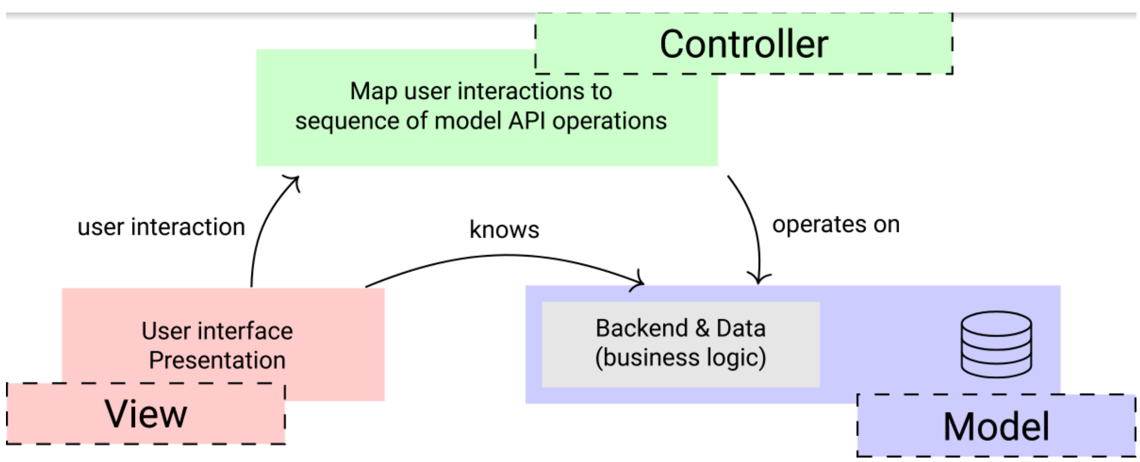
- Pipes:
  - Unidirectional, point-to-point channels from data source to target
  - Allow any data format, although smaller data formats are preferred for better performance
- Filters:
  - Self contained and independent from other filters
  - Stateless, does not depend on past data and realizes exactly one task

Different Types of Filters

- Producer:
  - Producer: Source of data
  - Transformer:
    1. Receives data from input channel
    2. Performs operations on the data
    3. Forwards result via output channel
  - Consumer: Sink of data, Display output, written file, database, etc.
  - Tester:
    1. Receives data from input channel
    2. Tests whether data satisfies certain conditions
    3. Redirects data accordingly to different output channels

Example of Pipes and Filters: Image Processing

5.5.3 Model-View-Controller (MVC)



- Seperates system into three parts:
- Model:
  - Business logic and data storage
  - Independent of input behaviour and output representation
- View:
  - Presentation of the model data to the user
    * Data obtained from model
    * Often more than one view
- Controller:
  - Translates user interactions to operations on the model
  - Each view has its own controller
  - All interactions with the model are done via the controller
- Controller and View are directly coupled with the model
- The model is independent of the controller and view

Change Propagation Mechanism

- Ensures consistency between the UI and the model
- Views register themselves at a model (Controllers too if behaviour depends on model state)
- Model notifies registered objects of changes

Trade-Offs

- Updates all registered objects, even if they are not affected
- Increase in complexity due to seperate view and controller components without gaining much flexibility
- High dependency between view and controller

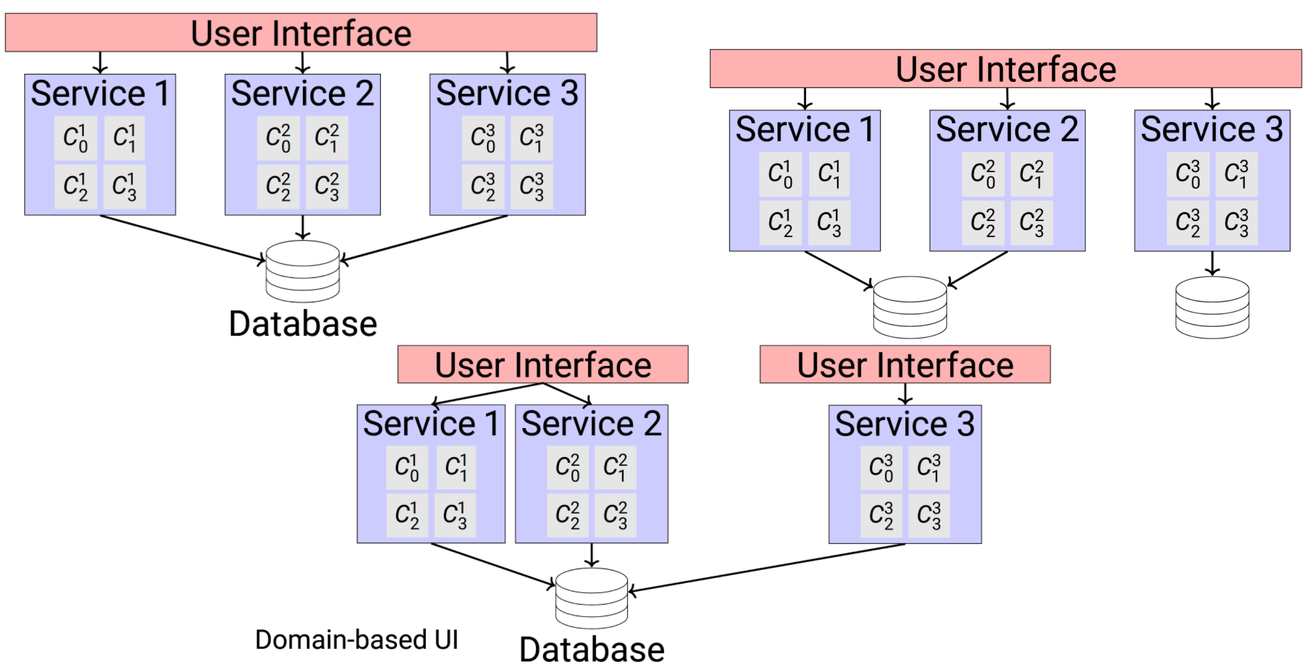
MVC should be used when:

- Application is interactive and:
- Number and kind of views are not fixed or unknown
- Display and application behaviour must reflect changed immediately
- Changing and porting the ui should not affect the applications core

5.6 Distributed Architectural Styles

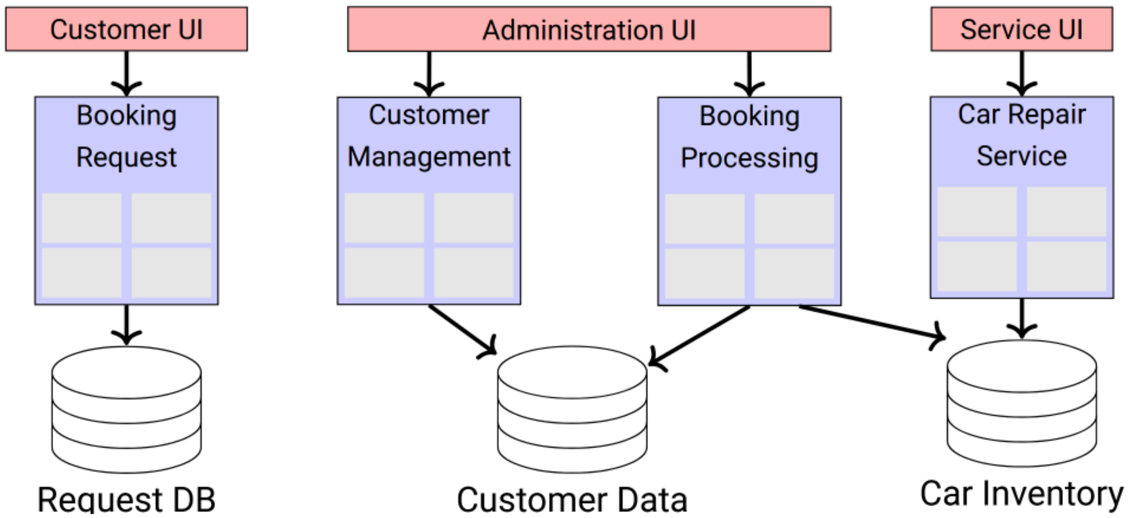
5.6.1 Service-Based

Service-Based Architecture is a style of software architecture where the application is decomposed into service components that are independent of each other and are able to be provided separately. They can take different forms:



In these examples access permissions must be defined. For example: One service might have read-only access to a database while another has write access.

More concrete this can look like this:



Choosing the right architecture style is tricky as it depends on many factors, such as the domain, the characteristcs, data architecture, organizational factors and the devolpment factors as a whole.

Therefore communication and documentation is very important.

6 Design Principles

A good software should have certain standards it should fulfill. The quality of a software can be measured.

Quality Assurance

To create software we should first create a quality assurance plan, which is integrated into the software development process.

- Constantly asses design quality (quantitative and qualitative characteristics)
- Apply time-tested desing principles where applicable
- Use tools and design techniques that help to achieve quality
- Use design patterns (designs used across multiple projects, problems, etc.)
- Systematically verify correctness & performance
- Validate fulfillment of Requirements

6.1 Good Software

What is good software?

- Internal quality factors
  - Perceivable only by computer professionals
  - White box view
  - Code, databases, documentation, etc.
- External quality factors
  - Perceived by the customer / user
  - Depend on internal quality factors
  - Black box view
  - UI, speed, ease of use, etc.

Internal quality factors

- **Modularity:** How easy is it to modify the software?
- **Comprehensibility:** Is the software easy to understand?
- **Cohesion:** Is it clear what each component does?
- **Concision:** How concise is the code? (Code duplication, overly lengthy code)
- **Correctness:** Does the software work as intended?

External quality factors

- **Validity:** Does the software work as according to the requirements?
  - Needs precise requirements
  - Depends on correct design
  - Often conditional / codependant on correctness of internal quality factors
- **Robustness:** How well does the software handle abnormal conditions and errors?
- **Extensibility:** Can the software be extended to fulfill new requirements?
  - Architecture must be flexible / extensible
  - Often dependant on modularity of internal quality factors
- **Reusability:** How well can the software be reused in different contexts?
- **Compatibility:** How well does the software work with other software?
- **Portability:** How well does the software work on different platforms (hardware & software)?
- **Efficiency:** How fast and resource-efficient is the software?
  - Often depends on algorithms and data structures
  - Should be implemented for the common case
- **Usability:** How easy is it to use the software?
- **Functionality:** How far does the software usage extend?
  - Features should be consistent in usage and design

Overall the most important quality factors of good software are:

- **Maintainability:** Can be adjusted over time to new requirements
- **Efficiency:** Is reasonably fast and resource-efficient
- **Usability:** Is relatively easy to use and responsive
- **Dependability:** Does not cause physical or economical damage in case of system failure

6.2 Measuring Software Quality

In general, there are no universal way to measure quality as different software varies wildly. Oftentimes some metrics need to be negelected in favor of others, depending on the context (Usability over Security, Modularity over Concision, etc.).

What can be done is to define standards / heuristics to indicate quality of code. These are usually called **software metrics** or **code metrics**.

Software Metrics Pros

- Can be computed mechanically
- Can be used to indicate bad design

Software Metrics Cons

- Does not take semantics into account
- False sense of correctness

Code Metrics

- **Fan-in / Fan-out:**
  - Fan-in: Number of functions that call a specific function
  - Fan-out: Number of functions that are called by a specific function
- **Length of code:** Number of lines of code, indicates complexity
- **Cyclomatic complexity:** Number of decision points in code (control-flow graph)
- **Depth of conditional nesting:** Number of nested conditional statements, hard to understand, hard to test
- **Weighted methods per class:** How many functions are in a class, functions are weighted dependend on size / complexity
- **Depth of Inheritance:** Number of levels of inheritance, hard to understand

6.2.1 Control-Flow Graph (CFG)

A CFG represents all execution sequences of a program.

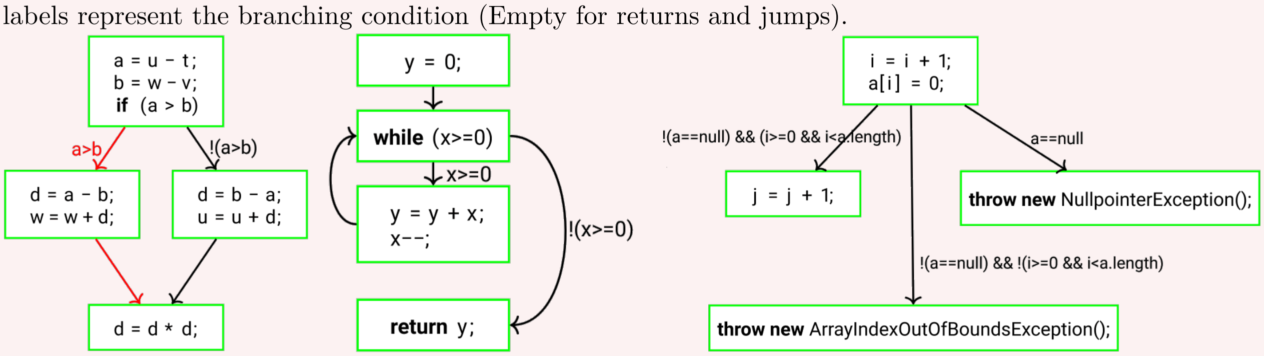
Basic Blocks in a CFG

A basic block is a maximal sequence of non-branching statements or instructions that are always executed together.
The execution of a basic block starts with the first statement, only the final statement can be a jump (branch or return).

```
a = u - t;
b = w - v;
if (a > b) {
  d = a - b;
  w = w + d;
} else {
  d = b - a;
  u = u + d;
}
d = d * d;
```

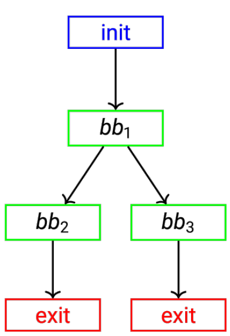
Control-Flow Graph

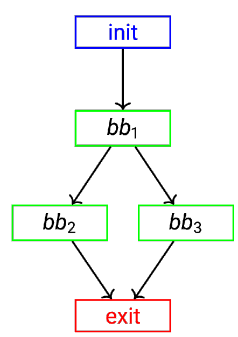
A Control-Flow Graph CFG(P) = (N, E, Label) of program P is a labeled directed graph, with nodes n ∈ N which represent the basic blocks of P and edges e ∈ E which represent the control flow of P. Hereby each edge e = (n_i, lb, n_j) ∈ E with n_i, n_j ∈ N and lb ∈ Label is a transition from n_i to n_j with label lb. The labels represent the branching condition (Empty for returns and jumps).



Red represents one possible execution sequence

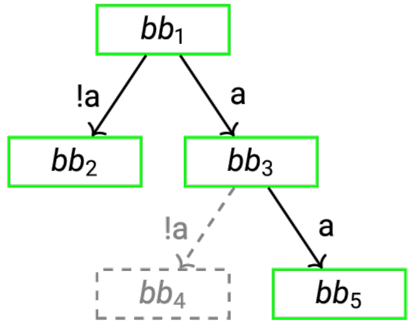
In many cases the definition of explicit initial and exit nodes is important.





As there might be different end states sometimes the different exit nodes are important. For other reasons, like code metric computation, sometimes the exit nodes should be handled as a single node.

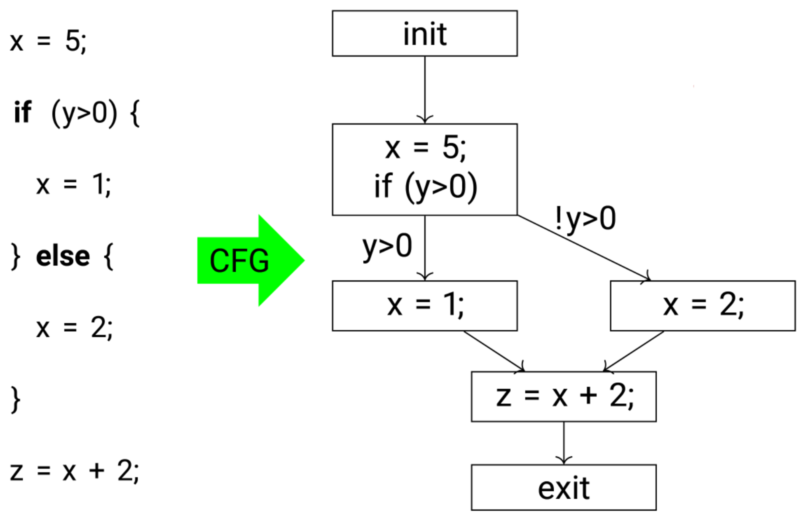
Sometimes code can result in unreachable nodes in one specific execution sequence. In this case the unreachable node / inactive edge should not be displayed.



Assumes bb_3 does not modify a

6.2.2 Code Metric: Cyclomatic Complexity

The cyclomatic complexity defines the number of independent paths through the code. It requires a CFG with a single exit node.



Hereby the number of independent paths through the code is 2. This can be calculated like follows:

$$C := E - N + 2P$$
with E edges, N nodes and P connected components

So in this example it would be:

$$C = 6 - 6 + 2 * 1 = 2$$

In general, a cyclomatic complexit C of 10 or higher is considered to be complex - Rethinking design and coding might be beneficial.

6.2.3 Code Metric: Class and Interface Coupling

A class or interface C is coupled to a class or interface D if C requires D directly or indirectly. Hereby a class or interface that depends on 2 classes is considered looser than one that depends on 8 classes.

Common types of Coupling in OOP

- Attribute referal: X has an attribute of Y
  - `class X { Y y; }`
- Expression referal: X contains an expression of Y
  - `class X { Object o = new Y(); }`
  - `class X { void m() { …if (o instanceof Y) …} }`
- Method referal: X calls a method of Y
  - `class X { void m() { …y.m(); …} }` (Object method)
  - `class X { void m() { …Y.m(); …} }` (Static method)
- Method-Instance referal: X has a method that references an instance of Y
  - `class X { X(Y y) { …} }` (Parameter)
  - `class X { Y f() { …} }` (Return type)
  - `class X { void m() { Y y = …; } }` (Local variable)
  - `class X { void m() { Object o = new Y(); } }` (Local Expression)
- Inheritance: X inherits from Y
  - `class X extends Y { …}` (Extension)
  - `class X implements Y { …}` (Implementation)

Design Principles: Tight and Loose Coupling

Tight coupling is generally undesirable

- Changes in couples classes may cause undesired changes in other classes
- Tight coupling makes it hard to understand a class in isolation
- Tight coupling makes it hard to reuse a class
- Tight coupling results in low modularity

Generic classes with high reusability must have very loose coupling. However, very loose coupling or no coupling in general is also undesirable.

- Goes aginst OOP principles
- Loose coupling may require a huge number of active objects, decreasing performance

However, the tightness of couplings needs to be determined on a case-by-case basis.

6.2.4 Code Metric: Cohesion

Cohesion measures the strength of the relation among elements of a class. All operations and data in a class should "naturally" belong to the concept modelled by the class.

Types of Cohesion

Ordered from undesirable to ideal:

- **Coincidental**: No meaningful relation among elements
- **Temporal**: Class elements are executed together
- **Sequential**: Result of one method in input of another
- **Communicational**: All functions access the same input or output
- **Functional**: All elements contribute to achieve a single, well-defined purpose: **Ideal**

Lack of Cohesion of Methods (LCOM)

Cohesion is often evaluated by the **Lack of Cohesion of Methods (LCOM)** metric. Hereby, a class C is defined as a set of instance fields F and methods M (excluding constructors). This set is then used to define an undirected Graph G(M,E) with vertices M and Edges E.

$$E = \{\langle m_1, m_2 \rangle \in M \times M | \exists f \in F : m_1 \text{ and } m_2 \text{ access } f, m_1 \neq m_2\}$$

The LCOM(C) is then defined as the number of **connected components (CC)** of G(M,E). This means that for a class C with |M| = n, the LCOM(C) $\in [0, n]$. Therefore a high LCOM value indicates low cohesion. An issue with this metric is that its definition needs to be refined for special methods, like the constructors, hashCode, toString etc. methods. While these are technically part of the class, they are considered standard components of each class and therefore do not count towards the LCOM.

Low Cohesion is generally undesirable. As the classes can be hard to comprehend, reuse and maintain. Low cohesion also often indicates too-corse abstraction, meaning classes take responsibility for too many tasks, that should be handled by other classes.

As a rule of thumb: A class with high cohesion can often be described in a single sentence.

6.3 Responsibility-Driven Design

Describes a systematic approach to think about the design of software objects and components in terms of **responsibilities**, **roles** and **collaborations**.

6.3.1 Responsibility Types

Responsibilities in general are related to

- **Obligations** of an object
- **Behaviour** of an object

in terms of its role in the software design.

Type: Doing Responsibility

Doing Responsibilities describe responsibilities that are related to performing a task.

- Doing it (perform a calculation, create an object)
- Initiate action in other objects
- Control and coordinate activities in other objects

For Example: A Bill object is responsible for calculating the total price.

Type: Knowing Responsibility

Knowing Responsibilities describe Responsibilities that are related to **knowing** and **providing** information to other objects.

- Knowing private, encapsulated data
- Knowing related objects

For Example: A Car is responsible for knowing the driven distance.

6.3.2 Responsibilities and Methods

A responsibility is **NOT** the same as a method. A responsibility can be modelled with a method, but in many cases its better to model it with multiple. Therefore a method is part of a responsibility and can be the whole responsibility, but a responsibility can also be split into multiple methods.

There is no real method to determining how to split responsibilities. It is often very circumstancial and needs to be adjusted to the needs at hand.

6.4 More Design Principles

Ideally a system design should follow the **Single Responsibility Principle (SRP)**. This means that a class / object should have a responsibility which is its primary reason to change. Therefore one responsiblity per class is the ideal.

6.4.1 Collaboration of Multiple Classes

Oftentimes a oolicy requires collaboration of multiple classes. This makes it hard to choose which of these classes should handle the responsibility. There is also no easy answer for this. Sometimes there is a clear class that makes access to the others easier. To figure this out multiple drafts may be needed.

6.4.2 Delegation vs. Inheritance

To figure out where to place specific responsibilities the concepts of delegation and inheritance are useful.

Delegation

Delegates responsibilities to other objects:

- Get objects from other class with the needed functionality
- Use the object to fulfil only the needed functionality
- Inheritance hierarchy remains unchanged

Inheritance

Inherit responsibilities from baseclass:

- Violates SRP
- All subclasses of the current class are forced to also inherit the responsibility
- Not required functionality from the baseclass is also inherited
- Inheritance hierarchy is changed →harder to maintain and understand

Most of the time delegation is preferred. As the design is more understandable and maintainable. It also is evaluated at runtime rather than compiletime. Inheritance should only be used when the responsiblity extends the functionality **organically**.

6.5 Encapsulation

Interface

An Interface declares the method signatures and public constants of its implementing classes. They provide independence of functionality from implementation.

Always Program to Interfaces

- Fields, return types, method parameters etc. should be declared with interface type
- Public methods should not expose implementation details
- Fields in implementing classes should be private. Retrieval & modification of information via getters and setters should be used.

This process has the advantages:

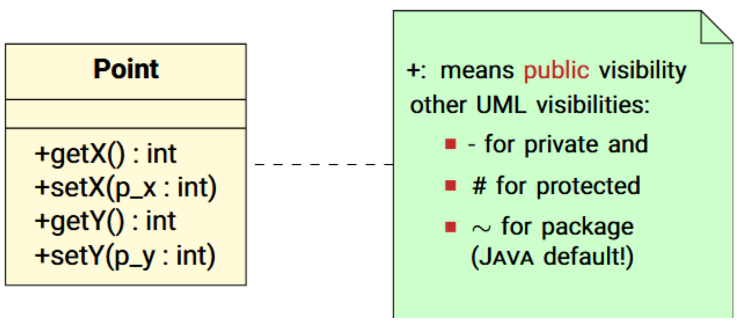
- Avoids unjustified assumptions about implementation
- Interfaces are more stable than implementations
- To change the implementation its sufficient to exchange the constructor
  - Can factor out implementation-independent code into abstract classes
  - Reduces coupling

6.5.1 Field Access

Instance fields should **NEVER** be public. This ensures that information is hidden and adheares to the "Always Program to Interfaces" rule.

It also makes sure that there is a distinction between implementation-specific data and public data. Client should not be able to access implementation-specific data.

6.5.2 Accessor Methods Should not Expose More Than Necessary



UML Access Modifiers

Accessors shouldn't always be used. Oftentimes the use of accessors is unnecessary and only creates more coupling. In many cases it's better to outsource the responsibility of the accessor / the surrounding responsibility to another class instead.

There are some reasons to use accessors though:

- Responsible for aspects of the UI (Data important for visualization)
- Class using accessors implements a policy
- Class is a static container

6.6 Design Knowledge: The God Class Problem

God Class

A class that contains most of the system logic:
- Promotes poorly distributed responsibilities
- Not object oriented design

To avoid this you can use the following criteria:

- Avoid classes with unclear responsibilities
  - Classes that fail the SRP principle
  - Solution: Split class and relocate the responsibilities
- Avoid classes with low cohesion and non-communicating classes
  - Classes with methods operating on a small subsets of its fields but not with other objects
  - Solution: Split class and relocate the responsibilities
- Avoid classes with public field accessors:
  - Public field accessors can indicate wrongly located responsibilities
  - Solution: Redesign interface and relocate the responsibilities

In general, when designing a system, one should try to model the real world. This however can produce complex systems. Therefore often it is advised to put the real world model aside and design the system as according to the design principles.

7 Design Techniques

7.1 Documentation

Readability

- Documentation: Explain design, architecture, etc.
- Source Code Comments:
  - API documentation: Packages, interfaces, classes, methods, etc.
  - Line comments: Small clarifications of the code
- Source Code

Code Specific Aspects

- Coding Style Conventions: Naming, formatting, bracket placement, etc.
- Restrictions on Usage of Language Features
- Naming of Identifiers: Coherent and descriptive naming
- Meaningful Comments

7.1.1 Comments

Comments can be seperated in two different categories:

API Comments

- **Intent:** Document API usage
- **Audience:** Third-Party developers using the code as a framework or library

Used to describe in detail when method can be called and how it behaves

- Restrictions on parameters besides types: non-negative, not null, etc.
- Side effects on object state
- Thrown exceptions (when and why)
- What is returned

Statement Level Comments (Line Comments)

- **Intent:** Describe implementation details, code structure
- **Audience:** Developers on the same code base

Should be concise and used sparingly

- Clear, well-written code is mostly self explanatory
- Might indicate poorly written or overly complex code →**Refactor**

7.2 Refactoring

Refactoring describes the process of restructuring code without changing its external behaviour.

This can be done in order to:

- Prepare for addition of new features (just preparing, not actually adding)
  - Reduce code duplication
  - Avoid nested conditionals
- Improve design (Cohesion, etc.)

- Increase comprehensibility
  - Choose a clear naming convention
  - Choose meaningful names
  - Simplify convoluted logic
- Improve maintainability

7.2.1 Refactoring Techniques

Extract Method

A method should be extracted if one method does multiple things or similar functionality is realized within one method or across multiple methods.
The method extraction should be done like this:

1. Create a new method (target)
2. Copy extracted code to target
3. Identify local variables used in extracted code
   - 3.1. Variables only used in extracted code can be declared as local variables
   - 3.2. Extracted code modifies exactly one outside variable: Check if target method can be query
   - 3.3. If more than one outside variable is modified: Extract Method is not possible
   - 3.4. Pass undeclared variables in target as method parameters
4. Replace extracted code with call to target

This unfortunately sometimes reduces cohesion as target method does not necessarily accesses all the same variables as the source method. This indicates that the class has to many responsibilities. Indicates that usage of **Move Method** is needed.

Move Method should be only be used if:

- Source method does **not** use features of the source class
- Source method does **not** override a method or is overriden by a subclass

Move Method

1. Create new method in target class
2. Copy source code to target method and adjust it to work there
3. Determine how to reference target object from source
4. Turn source method into delegating method
5. If not needed: Remove source method and accessors in target class

If a class has **two or more independent responsibilities** and no other class can handle these responsibilities, the **Extract Class** technique should be used.

Extract Class

1. Create new class
2. Link new class from old class (e.g. attributes), may require a new accessor
3. Apply refactoring **Move Method** and **Move Field**
4. Review and reduce class interfaces (unnecessary accessors, etc.)

8 Design Patterns

A design pattern describes

- a problem that reoccurs regularly in the domain
- the core of a solution to this problem, such that one can reuse the solution in other contexts (might not be exactly the same)

Template Method Pattern

Implements an algorithm in a manner that allows adaptation to different implementations.

- Define skeleton algorithm, but defer implementation of some concrete parts to subclasses
- Often used in frameworks and APIs

Some benefits:

- Separation of variant and invariant parts
- Avoidance of unnecessary code duplication
- Control of subclass extensions

Design Pattern Template

1. - Name: Short mnemonic to extend the design vocabulary
   - Intent: Goals and reasons why to use the pattern

---

2. - Motivation: States problem situation
   - Applicability: Context in which the pattern can be used

---

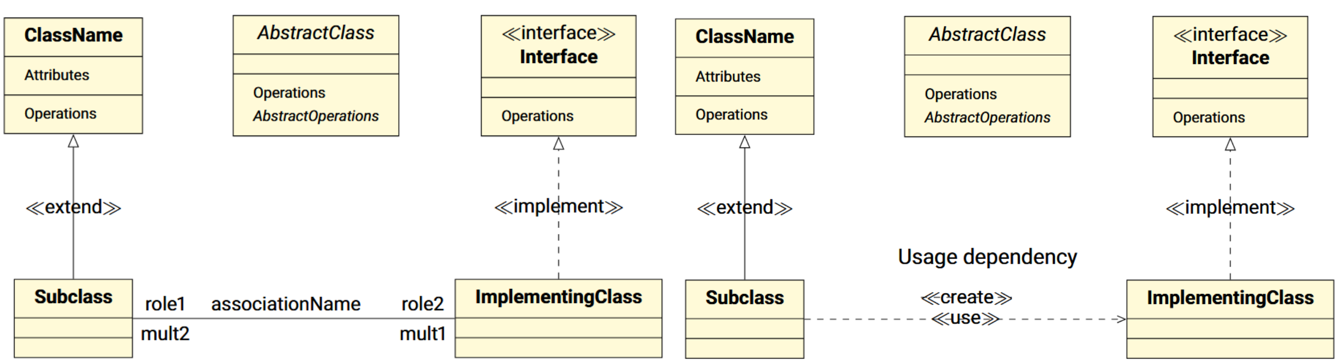
3. - Structure: Static structure of the pattern (UML Class diagram)
   - Participants: Which classes are involved
   - Collaborations: How the classes interact
   - Implementation: How to implement the pattern

---

4. - Consequences: Gains and trade-offs

---

5. - Known Uses: Examples of using the pattern
   - Related Patterns: References to and discussion of related patterns



8.1 Subject - Observer Pattern

The subject - observer pattern utilizes Object-Oriented Analysis and Design (OOAD). OOAD organizes a problem by breaking it down into managable subtasks →Objects are responsible for subtasks. Collaboration might be required for complex problems.

Advantages:

- Easy to understand, implement, understand, maintain and reuse objects - Divide and Conquer approach possible
- Flexible combinations for different problems possible

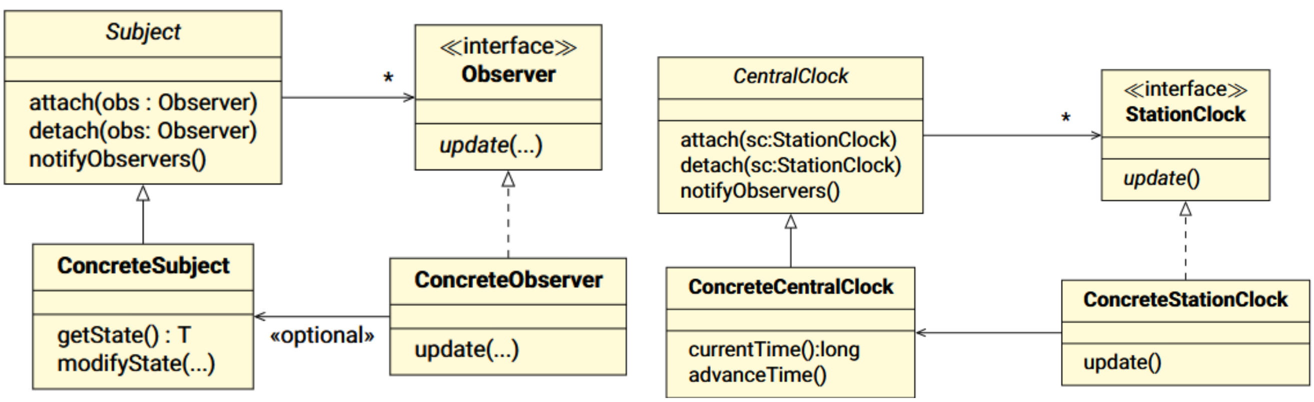
Disadvantages:

- Behaviour is distributed across objects →Lack of clarity of design
- Any state change might affect others

Hereby the communication of the objects is desirable to be done with low coupling, as change in one object should not necessarily result in change in another. This way the objects can be reused in different context.

8.1.1 Subject - Observer Pattern: Pull Mode

Subject is an object with state changes that are independent of the observers. The observers get information from the subject and react accordingly.



- Subject should not need to know details about its observers
- Identity and number of observers not predetermined or fixed
- New observers can be added dynamically

Observer Pattern - Advantages

- Abstract coupling between **subject** and **observer**
- Support for broadcast communication
  - Sender does not know the type of the receiver

Observer Pattern - Disadvantages

- Danger of update cascades from observers to their dependant objects
- Update sent to all observers - might not be interesting to all observers
- No change details - Observers need to find out what has changed
- Uniform interface for all observer updates - Subject cannot send optional parameters to observers

Example: Car Inventory

```java
class CarInventoryView implements Observer {
    @Override
    public void update(Subject subject) {
        fillTable((CarInventory) subject);
    }
}
```

8.1.2 Subject - Observer Pattern: Push Mode

Instead of just passing the current state of the subject to every observer additionally also provides the change in the state. Observers can then handle their actions according to the change and do not necessarily need to access the state of the subject

Example: Car Inventory

```java
class CarInventoryView implements Observer {
    @Override
    public void update(Subject subject, Change change) {
        if(change.getKind() == Change.CarDeletion) {
            deleteRowForCar(change.getCar());
        } else(change.getKind() == Change.CarAddition) {
            ...
        }
    }
}
```

8.1.3 Subject - Observer Pattern: Interest Mode

When registering observer to subject, specify what kind of updates the observer is interested in. This way the observer doesn't have ot check whether the update is of interest or not, and can skip right to handling it. This, of course, means that the subject has to handle more as it can't just send out a pure update notification.

For example: Java Action Listeners, Mouse Listeners etc.

8.2 Factory Method

In a framework that needs to be able to present multi-format documents like PDF, HTML, Word etc. the framework should be able to do so, while offering common functionality. (open, close, save, print, etc.)

This can essentially be done by letting these different classes implement a common interface, but leave instantation of a specific class to the factory.

Example: Factory Method

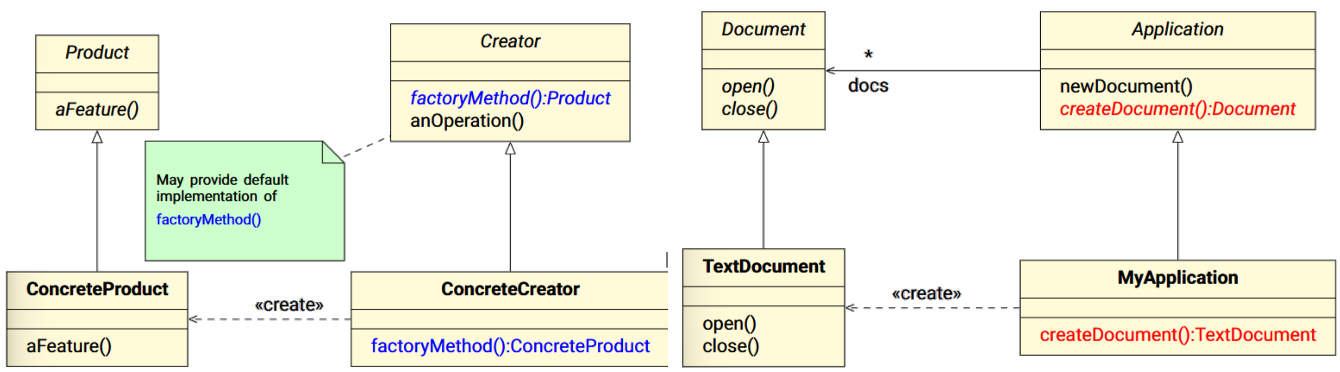
Abstract:

```java
public abstract class Document {
    public void open();
    public void close();
    ...
}

public abstract class Application {
    private List<Document> docs = new ArrayList
    <>();

    public void newDocument() {
        Document doc = createDocument();
        docs.add(doc);
        doc.open();
    }
    public abstract Document createDocument();
}
```

Concrete:

```java
public class TextDocument extends Document {
    ...
}

public class MyApplication extends Application {
    public Document createDocument() {
        return new TextDocument();
    }
}
```

The creator can also be implemented concretely, providing a reasonable default implementation.



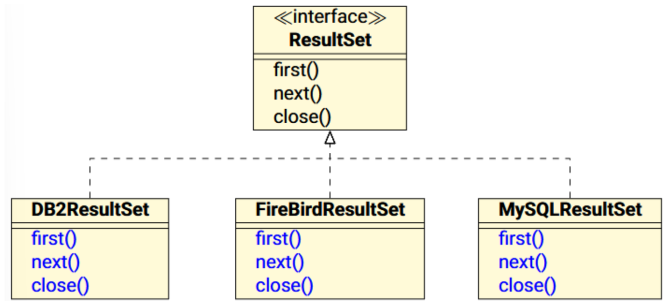
Factory Method - Consequences

- Client application code only knows product interface →Works for any ConcreteProduct
- Product provides a hook for subclasses →Extended version of object via hook

8.3 Abstract Factory

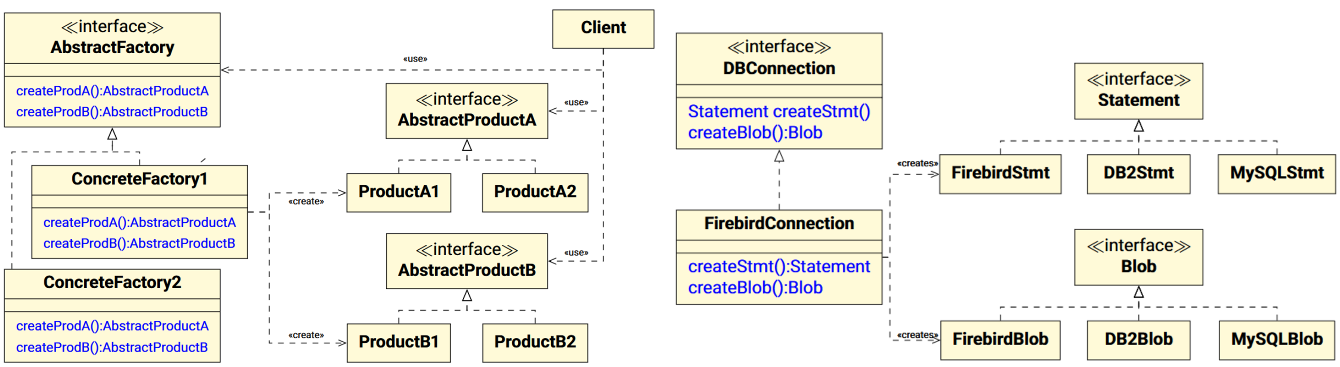
Abstract Factories provide a system to create a family of related classes that implement common interfaces.

For example, taking a search engine that needs to search for a query in multiple databases of different formats:



This creates a problem of that this usually needs to still have concrete implementations of the subclasses as the class hierarchy might differ and the interacting code might be different. Additionally, this method requires that the format is specified at the time of creation, which is undesirable.

The solution is to have an interface, the abstract factory, that is used to instantiate the concrete factories, which in turn can then be used to create the concrete classes.



Abstract Factory - Advantages

- Abstracts concrete products - Client is unaware of the concrete product they're using
- Changing formats/families is easy
- Consistency among products

Abstract Factory - Disadvantages

- Adding unforseen additional products is expensive - Abstract family and all its subclasses need to be changed
- Object creation followm non-standard pattern - Factory instead of constructor

8.4 Factory Method vs. Abstract Factory

	Factory Method	Abstract Factory
Product	Single	Family
Product declaration	Client	Startup
Product exchange	Concrete Product	Abstract Product
Creation	Local (Creator)	Selection of Factory

9 Verification

9.1 Introduction

9.1.1 Validation vs. Verification

Validation

Validation ensures that the software meets the costumers expectation.

Are we building the **right** system?

- Is the feature set as intended and complete?
- Does it fit into the organizations workflow?

Validation is reliant on correctly performed requirements analysis and thus cannot be done without involving the user / customer.

Verification

Verification ensures that the software is correct in respect to the system specification.

Is the system built **right**?

- Do the features work as specified?
- How does the system react to faults?

Verification is performed in solution space by the developer and thus doesn't necessarily involve the user.

9.1.2 Verification Techniques

Static

Static techniques do not require code execution.

- **Code Review:** Can be done at any stage of development
- **Static Checking:** Automated analysis of source code (Type checks, bug finders, etc.)
- **Formal Verification:** Ensures that a program satifies a formal specification

Dynamic

Dynamic techniques require code execution.

- **Testing:** Assert correct behavior for specific inputs
- **Runtime Monitoring:** Instrument program with safety assertions, whose violations are detected at runtime

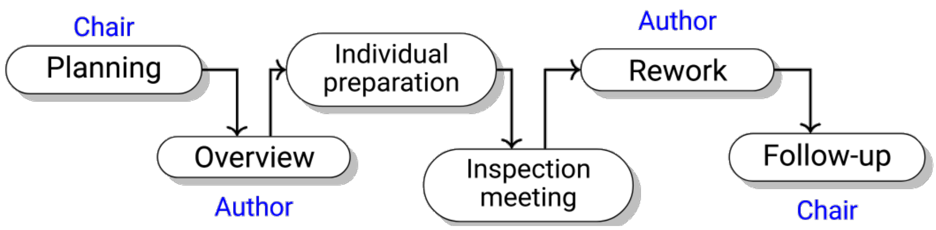
9.2 Code Review

A **Code Review** is an inspection process in which a team reviews project code, with the goal of identifying errors, bugs, deviations from conventions, etc.

9.2.1 Building Blocks

In a code review all team members take on specific roles:

- **Author / Owner:** Writing the code
- **Inspector / Reader:** Inspecting the code
- **Scribe:** Writing down comments, questions, requirements, etc.
- **Chair / Moderator:** Leads project and manages discussion and requirements



9.2.2 Check Lists

Code Reviews are often driven by a **Check List**, which is used to specify specific fault classes that should be checked. These are often individual to the project environment based on language, coding standards, etc.

Check List Example

Data Fault	Are all variables initialized?
I/O Fault	Are all input variables used?
...	...

Advantages

- Empirical evidence that they work and save cost
- Distribute knowledge of the codebase to all team members
- Find defects before they might cause problems in tests
- Improves code quality
- Code does not need to be executed

Disadvantages

- Team members might feel criticized
- Time pressure might become worse because of the review
- Only works when properly conducted

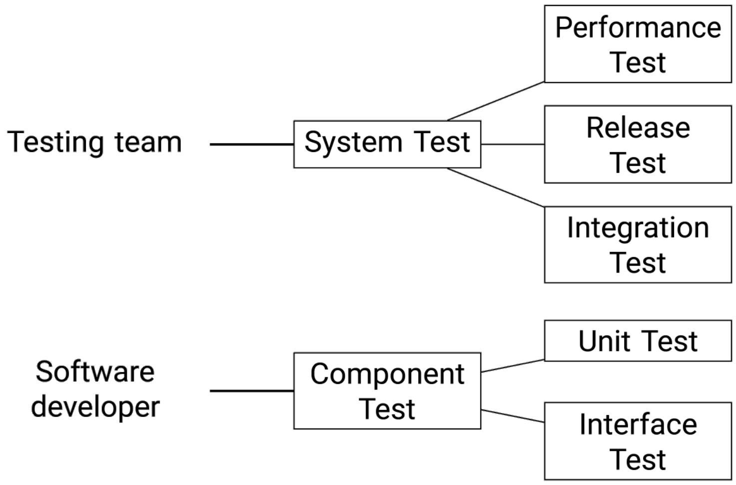
9.3 Software Testing

Testing can never show all faults, but it can reduce them. (Falsifiability)

9.3.1 Constituents

- **System or Implementation under test (SUT, IUT)**
- **Test Inputs**
- **Test Harness:** Runtime environment, preparation and execution, clean up
- **Test Verdict:** Given by **Test Oracle**

9.3.2 Test Levels



9.3.3 Test Plan

A test plan contains detailed descriptions of the testing process. It is a document intended to be read by humans.

- **Work Plan:** Phases, schedules, etc.
- **Testing Procedures**
- Explanation of the **test design**
- **Test documentation just as important as code documentation**

9.3.4 Test Design

1. Identify and analyze responsibilities of the IUT
   - **Pre- and Postconditions** in use cases
   - **Minimal and Success Guarantees** in use cases
   - Analyze distribution of responsibilities
2. Add test cases based on
   - Use Case-, Design- and Code Analysis
   - Suspicions, minimal success guarantees
   - General Heuristics (Domain / Expression boundaries)
3. Determine for each test case how verdict is reached: Provide expected results, programmed or human test oracle

9.3.5 Test Automation

Testing is very expensive; Typically about 15 - 20% of development costs are allocated towards testing.

These can be reduced automating tests:

1. **Running the tests:** Nightly, after each change, etc.
2. **Generating test cases:**
   (a) Code-driven
   (b) Model-driven
   (c) Data-driven
3. **Generating test verdict:**

(a) Test oracle as a small program, added to the test harness

(b) Oracle generated automatically from formal specification

Tasks of a Test Automation System

1. Set up test environment
   - Start servers, establish connection, register services
2. Start IUT
3. Bring IUT to required pretest state
   - Load required data, create required objects
4. Set tests inputs
5. Evaluate output and test verdict
6. Clean up environment
   - Delete files, stop services, reset data

Not everything is possible to automate, some manual tasks remain (Test inputs, test oracle)

9.3.6 Test Goal

Establish **sufficient** trust, that system is operational by exercising the interfaces between its parts.

9.3.7 Test Input

Test (Data) Point

A **Test Point** is a specific value for

- a test case input
- a state variable

The test point is selected from a domain. A **domain** is the set of values that input or state variables can assume.

Heuristics for Test Point Selection

- **Equivalence Classes:** Singular test point for expected equivalent outcome
- **Boundary Values:** Min/Max of ordered domain, pivot for comparison
- **Special values:** Null, other values with specific semantics

9.3.8 Other Definitions

Test Case

A **Test Case** consists of

- Pretest State of IUT
- Test Point / Conditions: test input
- Expected result

A collection of test cases is called a **Test Suite**.

Test Run

A **Test Run** is the execution of a test suite on a single IUT. A test whose results are equal to the expected results gets the **verdict Pass**, otherwise a **Fail**.

Test Driver & Test Harness

A **Test Driver** is a class or program that applies test cases to an IUT.
A **Test Harness** is a system of test drivers and other components that support test execution.

Fault & Failure

A **Fault** is missing or incorrect code.
A **Failure** is teh manifested inability of a system to perform a required function within specified limits (Time, memory, etc.)

9.3.9 JUnit Test Framework

Example of JUnit Test

```java
public class AccountTest {
    Account account;
    @BeforeEach // Runs before each test (Pretest State)
    public void setUp() {
        account = new Account(100);
    }
    @AfterEach // Runs after each test (Cleanup, Posttest State)
    public void tearDown() {
        account = null;
    }
    @Test
    public void successfulWithdrawTest() {
        assertTrue(account.withdraw(50)); // Delivers a passing verdict if value is true
    }
    @Test
    public void failedWithdrawTest() {
        assertFalse(account.withdraw(150)); // Delivers a passing verdict if value is false
    }
}
```

9.4 Test Coverage

9.4.1 When to Stop Testing?

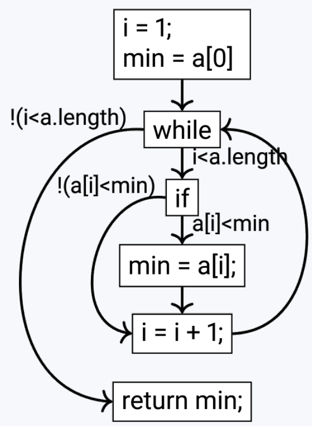
Structural Criteria (Code Structure)

Based on the **Control Flow Graph (CFG)** of a program.

Statement Coverage (SC)	Each **statement executed** at least once
Basic Block Coverage (BBC)	Each **basic block** executed at least once (implies **SC**)
Branch Coverage (BC)	Each **outgoing edge** from a node in the CFG is executed at least once (implies **BBC**)
Path Coverage (PC)	Each **path** through the CFG is executed at least once (implies **BC**), unachievable in practice, # of paths grows exponentially

Example of Structural Coverage

```
i = 1;
min = a[0];
While i < n.length do
    If a[i] < min then
        min = a[i];
    i++;
return min;
```



Definitions

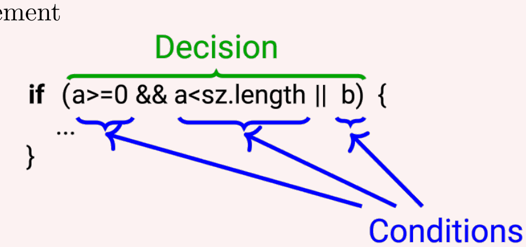
Execution Path	A complete path through the CFG
Feasible	A path that can actually be taken

Logic-Based Criteria (Logical Case Distinctions)

Condition Coverage (CC)	Each **condition** evaluated as true and as false in at least one test run
Decision Coverage (DC)	Each **decision** (guard) evaluated as true and as false in at least one test run
Modified Condition Decision Coverage (MCDC)	Combines **CC** and **DC** & independence test
Multiple-Condition Coverage (MCC)	All true-false combinations of conditions are tested in at least one test run

Condition vs. Decision

Condition	A condition is a Boolean Expression & cannot be divided into sub-expressions
Decision	A decision is a Boolean Expression constituting the guard of a conditional or loop statement



Modified Condition Decision Coverage (MCDC)

For one occurrence of condition c **inside** decision d, MCDC is satisfied if:

1. Evaluates d at least twice
   - once where c is true
   - once where c is false
2. d evaluates differently in both cases
3. all other conditions in d evaluate identically in both cases **or** are not evaluated at all in at least one case.

9.5 Test Automation & Tool Support

9.5.1 Automated Test Case Generation (ATCG)

As writing test cases with good coverage is very time consuming, automated test case generation (ATCG) is often used instead.

Fundamental Approaches

White Box: (**Code Based**) Code of IUT is analyzed to achieve coverage.
- **Syntactic Approach:** Scan for conditions, evaluation. Achieves logic-based criteria.
- **Symbolic Execution:** Unwinding CFG with symbolic values. Achieves structural coverage criteria.
  → **Under-Approximation:** Unreached code.

Black Box: Analysis of input data or model of IUT.

9.5.2 Further Test Automation

Test Coverage Recording

1. Instrument IUT
2. Run test suite, collect information during test runs
3. Analyze and display achieved coverage statistics

Does not help with writing tests, but helps with knowing when to stop.

Test Oracle Synthesis

- **Human Oracle:** Time consuming and error-prone
- **Verdict as Code (assert):** Need expertise, hard to maintain

→ Write test oracle in **formal specification language**, sythesize code from specification

9.5.3 Automatic Static Verification Techniques

Static Checking

Typically based on CFG of IUT and constraint solving
- Runtime exceptions, liveness, information flow
- Fully automated
- Over-approximation, possibly many false positives
- Scales reasonably

Bug Finding

Based on pattern matching and heuristics

- Fast, scales well, handles full Java
- Over- and Under approximation (false positives and incomplete)

SpotBugs is a static analysis tool utilizing bug patterns to find bugs in Java programs. It works at a byte code level.

It sources its bug patterns from **complex language features**, **misunderstood API methods or invariants** and **typos and wrong usage of operators**

9.5.4 Formal Verification

Formal Approaches

- Mathematical Foundation (logic and set theory)
- Sound relative to formal model (strong guarantee)
- Not necessarily complete (not all true properties can be proven)

Often checked for by external programs that use either **Model Checking** or **Deductive Verification** by feeding it the source code and the formal specifications.

9.5.5 Design-by-Contract

Design-by-Contract Example in JML

```
/*@ private normal_behavior
    // What needs to be true for this method to work correctly
    requires 0 <= low <= up <= a.length;
    requires (\forall int x,y;
            0 <= x < y < a.length; a[x] <= a[y]);
    // What this method guarantees to be true after execution
    ensures \result == -1 || low <= \result < up;
    ensures (\exists int idx;
            low <= idx < up; a[idx] == v) ?
        \result >= low && a[\result] == v
        : \result == -1;
    // What the method may modify
    assignable \nothing;
    // Specifies the termination metric
    measured_by up - low;
@*/
private int binSearch(int v, int low, int up) {...}
```

Design-by-Contract

Formal specification:

- Pre- and Postconditions, side effects for each method
- Class and loop invariants

Verification tool proves that each method

- fulfills its contract in all possible runs
- preserves loop and object invariants

9.5.6 Java Modelling Language (JML)

JML is a **contract-based specification language** tailored to java

General JML Philosophy

Integrate
- JML specification
- Java implementation

within a single language

JML is not external to Java, but integrated

9.5.7 Deductive Verification

Working Principle: Path Exploration

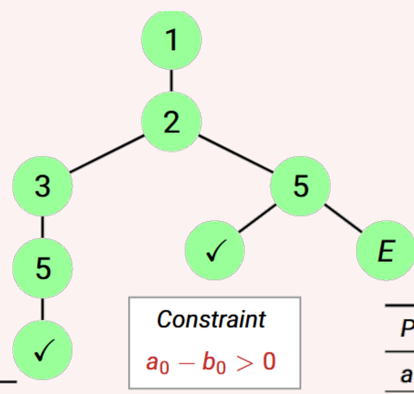
Symbolic execution explores all paths in CFG of straight-line programs (no jumps, no loops, no method calls)
- Finite number of paths
- Uses symbolic values to represent all inputs
- Loops approximated by all invariants

Scalability of Deductive Verification

Approximate effect of a method call with a contract
- During symbolic execution, replace called method with contract
- Substitution and first-order deduction instead of path exploration

Symbolic Execution Example

```
a = a - b;
if (a < 0) {
    a = -a;
}
r = b / a;
```



Constraint: $-a_0 + b_0 < 0$

PV	Value
a	$-a_0 + b_0$
b	b_0
r	$b_0/(-a_0 + b_0)$

Constraint: $a_0 - b_0 > 0$

PV	Value
a	$a_0 - b_0$
b	b_0
r	$b_0/(a_0 - b_0)$

PV	Value
a	$a_0 - b_0$
b	b_0
r	r_0

Constraint: $a_0 - b_0 = 0$