Software Engineering Lecture Notes WS 24/25

Igor Dimitrov

2024-10-14

Table of contents

Preface

Lecture notes for the course “Software Engineering” at Heidelberg University WS24/25.

1 Introduction

1.1 Chapters

1. Introduction
2. Communication in a Project link
3. Requirements Engineering (Communication with the Users) link
4. Design (Communication with Developers) link
5. Quality Management link
6. Evolution link
7. SWE-Process (Summary and Project Management) link

2 Introduction to SE

Core goals of software engineering:

- **quality**: software fulfills the requirements $\Leftarrow$ **this**
- **users**: software is useful for the users
- **developers**: it is easy to develop, maintain and modify the software
- **cost / time**: software can be implemented within the given cost & time constraints $\Rightarrow$ project management

Main task categories of software engineering:

- Development:
  - Determining the software context
  - requirements engineering
  - architecture
  - detailed design
  - implementation
  - configuration management
- Quality management
  - product: testing, inspection, metrics
  - process: measurements, improvements
- Evolution:
  - maintenance
  - further development / modification
  - re-engineering / change management
- Project management:
  - team
  - costs
  - time / deadlines
  - risks
  - contractor / client

3 Communication in a Project

Following topics relate to and determine communication within a project:

1. Number of participants and their roles in the project
2. Type of the contractual relationship
3. Team Organization: The way developers communicate within the project
4. Collaborative Coding

3.0.1 Projects and Participants

terms relating to project and process:

- process:
- project
- process model:

characteristics of a project:

- limited time
- creator
- purpose
- client
- results
- means and tools
- organization and planning

participants:

- client
- user
- manufacturer

3.0.2 Contractual Relationship

project types:

- EP (Entwicklungsprojekt) -> development project
- AP (Auftragsprojekt) -> Commissioned Project
- EDP (EDV-Projekt, EDV = Elektronische Datenverarbeitung) -> IT Project
- SP (Systemprojekt) -> System Project

3.0.3 Team Organization

types of team organization:

- single person
- 2-person team
- anarchic team
- democratic team
- hierarchical team
- chief-programmer team
- agile team

types of secondary organization:

- functional
- project-based
- matrix

3.0.4 Collaborative Coding

- Pair programming
- Distributed development

4 Requirements Engineering

requirements engineering can be understood as corresponding to the communication with the users / clients. Deals with the following topics

1. Introduction to communication with users/clients.
    1. Clients and Requirements
    2. Description and specification of requirements
    3. Defining Requirements Engineering
    4. Outcome of Requirements Engineering
    5. Benefit of specification
    6. Complexities of RE
2. Usage modelling / description
    1. Introductory Example
    2. Introduction
    3. Tasks, Roles, Persona
    4. Domain Data
    5. Functions, UI-Structure
    6. GUI
3. Documentation Quality
    1. Introduction and Templates
    2. Characteristics and style guide
4. Usability
5. Quality assurance with the client
    1. acceptance test
    2. usability test
6. Quality requirements
    1. Motivation
    2. Quality attributes
    3. QR-description
    4. QR-test
7. Use-cases (not relevant to the exam)

    1. Description of Uses Cases
    2. Use for system testing

8. RE procedure

    1. Introduction
    2. Gathering requirements
    3. Specificying requirements

4.0.1 Communication with Users / Clients

- requirements engineering:
    - collection of the requirements from the client
    - specification / formalization of the requirements
    - testing / examination of the requirements
    - management of the requirements
- requirements engineering result:
    - document:
        * description using system functions
    - prototype
- Advantages and Uses of Specification:
- Disadvantages of a missing specification
- Difficulties related to RE:

4.0.2 Usage Modelling

- Task-oriented Requirements Engineering:
    - Task level: tasks, roles, persona
    - Domain level: subtasks (as-is & to-be), domain data
    - interaction level: system functions, ui-structure
    - system level: gui, screen-structure (virtual window)

Roles, Persona, Tasks

Roles, Persona

UDC (User-centered design)

Roles and Persona

- User role:
- User profile:

How are personae described:

- name
- biographic facts: age, gender, etc
- knowledge and attitude with respect to the tasks and technology:
- needs: main use-cases in which the user wants to apply the software -> Tasks
- frustrations:
- ideal features:

Tasks

How tasks are described:

- Goals
- Decisions
- Causes
- Priority
- Execution profile (frequency, continuity, complexity)
- Precondition
- Info-in (input)
- Info-out (output)
- resources (means, participating roles)

sub-tasks:

Persona-task correspondence:

- needs <=> sub-tasks => combination of system-functions
- frustrations <=> problems in sub-tasks
- ideal features <=> ideas, system functions

Domain Data

Domain data explain the terms used in the task descriptions.

Goal: describe/model the entities of real world, including their relationships / associations to each other, in order to understand the tasks.

- domain data describe **entities** that are relevant within the context of the sofwtare. They correspond to the terms used in the task description => independent from the software.
- described with simple **class diagrams** => **domain data diagram** (no operations, no aggregation, no inheritence, only associations)
- sometimes **glossary** is sufficient.

sometimes not sufficient, because there additional data necessary on the UI level => **interaction data**. (not relevant in our MMAP case)

Functions & UI-Structure (Interaction Level)

Goal: implementation of the User/Machine boundary with respect to the task descriptions.

consists of 2 parts:

1. System functions: which functions are provided by the system?
2. UI-Structure:

    - in which context can the user call which functions,
    - which data is available/visible in those contexts?
    - how are functionalities divided among the sub-parts?

⚠ Warning

UI-Structure is **not** GUI-structure, i.e. the concrete layout is not yet determined.

System Functions

how is a system function described:

- name: nomenclature verb-object, describe what will be achieved on the user data (e.g. unlinkMovie => movie will be unlinked)
- input:
    - context (i.e. workspace) in which the SF accessible

- concrete input: data that is gathered during the interaction with the user. Such data is not yet determined when the SF is first called on the GUI, but first provided by the user during the interaction.
- output: changes of the UI
- description:
- exceptions: cancel/discard by the user
- rules:
- quality requirements:
- precondition:
- postcondition:

UI-Structure

Consists of

- workspaces:
  - bundle related system functions and data similar to a class (but only from an abstract user point of view. Actual structure in code can be completely different)
  - only system functions that can be triggered by the user are listed
- navigation links between workspaces

UI-structure abstracts from a concrete screen-layout. Logical represents a logical view of the interaction structure.

> ❗ Important
>
> UI-structure is created concurrently with the System functions, because their close interrelation. (Workspaces contain System functions and data)

Design of System Functions and UI-Structure

- how is the system function specification template filled in?
- how are ui-structure decision made concurrently to SF specifications?
- initial test considerations?

There's still lots of wiggle room for specific design decisions.

TODO:

GUI

GUI= concrete layout of the UI:

- data representation
- function representation
- window structure
- dialogue description (how user controls the execution of functions)

Design principles:

- law of proximity
- law of closure
- law of good continuation
- law of similarity

Types of functions:

- semantic functions: actual data manipulation in the system, e.g. save, open, calculate something etc
- help functions / auxillary functions: data manipulation on the screen, e.g. text size
- search
- navigation

How functions are represented:

- buttons
- checkbox
- menu-selection
- shortcut
- icon
- scroll-bar
- drag-and-drop

Views (Window / Screen): concrete version of Workspaces:

- how is data represented
- how are functions made available / represented

Documentation of a view: Virtual Window (Mock-up)

4.0.3 Documentation Quality

following topics relate to it:

- document templates
- features and sytle-guide

many documents exist:

- Software context-design:
  - problem-description
  - contract
  - acceptance test plan
- Requirements Engineering:
  - Client-requirements
  - Usage test plan
  - Software specification
  - System test plan
- architectural specification:
  - architectural specification / definition
  - sub-system specification
- Detailed design:
  - component specification
  - integration test plan
- Implementation:
  - Code
  - Component test plan

Communication happens via / is facilitated by documents

4.0.4 Usability

The degree to which a product can be used efficiently and adequately for specific tasks / goals, in a specific usage context:

- effectivity
- efficiency
- satisfaction

7 interaction (dialogue) principles :

- Task appropriateness: no unnecessary repetitive actions must be taken bythe user
- Self-descriptiveness: user knows what the actions achieve, what's the input, output and the systems response.
- Controllability: user can decide the order of the operations, can terminate the operation and resume at any given moment without loss.
- Expectation-conformity: system is consistent, confirms to users real-life experience and other software conventions
- Error tolerance: even at the hands of an incorrect input the intended result can be achieved with minimum correction effort.
- User engagement: system must be appealing and inviting and provide users a positive experience.
- Learnability: users are supported and guided during the learning of the software

4.0.5 Quality Assurance with the Clients

- verification: whether the software being built confirms to the specifications derived from the requirements and whether the requierements documents meet quality standards
- validation: whether the specifications actually correspond to clients requirements ⇒:
  - usability test
  - acceptance test

Acceptance Test

- Tests whether the client accepts the system ⇒ validation.
- Uses **system tests** provided by the client
- includes particularly also **usability tests**.
- tests are carried out in the production environment (live system where the software is fully deployed and used by the clients)

Usability Test

A representative group of users from the target demographic of the product take part in the test to determine to what extent the usability criteria is met.

Consists of:

- users
- observers

how:

- develop a test plan:
  - goal
  - description of the problem
  - description of the users
  - test design
  - list of tasks
  - test environment
  - evaluation criteria
- prepare the test:
  - recruiting a representative group
  - recruiting observers
  - distribution and review of the test plan with the participants
  - prepare scenarios
  - prepare surveys: backgroung, pretest, posttest
  - prepare test env
- execute the test:
- analyse the results

4.0.6 Quality requirements

Motivation

- functional requirements $\Rightarrow$ what?
  - tasks
  - system functions
  - gui
- non-functional requirements (NFR) $\Rightarrow$ how good?
  - Quality of the system
- QR = quality requirements: describe product considerations
- QA = quality attributes

Quality Attributes

QA describe various types of software quality.

Categories:

- Quality in Use: Direct validation with the users
  - Beneficialness: how well are users supported
    - usability
    - accessibility
    - suitability
  - freedom from risk: general impact
  - acceptability (how good is the system from the point of view of the user)
    - support experience
    - trustworthiness
    - compliance (regulations, laws)
- Software Product Quality: verification (internally), validation (externally), continuously during the whole development process
  - functional stability
    - functional completeness
    - functional correctness
    - functional appropriateness (helpful)
  - performance efficiency
    - time behavior
    - resource utilization
    - capacity
  - compatibility
    - co-existence with other software
    - interoperability
  - interaction capability (usability)
    - recognizability
    - learnability
    - operability
    - user error protection
    - user engagement
    - inclusivity
    - user assistance
  - reliability
    - faultlessness
    - availability
    - fault tolerance
    - recoverability
  - security
    - confidentiality

        - integrity
        - non-repudiation
        - accountability
        - authenticity
    - maintainability
        - modularity
        - reusability
        - analysability
        - modifiability
        - testability
    - flexibility
        - adaptability
        - scalability
        - installability
        - replaceability
    - safety
        - operational constraint
        - risk identification
        - fail safe
        - hazard warning
        - safe integration

Describing QR

- QR should be made as measurable and as concrete as possible.
- QR are defined in parallel with the FR with as much detail as possible, for all levels:
    - Tasks
    - Domain data
    - Functions
    - GUI

Testing QR

- Different QA categories require different testing methods.
- Usually only possible at the level of system testing, especially acceptance testing.
- Usually conducted by acting out scenarios

4.0.7 Use Cases

So far we described the requirements via:

- (Sub)-Tasks
- Domain Data
- Roles / Persona
- System functions
- Interaction data
- UI Structure

At the system level this is further refined via:

- concrete views
- interaction models

But this is only an indirect description of users interaction with the system.

Use Cases describe the specifics of users interactions with the system, specifically the particular execution sequence of system function to complete a certain (sub-)task. Thus, they are derived from **Tasks**

use cases vs TORE:

- use cases cover only a subset of TORE. Particularly
  - No domain data model
  - No UI-Structure (which SF & Data are available in which view, how do you navigate between views)
- UC integrate SF within the execution flow, in TORE SF are contained in the subtask description (although without the execution flow)

Description of a UC

Example: manage movie or performer ratings in MMAPP.

- Name: Manage Rating
- Actor: Person
- Goal: to rate movies or performers
- Precondition: at least one movie exists. Depending on the current state wither A1 or A2 is the starting point:
  - Flow of events:
    - Actor: A1) W0.1 Movie Master View ... System: System shows the
    - ...

    - Rules:
    - Quality requirements:
    - Data, System Functions:
    - Postcondition:

Use of System Tests

Scenario vs UC:

- Scenario: a specific interaction flow
- UC: an abstract description of a set of scenarios

A choice of a typical set of scenarios can be used as a template when creating a UC. On the other hand a UC can be validated by a subsequent creation of a set of test scenarios.

What is a typical flow?

- normal flow
- coverage of branches / exceptions
- dealing with large input (stress test)
- dealing with complex execution flows (many functions are called)

4.0.8 RE Procedure - Method of RE

So far we only concerned ourselves with describing the requirements. Here we explain how to gather and manage them.

Introduction

Stakeholder (client) requirements: the requirements stipulated by the client to achieve a certain goal or solve a certain problem.

1. requirements are gathered
2. requirements are formulated / formalized in the specification
3. specifications are validated and verified
4. specifications are translated into a design
5. the design is implemented
6. the implementation is tested
7. the implementation is deployed to the clients

Gathering Requirements

what information should be gathered:

- previous method to solve the problem
- problems related to the previous method
- goals for the new system / method to solve the problem
- success criteria
- rough system architecture (how many components, distribution)
- realistic solutions
- consequences and risks

how it is gathered:

- observation
- user surveying
- ethnographic studies
- use cases
- workshops
- object oriented analysis
- entity relationship diagrams
- conceptual modelling
- data flow diagram
- formal specification
- prototypes
- observations of other products
- literature research
- studying standard software

requirements can also be gathered via user feedback

Specifying Requirements

contents and types of requirement specification:

- functional requirements: how should the software support the users?
- non-functional requirements:
  - quality requirements: how well should the software support the users?
  - edge conditions
- rationales: reasoning behind the decisions?

requirement specification is textual as well as schematic (UML, ER etc)

There are two types of requirements:

- client (user) requirements
- system (developer) requirements: requirements from the point of view of the developer with more technical detail.

Usually two different documents are created.

TORE again:

1. task level:
    - tasks
    - roles, persona
2. domain level:
    - sub-tasks
    - domain data
3. interaction level:
    - system functions
    - interaction
    - ui-structure
4. system-level: GUI
    - screen-structure (virtual window)

above levels can be categorized as follows:

- 1, 2: client requirements
- 3, 4: system requirements

Two different specification documents:

- lastenheft (client specification document)
- pflichtenheft (developer specification document)

5 Design

Design can be understood as communication with and within the developers.

Goal: The software system can be further developed efficiently, and is easy to understand for new developers.

Belongs to the development and specifically detailed design category of software development.

Deals with the following topics:

1. Introduction to Modelling
2. Class diagrams
3. Interaction diagrams
4. State Diagrams
    1. UML State diagrams
    2. Dialog models
5. Class design with OOAD
    1. OOAD introduction
    2. OOAD: Analysis Class diagram
    3. OOAD: Design Class Diagram
6. Design Patterns
    1. Introduction
    2. Creational patterns
    3. Structural patterns
    4. Behavioral patterns
7. Rationales (Communication of decisions)
8. Summary of modelling techniques

5.0.1 Introduction to Modelling

A given team (approx 7 people) developes an understanding of the structure of the Software System, albeit without using code, since code is too detailed and is not conducive to understanding the overall structure of the software.

Instead an appropriate abstraction level is necessary $\Rightarrow$ modelling languages.

Model: Abstraction of a system, expressed in a formal language / notation where irrelevant details are omitted.

Characteristics of models:

- mapping
- incompleteness
- pragmatic (a model is created with a certain purpose / goal)

Characteristics of a formal notation:

- syntax
- semantics
- pragmatics
  - analysis techniques: type checking, consistency checks
  - simulation techniques
  - transformation techniques (refactoring, (algebraic) simplification)
  - generation techniques

A notation is a 'theory' that enables to reason about and manipulate representations of objects from the domain.

5.0.2 4 + 1 Architectural View Model

4 different viewpoints representing different stakeholders:

1. **Logical View** $\rightarrow$ **End-users**: functionality that the system provides to the end-users / clients.

   UML diagrams:

   - class diagrams
   - state diagrams

2. **Process View**: dynamic aspects of the system, description of the system processes, how they communicate, run-time behavior of the system: concurrency, distribution, performance, scalability, etc.

   UML diagrams:

   - sequence diagram
   - communication diagram
   - activity diagram

3. **Development (Implementation) View → Programmers**: system from programmers' perspective, concerned with software management.

   UML diagrams:

   - package diagram
   - component diagram

4. **Physical View → System-engineer**: topology of the software components on the physical layer as well as the physical connection between these components.

   UML diagrams:

   - deployment diagram

- (+1): **Scenarios / Use Case View**:
  - Architecture is described using a small set of use cases or scenarios
  - Scenarios are Sequences of interactions between objects and between processes.
  - used to identify architectural elements and to illustrate and validate the architectural design.
  - used as a starting point for tests of an architecture prototype

5.0.3 UML

UML diagram categories:

- Structure diagrams → statics of the system:
  - **class**
  - **object**
  - component
  - deployment
  - package
  - composite structure
- Behavior diagrams → dynamics of the system
  - interaction:
    - **sequence**
    - communication
    - interaction overview
    - timing
  - Use Case
  - **Dialgue Model / Activity**
  - **State**

In the lecture especially:

- class diagrams
- object diagrams
- interaction diagrams
- state diagrams
- Dialogue Model / Activity Diagrams

Structure Diagrams

- Design:
  - Class Diagram (analysis and design) (logical view)
  - Object diagram (special cases) (logical view)
  - package diagram (bundles of classes) (development view)
- Architecture:
  - compositional structure diagram
  - logical component diagram (internal and external view)
  - distribution diagram (over the physical components)

Class Diagrams

important components of class diagrams:

- classes (objects)
- associations between classes:
  - aggregation
  - komposition
- attributes
- operations
- generalization / specialization relationship (inheritance)
- interfaces: a view on a class (a contract)

Behavior Diagrams

- Flows:
  - Use case diagram (overview of the use case)
  - Dialogue model / activity diagram (sequences of activities)
  - state diagram (sequences of states)

- interaction:
    - sequence diagram (sequence of messages)
    - communication diagram (focus on a component)
    - time diagram (communication between automata)
    - interaction overview diagram (interaction of multiple interactions)

State Diagrams

UML extension of traditional state diagram aims to overcome some of the limitations FSMs. Thus UML state machines is an extension to the traditional mathematical FSMs.

specific to UML:

- hierarchically nested states
- orthogonal regions
- extended notion of actions
- characteristics of both Mealy and Moore machines:
    - Mealy: actions that depend both on the state and the triggering event
    - Moore: entry and exit action (only associated with states, not transitions)

Many software systems are **event-driven** (also called reactive):

- mouse click
- button press
- time tick
- arrival of a data packet
- ...

Software systems react to events by taking **actions** and changing to another state ⇒ **state transition**. Advantages of using an FSM model to underlie the code:

- reduce the number of execution paths through the code
- simplify the conditions tested at each branching point
- simplify switching between different modes of execution.

Basic UML State Diagrams

- state: rounded rectangles labeled with state names
- transitions: arrows labeled with the triggering **events** optionally followed by list of **actions**
- event / trigger: the thing that causes the state transition
- action: the optional thing carried out by the system as a response to the event

- initial transition: originates from the solid circle (the default initial state when the system first begins - entry point) every diagram should have it. initial transitions can have associated actions

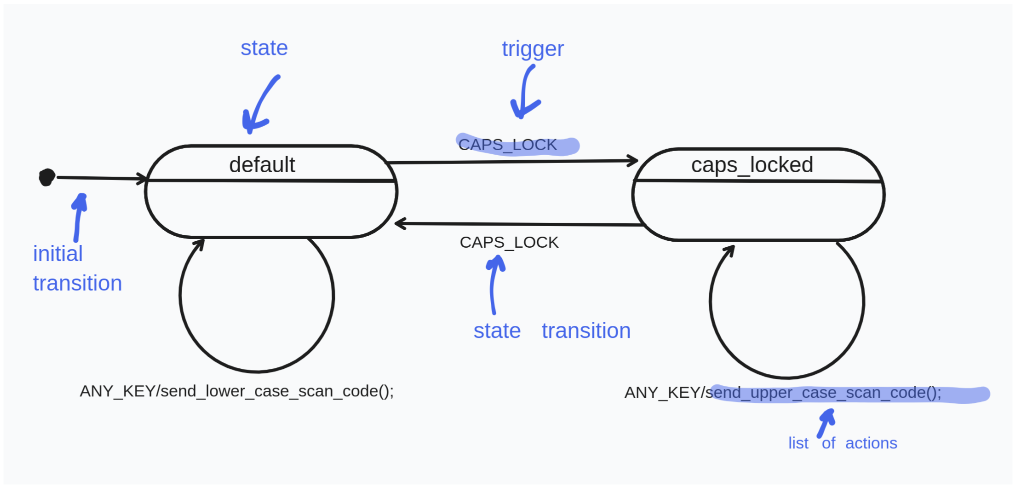


Figure 5.1: basic uml state diagram

Elements of UML state machines:

- **event / trigger**: something that happens that effects the system.
  - can have associated parameters
  - an event instance can have a long life-cycle
- **state**: Governs the reaction of the state machine to events.
  - a state can abstract away all possible (but irrelevant) event sequences and only capture the relevant ones.
  - in the context of software systems a state is a single variable that can have only a limited number of values, usually an provided by an enum type ⇒ the value of the state variables fully defines thet current state of the system at any given time
- **extended state**: However interpreting the whole state of machines can become impractical very quickly (state explosion) ⇒ in UML state is spilt up into:
  - enumerable state variable. Corresponds to the qualitative aspect of the whole state
  - all other variables which (**extended state**). Corresponds to the quantitative aspect of the whole state. Quantitative aspects do not lead to a state change. (See below for more explanation)
- **guard condition**: in extended UML a transition can have a guard ⇒ transitions only fires if the guard is evaluated to true, related to extended states, simplifies number of states.

- **actions and transitions**: response of the system to an **even/trigger**.
  - **action**:
    - changing a variable
    - performing I/O
    - invoking a function
    - generating another event instance
    - changing to another state
    - ...
  - **state transition**: switching from one state to another state
- **run-to-completion execution model (RTC)**: processing of each event must carried out to completion before the next event can be carried out.
  - incoming event can not interrupt this and are instead stored in an event queue
  - This avoids internal concurrency issues within a single state machine.
  - During even processing the system is unresponsive / unobservable.
  - Advantage: simplicity
  - Disadvantage: responsiveness of a state machines is determined by its longest RTC step.
- **hierarchically nested states**: repeating transitions common to a group of states can be factored out as a super (outer) state.
  - the complexity of the state machine doesn't explode (see calculator example below for more details)
  - semantics:
    - if a system is in a substate surrounded by a superstate, it is automatically also in that superstate, i.e. the superstate is inherited.
    - the state machine will try to handle any event first in the context of the substate, but if it is not defined, it will delegated to the superstate.

- **orthogonal regions**: compatible and independent regions of states that happen simultaneously. (AND-decomposition). $\Rightarrow$ reduction of combinatorial explosion of sates.
- **entry and exit actions**: always automatically executed upon entry to state or an exit from a state. $\Rightarrow$ guaranteed *initialization* and *cleanup*.
- **internal transitions**: self transitions can be instead represented as internal transition (without executing the entry and exit actions)
- transition execution sequence:
- local versus external transitions:
- event deferral:

Extended State Example

Assume we want to introduce a limit to the number of times keys can be pressed on keyboard, from the beginning of its execution. In traditional way we would have to introduce 1000 states that would approach the final state incrementally (state explosion). Instead this **quantitative information** can be captured in an additional variable that does not effect the **qualitative state** of the system, and is managed via **guard conditions**:

basically, extended state == variable not represented as state, that interacts with guard conditions.

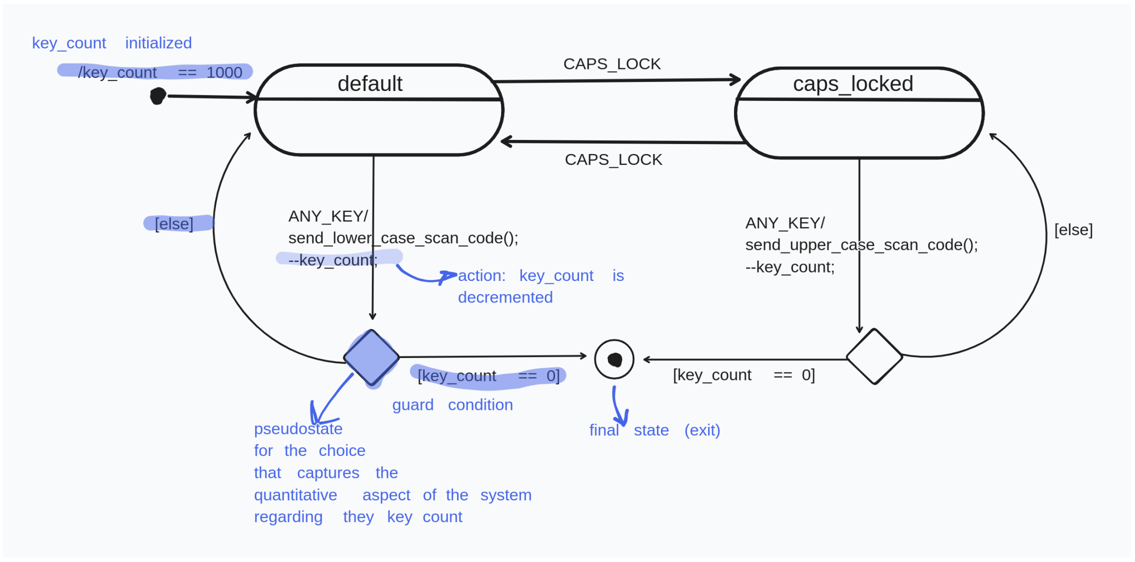


Figure 5.2: extended uml state machine exmaple

Hierarchical States Calculator Example

Modelling a simple pocket calculator with traditional FSM introduces many repetitive transitions:

- for every state the event 'C' changes the state to **operand1**
- for every state the event 'OFF' changes to the exit state.

Using the superstate **on**, this common behavior factored out and the total number of transitions is greatly reduced. Now when the system is, say, in the state **opEntered** and event 'C' takes place, it can't be handled by the inner state since it is not defined. Then, by the semantics of the UML state machines, this event is relegated to the outer state, where it is defined to transition to itself. From there it enters the inner states entry point, which changes the state to **operand1**.

Analogue for 'OFF'.

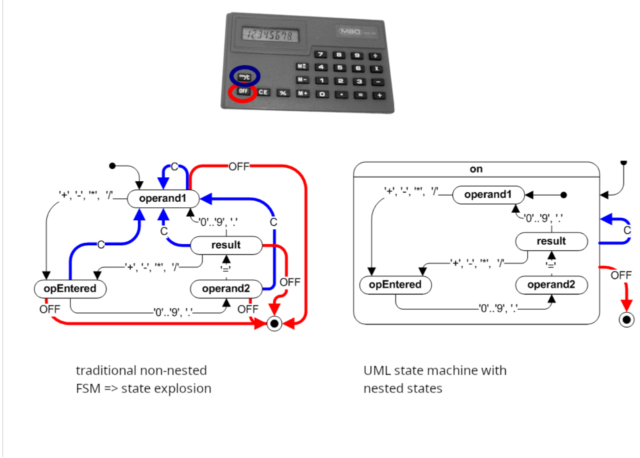


Figure 5.3: hierarchical states calculator example

Dialog Models / Activity

Dialog Model: state diagrams applied to UI.

- state ⇒ view
- transition ⇒ execution of a function

History Sates:

- H*: the previous innermost view / state
- H: the previous superstate.

Interaction Diagrams

interaction diagrams describe communication between various actors.

Various uml interaction diagrams:

- **sequence diagrams**
- communication diagrams
- time diagram
- interaction overview diagram

Sequence Diagram

In programs many objects collaborate with each other to carry out functionality. An objects collaborates with another by **sending messages** to it.

- sending message to an object ⇒ calling it's method: `myCDplayer.play("Song 1");`
- `myCDplayer` itself can collaborate with it's own instance (member) variables, and the input parameter object `"Song 1"`
- Collaboration between objects java implies having references to the same object from several locations in the program.

Also called event diagrams or event scenarios.

Show process interactions arranged in time **sequence**. Depicts the processes and objects involved and the sequence of messages exchanged to carry out functionality.

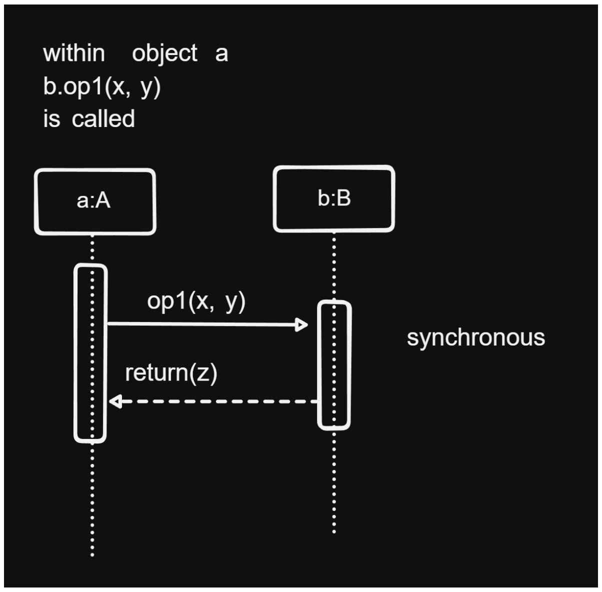


Figure 5.4: seq diag

rectangles represent the duration during which the object is active.

5.0.4 Class Design with OOAD

Steps of the software process:

1. analysis and specification: gathering and representing requirements
2. **design**: structure and architecture of the software is defined
3. coding: implementation of the end product in detail.

Design has the following goals:

1. partitioning the system into manageable units
2. laying out a structure: relationship between the units / parts that make up the whole system
3. hierarchical partitioning: abstraction that helps understand and maintain a large software system

results from requirements:

- Domain / Interaction data diagram: specification of the data in detail
- SFs or UCs: specification of functionality / how functionality is carried out
- Workspaces (UI-Structure): How and where is the data & SF represented and provided for the user, as well as the navigation between the workspaces.
- View (Virtual Window): the mockup of the GUI

Consideration when making design decisions:

- Data management classes: implementation of domain / interaction data diagram
  - which data which classes?
  - can some entities be directly taken as classes?
  - what operations are necessary?
  - how is data storage implemented?
- Classes for internal processing: Implementation of SFs
  - how are SFs implemented / distributed among operations of which classes? (note that SFs do not generally correspond to individual operations, but are combinations of multiple ops)
- UI classes: Implement the views (virtual windows) and the navigation from the UI-structure.

Design Principles: Cohesion and Coupling

Cohesion

A measure of or degree to how much the elements of a component belong together.

Goal: high cohesion, i.e. elements within a module or component belong together strongly, are highly interdependent $\Rightarrow$ good for maintenance.

We can't easily partition a highly cohesive module into subparts, where the subparts are independent of each other.

How to achieve high cohesion:

- principles of object orientation (data encapsulation)
- using appropriate design patterns for coupling and decoupling

Coupling

A measure of the degree of how strongly different components depend on one another.

Goal: low coupling, i.e. low interdependence between different components

Low coupling is good for performance (communication is simpler)

how to reduce coupling:

- interface coupling: information exchange takes place only via interfaces.
- components should call other components as little as possible
- data coupling should be avoided: no shared data among different components
- structure coupling should avoided: no shared structure among different components

Simple and complex operations:

- Simple operation: only direct access of attributes (class access it's own attributues)
- Complex operation: direct as well as indirect access of attributes (indirect access: class A access attributes of class B)
  - increase coupling
  - hinder cohesion

Complex operations should therefore be split up in smaller operations, or encapsulated in own class whose data they access.

OOAD

Object-oriented analysis & design ⇒ systematic development of the design model

the process of: requirements ⇒ class diagram:

- what decision should be made and how?
- what classes are needed, with what operations and attributes should the classes be equipped?

OOAD 2 step method:

1. Analysis class model: defines class structure based on the requirements
2. Design class model: concrete implementation of the analysis class model using frameworks, libraries and concrete classes, taking design goals, especially NFRs.

Analysis Class Model / Diagram

4 Steps:

1. Determine classes, attributes, and association from the requirements, with appropriate names.
2. Determine the operations of the classes: distribute basic operations and SFs as methods (operations) in classes, taking high cohesion and low coupling into account.
3. Determine complex associations and inheritance relationships
4. Consolidate the class diagram: possibly dissolve interaction / ui classes, consolidate associations.

3 types of classes:

1. entity class: describes objects with permanent existence
    - e.g.: film, actor.
2. control class: describes processes. SFs are initially modelled as such. Serve as placeholder for complex operations, ultimately are dissolved and distributed as methods among various appropriate classes
3. boundary / interaction / dialogue class: bundles data and operations that are provided on the UI.

Step 1

Goal: Determine / derive classes from the requirements

Rules:

1. Entity classes:
    - Entities from the data diagram become entity classes, e.g. film and actor.
    - associations from the data diagram become the associates between the classes
2. Control classes:
    - SF are initially modelled as such
    - control classes are linked to the entity classes, whose attributes they access
    - control classes are linked to other control classes, which they cooperate with.
3. Boundary classes:
    - Workspaces ⇒ boundary classes
    - are linked to the entity classes, that are displayed in the workspace
    - are linked to the control classes of the SFs that are provided on the UI interface.
    - navigation between workspaces ⇒ links / associations between boundary classes.

Step 2

Goal: Determine the operations of the classes, dissolve control classes by distributing them among the various classes as methods. (occasionally a control class can be retained as a concrete class)

Rules:

1. Dissolve control classes by distributing them among entity and dialogue classes:

    1. canonical solution: if control class operates only on the attributes of a single class, it becomes a method in that class.
    2. simple solution: if multiple classes are involved, but the input and output are associated to one class respectively, then the the control class can be … ?
    3. complex solution: if multiple classes are involved, and the output and input relate to multiple classes, then … ? $\Rightarrow$ sometimes objectification of the control class

2. Operations can be split up or united, depending on the situation.

    - an operation can cover multiple SFs (only for the canonical solution)
    - SFs can be implemented by multiple operations

Step 3

Goal: Use inheritance and model complex associations

Step 4

Consolidate the model and possibly dissolve the interaction classes

Rules:

1. place interaction (ui) classes in a separate layer (e.g GUI layer)
2. alternatively assign dialogue classes to some other classes: in this case the class is responsible both for data and representation on the GUI
3. revising the associations: cover all possible communication links

    - each class that has a complex operation must be associated to the classes that they call
    - consolidate the associations
    - provide multiplicities (cardinalities)

4. no redundant links: associations that are not used in any SF should be removed.

Design Class Diagram / Model

Goal: preparation for coding by taking design goals into account.

Analysis class diagram gets refined with infrastructure classes (e.g. library) and completed, often using design patterns.

Steps:

1. complete the list of attributes and operations
2. determine the data types and access specifiers (private, protected, public, etc)
3. specify operations (pre- and postconditions)
4. define exceptions
5. specify concrete data structures that realize associations
6. eliminate multiple inheritance
7. Consolidation via sequence diagrams

Step 3 - Specifying Operations

Specification of an operation specifying the behavior of an operation by providing a contract, without providing the algorithm

Design by Contract specifying an operation by providing pre-, post-condition and an invariant

- Usually only textual
- Sometimes directly supported by programming languages, e.g
  - Javadoc comments in Java
  - Assert in Java

Step 4 - Specifying Exceptions

Error situations in code are usually represented as special output values, however this does not guarantee that the caller will deal with the error.

Instead some programming languages have explicit error handling mechanisms called exceptions

in java:

```java
try {
  // code that throws and exception of type E
} catch (E e) {
  // what to do with the caught exception
}
```

Exception types:

- Exceptions in the domain: input doesn't satisfy precondition, business logic is violated
- Technical failure: Connection to server failed, called object doesn't exist

Ways to deal with exceptions:

- directly: using a specific operation
- handing over to the outside: via try-catch block
- not treating: letting the run-time system eventually catch the exceptions

Step 5 - Determining Concrete Data Structures

- Concretizing associations with a qualifier data structure:

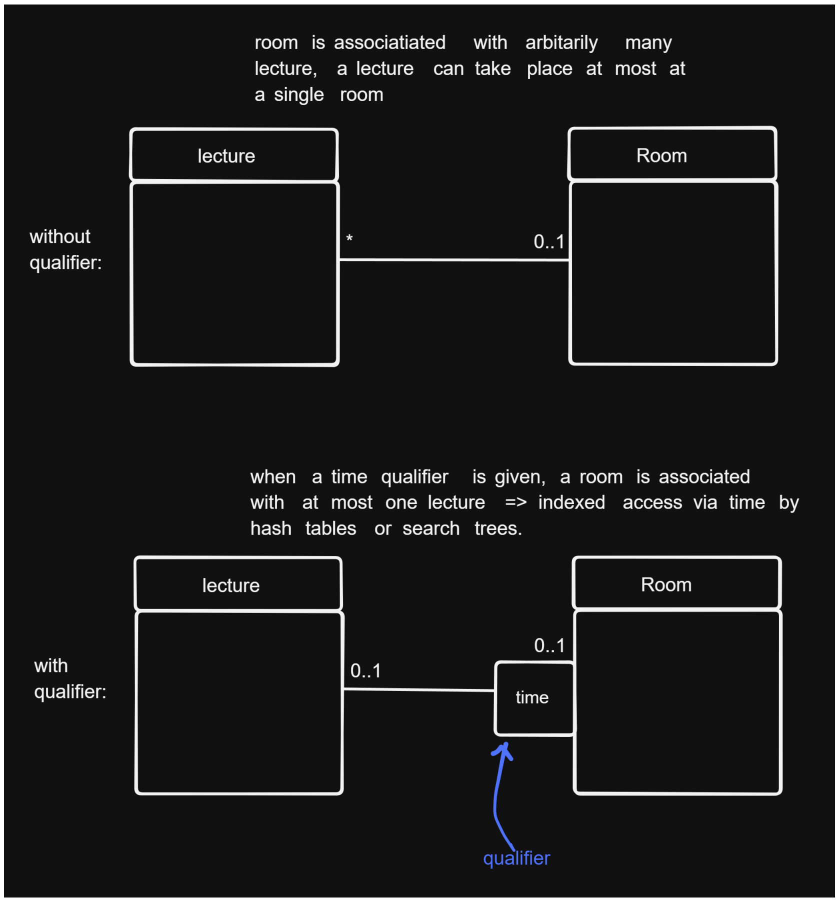

room is associatiated with arbitarily many lecture, a lecture can take place at most at a single room
lecture
Room
without qualifier:
*
0..1
when a time qualifier is given, a room is associated with at most one lecture => indexed access via time by hash tables or search trees.
lecture
Room
with qualifier:
0..1
0..1
time
qualifier

Figure 5.5: concretizing with qualifier

- concretizing associations by providing a fixed order: if there is a fixed order the objects associated to some other objects exist, this order can be fixed via a list or iterators.

Step 6 - Eliminating Multiple Inheritance

It is common to have multiple inheritance relationships in analysis models. This should be eliminated in design models using **interfaces**.

Step 7 - Consolidation via Sequence Diagrams

Implementations of complex operations (SFs) should be checked using sequence diagrams for:

- high coupling: are too many objects and too many method calls used for the implementation? Yes ⇒ reduce it by repartitoning the SF into operations
- low cohesion: is there an operation of a class that works with disjoint sets of attributes? Yes ⇒ can the class be partitioned into multiple classes and will it help with the complexity?

5.0.5 Design Patterns

Transferring experience of good class structure design. Design patterns describes the roles of classes, their dependencies and associations.

Types of design patterns:

- creational patterns (Erzeugungsmuster)
- structural patterns
- behavioral patterns (Verhaltensmuster)

How a pattern is described:

- name:
- problem: motivation, field of application, problem class
- solution:
  - structure (class diagram)
  - elements (classes, their associations, and their operations)
  - interactions of objects (sequence diagram)
- discussion:
  - advantages and disadvantages: when and why should be applied
  - dependencies, limitations
  - special cases
  - known applications (how common is it, how mature is it)

advantages:

- proven solutions to recurring problems
- better readability & maintainability of software design and source code
- easier communication via a common vocabulary / language of patterns
  - among architects and developers
  - among developers

disadvantages:

- using a pattern in a false context: overhead due to unnecessary classes, bad readability, maintainability etc

Creational Patterns

Deal with creation of objects:

- try to hide, unify or simplify it
- describe what is created, how and when

Examples:

- singleton
- abstract superclass
- factory method
- prototype
- builder

Singleton

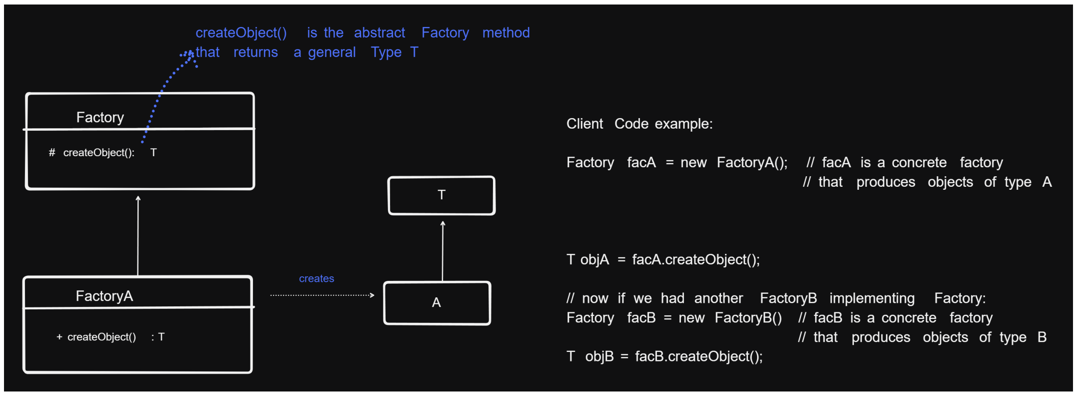
- **problem**: there can exist only a single instance object of a given type.
- **solution**:

  ```
  public final class S {
    private static S instance;
    private S(); // private constructor
    public static S getInstance () {
      if (instance == null) instance = new S();
      return instance;
    }
  }
  ```

```
// usage of singleton
// S s = new S() can't be called because constructor is private, instead
S s = S.getInstance(); // s refers to the single central initialized static instance
```

Factory Method

- **problem**: When creating an object, we don't want to have to specify their exact classes, i.e we should be able to choose between variouis variants of a product.
- **solution**:
  - Rather than creating an object by calling a constructor, the creation of the object is delegated to a factory method
  - The code that creates the object is outsourced to an own class called Factory or Creator. Optionally the Creator / Factory class can be abstract or an interface and the object creation can be implemented in subclasses inheriting from it ⇒ subclass decides which object type is to be created.



```
Factory facA = new FactoryA();   // facA is a concrete factory
                                 // that produces objects of type A

T objA = facA.createObject();

// now if we had another FactoryB implementing Factory:
Factory facB = new FactoryB()  // facB is a concrete factory
                               // that produces objects of type B
T objB = facB.createObject();
```

Figure 5.6: factory design pattern

Builder

- **problem**: Objects of the a class can be very different if they are complex, requiring a different constructions. How can the creation of a complex object be simplified, specifically, how can the creation of a process be abstracted from its representation, so that the details of the way the object is created can be later easily changed, without having to modify the class?
- **solution**: Encapsulate the building of a complex object in a separate Builder class, to which the creaetion of the object is delegated to, instead of creating it directly

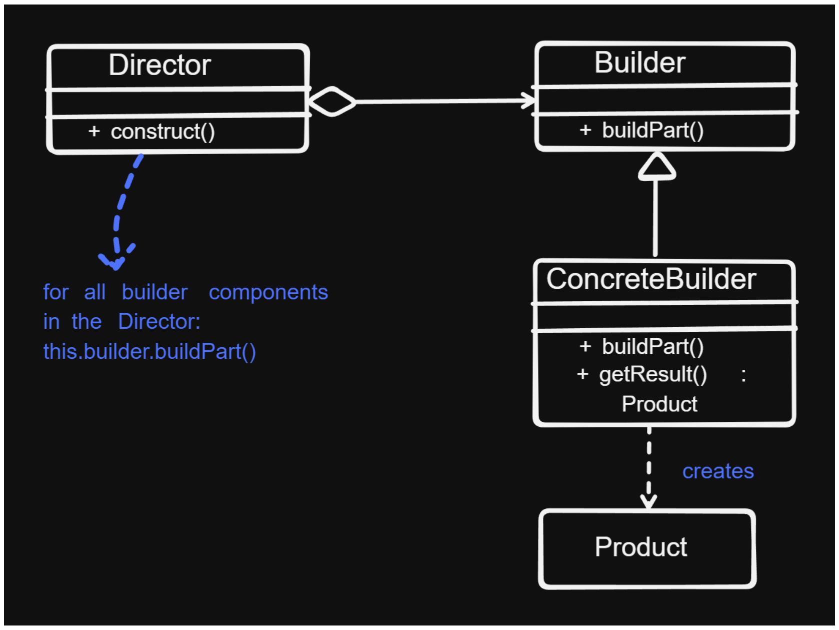


Figure 5.7: builder pattern

- **Director**: assembles the object from the sub-parts whose construction is delegated to `Builder` objects.
- **Builder**: the abstract interface for creating objects (products)
- **ConcreteBuilder**: provides the implementation for the builder

Abstract Superclass

- **problem**: different classes contain identical groups of attributes and methods
- **solution**: refactor the classes, such that the common groups of attributes and methods are separated in an abstract super class, from which the old classes inherit.

Structural Patterns

Deal with combinations and relationships of classes ⇒ allow building larger structures

examples:

1. **Composite**
2. **adapter**
3. **proxy**
4. bridge
5. director

6. facade
7. flyweight

Composite Pattern

- **problem**: modelling and implementing hierarchical (tree) structures
- **solution**: a uniform abstract super class from which a leaf and a node class inherits

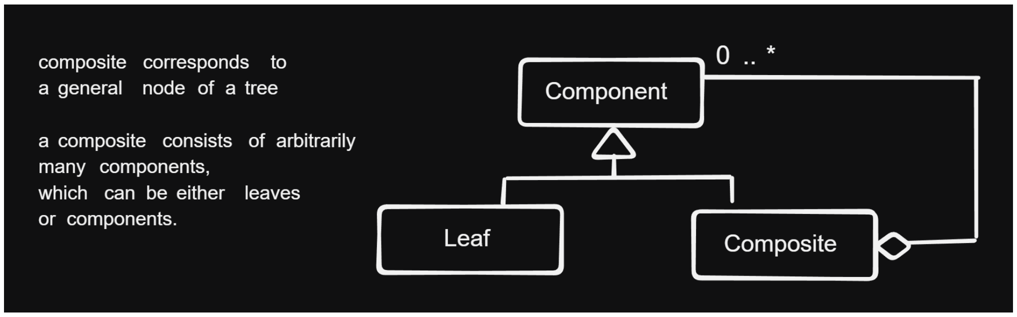

composite corresponds to a general node of a tree
a composite consists of arbitrarily many components, which can be either leaves or components.
0 .. *
Component
Leaf
Composite

Figure 5.8: composite design pattern

Adapter Pattern

Also called wrapper

- **problem**: A Class A (client) requires a class with a certain interface. There is another class B, that provides a different kind of interface. We would like to use class B in class A, even though the interface that B provides is not what A requires.
- **solution**: Define a separate adapter class that converts the incompatible interface of class B (adaptee, i.e. the class to be adapted) into the interface that A expects. (this interface is called the target interface).

Proxy

- **problem**:
  - Access to an object should be controlled (because it is expensive or sensitive from a security standpoint)
  - Additional functionality should be provided when accessing an object
- **solution**: A separate proxy object that can be used as a substitute for the other object (subject). Proxy implements necessary additional functionality.

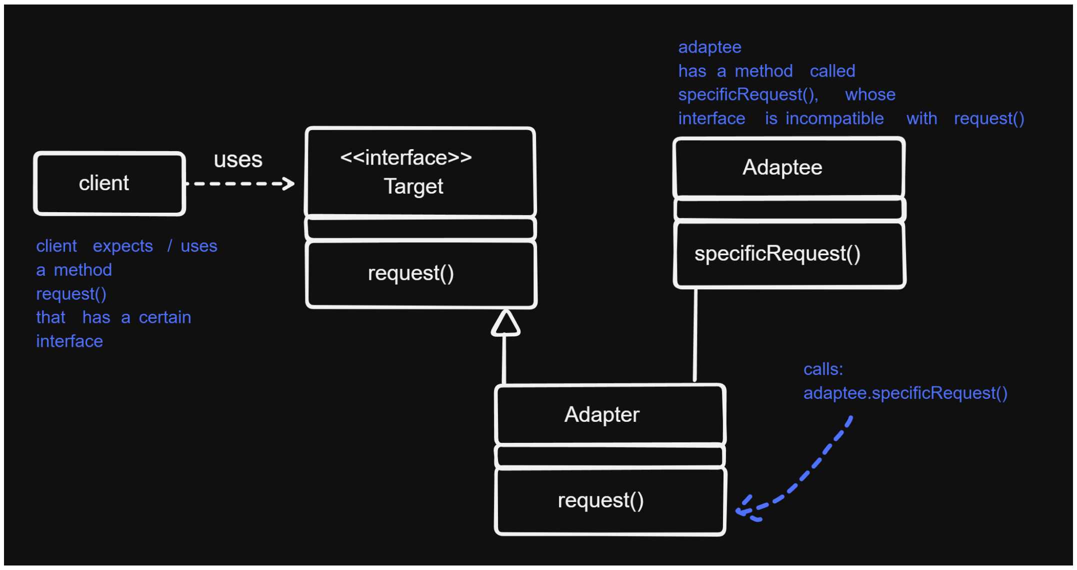


Figure 5.9: adapter design pattern

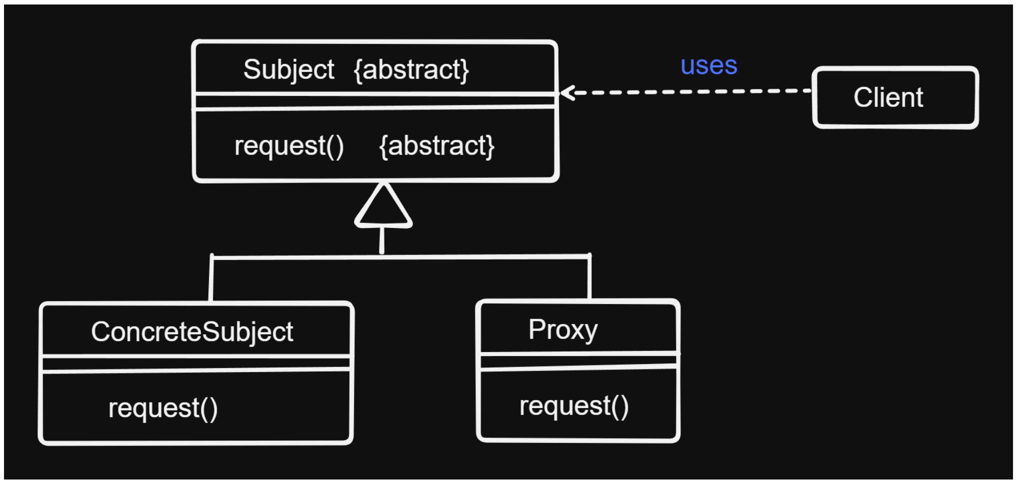


Figure 5.10: proxy design pattern

Behavioral Patterns

Deal with algorithms and assigning states to objects. Describe not only classes and objects but their interactions.

examples:

1. **template method pattern**
2. command
3. **observer**
4. **visitor**
5. interpreter
6. memento
7. **strategy**
8. iterator
9. mediator
10. state
11. **chain of responsibility**

Template Method Pattern

- **problem**: a method consists both from fixed but also from changeable elements
- **solution**:
  - define a template method, that comprises the scaffolding of the method
  - the algorithm is concretely implemented in the inheriting subclasses
  - fixed components making up the method are factored to the superclass

Chain of Responsibility

- **problem**:
  - The sender of a request / message and the receiver of the message shouldn't be coupled, or only loosely coupled.
  - It should be possible for more than one receiver to handle the request in flexible ways
- **solution**: A chain of receiver objects that, depending on run-time conditions, either handle the request or forward it to the next receiver. This enables a flexible handling of the request. In other words the sender does not know which object will handle the request or how exactly.

Observer

- **problem**: There is a one to many relationship between an object called **subject**, and its dependencies, called **observers**. Observers must be notified whenever the state of the subject changes and react to it. In other words observers "observe" the subject and react to its state changes.
- **solution**:
  - Subject maintains a list of observers. Observers have methods to subscribe to or unsubscribe from the list.
  - Subject has a method called `notify()`, that calls the `update()` method in each of the observers that are in the subscribers list. The `update()` method of each observer updates each one, making it to react to the state change of the subject.

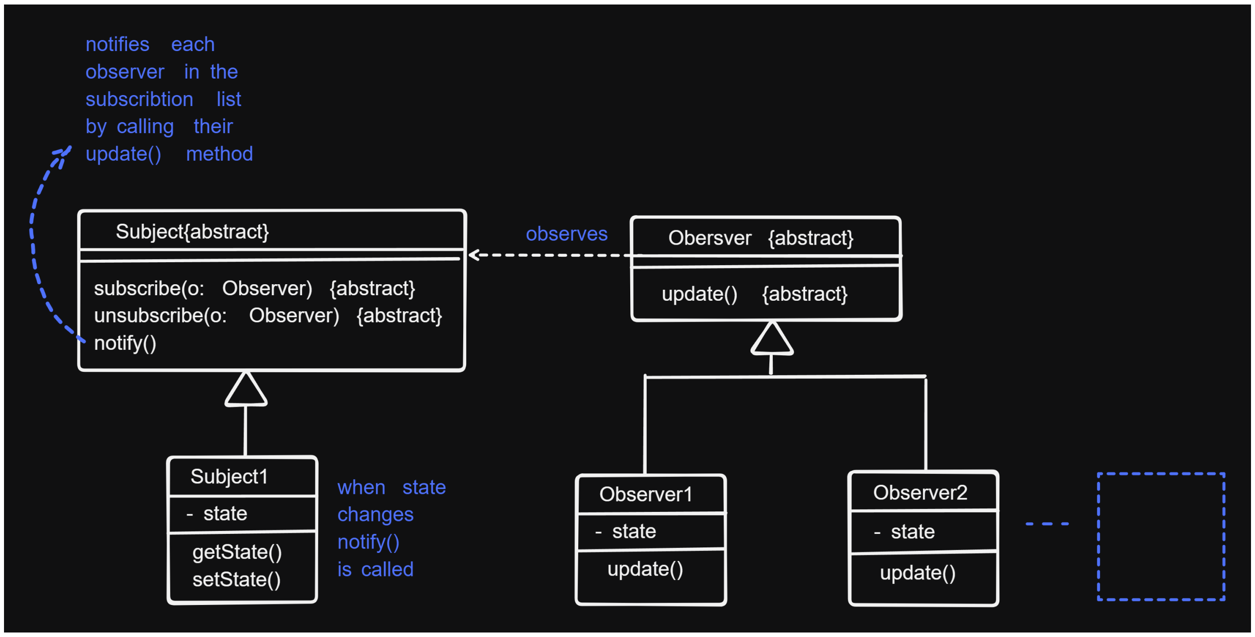

notifies each observer in the subscribtion list by calling their update() method
Subject{abstract}
subscribe(o: Observer) {abstract}
unsubscribe(o: Observer) {abstract}
notify()
observes
Obersver {abstract}
update() {abstract}
Subject1
- state
getState()
setState()
when state changes notify() is called
Observer1
- state
update()
Observer2
- state
update()

Figure 5.11: observer pattern

Visitor

Strategy

5.0.6 Rationales (Communication of Decisions)

Documents contain only the last decision. Communication of all sorts of decisions and history of discarded decisions via Rationales.

How are rationales described:

- questions: concrete problems that don't have an obvious solution
- options: describe alternative solutions to a problem
- criteria: quality requirements
- arguments: condensate and summarize discussions
- decisions:
  - relates to one or more open questions
  - summarizes the chosen options and arguments that support it

6 Quality Assurance

Quality: Software satisfies the requirements

topics:

1. Introduction
2. Organizational quality assurance
3. Testing:
    1. Intro
    2. Test-case specification
    3. Black-box component testing
    4. White-box component testing
    5. System testing
    6. Integration testing / overall-component testing
4. Static testing
    1. Static Analysis
    2. Metrics
    3. Inspection
5. Analytical Quality assurance at large

6.0.1 Introduction

Core goals of software engineering:

- **quality**: software fulfills the requirements $\Leftarrow$ **this**
- **users**: software is useful for the users
- **developers**: it is easy to develop, maintain and modify the software
- **cost / time**: software can be implemented within the given cost & time constraints $\Rightarrow$ project management

Methods of quality control are based on the following questions:

- what are the requirements?
- what errors are there in the system?
- how do the errors originate and manifest in the system?
- how to prevent errors?

Terms

Validation vs Verification:

- **Verification**: the product fulfills requirements
- **Validation**: the requirements correspond to users wishes

Error and deficiency

- **Error**: discrepancy between the product and the requirements
- **Deficiency**: a requirement or an expectation is fulfilled insufficiently

Further terms:

1. individual mistake by a person ⇒ standards, norms, training
2. erroneous state / deficiency in a program ⇒ debugging
3. error that manifest in the system ⇒ testing

Above 1 ⇒ 2 ⇒ 3.

How is quality control achieved:

- **Quality management**: general
- **Quality assurance / QA**: concrete processes to achieve quality
    - Constructive QA ⇒ design, implementation, programming
    - Analytical QA ⇒ formal proof, inspection, static linting, dynamic testing
    - Organizational QA ⇒ project management

6.0.2 Testing

Introduction

Goals is to find errors that manifest in the system (Fehlerwirkung) ⇒ Systematic test:

- Pre-/Postconditions are defined precisely
- Inputs are systematically specified
- Results are documented and analyzed w.r.t testing criteria

combinatorial state explosion ⇒ Complete testing is never possible

Terms:

- **Base**: all documents that the test case is derived from (requirements etc)
- **Test case**: consists of
    - collection of inputs

  - preconditions and edge cases
  - expected results
  - expected exceptions

- **Precondition**: the state of the object / environment, that must be specified, s.t. a test-case can be run
- **Postcondition** the state of the object / environment after the execution of the test case.
- **Test run**: execution of test or the suit of tests on a specific version of the test object
- **Reaction of the test object**: The sequence of internal states, reactions and outputs of the tested object. They must agree with the requirements / expectation (ideally tested automatically)
- **specification**: determining test objects and their test cases, choosing the testing methods

  - derivation from the documentation and logical test cases
  - condition to end the test

Test stages:

1. Component / Unit Test
2. Integration test: integrating components with each other
3. System test: whole system
4. acceptance test: after release, by the user / client

(1)$\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (4)

Regression Test: after changes in the software test to see if changes craeted new errors

Test-case Specification

Elements of a test case:

- expected behavior based on a **test oracle**:

  - requirements specification
  - user manual
  - executable prototype (formal specification)
  - old version

- two abstraction levels when describing the test case

  - logical: range of input / output, possibly via equivalence classes
  - concrete: specific input / output values, possibly representatives of equivalence classes

Description of a test case:

- name
- tested requirement / relation to a requirement
- type: component, integration, system, acceptance
- precondition
- postcondition
- test infrastructure
- description of test steps. For each step:
    - input
    - expected output
    - expected exception

Component

Component: a self-contained code unit ⇒ class, function, module

typical erroneous behavior of a component:

- non-termination
- incorrect or incomplete result
- unexpected / incorrect error message
- inconsistent memory
- superfluous resource load
- unexpected exception behavior, e.g. crash

Component test types:

- Black-box: no knowledge of the internal implementation of the object, only the interface and specification.
- White-box: purposeful testing inner elements and the flow of execution, using the knowledge of its internal structure.
- Intuitive: based on knowledge / experience of typical errors ⇒ supplementary to the two systematic test above.

Black-box Component Testing

Properties of a black-box test:

- Test cases are derived from the input / output behavior of the operation (specification)
- The goal when deriving the text cases is the coverage of:
    - input values
    - output values
    - specified exceptions

- especially tests that all the requirements on the operation are satisfied.

Test case description for a an operation / method test:

- name
- tested component (class)
- type: component test
- precondition: regarding relevant component data and restriction on the input
- postcondition: updated component data
- test steps: detailed description of the steps,
  - input
  - expected output
  - expected exception

Equivalence Class Tests

Equivalence class:

- **Idea**: Partitioning of the range of input / output values into classes, such that values from the same equivalence class demonstrate conceptually same behavior and a single value from that class can be chosen as a representative.
- **Boundary values**: if the rang of values of an EQ are ordered, then
  - values on both of the exact boundaries (min and max)
  - as well as the neighbors of the boundaries: pred(min), succ(min), pred(max), succ(max)

typical equivalence classes:

- input:
  - Valid input EQ (GEK): Valid input range, possibly subdivided w.r.t. the boundary values
  - Invalid input EQ (UEK): invalid input range
- output:
  - Valid output EQ (GAK): partitioning of **valid input ranges** s.t. various typical output values are covered (also possible subdivisions w.r.t boundary values)
  - Invalid output EQ (UAK): Exceptions

Deriving test cases based on EQ

- if there are **multiple** inputs:
  - combine all valid EQs of various inputs (cartesian product) ({GEK1, GAK1} x … x {GEKn, GAKn})

   - combine all valid EQs with all possible invalid EQs {GEK1, GAK1, ... , GEKn, GAKn} x {UEK1, UAK1, ..., UEKn, UAKn}

Simplification:

- only frequent combinations
- only test cases with boundary values
- only pair-wise combinations

Minimal:

- each valid equivalence appears in one test case

White-box Component Test

white-box component test:

- Test cases are derived from the knowledge of the code base.
- The goal is to test weather all of parts of the code are correct.
- Requirements specification is additionally necessary as a **test oracle** , i.e to determine what the results of the tests should be.

Main goal is the achieve **coverage** of the code:

- Statement Coverage
- Branching Coverage
- Path Coverage

Coverage can be seen on the **control-flow graph**

Control-flow Graph

Control-flow graph is an abstraction of the code, s.t.:

- consecutive statements are united
- all branches are visible: multiple execution flows from a statement ⇒ individual node.
- jumps ⇒ individual nodes
- possibly few nodes!

Rules:

1. consecutive statements until the end of a block, especially until the next if-branching or while-loop are combined to a single node. (the branching or loop condition can be included in the node as well)

```
1  int binsearch(int[] a, int k) {
2      result.found = false;
3      result.index = -1;
4      if (a.length != 0) {
5          int bottom = 0;
6          int top = a.length - 1;
7          int mid;
8          while (bottom <= top) and !result.found {
9              mid = (top + bottom) / 2;
10             if (a[mid] == k) {
11                 result.index = mid;
12                 result.found = true;
13             } else {
14                 if (a[mid] < k)
15                     bottom = mid + 1;
16                 else
17                     top = mid - 1;
18             } // end else
19         } // end while
20     } // end if
21     return result.index
22 } //binsearch
```

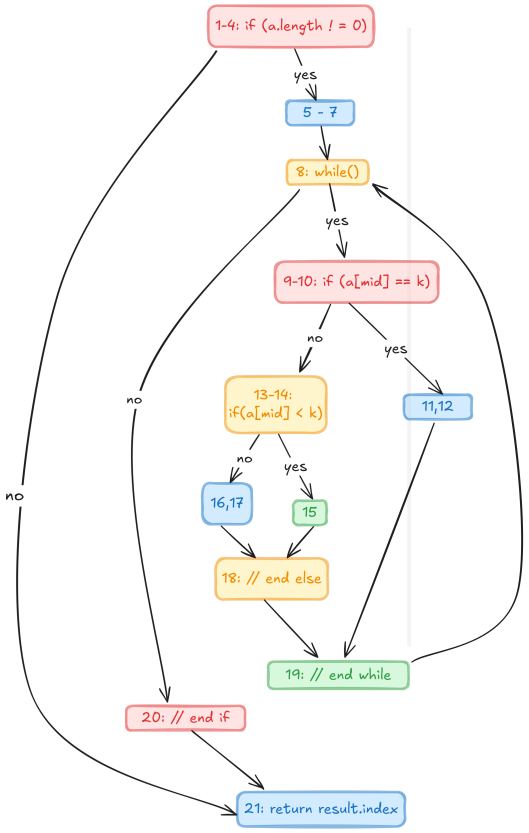


Figure 6.1: control-flow graph example

2. if a block consist of multiple nodes the brace closing the block gets its own individual node. (other than the brace closing the program)
3. The node of the negative case of an if-else branch contains the else line and the following statements.
4. The condition of a loop-statement get an individual node (due to the jump).
5. If earlier `return` statements exist in the program, the `return` statement becomes a node that can be reached in various ways.

Statement, Branching, Path and Term Coverage

Statement Coverage

Nodes in the control flow graph are (combinations) of statements. 100%-statement coverage ⇒ all nodes of the graph are visited. In the previous example:

- (1-4) ⇒ (5-7) ⇒ (8) ⇒ (9 - 10) ⇒ (13,14) ⇒ (15) ⇒ (18) ⇒ (19) ⇒ (8) ⇒ (9-10) ⇒ (13-14) ⇒ (16,17) ⇒ (18) ⇒ (19) ⇒ (8) ⇒ (20) ⇒ (21)

Which input achieves this sequence? ⇒ `key == 15, a == [1, 15, 17, 19, 20]`

Note

100% node coverage does **not** imply that all equivalence classes are covered

Branch Coverage

branch coverage:

- edges ⇒ branches.
- branch coverage ⇒ all edges are visited

previous example doesn't cover the edge (1-4)⇒(21). This corresponds to the case when the input array is empty: `a == []`, which in turn is a UAK.

Term Coverage

- For complex branching and loop conditions every part making up the condition should be covered, i.e. every term making up the complete expression is "activated" at least once, for both when the whole condition is true, as well as false.
- Goal is to discover possible bugs due to individual terms.

Example: `if(X or Y)`. Here `X` and `Y` are terms and `X or Y` is the complex condition.

Path Coverage

- Path: Sequences / combinations of edges
- 100% path coverage not feasible & not easy to define.

Grey-box Testing: White-box + Black-Box

1. first define black-box test cases: simpler, since only requirements are sufficient.
2. check which portions of code are not reached by the black-box tests. (e.g. with coverage plugins)
3. complete the test suite by writing additional white-box tests such that 100% statement and brach coverage is achieved.

Class / Object Test

Previously we discussed testing smaller units, like methods. We can also test larger components like classes (objects).

Individual methods of class ⇒ gray-box.

But for testing interoperability of the methods of the whole class ⇒ state based test based on the state diagram of the object. Testcases are then sequences of methods calls on the object that traverse various states of the object.

System Test

Integration Test

7 Evolution

All activities that facilitate re-use and further development. (All activities that take place after the initial development phase)

topics:

1. Intro
2. Architecture
3. Re-use
4. Further development and change management
5. DevOps & IT-Governence
6. Re-engineering

8 SWE Process & Project Management

Making sure that the Software system is developed withing the time money constraints.

topics:

1. Project management
2. SWE-process models & methods