

62531 Development Methods for IT Systems

Course plan

Deena Francis, Lei You

August 24, 2023

Text book: [R1] Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, ISBN-13:9780131489066
Note: other references or links will be provided for certain lectures. For details see below.

Lecture 1 (Main theme: *Introduction to software development process*)

Date and time: 29 August 2023, 13:00 to 17:00

Reading: [R1] Chapters 1 - 3

- Introduction to the course
- Introduction to:
 - Models of software development
 - Unified Process
 - Object Oriented Analysis and Design
 - Unified Modelling Language

Lecture 2 (Main theme: *Requirements specification and use cases*)

Date and time: 5 September 2023, 13:00 to 17:00

Reading: [R1] Chapters 4 - 6

- The UP inception phase
- Requirements specification
 - Types
 - Stakeholders and actors
 - Use cases
 - Requirements gathering techniques

Lecture 3 (Main theme: *Detailed look into UML*)

Date and time: 12 September 2023, 13:00 to 17:00

Reading: [R1] Chapter 6

- Actors
- Use case diagrams

- Inheritance
 - Inclusions
 - Extensions
-

Lecture 4 (Main theme: *Elaboration phase, introduction to risk management*)

Date and time: 19 September 2023, 13:00 to 17:00

Reading: [R1] Chapters 7 - 8

- Other requirements
 - Introduction to risk management
 - Inception to elaboration
 - Introduction to domain models
-

Lecture 5 (Main theme: *Analysis classes, sequence diagrams*)

Date and time: 26 September 2023, 13:00 to 17:00

Reading: [R1] Chapters 9 - 12, 15 section 15.4

- Domain models
 - More about objects and classes
 - Analysis classes
 - System sequence diagrams
 - Sequence diagrams
 - Packages
-

Lecture 6 (Main theme: *Activity diagrams, introduction to code documentation*)

Date and time: 3 October 2023, 13:00 to 17:00

Reading: [R1] Chapter 28

- Activity diagram
 - Introduction to code documentation
-

Lecture 7 (Main theme: *Class diagrams, General Responsibility Assignment Software Patterns (GRASP)*)

Date and time: 10 October 2023, 13:00 to 17:00

Reading: [R1] Chapters 16 - 17

- Architecture
 - Design class diagrams
 - GRASP
 - Evaluation
-

Lecture 8 (Main theme: *More GRASP and design*)

Date and time: 24 October 2023, 13:00 to 17:00

Reading: [R1] Chapters 17 - 18

- Feedback for the evaluation
 - GRASP
 - Gang of Four (GoF) patterns
 - Usecase realization
-

Lecture 9 (Main theme: *Design to code*)

Date and time: 31 October 2023, 13:00 to 17:00

Reading: [R1] Chapters 19, 20, 31 sections 31.8 and 25.1

- Objects - knowledge about each other
 - Design to code
 - Inheritance
-

Lecture 10 (Main theme: *State diagrams, review technique*)

Date and time: 7 November 2023, 13:00 to 17:00

Reading:

- [R1] Chapter 29 - How to conduct a design review: link
 - Code Review: link
 - Constructive Feedback: link
 - Making Great recommendations: link
 - State diagrams
 - Review technique
 - Peer review - report
-

Lecture 11 (Main theme: *Review, project management*)

Date and time: 14 November 2023, 13:00 to 17:00

Reading:

- [R1] Chapter 29
 - Kogon Blakemore and Wood: Project management for the unofficial project manager: Extract from Chapter 4: Project Planning: link
 - Feedback
 - Project plan
 - Risk matrix
 - Risk management
-

Lecture 12 (Main theme: *Real-life project development, Final evaluation, Sample exam*)

Date and time: 21 November 2023, 13:00 to 17:00

- How does project development take place in a real software company?
 - Guest lecture
 - Evaluation
 - Test exam
-

Lecture 13 (Main theme: *Review of topics*)

Date and time: 28 November 2023, 13:00 to 17:00

- Review of topics
 - How did the mock exam go?
 - Q&A
-

Exercises

During 15:00 to 17:00, the students will work on their weekly exercises. TAs and the lecturer will be available to help clarify doubts and provide general help.

Project

There are **4 reports** that need to be submitted to learn during 4 weeks. These reports correspond to CDIO 0, CDIO 1, CDIO 2 and CDIO 3 projects. The students will receive constructive feedback for these reports.

Exam

The final exam will be a multiple choice exam lasting 2 hours.

62531 Development Methods for IT Systems: Lecture 1

Deena Francis

Assistant Professor,

Section for AI, mathematics and software,

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

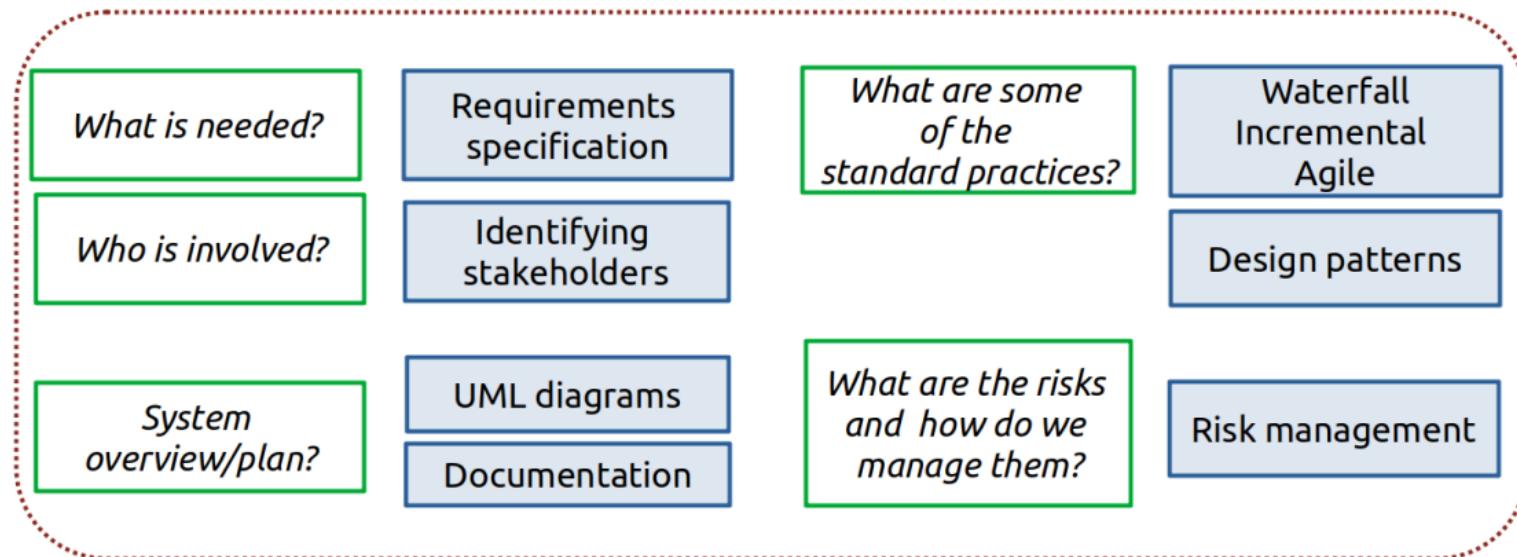
$\Delta \int_a^b \Theta^{\sqrt{17}} + \Omega \delta e^{i\pi} = -1$

$\varepsilon \in \{2.7182818284\}$ $\lambda \circ \alpha$

$\infty = \chi^2 \gg 000, \approx$

$\Sigma!$

Course overview



Today (Lecture 1) (Main theme: *Introduction to software development process*)

- Introduction
- Models of software development
- Unified Process (UP)
- Modelling
- Object oriented analyses and design
- Unified Modelling Language (UML)

Learning objectives (Today)

- Describe a modern software development process
- Prepare a requirements specification in the form of use cases.
- Describe and draw UML diagrams for a use case

Small discussion

Discuss with your neighbors for 2 minutes.

Suppose you are developing a software, what do you think you will need to do?

Small discussion

Discuss with your neighbors for 2 minutes.

Suppose you are developing a software, what do you think you will need to do?

- Know clearly **what** to develop
- Have knowledge of a **programming language**
- Check whether the functionalities are **implemented correctly**
- Constraints, situations of failure
- ...

Software development process

- Consists of several crucial steps
- Not just coding!

A short discussion

Can you think of a few scenarios of how a software project could fail?

A short discussion

Can you think of a few scenarios of how a software project could fail?

- Fails to meet stakeholder requirements
- System crashes
- Unexpected results
- Time delays
- ...

Motivation

Ariane flight V88:

- Launch ended in failure due to:
 - No integer overflow handling
 - self destructed!
- Failure resulted in a loss of more than US\$370 million



Watch_link

Some causes of failures

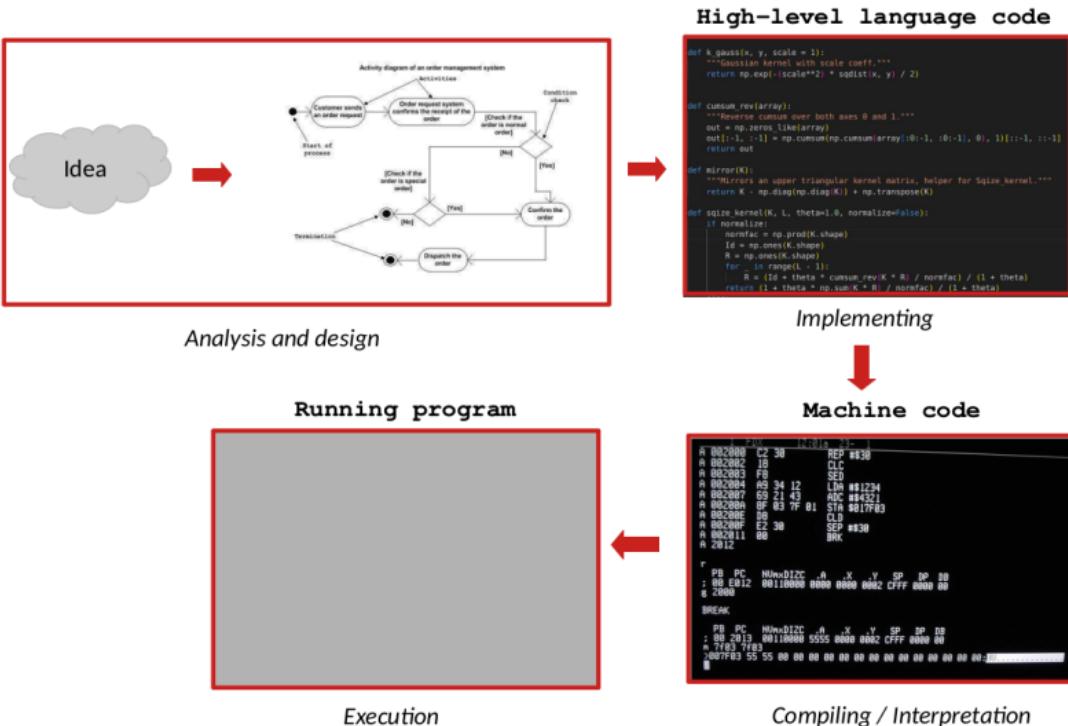
- Poor project management
- Poor cost estimation
- Poor requirements specification
- ...

Reference: [1]

Programming

Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task.

Idea → Code



Development tasks

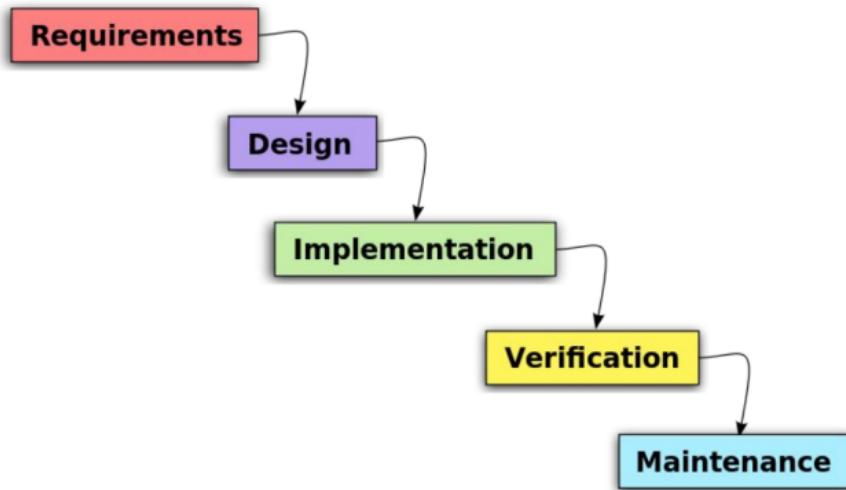
- What?
 - requirements, constraints, usecases
- When?
 - deadlines
- How?
 - tools for development, communication, tracking

Software development methods

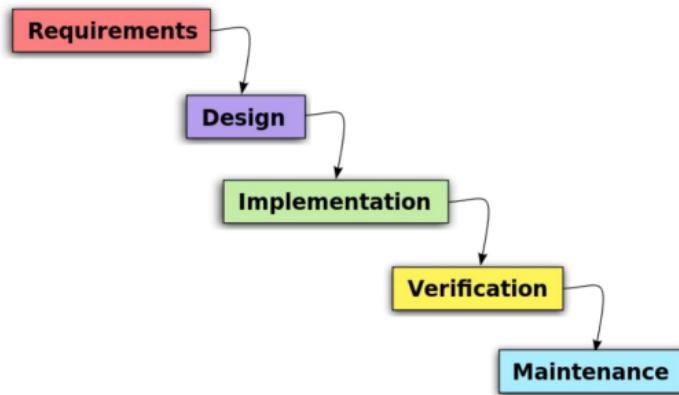
- Waterfall
- Agile

Water fall

- Project activities occur in linear sequential phases



Water fall



Pros:

- Easy to understand
- Easy to track
- Well documented

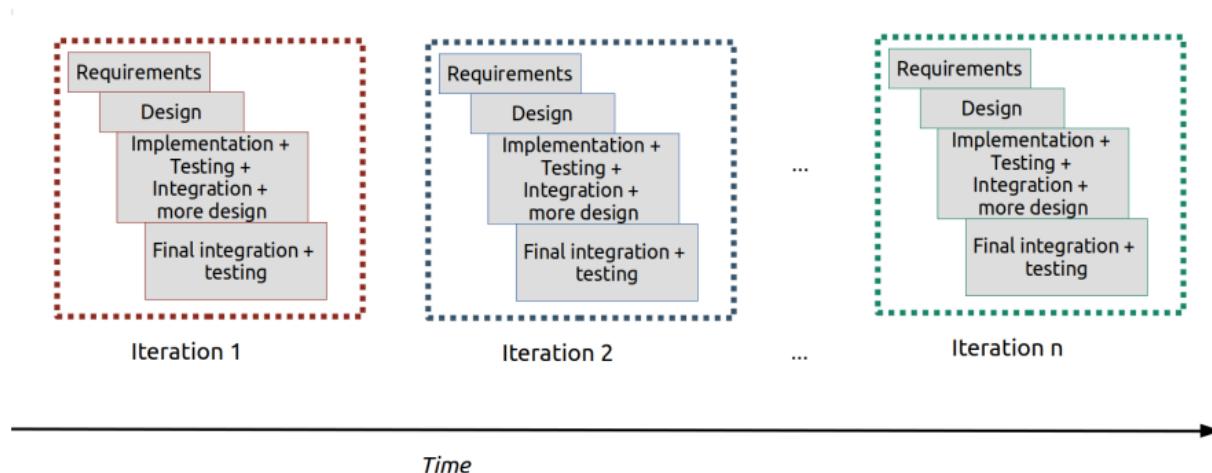
Cons:

- Inflexible
 - Requirements defined at start
 - Difficult to change goals
- Late testing

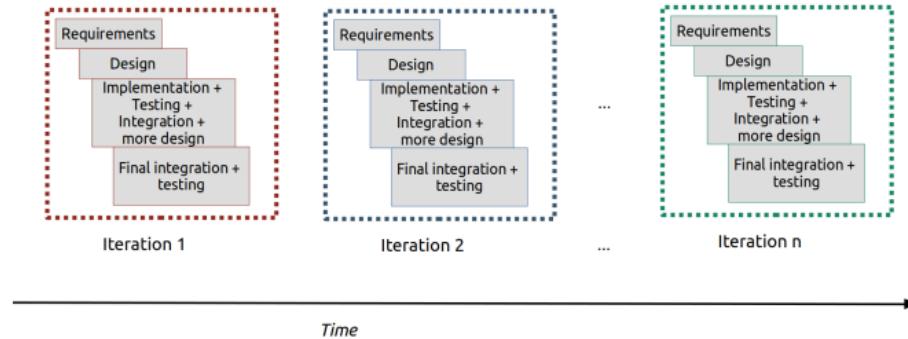
Agile

Development is organized into a series of:

- **Iterations:** short, fixed-length mini-projects
- Each iteration has its own: Requirements, Design, Implementation and testing



Agile



Pros:

- Early feedback, mitigation of risks, view of progress
- Flexible

Con:

- Finding the right iteration length can be tricky

Agile methods

- Agile Unified Process
- eXtreme Programming
- Feature-driven Development
- Kanban
- Scrum
- ...

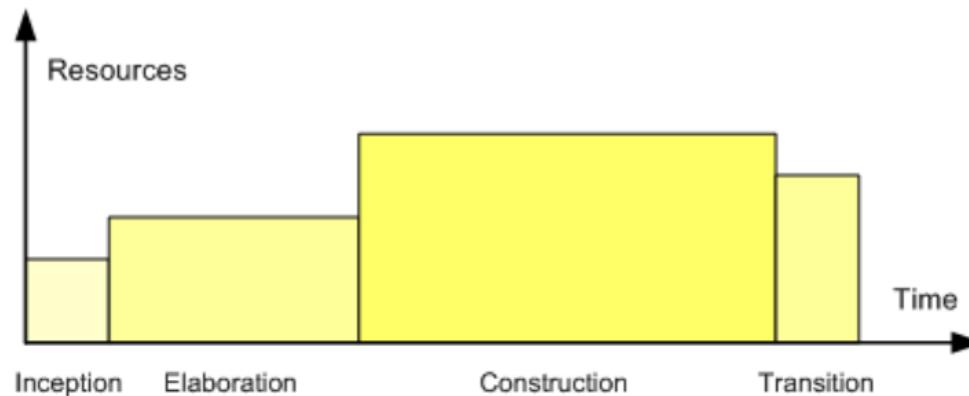
Commonality: flexibility and feedback loops!

Unified Process (UP)

An iterative and incremental software development process.

Iterative = involving repetition

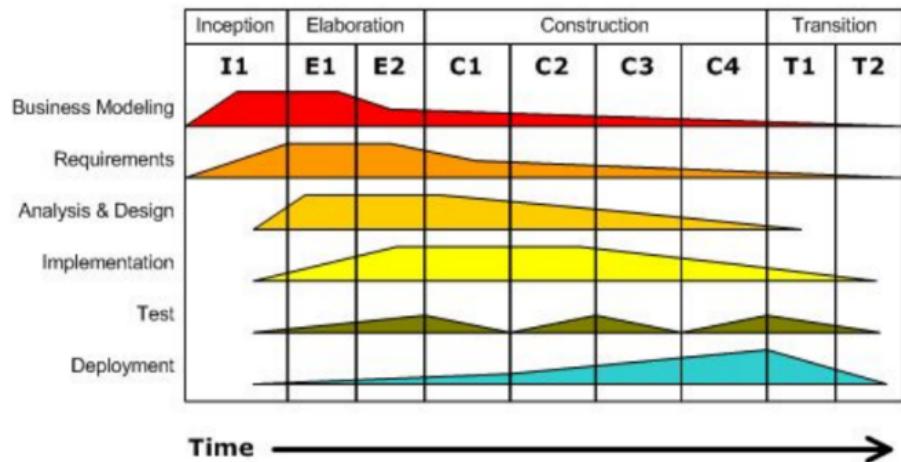
Incremental = in a series of additions or increases



Unified Process - Phases

- ① Inception
- ② Elaboration
- ③ Construction
- ④ Transition

Iterative Development
Business value is delivered incrementally in time-boxed cross-discipline iterations.



UP - an important deliverable

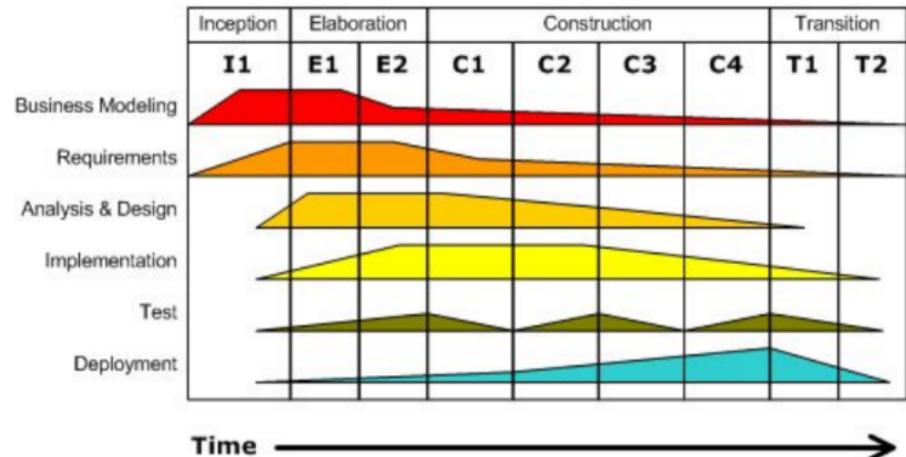
Executable architecture baseline:

- Partial implementation of the system
- Developed during the elaboration phase
- Serves as basis for further development

UP - disciplines or set of activities

- ① Business modelling
- ② Requirements
- ③ Design
- ④ Implementation
- ⑤ Test
- ⑥ Deployment
- ⑦ Supporting disciplines:
 - Configuration and change management
 - Project management
 - Environment

Iterative Development
Business value is delivered incrementally in time-boxed cross-discipline iterations.



UP - Principles

- Iterative and incremental
 - System grows incrementally, iteration by iteration
- Use case driven
 - What users want
- Risk-driven
 - Identify and lower risks
- Architecture-centric
 - Building, testing, stabilizing core architecture

UP - increments

Iterations:

- Divided into fixed timeslots
 - Timeboxed (2-6 weeks)
- Each iteration: All phases involved

A short discussion

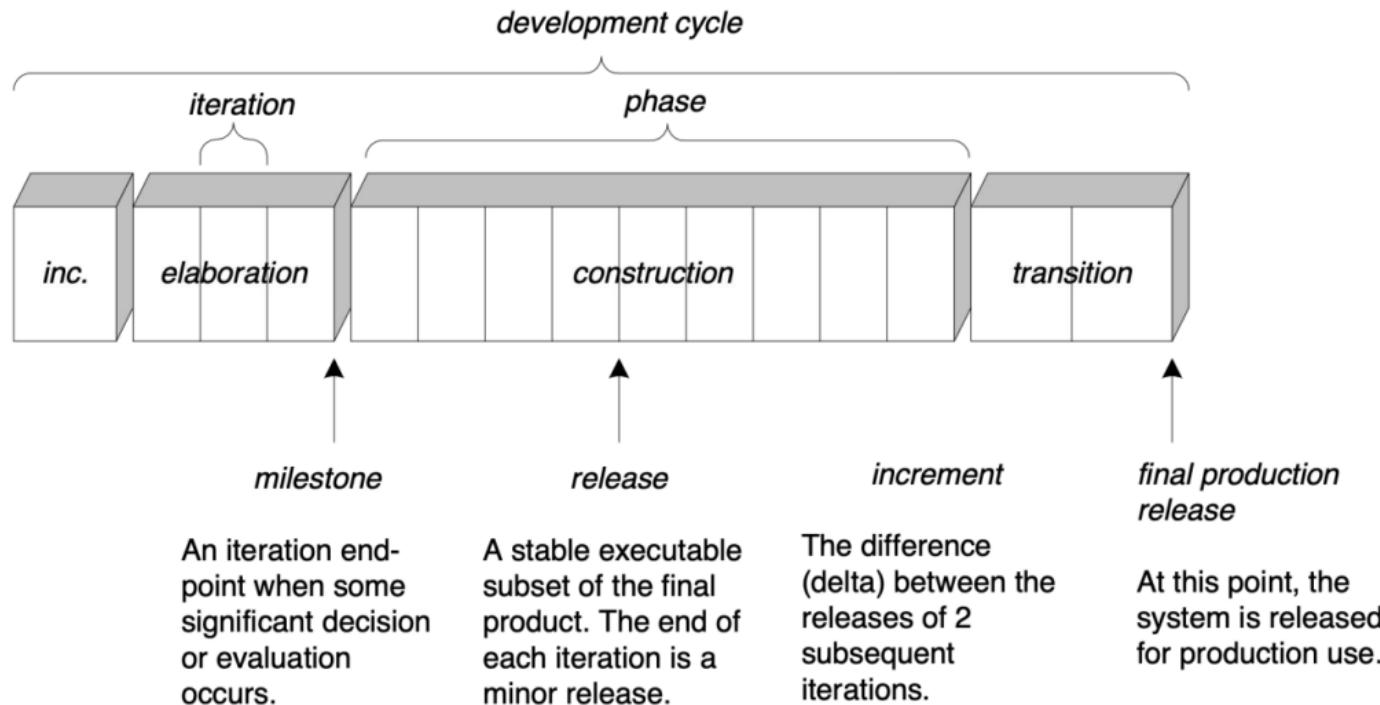
Can you think of the advantages of using iterative development?

A short discussion

Can you think of the advantages of using iterative development?

- Short, manageable development steps
- Visible progress in the work at an early stage
- Requirements can be modified if the need changes
- Early feedback and user participation ensure that the system eventually meets the desired requirements
- High-risk issues can be addressed first

UP - phases and activities



Requirements specification

Use case

It is the user's description of the required functionality.

Examples:

- Create an account
- Search for book
- Play music
- Calculate average score
- ...

Use case

- Verb + Noun

Examples:

- Play DiceGame
- Write book
- Watch movie
- Save file
- ...

Use case

- Description: step by step

Example - Usecase: Play DiceGame:

- ① Player throws two dice
- ② System displays the results
- ③ If the sum of the dice is 6 then the Player wins, else he loses.

Object-Oriented Analysis and Design (OOAD)

Natural way of looking at the world!

C	Movie
-	identifier
-	name
-	genre
-	runningTime
+	displayDetails()
C	Button
-	xLocation
-	yLocation
-	text
+	draw()
+	press()

Object:

- Instances of classes (structure + behaviour)

OOAD

Analysis:

- Finding objects in the problem domain
- Describing objects in the problem domain

Design:

- Define the objects
- Define how objects operate or interact

Modelling

Is a way to understand the problem or solution space.

- Visualization is key

How does this help?

- Improves understanding and communication
- Quickly explore alternatives

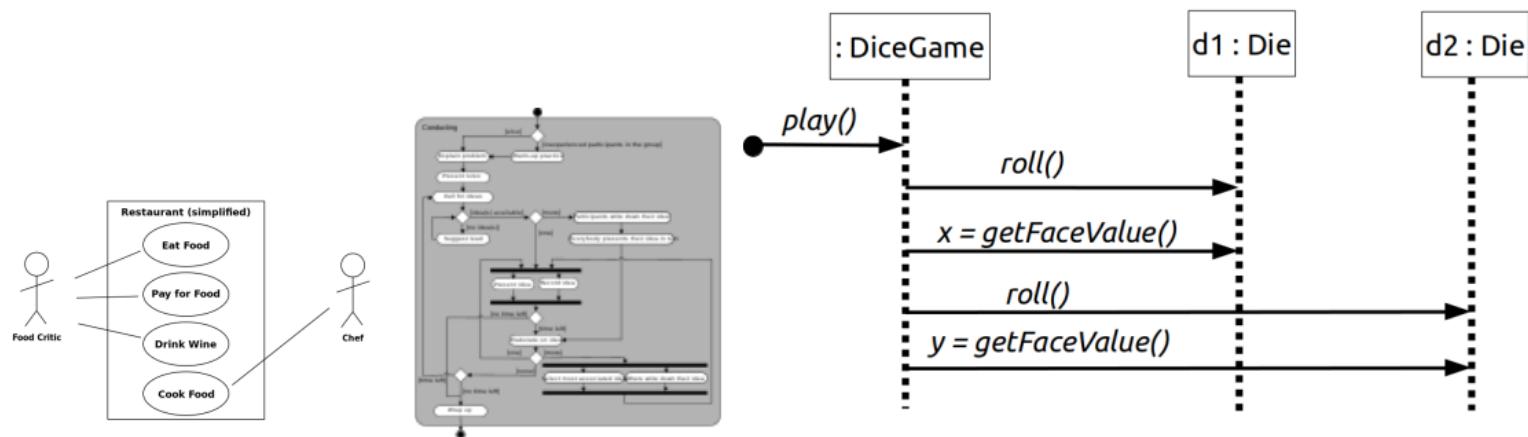
Modelling - an example

A player starts to play a dice game by rolling two dice. If the sum of the values of the dice is equal to 6 then the player wins, else he loses.



Unified Modelling Language (UML)

A visual language for specifying, constructing and documenting the artifacts of the system.



UML - features

Shows:

- Diagrams and descriptions that visualize the system
- A partial representation of the system
- Static and dynamic conditions in the system

UML - Advantages

- Designed to be used for large systems
- Can be read by almost everyone
- Several diagrams can be developed simultaneously

UML

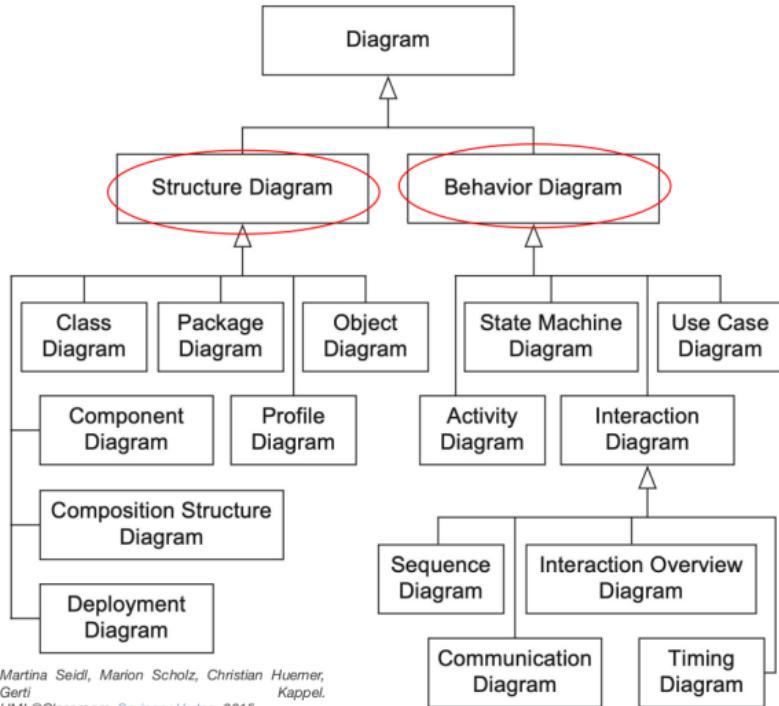


Figure 2.1
UML diagrams

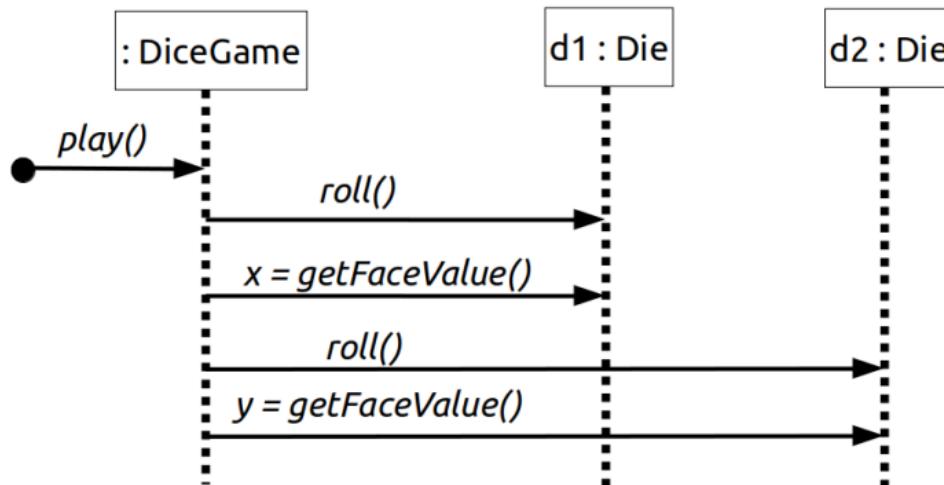
Modelling - First model

Domain model: A diagram that shows *concepts* and their *associations* and *attributes*.



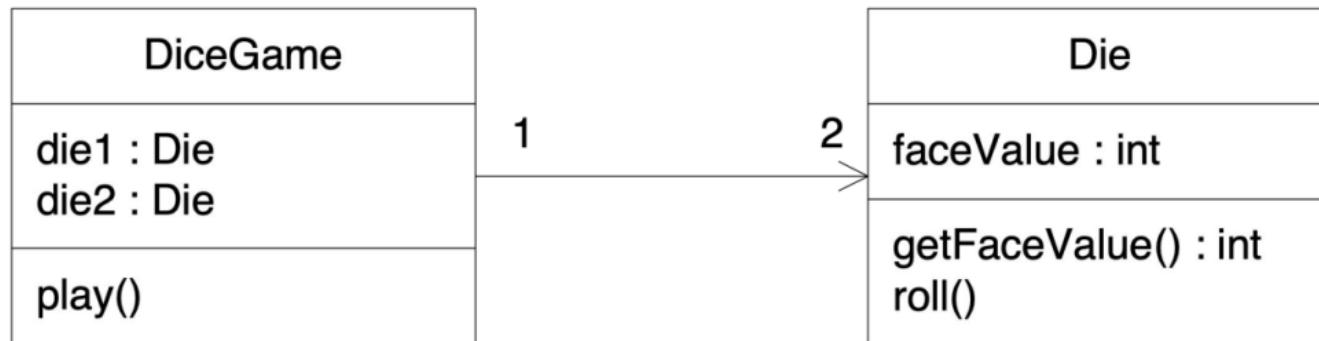
Modelling - Second model (UML)

Sequence / interaction diagram: shows the flow of messages between objects.



Modelling - Third model (UML)

Design class diagram: shows the static view of class definitions



OOAD example - Analysis: Use case

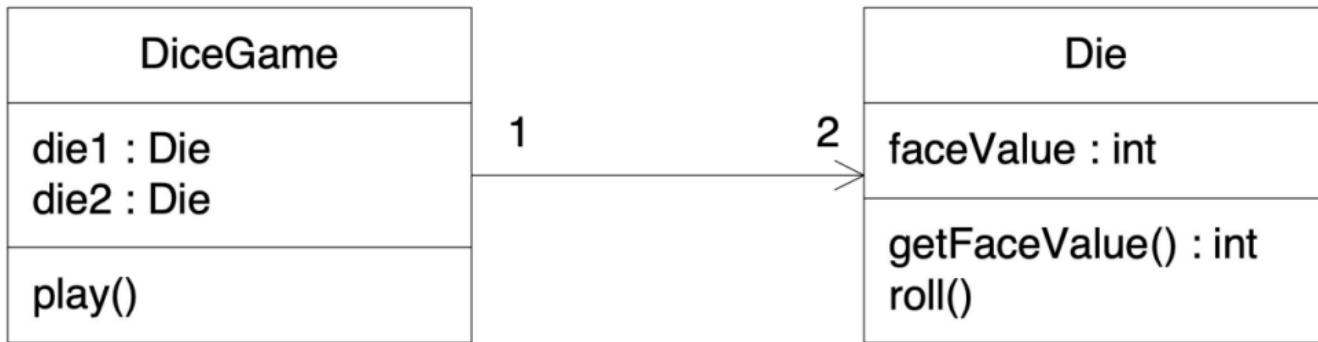
UC1: Play DiceGame

- ① Player rolls two dice
- ② The System displays the results
- ③ If the sum of the dice is 6, the Player wins, otherwise he loses

OOAD example - Analysis: Domain model



OOAD example - Design: Design class diagram



Comparison

The domain model:



The design class diagram



- Conceptual classes
- Relationships
- Possibly very important attributes
- Raw UML class diagram to visualize real-world concepts
- Describes software components, attributes, methods and relationships
- Specification or implementation perspective
- Raw UML class diagram to visualize software elements

Note

- The aim is to understand and communicate – not just to document
- Use the simplest tool
- Keep it simple and use only necessary products

Exercises

- From 15:00 to 17:00, you will work on exercises given in Week1_exercises.pdf on Learn.
- TAs will assist you in these tasks.
- Rooms:
 - 358/holdlokal 065
 - 358/holdlokal 066
 - 358/holdlokal 070
 - 358/holdlokal 069
 - Or here (306/34)

Thank you!

- [1] Soren Lauesen. It project failures, causes and cures. *IEEE Access*, 8:72059–72067, 2020.

62531 Development Methods for IT Systems: Lecture 2

Deena Francis

Assistant Professor,

Section for AI, mathematics and software,

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$\int_a^b \Theta^{\sqrt{17}} + \Omega \delta e^{i\pi} = -1$$

$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$

$\Sigma!$

∞

χ^2

\gg

\approx

\circ

λ

$\{2.7182818284\}$

μφερτυθιοπσδφγηξκλ

Today (Lecture 2) (Main theme: *Requirements specification and use cases*)

- The UP inception phase
- Requirements specification
 - Types
 - Stakeholders and actors
 - Use cases
 - Requirements gathering techniques
- CDIO 0 report (deadline on 15/09/2023 18:00)

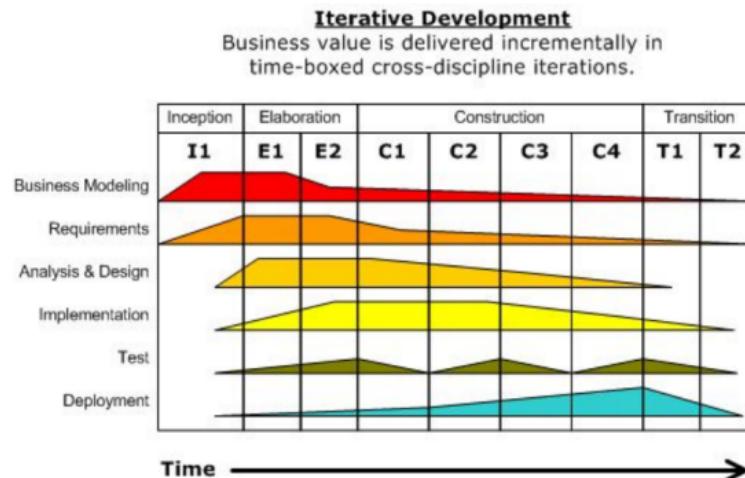
Learning objectives (Today)

- Describe the components of UP inception phase
- Describe and identify requirements
- Describe and define use cases
- Identify stakeholders and actors
- Draw a basic use case diagram

UP - phase 1: Inception

Initial short step to establish a common vision and basic scope for the project.

- One or few weeks long
- Go ahead or not?



What does vision look like?

Think about the following:

- Need
- Approach
- Benefit
- Competition

- This product _ is a complete solution to _ that helps _.
- It uses a unique method _.
- Its key benefits include _.
- Our product differs from the current products _ in terms of _.

The vision - longer and more detailed edition

Page 109 - 111 of the textbook

The vision - short edition

Mission

Maternity Foundation works to improve maternal and newborn health. We develop and integrate scalable programs and digital solutions that empower birth attendants, pregnant women and new mothers in low- and middle-income countries.

Vision

A world where no woman or newborn suffers preventable harm or death related to pregnancy or childbirth.

Inception artifacts

These are the things that are created in the inception phase.

Inception artifacts

Artifact	Comment
Vision	High level goals and constraints
Business case	Describes the business requirements
Use case model	Functional requirements, identify most of the use cases, 10% of them are analyzed in detail
Supplementary specification	Other non-functional requirements
Glossary	Key domain terminology and data dictionary
Risk list and risk management plan	Risks and ideas for mitigation
Prototypes and proof-of-concepts	Clarify vision, validate technical ideas
Iteration plan	What to do in the first elaboration iteration
Phase plan and software development plan	Tools, people, education and other resources. Low-precision guess
Development case	Description of the customized UP steps and artifacts for this project

The inception phase

- Create a few diagrams for improving understanding
- Use only the artifacts that are important to the project
- This is a short phase 1-2 weeks

A new chatting app



Artifacts in requirements specification

- Stakeholders
- Actors
- Requirements
- Use cases

Stakeholders

- Affects or is affected by the system
- End users, supporters, customers, organizations



- Funding partners
- Private organizations
- Schools, universities
- General population

Identifying stakeholders - activity

Paint program for kids

This program will be developed for kids 2+ which will feature basic drawing and coloring tools such as rectangles, circles, lines and a paint bucket. This software will be used in kindergarten schools to assist kids in learning colors and shapes. Teachers will give a demonstration of the tool and then the kids will try to replicate it.

Can you identify the stakeholders?

Actors

- Not a part of the system
- Have a role
- Two types:
 - Primary
 - Secondary



- Students
- Employees of private organizations
- Any person

Primary and secondary actors

Primary actors

- initiate use cases and interact with the system
- have their goals met
- placed on the left side of use case diagrams

Secondary actors

- used by the system but they do not interact with the system on their own
- obtains information
- placed on the right side of use case diagrams

Identifying actors - activity

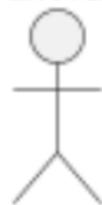
Paint program for kids

This program will be developed for kids 2+ which will feature basic drawing and coloring tools such as rectangles, circles, lines and a paint bucket. This software will be used in kindergarten schools to assist kids in learning colors and shapes. Teachers will give a demonstration of the tool and then the kids will try to replicate it.

Can you identify the actors?

Actors in UML notation

«Human»



User

«Application»



Main Database

The «» denotes a stereotype, an extra text annotation.

Requirements

Requirements: are the capabilities and conditions that the project must conform to.

- It must be measurable

Examples of measurable attributes

Property	Measure
Speed	Transactions/second, response time
Size	Mbytes
Ease of use	Training time, number of help modules
Reliability	Mean time to failure, Rate of failure
Robustness	Time to restart after failure, percentage of events causing failure
Portability	Percentage of target dependent statements, number of target systems

Requirements specification

- Do it iteratively
- Do it skilfully
- Document and communicate them effectively

Requirements specification - types: FURPS+

- Functional - features, capabilities, security



- Secure and fast end-to-end encrypted chatting software
- Allows one-to-one and multiple participants in a chat
- Speech to text
- Screen sharing
- White board

Requirements specification - types: FURPS+

- Usability - human factors, help, documentation



- Supports 100+ languages
- Comprehensive documentation of features
- Help

Requirements specification - types: FURPS+

- Reliability - frequency of failure, recoverability, predictability



- Should not crash
- Save chat history
- Warn users about network conditions prior and during chat

Requirements specification - types: FURPS+

- Performance - response time, throughput, accuracy, availability, resource usage



- Low memory usage
- Fast

Requirements specification - types: FURPS+

- Supportability - adaptability, maintainability, internationalization, configurability



- Users can configure look and functionality using the menu
- Bug reports can be made in the app, and development team responds adequately

Requirements specification - types: FURPS+

- + ancillary or sub-factors - implementation, interface, operations, packaging, legal



- C++, Javascript
- GPL 3.0 license

Prioritizing of requirements - MosCoW

- Must have
 - The app must have text, video chat functionalities
- Should have
 - The app should allow sharing of files
- Could have
 - The app could have support for 100+ languages
- Would be nice to have
 - The app could have option to change look and feel of GUI

Use cases

Text stories, used to discover and record requirements.

Use cases and user stories

User stories

- Simple, single sentence description of a user's task that involves using the system
- Usually made by the users themselves

Use cases

- More detailed text stories, involving steps
- Made by an expert

Guidelines for finding use cases

① Define system boundary:

- Software application? Hardware application?
- Does one user/organization use it?

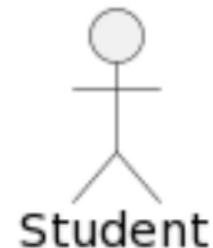
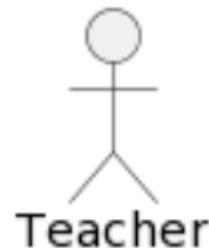
Drawing application

Guidelines for finding use cases

② Identify the primary actors and their goals:

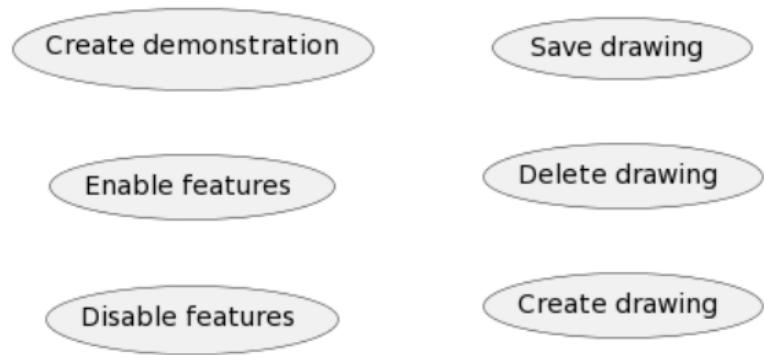
The system fulfills their goals

Actor	Goal
Teacher	<ul style="list-style-type: none">• Create a demonstration of drawing• Enable some features• Block some features• Save drawings• Delete drawings
Student	<ul style="list-style-type: none">• Create a drawing



Guidelines for finding use cases

- ③ Define use cases that satisfy user's goals:
name them according to their goal.



Formats of use cases

- Brief - one paragraph
- Casual - Multiple paragraphs
- Fully dressed - All steps in detail. Example: page 68 - 72 of the textbook.

Use case description

- Name - actor's goal
- ID - a unique number for identification
- Participating actors: Primary (starts the use case) and secondary (contributes)
- Pre-conditions - entry conditions
- Flow of events - Numbered list of actions
- Post-conditions - exit conditions

Requirements and use cases

- All requirements are traceable to use cases
- All use cases are linked to requirements

	Use cases				
		U1	U2	U3	U4
R1					
R2					
R3					
R4					
R5					

Requirements
Traceability
Matrix

Use case - Some details and an example

What is it?

- Describes an actor's task
 - Text description
 - Sequence of actions
 - Named (Verb + noun)
- Described from the actor's point of view
- Interaction with the system
- Have success and failure scenarios

UC1: Send message

Actor:User of chat app

Use case text:

- ① The user opens the app.
- ② He/she presses start chat button.
- ③ He/she selects the person(s) to whom the message is to be sent.
- ④ He/she types the message in the text box.
- ⑤ He/she presses send message button.
- ⑥ If the message has been received, user gets confirmation, else gets status report.

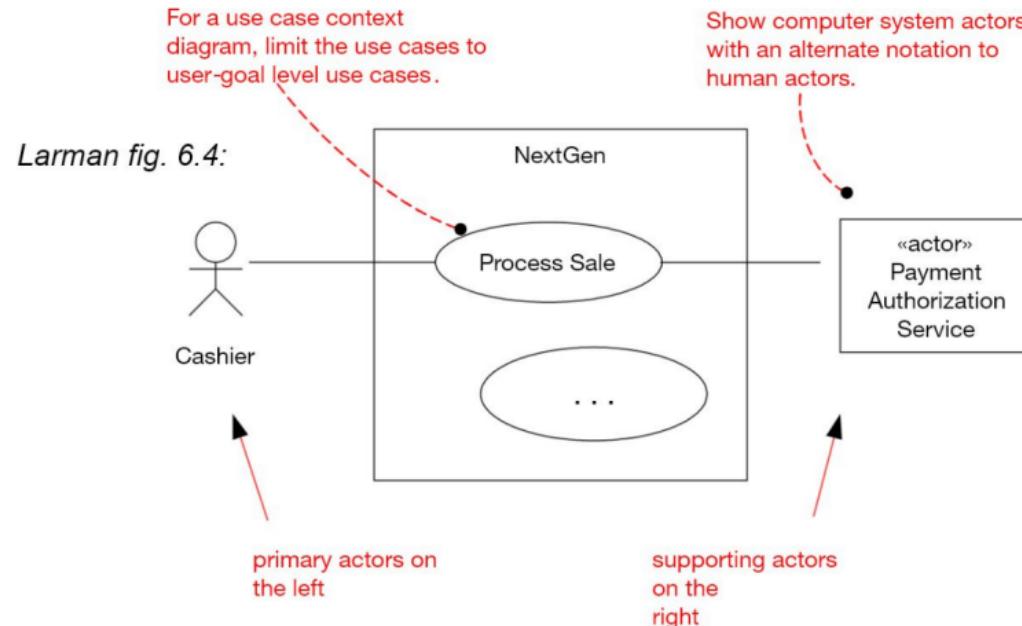
Use cases - characteristics

- Description of tasks
- Technology-independent! (Must be able to be implemented without IT)
- Describes **WHAT** the system does - NOT how

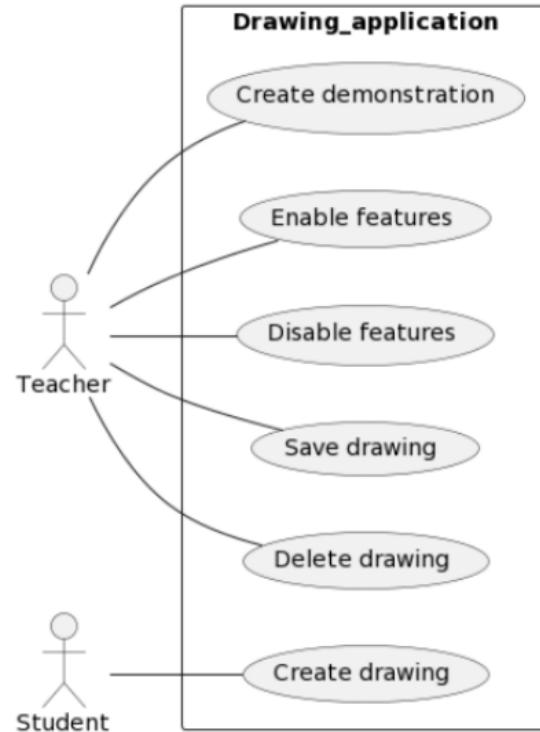
Use case diagram

It shows an overview of actors, use cases and their relationships.

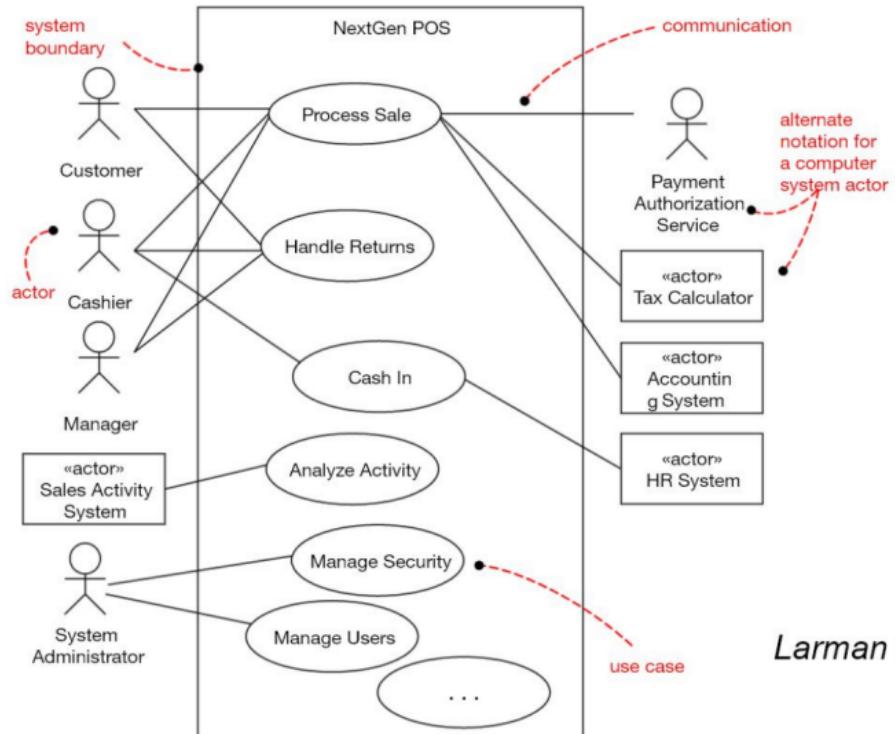
Use case diagram example



Use case diagram example



Use case diagram example



Larman fig. 6.3

Report submission

- Deadline: 15/09/2023 18:00

Report - CDIO 0

Create a report for the project with the following contents.

- Front page
- Table of Contents
- Division into subsections - headings
- Page numbers (also in any code appendix) – on all pages
- Bibliography
- Footnotes
- Naming figures

Report

Quality requirements:

- The text must be clear and legible with an appropriate margin
- Chapters are numbered 1, 2, 3, etc.
 - Each chapter is divided into several sections (1.1, 1.2, etc.). Not too many levels!
- Figures are numbered and named and have figure text.
 - Use chapter number followed by a consecutive number. In each chapter, the numbering starts from the beginning
- The bibliography is listed at the end.
 - Use the format recommended by DTU Library. (Typically sorted by first-mentioned author's surname - with publisher, year of publication and ISBN number)
- Remember to also indicate literature from web pages
- Beware of plagiarism!

Front page

- Subject
- Project name
- Group/team no.
- Delivery deadline
- Who is in the group:
 - Names
 - Study numbers
 - Image (portrait of the participant in question)



The front page template for DTU includes the following elements:

- DTU Logo:** The official logo of the Technical University of Denmark (DTU) is positioned at the top right.
- Phone Number:** The number 02312 02313 02315 is displayed below the logo.
- Course Information:** The text "INDLEDEDENDE PROGRAMMERING, UDVIKLINGSMETODER TIL IT-SYSTEMER OG VERSIONSSTYRING OG TESTMETODER" is centered above the CDIO 0 section.
- CDIO 0:** A horizontal line separates the course information from the participant details.
- Participant Portraits:** Six small portraits of participants are arranged in two rows of three. Each portrait includes the participant's name and study number.
 - Ian Bridgwood (8123456)
 - Inge-Lise Salomonsen (8234567)
 - Bjørn Høgh (8345678)
 - Maria Nyborg (8456789)
 - Sune Nielsen (8567890)
 - Daniel Rubin-Green (8678901)
- Date:** The date "18. september 2018" is located below the participant portraits.
- Logos:** Logos for "DTU Compute" and "DTU Diplom" are located at the bottom, along with their respective descriptions.

Contents

- Summary
- Hourly accounting
- Table of Contents
- Introduction
- Project planning
- Requirements/Analysis
- Design
- Implementation
- Testing
- Conclusion
- Appendix
 - Literature
 - Code

Exercises

- From 15:00 to 17:00, you will work on exercises given in Week2_exercises.pdf on Learn.
- TAs will assist you in these tasks.
- Rooms:
 - 358/holdlokal 065
 - 358/holdlokal 066
 - 358/holdlokal 070
 - 358/holdlokal 069
 - Or here (306/34)

Thank you!

62531 Development Methods for IT Systems: Lecture 3

Deena Francis

Assistant Professor,

Section for AI, mathematics and software,

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

Today (*Main theme: Detailed look into UML*)

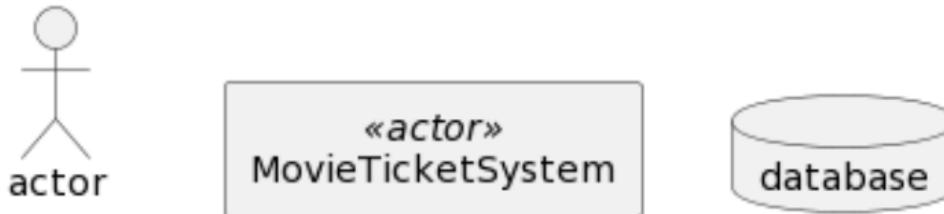
- Actors
- Use case diagrams
- Inheritance
- Include
- Extend

Learning objectives (Today)

- Identify actors and types of actors
- Understand when to use inheritance
- Understand and apply the «include», «extend» operations
- Identify and define alternative flows in use cases

Actors

- Users of the system
- Named after the role they play
- Involved in the use cases



Exercise

UC1: Get movie tickets

- ① User selects the movie
- ② User selects the number of persons
- ③ User selects date and time
- ④ User selects the seats
- ⑤ The system checks for errors and displays the selection
- ⑥ User completes payment and gets the ticket

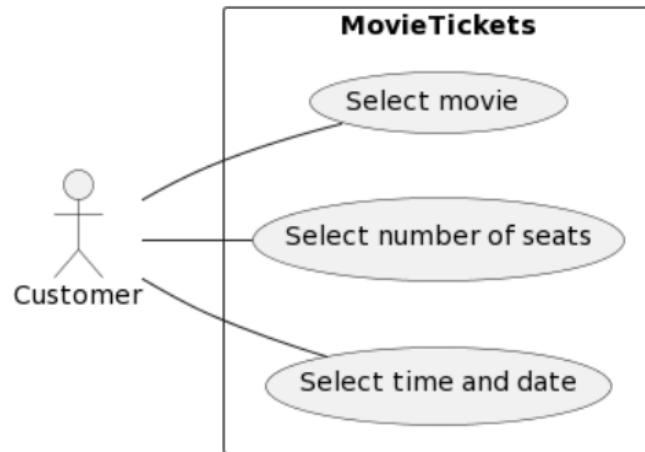
Identify the actors

Actors - types

- Human / non-human
- Primary / secondary
- Active / passive

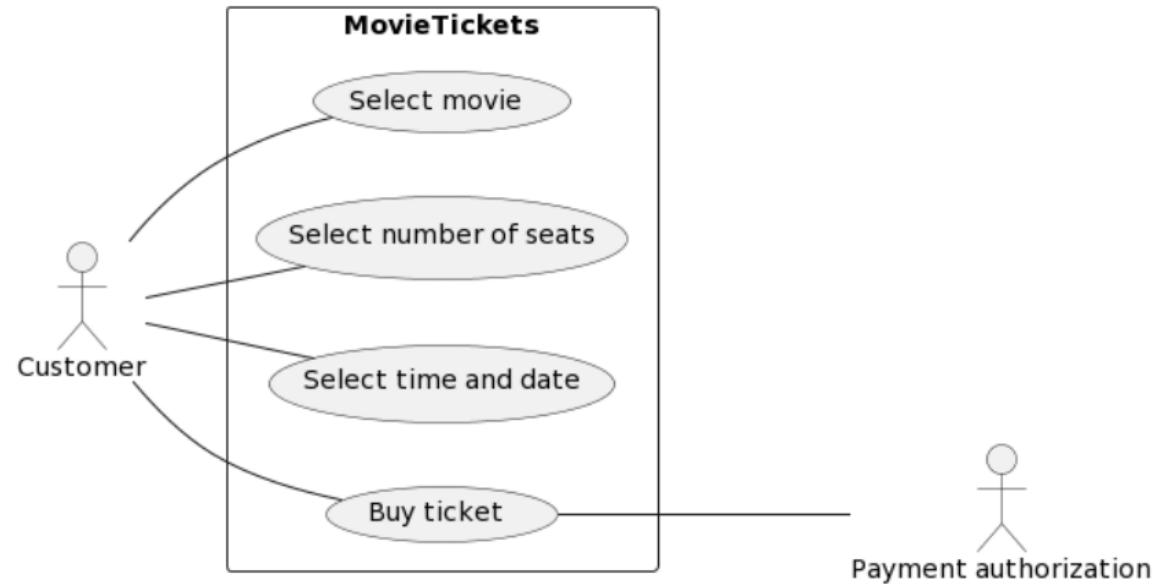
Primary actors

- Starts the use case
- Goals of this actor are met by the execution of the use case



Secondary actors

- Never initiates the use case
- Supplies / receives information
- Does not necessarily benefit from the use case

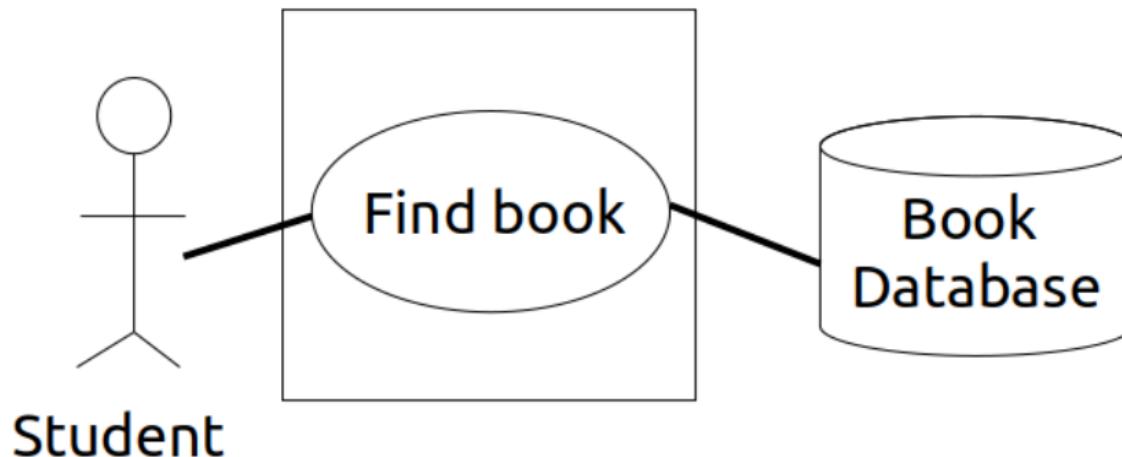


Active actors

- Uses the system
- Examples: User, Doctor, Customer, ...

Passive actors

- Used by the system
- Example: Payment gateway, Databases, ...

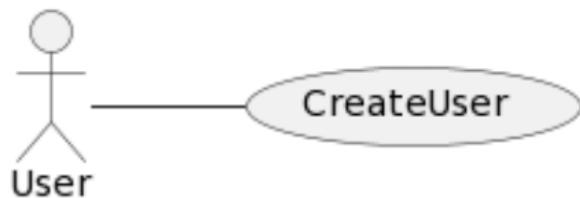


Relationships in use case models

- Actors and use cases
- Actors and actors
- Use cases and use cases

Relationship between actors and use cases

- Association



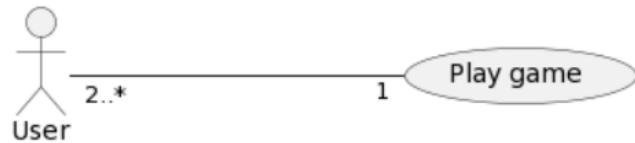
Association

Two-way connection between actors and use cases

- Represented by a line connecting actor and use case

Multiplicity

Denotes the number of elements.



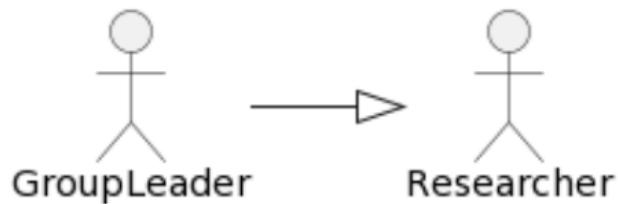
Two or more users play a game.

Multiplicity

Symbol	Meaning
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..5	1 to 5

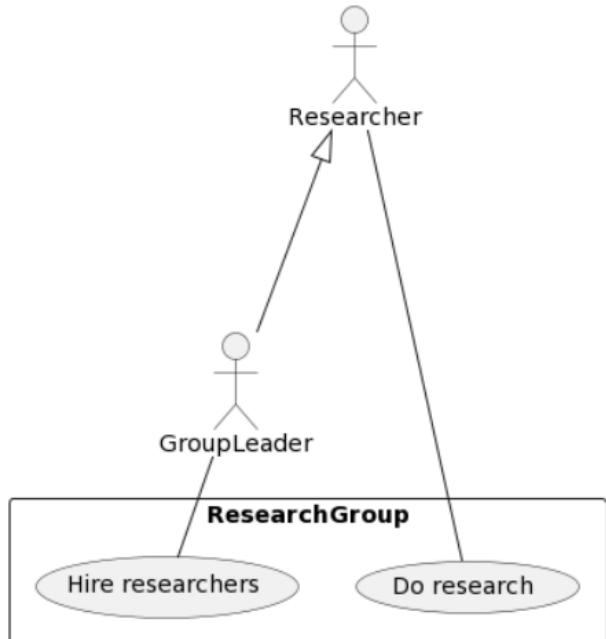
Relationship between actors

- Generalization or inheritance



Inheritance

- Some actors (e.g. GroupLeader) have extended privileges
- GroupLeader is also a Researcher
- Use cases of Researcher are inherited by GroupLeader

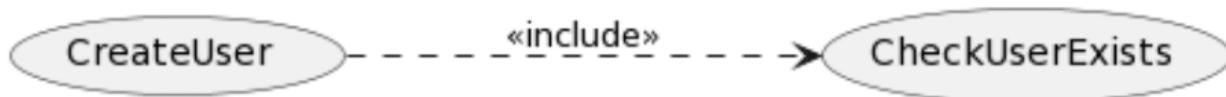


Relationships between use cases

- Include
- Extend
- Inheritance

Include relationship

It is used when a use case contains a 'sub-use case'



Include relationship

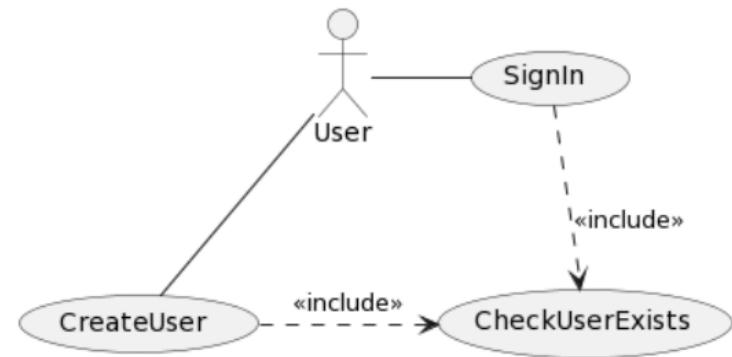
When is it used?

- When the use cases are complex

Include relationship

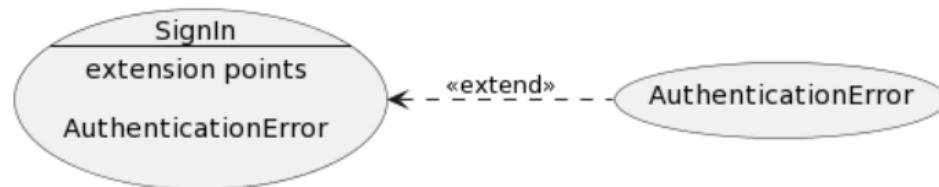
UML notation:

- dashed arrow to the sub-use case with «include»



Extend relationship

It is used when one use case extends the behavior of another use case.

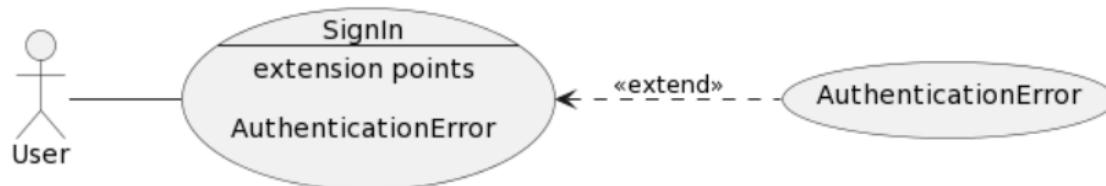


Added at extension points: points in the behavior of the use case where a behavior can be extended by some other (extending) use case

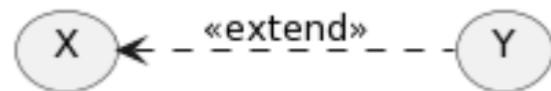
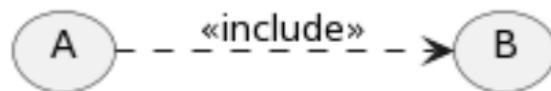
Extend relationship

UML notation:

- dashed arrow from the extending use case with «extend»



Include and Extend



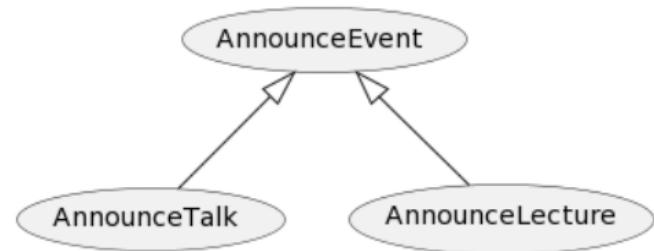
Include	Extend
B is required by A (Mandatory)	Y adds extra functionality to X (Optional)

Inheritance or generalization

- Common behavior can be drawn out in a general / abstract use case

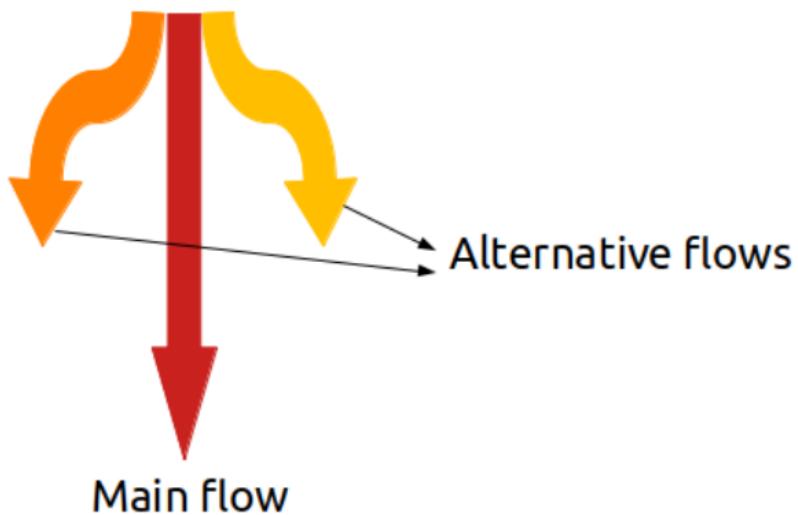
UML notation:

- Inheritance arrow



Flows in a use case

- Main flow
- Alternative flow



Alternative flows in use cases

Name:	Buy movie tickets
Short description:	A user buys one or more movie tickets.
Precondition:	User selects buy tickets option from menu
Postcondition:	One or more movie tickets are bought
Error situations:	There are no tickets available
System state in the event of error:	User does not get a ticket
Actors:	User, MovieTicketSystem
Trigger:	User wants to buy ticket(s)

Continued ...

Alternative flows in use cases

Standard process	<ol style="list-style-type: none">① User selects the movie② User selects the number of persons③ User selects date and time④ User selects the seats⑤ User completes payment
Alternative process	<ol style="list-style-type: none">④a The User's selected seats are not available④b MovieTicketSystem provides alternative seats④c User selects the alternative seats and completes payment

Branching in use cases - *If*

Use case: Select seat
Brief description: User selects one or more seats in a movie theatre
Primary actors: User
Preconditions: User has selected date, time and movie
Main flow:
<p>① User selects number of seats and their locations</p> <p>② If User selects "change seat locations":</p> <ul style="list-style-type: none">②a MovieTicketSystem displays new list of available seats②b User selects new seats <p>③ If User selects "add seats"</p> <ul style="list-style-type: none">③a MovieTicketSystem displays available seats③b User selects new seats
Postconditions: None
Alternative flows: None

For each in use cases

Use case: SelectMovie

Main flow:

- ① MovieTicketSystem displays a list of movies currently playing
- ② User selects movie(s)
- ③ For each selected movie:
 - ③a MovieTicketSystem displays a thumbnail sketch
 - ③b MovieTicketSystem displays summary of movie
 - ③c MovieTicketSystem displays the price for one ticket

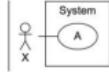
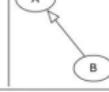
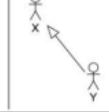
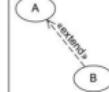
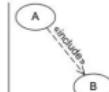
Alternative flows - *While*

Usecase: CreateNewCustomerAccount
ID: 5
Brief description: The system creates a new user for the account
Primary actors: Customer
Main flow: <ul style="list-style-type: none">① Customer selects "create new account"② While the Customer details are invalid<ul style="list-style-type: none">②a Systems asks Customer to enter details again for confirmation②b System validates the Customer's details③ System creates a new account for the Customer
Postconditions: A new account is created for the Customer
Alternative flows: InvalidEmailAddress InvalidPassword Cancel

Alternative flows in a separate use case

Alternative flow: CreateNewCustomerAccount: InvalidEmailAddress
ID: 5.1
Brief description: The System informs the Customer that an invalid email address has been entered.
Primary actors: Customer
Secondary actors: None
Preconditions: The Customer has entered an invalid email address
Alternative flow: (<i>The alternative flow begins after Step 2b of the main flow</i>)
① System informs Customer that an invalid email address has been entered
Postconditions: None

Summary

Name	Notation	Description
System		Boundaries between the system and the users of the system
Use case		Unit of functionality of the system
Actor		Role of the users of the system
Association		X participates in the execution of A
Generalization (use case)		B inherits all properties and the entire behavior of A
Generalization (actor)		Y inherits from X; Y participates in all use cases in which X participates
Extend relationship		B extends A: optional incorporation of use case B into use case A
Include relationship		A includes B: required incorporation of use case B into use case A

Thank you!

62531 Development Methods for IT Systems: Lecture 4

Deena Francis

Assistant Professor,

Section for AI, mathematics and software,

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

$\Delta \int_a^b \Theta + \Omega \delta e^{i\pi} = -1$

$\infty = \{2.7182818284\}$

$\chi^2 > 000$

$\Sigma!$

Today (*Main theme: Elaboration phase, introduction to risk management*)

- Other requirements
- Introduction to risk management
- Inception to elaboration
- Introduction to domain models

Learning objectives (Today)

- Describe other requirements in a software development process
- Identify, estimate and prioritize risks for a given project
- Describe the process of transitioning from inception to elaboration
- Create a basic domain model

Other requirements

Requirements that are not captured in the use cases.

- Glossary
- Supplementary specification
- Vision
- Business rules

Glossary

It documents all the terms in your project.

Why?

- Every stakeholder and members of your team understand and agree on terms used.

Glossary

Glossary

Algorithm: The steps required to perform a specific operation.

Database: Collection of data collected in the project.

Graphical User Interface (GUI): a digital interface in which a user interacts with graphical components such as icons, buttons, and menus

.

.

.

Supplementary specification

It is a document that will contain textual description of all other requirements.

- Reports, documentation, packaging, supportability.

Example: page 84 of the textbook

Business rules

They are statements that provide the criteria and conditions for making a decision.

Example: **Net sale** is defined as the total sales price of an order before discounts, allowances, shipping, and other charges.

They are not part of the software requirements!

Risk management

Risks

Risk = Probability of an adverse circumstance.

- Project risks
- Product risks
- Business risks

Project risks

It affects the project schedule or resources.

Examples:

- Loss of an experienced developer
- Delays in obtaining hardware components

Product risks

It affects the quality or performance of the software being developed.

Examples:

- Failure of a purchased component to perform as expected
- A part of the system does not work as planned

Business risks

It affects the organization developing or procuring the software.

Examples:

- A competitor introducing a new product

Risks

- Project risks
- Product risks
- Business risks

The risk types can overlap!

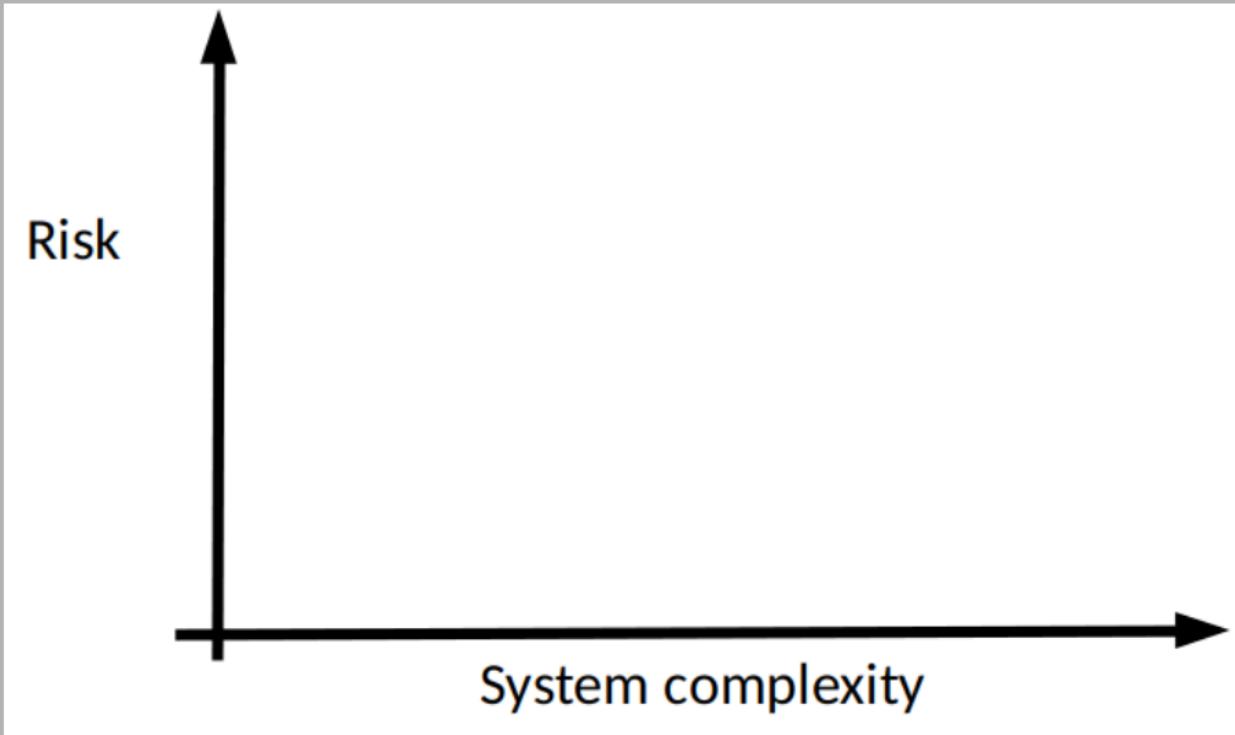
Common risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave project before it is finished
Management change	Project	There will be changes in company management with different priorities
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule
Requirements change	Project and product	There will be large number of changes to the requirements than anticipated

Common risks

Risk	Affects	Description
Specification delays	Project and product	Specifications and essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated
Software tool and underperformance	Product	Software tools that support the product do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is supervised by new technology
Product competition	Business	A competitive product is marketed before the system is completed

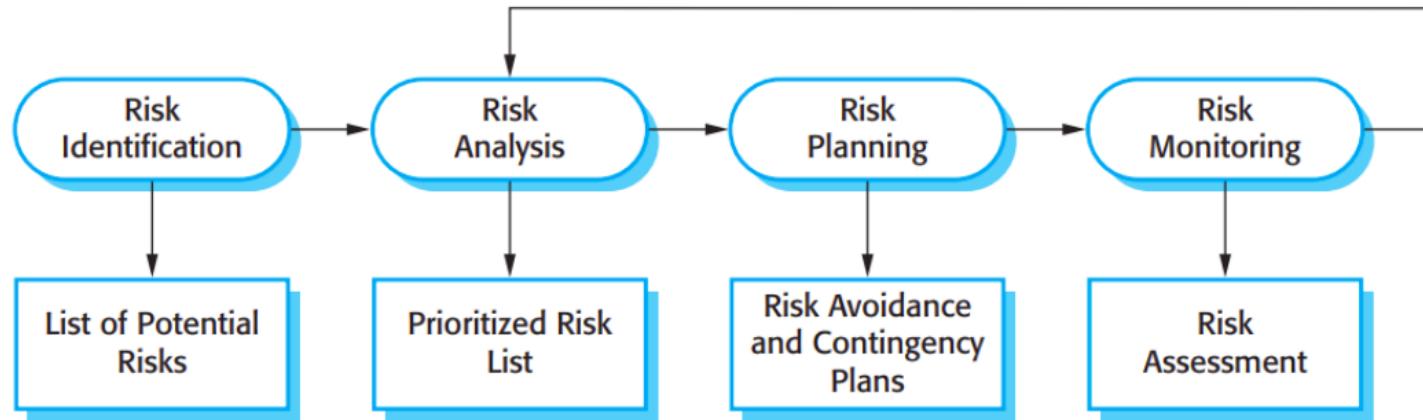
Exercise - What do you think this graph will be like?



Stages of risk management

- Risk identification
 - Identify possible project, product, and business risks
- Risk analysis
 - Assess the likelihood and consequences of risks
- Risk planning
 - Make plans to avoid or minimize risks
- Risk monitoring
 - Monitor the risks throughout the project

Risk management phases



Reference: [Sommerville(2011)].

Risk identification

Risk type	Possible risks
Estimation	<ul style="list-style-type: none">① The time required to develop software is underestimated② The rate of defect repair is underestimated③ The size of the software is underestimated
Organizational	<ul style="list-style-type: none">④ The organization is restructured so that different managements are responsible for the project⑤ Organizational financial problems force reductions in project budget

Risk identification

Risk type	Possible risks
People	<ul style="list-style-type: none"><li data-bbox="518 343 1318 384">⑥ It is impossible to recruit staff with required skills<li data-bbox="518 415 1318 456">⑦ Key staff are ill and unavailable at critical times<li data-bbox="518 486 1209 517">⑧ Required training for staff is not available
Requirements	<ul style="list-style-type: none"><li data-bbox="518 609 1500 691">⑨ Changes to requirements that require major design rework are proposed<li data-bbox="518 722 1555 763">⑩ Customers fail to understand the impact of requirements changes

Risk identification

Risk type	Possible risks
Technology	<ul style="list-style-type: none">⑪ The database used in the system cannot process as many transactions per second as expected⑫ Faults in reusable software components have to be repaired before these components are reused
Tools	<ul style="list-style-type: none">⑬ The code generated by software code generation tools is inefficient⑭ Software tools cannot work together in an integrated way

Risk analysis

Consider each identified risk and make a judgment about the probability and seriousness of that risk.

Not easy, use your own judgement and experience!

Risk analysis

- Probability:
 - Very low (10%)
 - Low (10–25%)
 - Moderate (25–50%)
 - High (50–75%)
 - Very high (>75%)
- Impact:
 - Catastrophic (threaten the survival of the project)
 - Serious (would cause major delays)
 - Tolerable (delays are within allowed contingency)
 - Insignificant

Risk analysis (ordered by effects)

Risk	Probability	Effect
Organizational financial problems force reductions in project budget	Low	Catastrophic
It is impossible to recruit staff with the skills required	High	Catastrophic
Key staff are ill and unavailable at critical times	Moderate	Serious
.		
.		
.		
The size of the software is underestimated	Moderate	Tolerable
Code generated by code generation tool is inefficient	Moderate	Insignificant

Risk analysis - Probability (likelihood) * Impact matrix

Risk Rating = Likelihood x Severity



Risk strategies / planning

- Transfer
 - Pass on responsibility
- Accept
 - Take the chance
- Mitigate
 - Minimize consequences
- Eliminate
 - Eliminate (remove) the problem

Risk planning

- Avoidance strategies: ways to avoid the risks
- Minimization strategies: ways to reduce the possible damage
- Contingency plans: ways to prepare for the worst

Risk monitoring

It is the process of checking that your assumptions about the product, process, and business risks have not changed.

Monitor risks regularly at all stages in a project!

Risk monitoring

- Risk becoming more or less probable?
- Effects of the risk have changed?

Risk monitoring: Assessing risk types - Indicators

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects
Organizational	Organizational gossip; lack of action by senior management
People	Poor staff morale; poor relationships among team members; high staff turnover
Requirements	Many requirements change requests; customer complaints
Technology	Late delivery of hardware or support software; many reported technology problems
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, etc.

Inception to elaboration

Inception

- ~ 1-2 weeks
- Requirements-oriented artifacts: short + brief
- Determine feasibility, risk, scope of project

Inception - activities, artifacts

- Most actors, goals and use cases named
- Most use cases written in brief format 10-20% use cases written in detail
- Most influential and risky quality requirements identified
- Version 1: Vision and supplementary specifications written
- Risk list
- User interface oriented prototypes, technical proof-of-concept prototypes
- Plan for first (elaboration) iteration

Elaboration

Build the core architecture, resolve high risk elements, define most requirements and estimate the overall schedule and resources.

Elaboration - time frame

2-4 iterations: each 2-6 weeks

Elaboration - artifacts

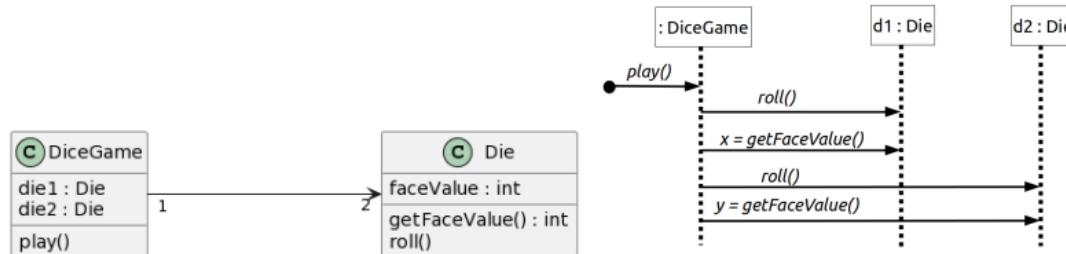
Domain model

Visualization of domain concepts.



Design model

Visualization of the logical design - class diagrams, sequence diagrams, package diagrams ...



Software architecture document

Summary of key architectural issues and their resolution in design.

When elaboration goes bad

- It is more than a few months long
- It has only one iteration
- All requirements defined before elaboration
- Risky elements not tackled
- Core architecture missing
- No executable architecture
 - there is no production code programming
- It is a requirements or design phase
 - preceding an implementation phase in construction

Best practices

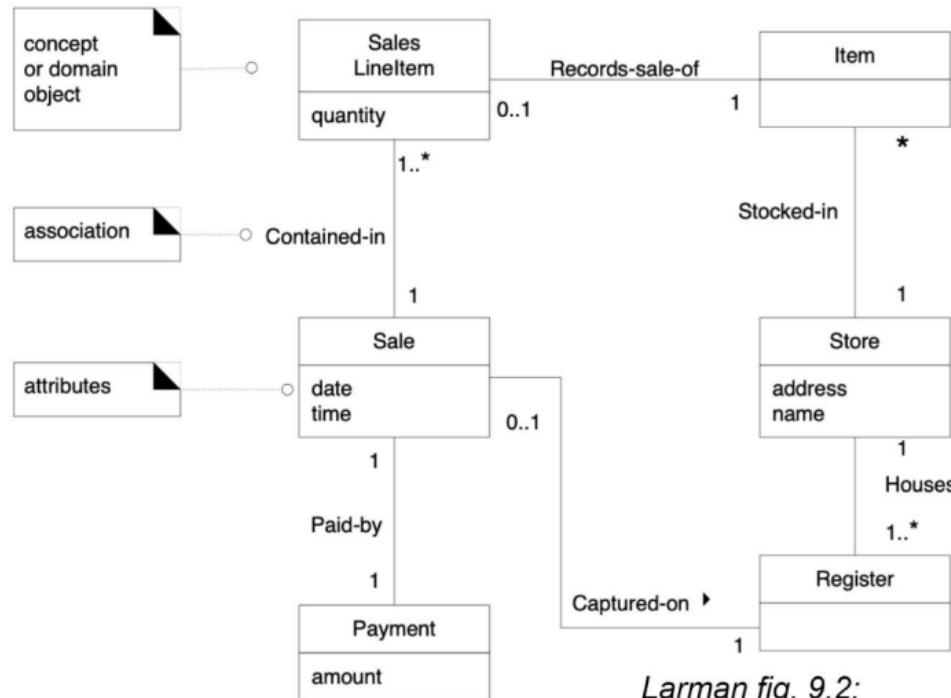
- Short iterations - fast deadline
- Start programming early
 - But remember to design before you code!
- Design and plan testing of risky elements
 - Adapt the design based on tests!
- Early test
 - Test driven development
- Feedback from tests, users and developers

Domain models

Capturing the domain

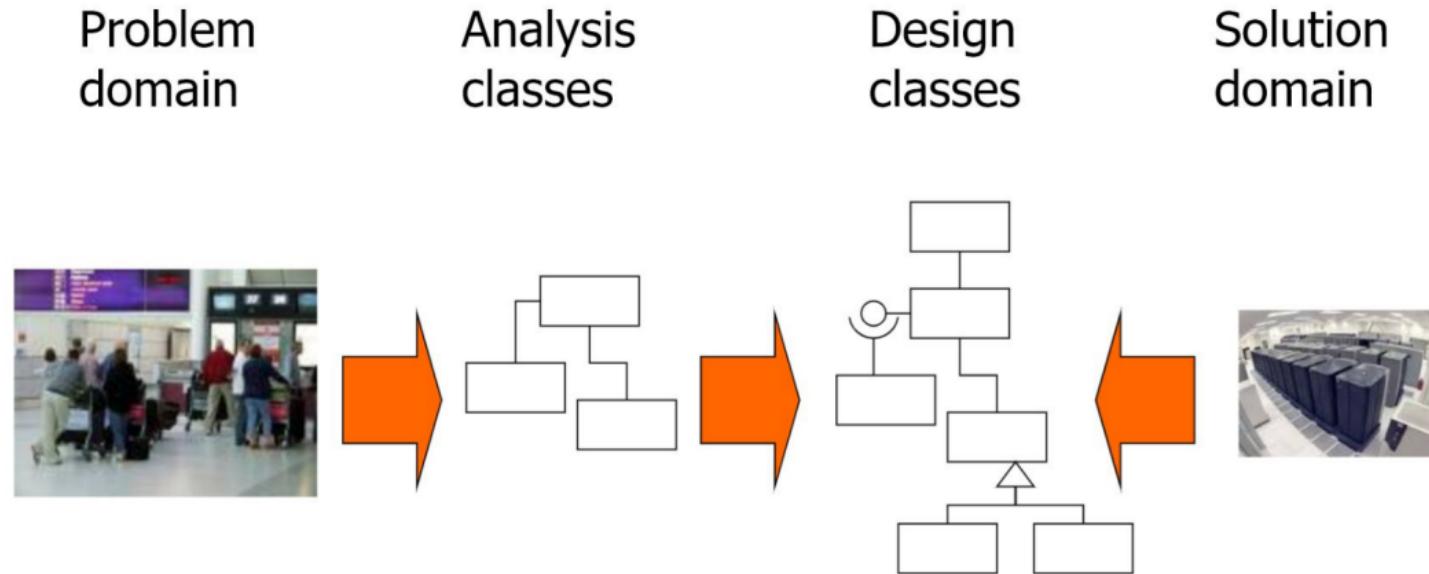
- Use an existing model and modify it
- Use a category list
 - Look at Figure 9.1 of the textbook
- Noun analysis
 - Describe the domain with text - identify nouns that are domain classes
 - Verbs can become use cases

Domain model example



Larman fig. 9.2:

Why use domain models?



Thank you!

References

 Ian Sommerville.

Software Engineering, 9/E.

Pearson Education India, 2011.

62531 Development Methods for IT Systems: Lecture 5

Deena Francis

Assistant Professor,

Section for AI, mathematics and software,

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

$\Delta \int_a^b \Theta + \Omega \delta e^{i\pi} = -1$

$\Sigma!$

∞ χ^2 \gg \approx

$\{2.7182818284\}$ \circ λ

$\text{πφερτυθιοπσδφγηξκλ}$

Today (*Main theme: Analysis classes, sequence diagrams*)

- Objects and classes
- Analysis classes
- Sequence diagrams
- Packages

Learning objectives (Today)

- Identify and create classes
- Develop class diagrams
- Develop sequence diagrams

Ways of solving a problem - programming approaches

Structured programming

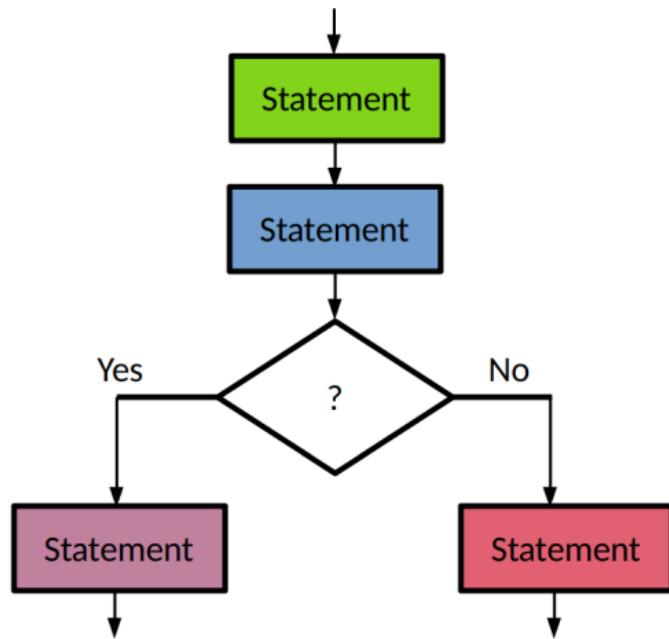
- Control structures: sequence of statements, loops, conditions (if-then-else, case)
- Subroutines: functions, methods

Object oriented programming

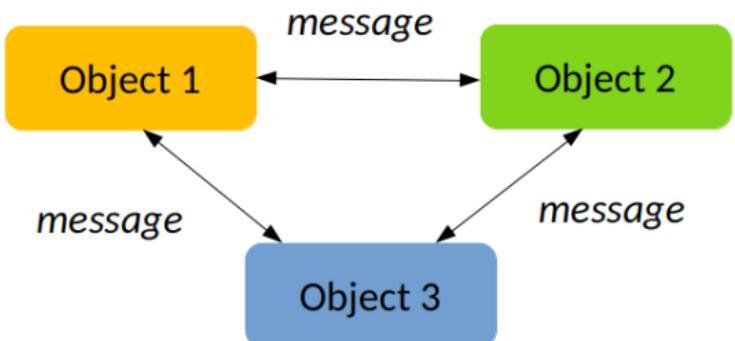
- Objects
- Interaction between objects

Ways of solving a problem - programming approaches

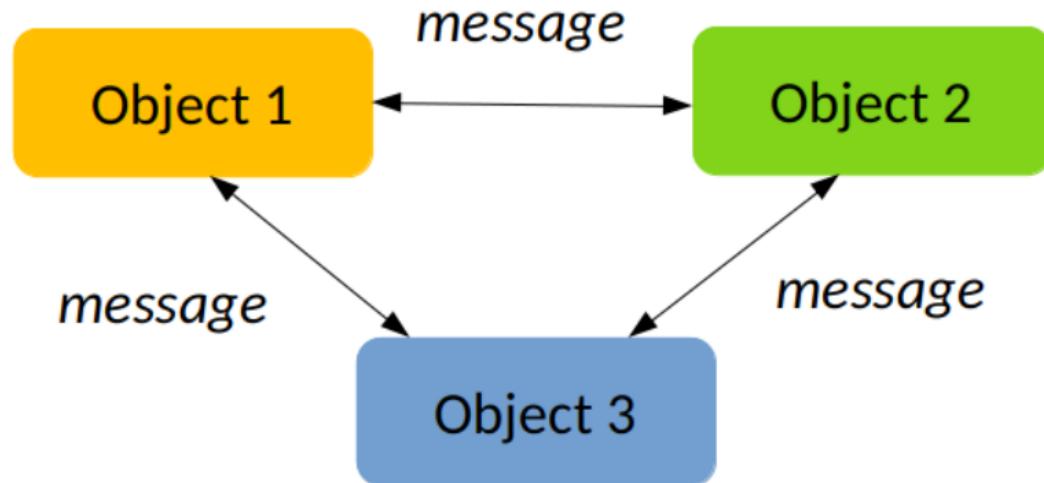
Structured programming



Object oriented programming



Object oriented programming

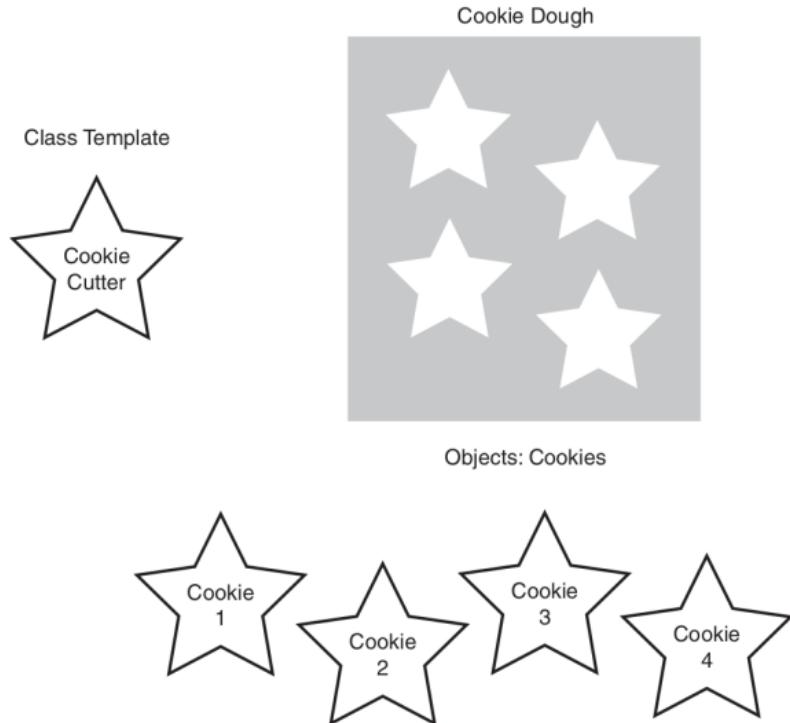


Objects

- *Variables*: contain the state of the object
- *Methods*: define the behavior of the object

Classes

They are blueprints for the objects.



Classes

- A sort of high-level data type

Integer and floating type variables

```
int x;  
float a;
```

Class variable i.e., object

```
myClass object1;
```

Objects

They are instances of the classes.

Object creation

```
myClass object1;
```

Naming conventions

Classes

- Capitalized first letter
- Noun
- Singular

Example: class MovieTicketSystem {}

Objects

- Lower case - first letter, just like a variable

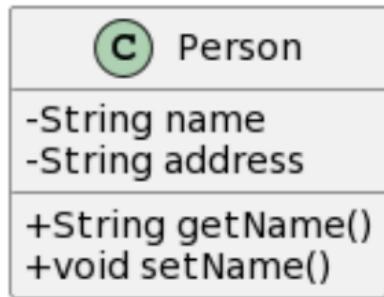
Example: MovieTicketSystem movieTicketSystem1;

Class

Example

```
public class Person{  
    //Attributes  
    private String name;  
    private String address;  
    //Methods  
    public String getName(){  
        return name;  
    }  
    public void setName(String n){  
        name = n;  
    }  
}
```

Classes in UML



Analysis classes

- Consists of data and behavior
- Software implementation details are not specified



Identifying analysis classes

We would like to have a system that can manage our badminton courts. We have players and coaches who can book the courts. There are fixed intervals you can book courts in. The coaches can book all courts in a hall. The coaches book first. Players can book the remaining courts.

Identifying analysis classes

We would like to have a system that can manage our **badminton courts**. We have **players** and **coaches** who can book the courts. There are fixed intervals you can book courts in. The coaches can book all courts in a **hall**. The coaches book first. Players can book the remaining courts.

- Identify the nouns: Court, Player, Coach, Hall

Design classes

- Consists of data and behavior
- Software implementation details are specified

Design classes

- Attributes
- Types
 - int, char, float, double, boolean, String, ...
- Visibility
 - + Public
 - - Private
- Methods

C	BankAccount
-name:	String
-number:	String
-balance:	double
+BankAccount	(name:String, number:String)
+deposit(m:double)	:void
+withdraw(m:double)	:boolean
+calculateInterest()	:double
+getName()	:String
+setName(n:String)	:void
+getAddress()	:String
+setAddress(a:String)	:void
+getBalance()	:double

Noun analysis

- Starting point is the relevant documentation
 - Vision
 - Use cases
 - Other requirements
 - Glossary
- Prepare a list of nouns
 - These are the candidates for classes and attributes
- Prepare a list of verbs
 - These are the areas of responsibility or use cases
- Select classes
 - Assign attributes
 - Assign Responsibilities

The Class, Responsibility, Collaborators (CRC) card

A brainstorming tool in object-oriented software.

- Nouns become classes
- Verbs become responsibilities of the classes
- Collaborators become the other cards that this class interacts with

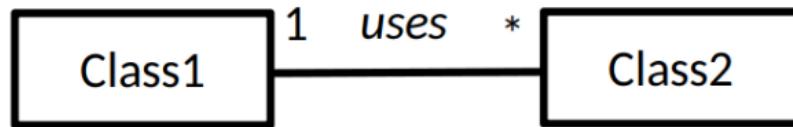
Class name: BankAccount	
Responsibilities	Collaborators
Maintain balance	Bank

Things that the
class does

Things the class
works with

Associations

- Connections between classes
- Relationship between objects
- Specify:
 - Line
 - Name
 - Multiplicity

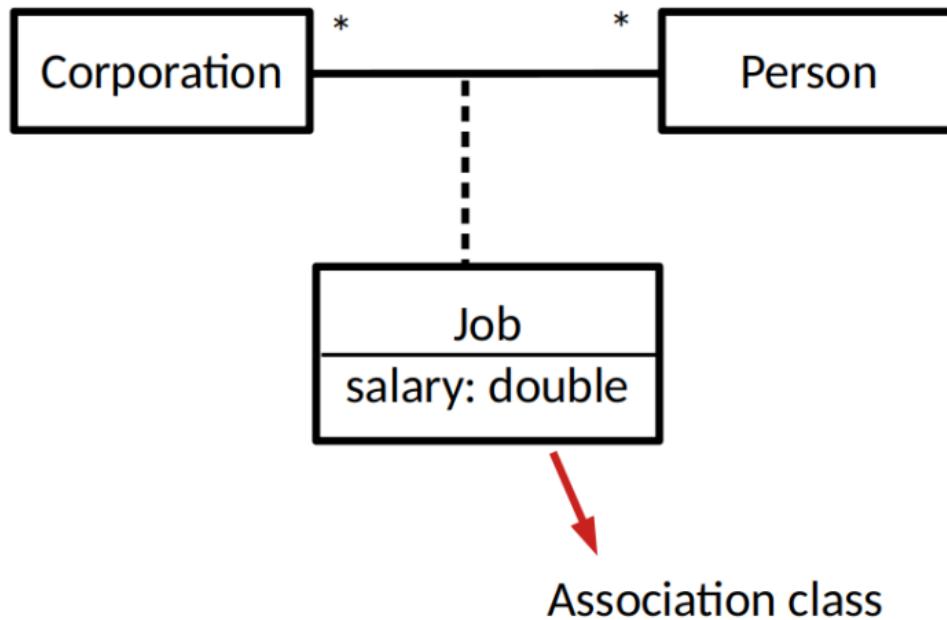


Multiplicity

Symbol	Meaning
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..5	1 to 5

Association classes

They are part of the association between two other classes.



Association classes

- Additional information about the association
- They are classes that contain:
 - Attributes
 - Methods
 - Associations

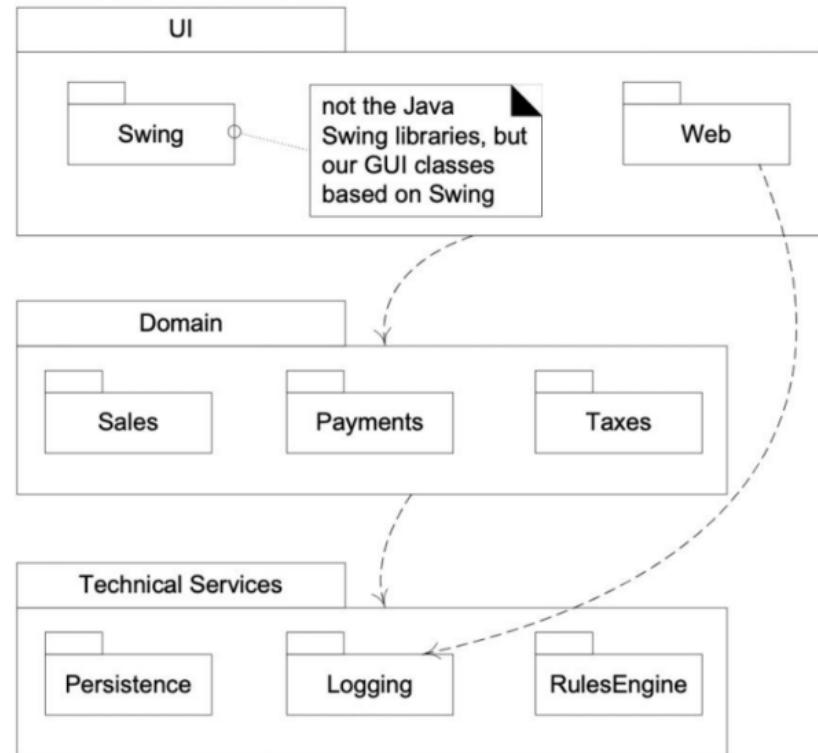
Packages

- Logical grouping of model elements such as classes
- Can show logical architecture of the system



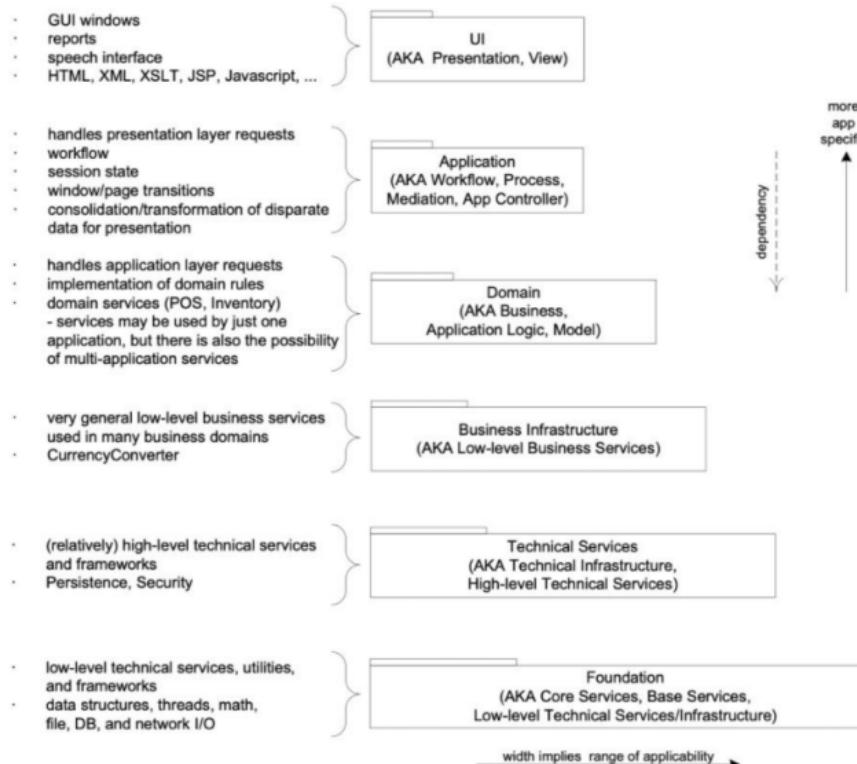
Packages - layered architecture

Larman Fig. 13.2



Packages - multi-layered architecture

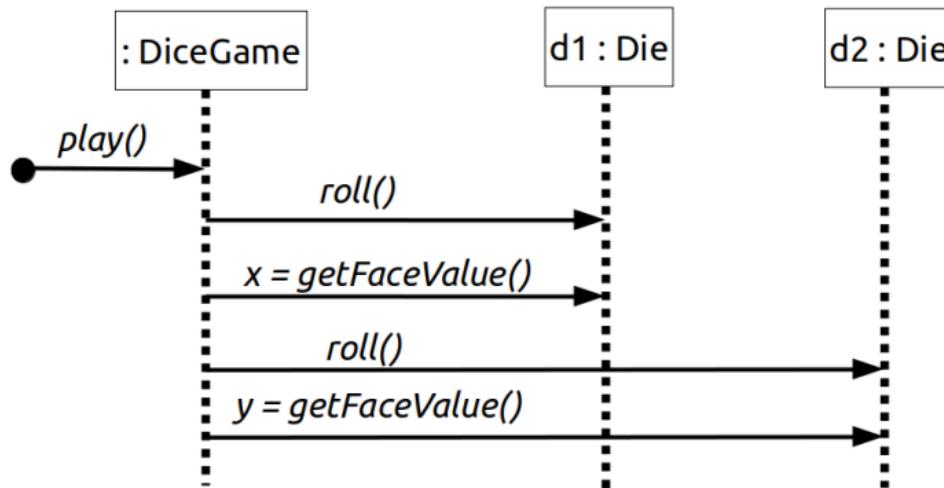
Larman Fig. 13.4



System sequence diagrams

Sequence diagram

Captures the course of events within a use case.

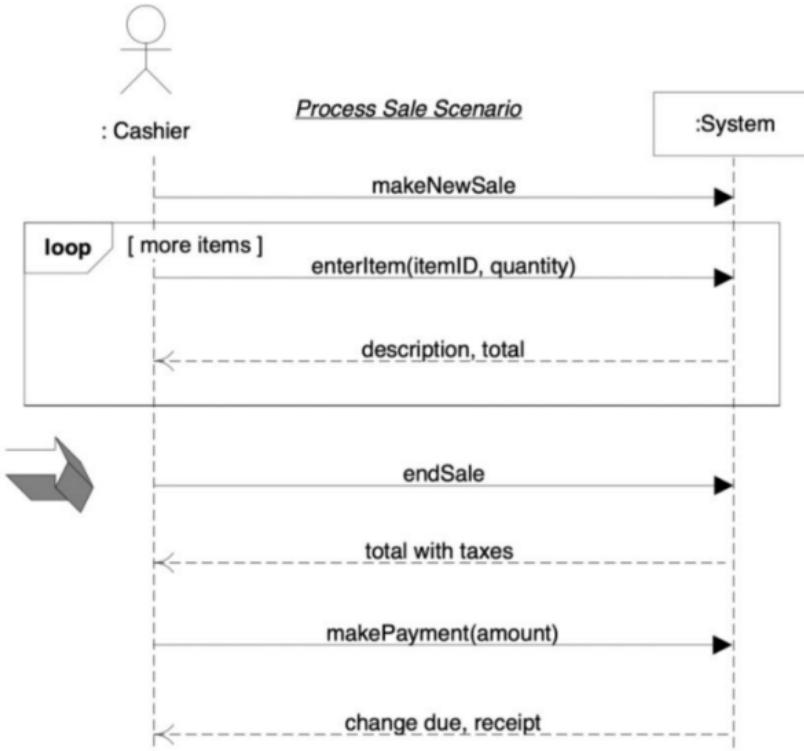


Sequence diagram: example

- Larman Fig. 10.3

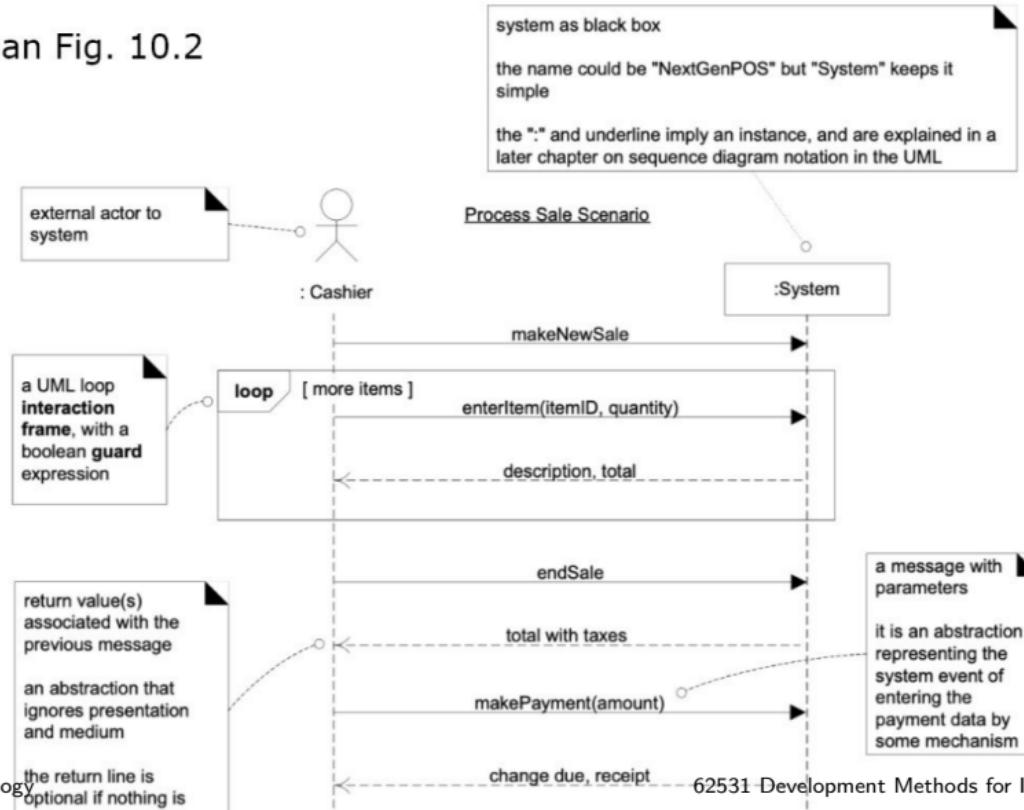
Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
- Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
- ...



Sequence diagram: example

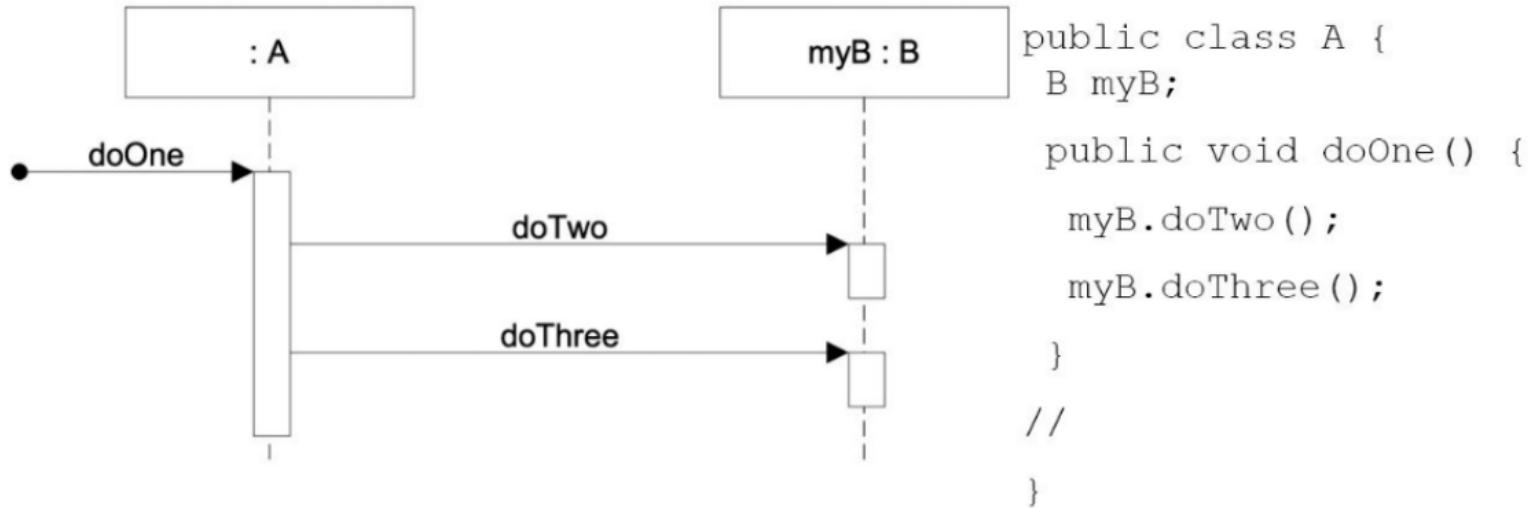
Larman Fig. 10.2



Why sequence diagrams?

- Tasks and their order
- Visual representation of the use case with the sequence of tasks

Sequence diagrams



The rectangular boxes (called activation boxes) show that processes are being performed in response to the message.

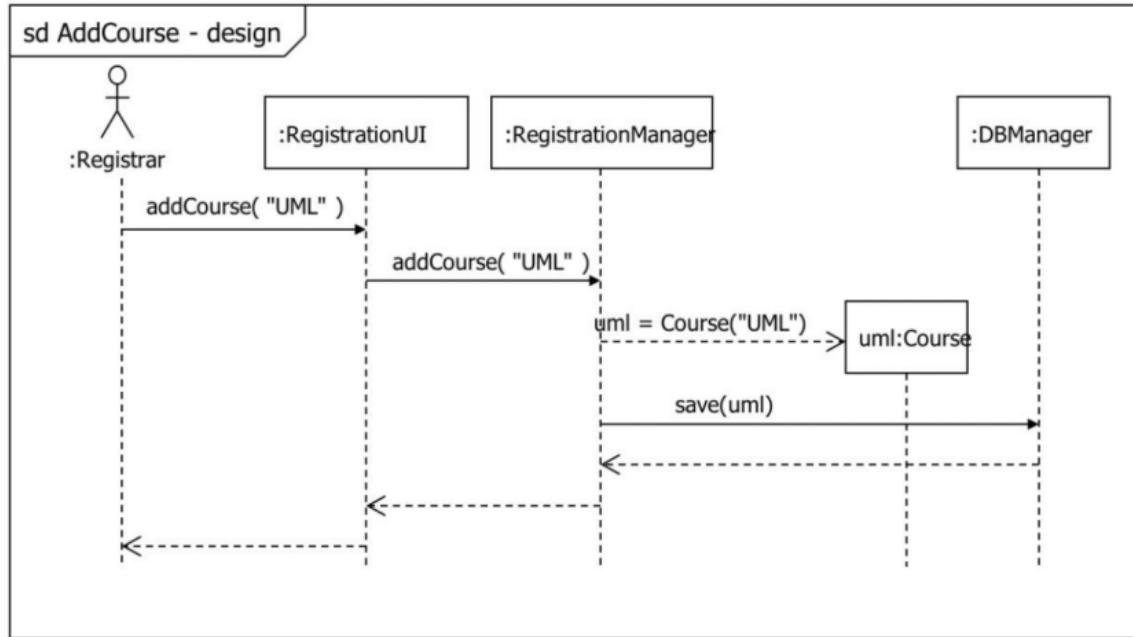
Sequence diagrams

<https://www.uml-diagrams.org/sequence-diagrams.html>

Example

Use case: AddCourse
ID: 8
Brief description: Add details of a new course to the system.
Primary actors: Registrar
Secondary actors: None
Preconditions: 1.The registrar has logged on to the system.
Main flow: 1.The Registrar selects “addCourse”. 2.The Registrar enters the name of the new course. 3.The system creates the new course.
Postconditions: 1. A new course has been added to the system.
Alternative flows: CourseAlreadyExists

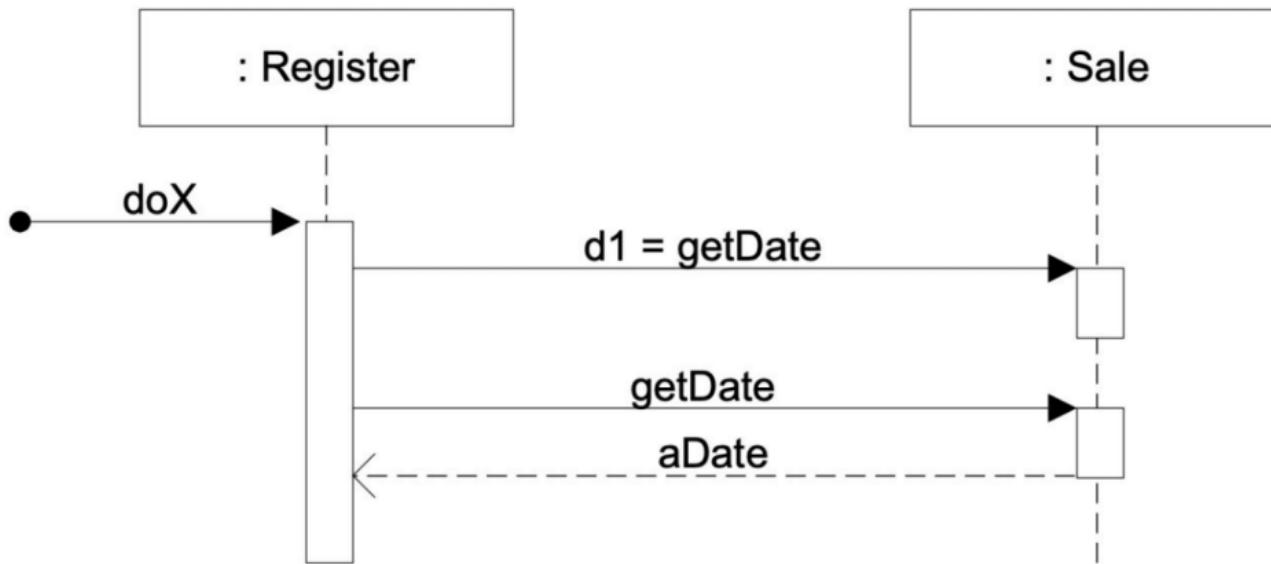
Example



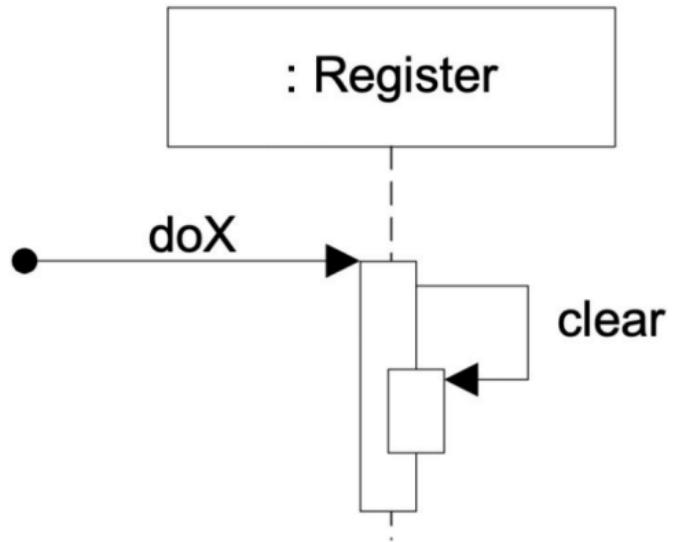
Example



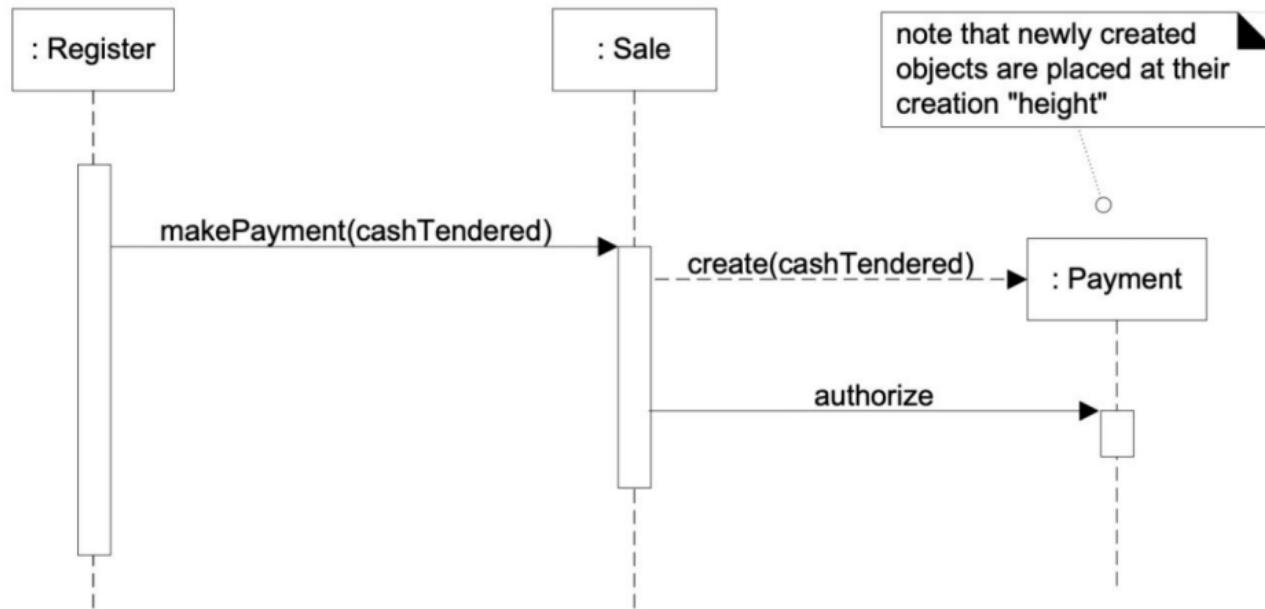
Call - return



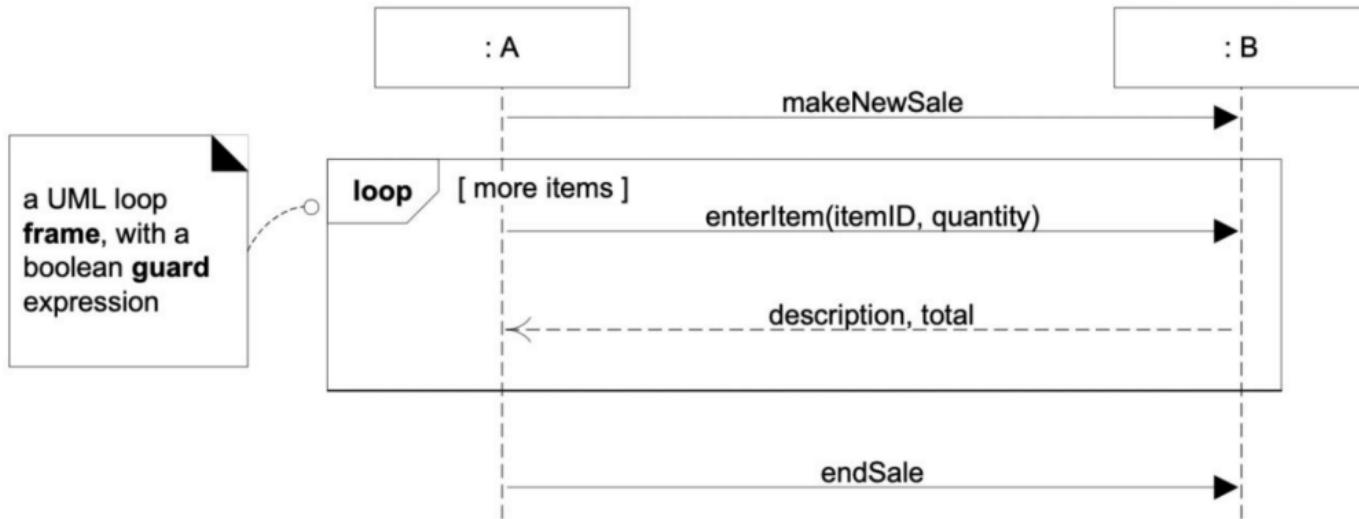
Call self



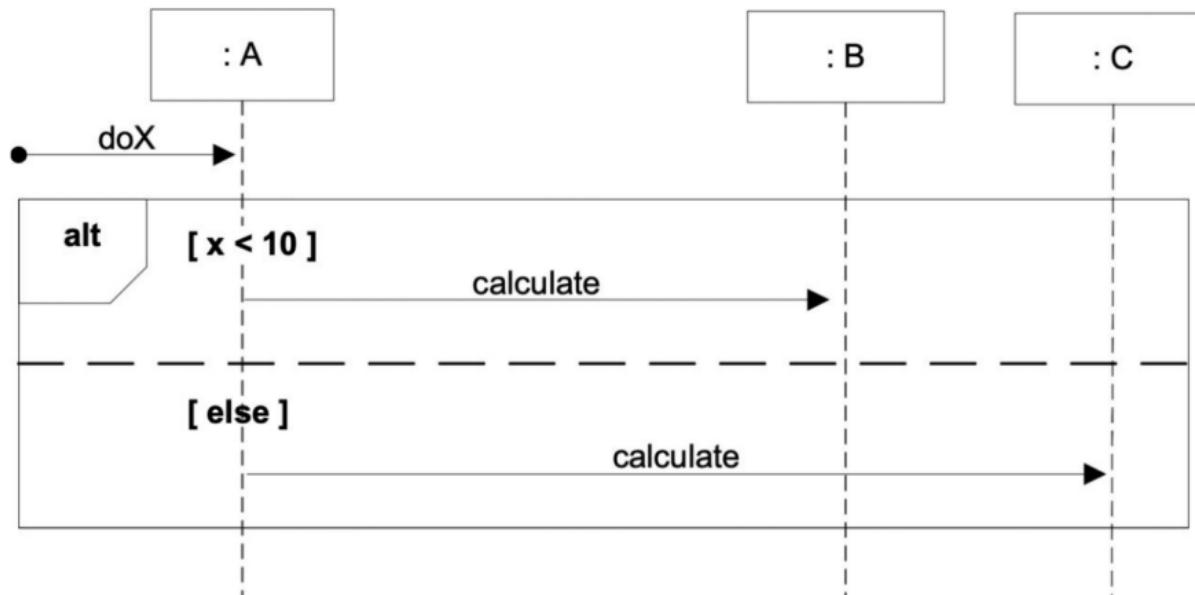
«create»



Loop



Branching



Full example

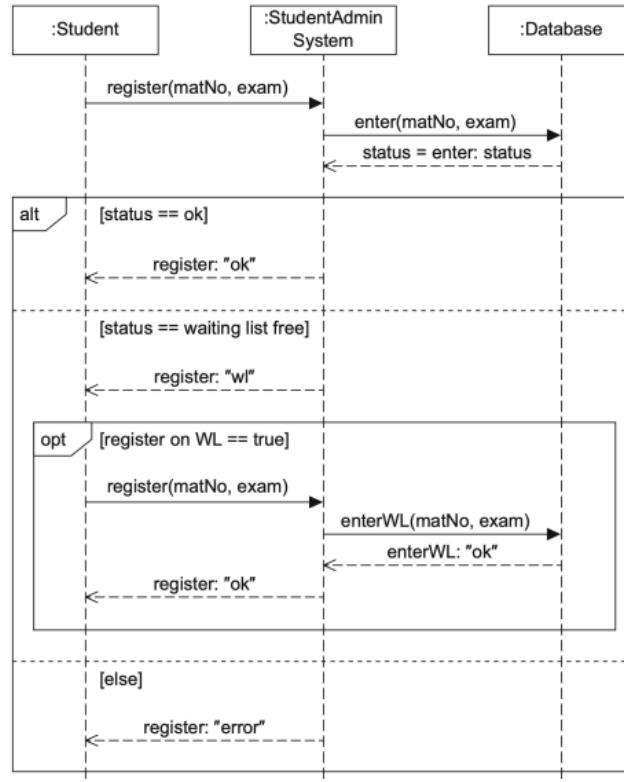


Figure 6.10
Example of an **alt** and an **opt** fragment

Report - CDIO 1

- Deadline 29 September 2023, 18:00

Thank you!

62531 Development Methods for IT Systems: Lecture 6

Deena Francis

Assistant Professor,

Section for AI, mathematics and software.

DTU Engineering Technology,

Technical University of Denmark (DTU)

email: dfra@dtu.dk

DTU Engineering Technology

$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$

Today (*Main theme: Activity diagrams, introduction to code documentation*)

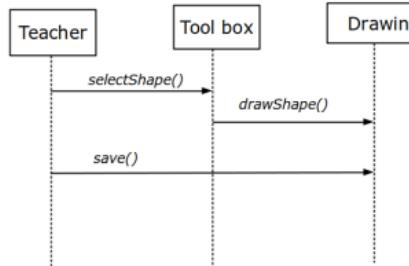
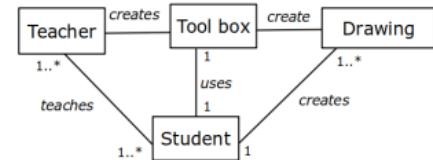
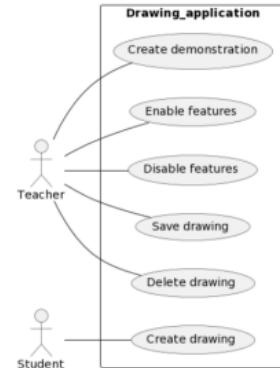
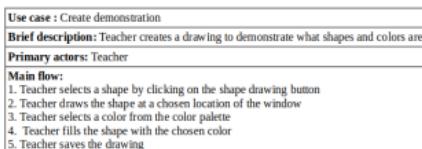
- Activity diagram
- Introduction to code documentation

Learning objectives (Today)

- Create activity diagrams
- Describe the difference between sequence diagrams and activity diagrams
- Understand methods for code documentation

So far we have seen ...

- Use cases
- Use case diagrams
- Domain models
- Sequence diagrams



Activity diagram - What?

It models the flow or dynamics of control, execution of your system.

It has:

- Edges or arrows indicating direction of flow
- actions
- decision
- fork and join
- merge
- partitions

Activity

- It is not atomic = can be divided into simpler actions
- A series of actions

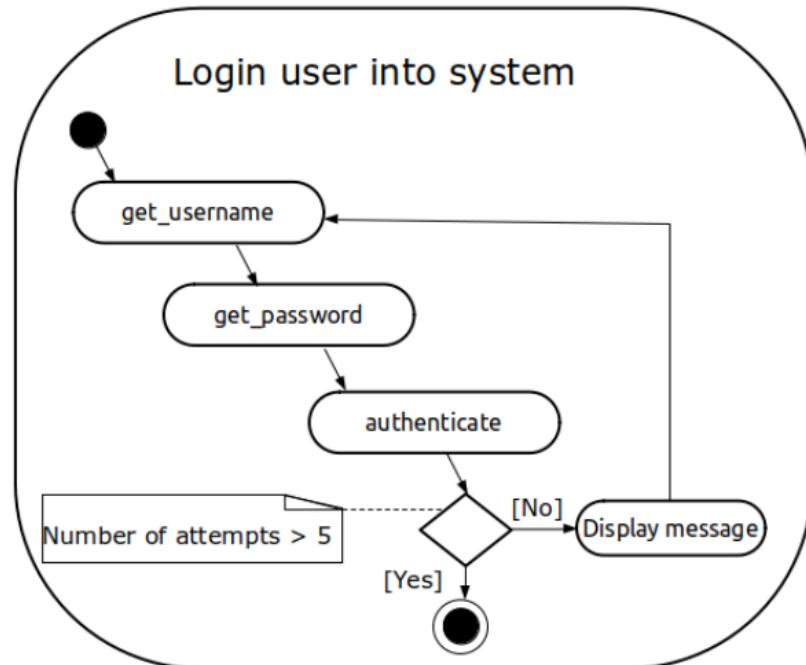
Log a user into the system

Display all chats

Create a new chat

Activity

Activity = behavior represented as coordinated flow of actions



Activity diagrams components - Actions

- Steps in an activity, operations
- It is atomic = cannot be broken down further



Activity and Action

Activity:

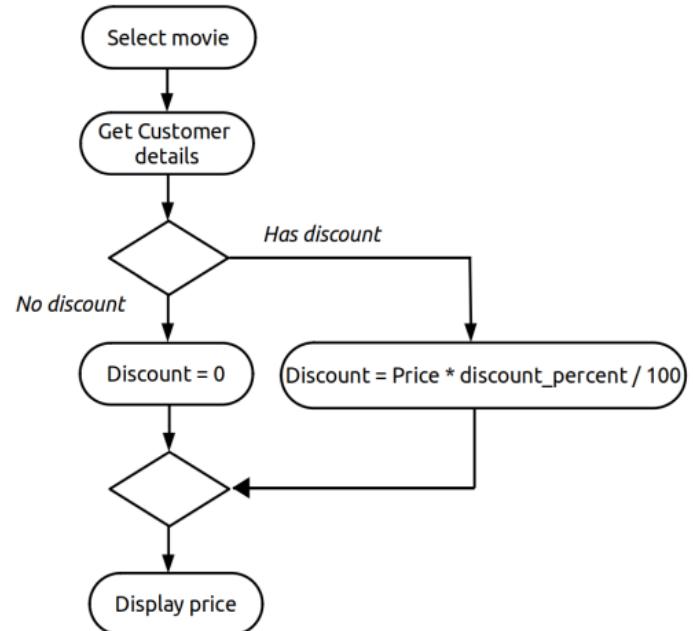
- Not atomic: can be broken down into simpler actions
- Takes a while to complete

Action:

- Atomic: cannot be broken down further
- Takes very little time

Activity diagrams components - Branch, merge

- **Branch:** A point where a decision is made to go in a certain direction based on a condition / guard
- **Merge:** A point where many ways meet for a common further road



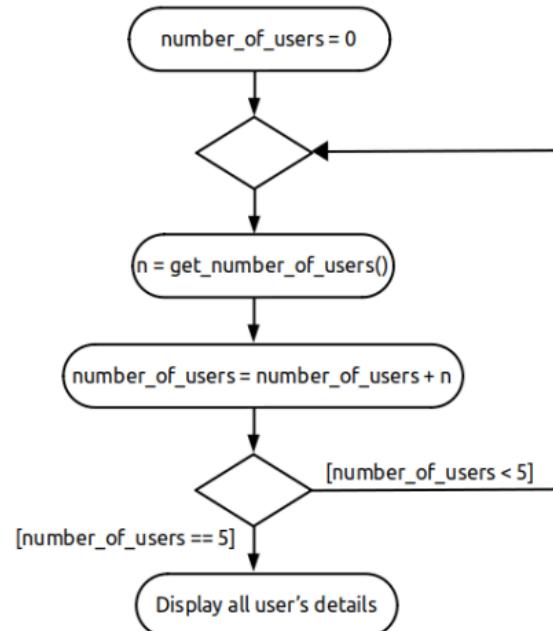
Activity diagrams components - Loops / iterations

Pseudocode for a while loop

```
number_of_users ← 0

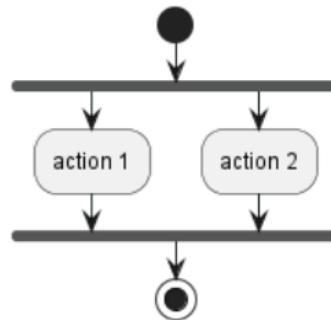
while number_of_users < 5 do
    n = get_number_of_users()
    number_of_users = number_of_users + n
end while

Display all user's details
```



Activity diagram components - Forks and joins

- Forks and joins: do several operations in parallel



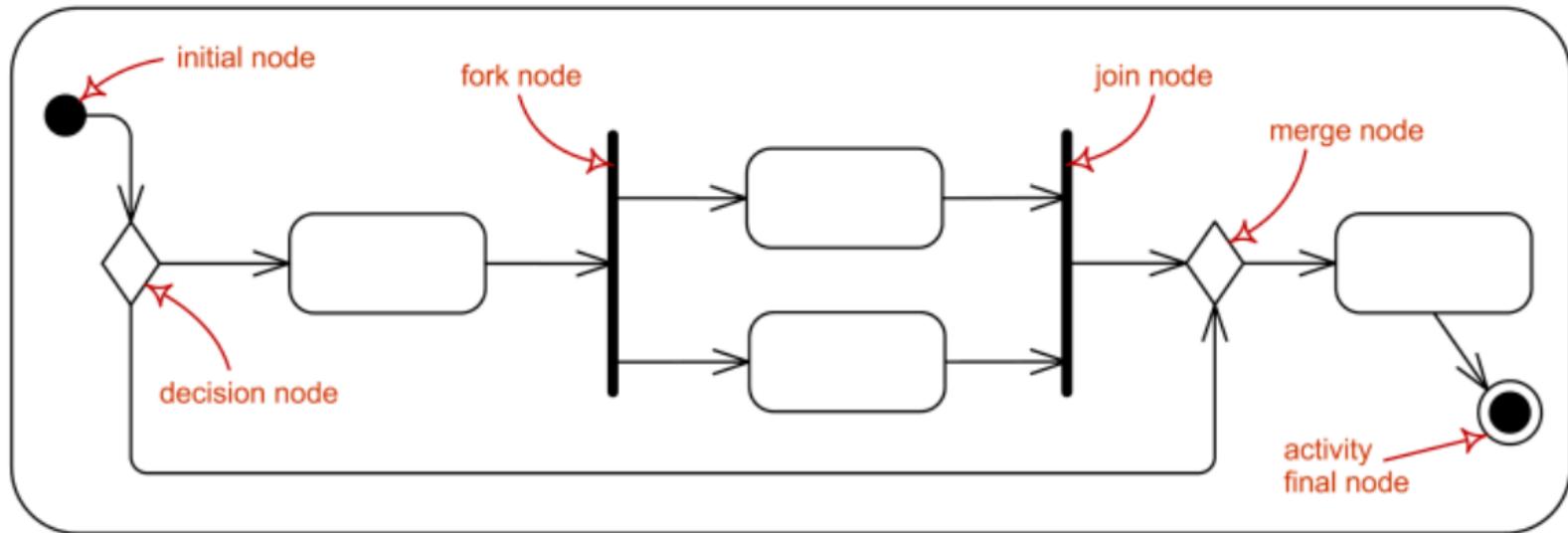
Activity diagrams components - Start and stop

- Start



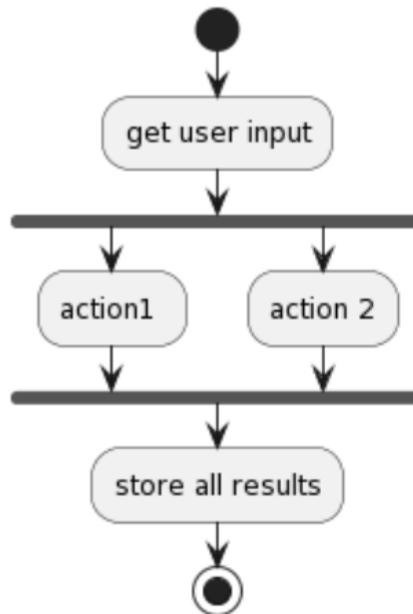
- Stop



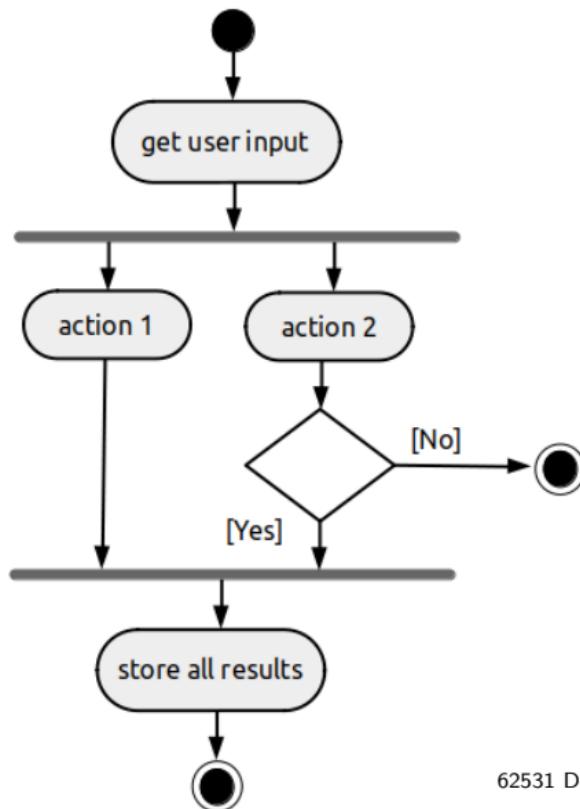


<https://www.uml-diagrams.org/activity-diagrams-controls.html>

Activity diagram example - single stop



Activity diagram example - multiple stops



Two types of final (stop) nodes

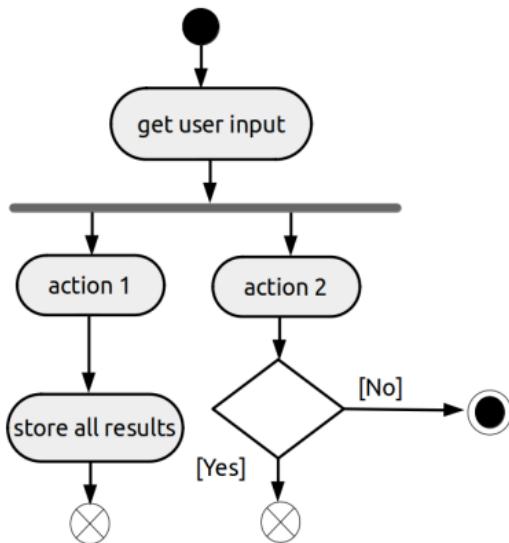
- Activity final: stops the entire activity



- Flow final: stops only the given flow

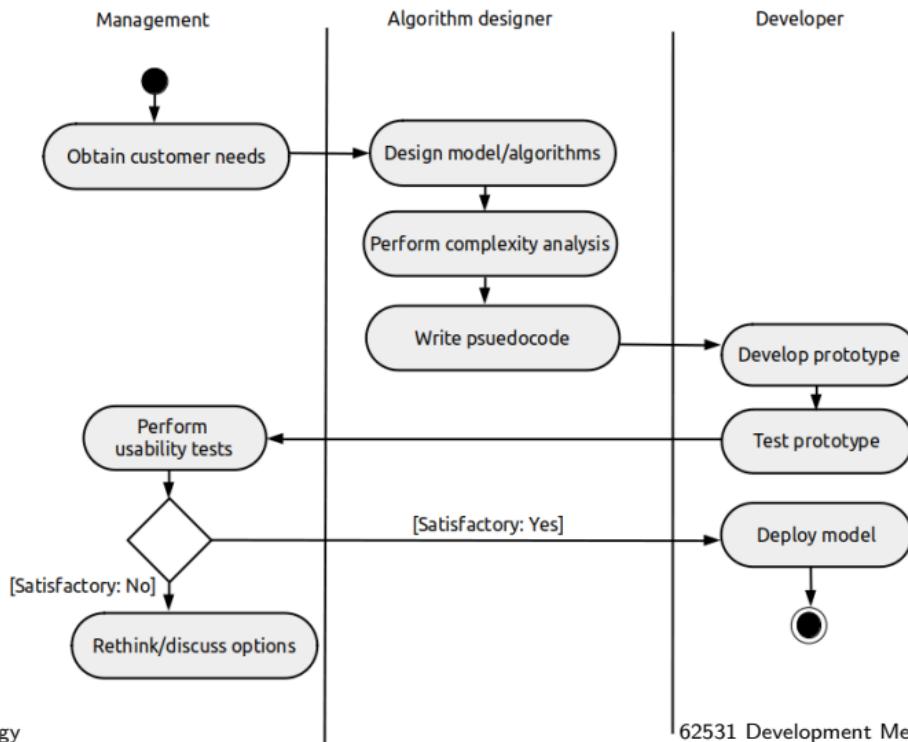


Activity diagram example - flow final node and activity final node



Activity diagram components - Swim lanes / partitions

It is used if there are several separate areas/systems that collaborate on an activity.



Event-based actions

- Accept event action: waits for an event to happen. Notation: concave pentagon



- Wait time action: this is a time event action. Notation: Hourglass



Event-based actions examples

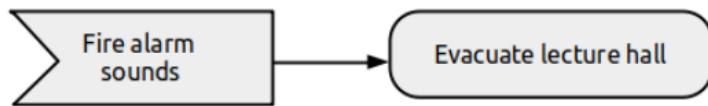


Figure: Accept event action



Every hour

Figure: Time event action

Code documentation

Software must be designed and implemented so that it is:

- easy to understand
- easy to test
- easy to maintain

Self documenting elements

- Class names
- Method names
- Variable names

Examples:

```
class LoginSystem
getUserName()
getUserPassword()
private String userName;
```

Layout

Divisions using:

- Extra lines between methods
- Separate file for each class (usually)
- Indentation

Indentation

```
class Second {  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

```
class Second {  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

Comments

- Makes code easier to understand
- The logic of the code can be briefly explained

Example:

```
/* Since performance matters,  
quicksort was used  
*/  
quicksort(numberArray);
```

Javadoc

- Built-in documentation tool for Java
- Generates HTML pages of API documentation from Java source files.
- Supported in most IDEs

Documentation in the report

- Design class diagrams
- Essential code snippets with explanation
- Essential sequence diagrams
- Code in appendix

Mid term evaluation

Please go to learn and complete the survey called: Mid term evaluation

Activities -> Surveys

Thank you!



Development Methods 62531

Lecture 7 - Class Diagrams, GRASP

Lei You - leiyo@dtu.dk

Today's Program

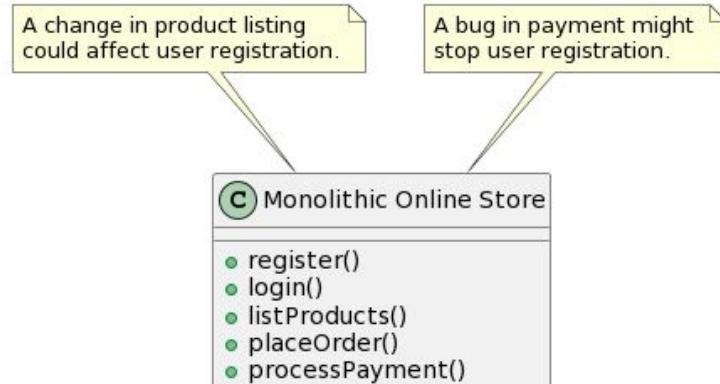
- Architecture
- Design class diagrams
 - UML class notation
 - UML class association
- GRASP
 - General Responsibility Assignment Software Patterns

Separation of Concerns

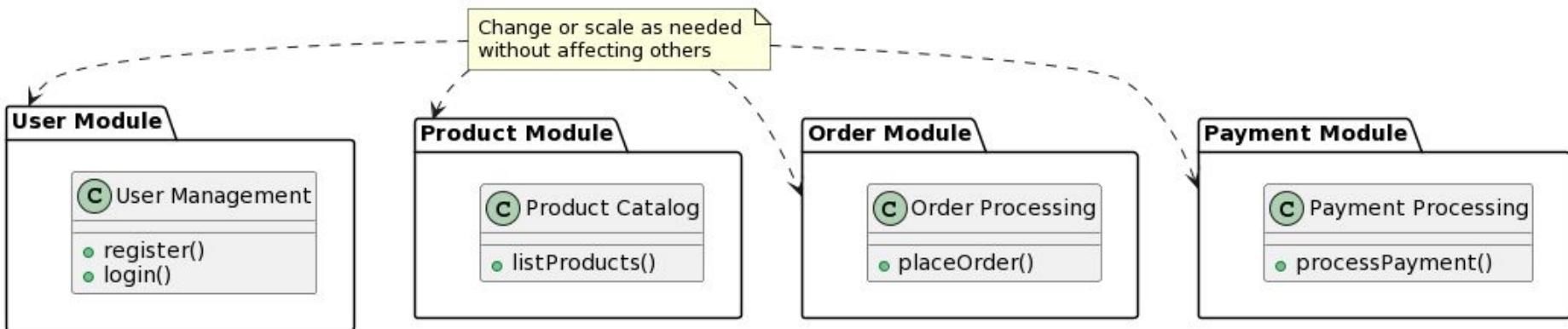
- Frequently occurring design principle
- Division of a system into separate areas
- Modularity in the code
- Examples:
 - The internet protocol stack
 - UI-related design, frontend/backend
 - Service-Oriented Architecture
 - Layered applications
- Why modularity?

Separation of Concerns

- Why modularity?
 - Division of development
 - Readability
 - Easier fault isolation
 - Easier tests
 - Recycling
 - Easier replacement
- Example:

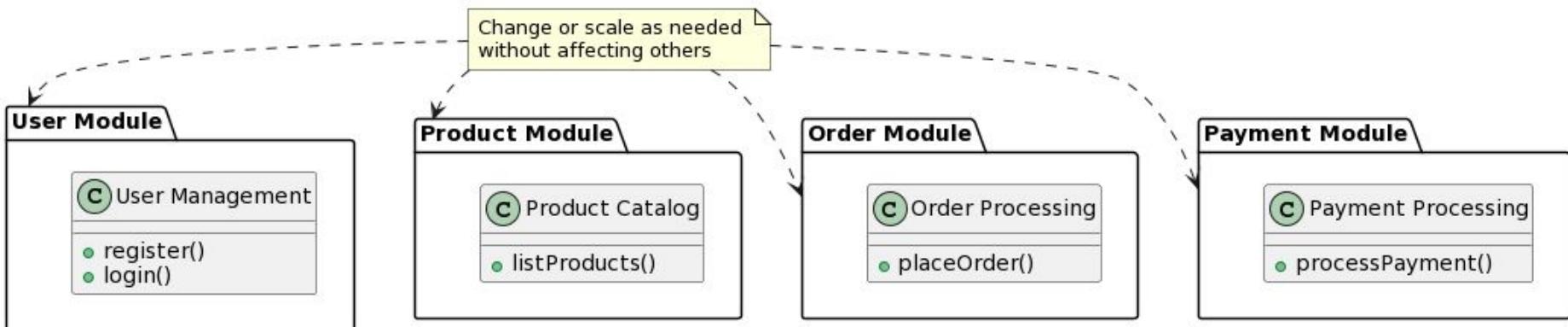


Separation of Concerns



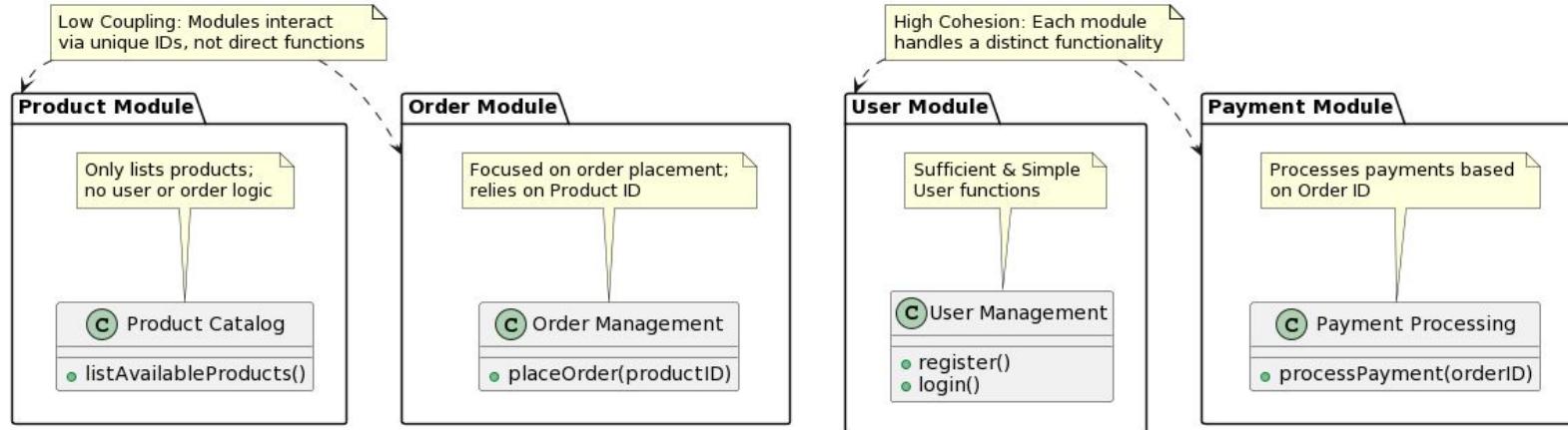
GRASP: Low Coupling and High Cohesion

- Low Coupling
 - A class should be as independent as possible
 - A class should be associated with only the few classes necessary for it to fulfill its scope of responsibility
- High cohesion
 - A class must have a well-defined area of responsibility
 - A class must have a set of operations that support the scope of responsibility



Good Class Design

- Complete and sufficient
 - Meets the client's expectations of functionality (no more, no less)
- Simple
 - methods are simple
 - Does not contain multiple ways to perform functionality
- Low coupling
- High cohesion



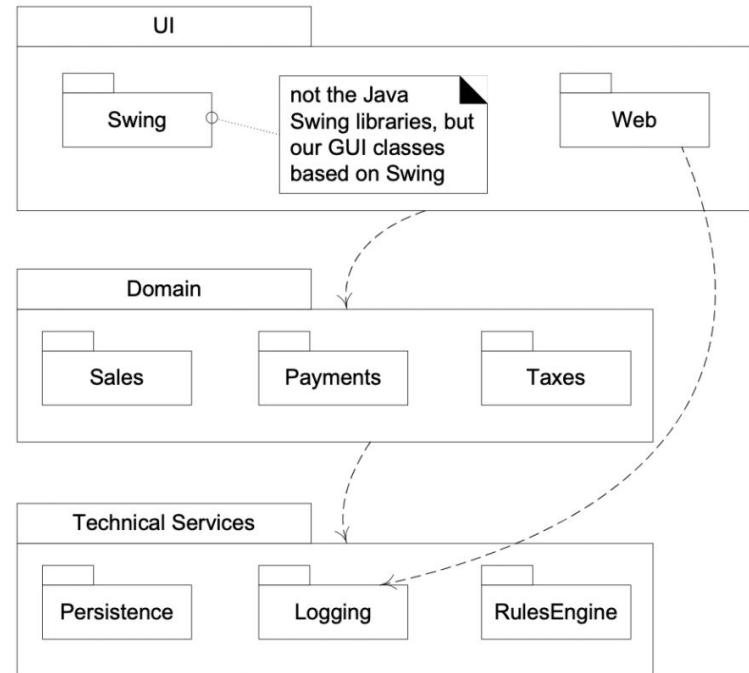
Layered architecture

A partial layered, logical architecture drawn with UML package diagram notation.

Often used to illustrate the logical architecture of a systems.

A UML package can group anything: classes, other packages, use cases, and so on.

- Larman Fig. 13.2



Layered architecture

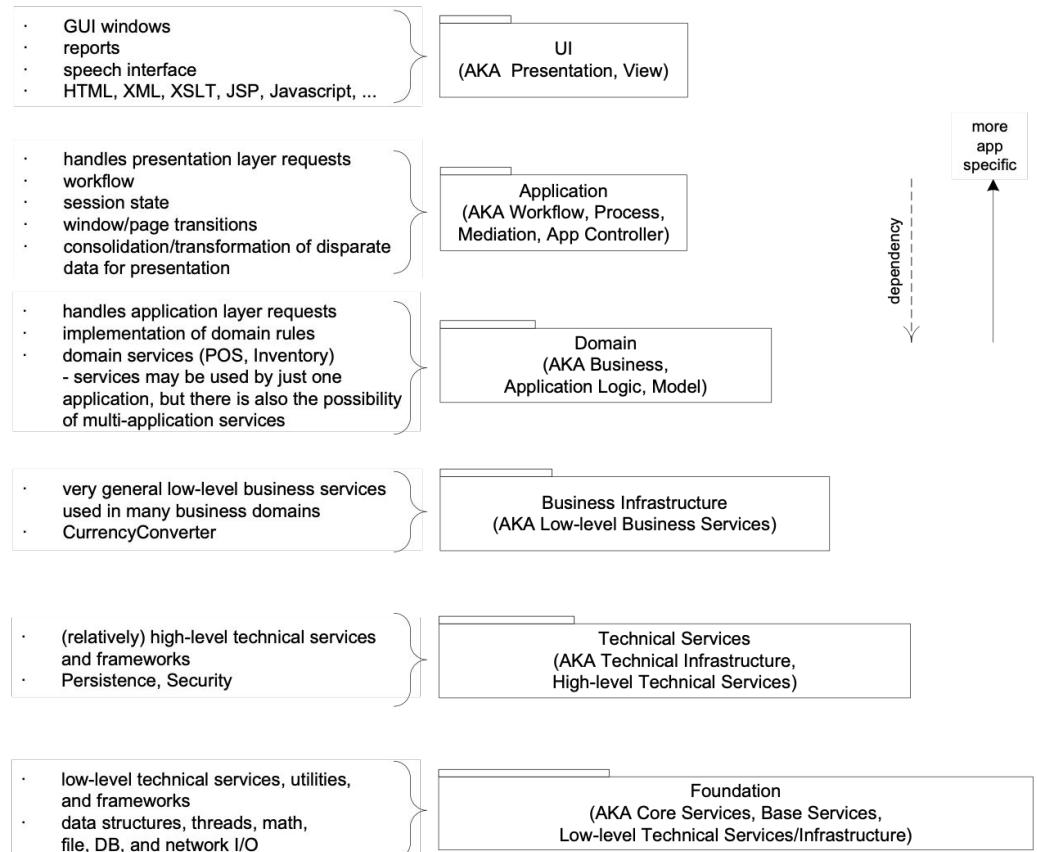
Modularity

The essential idea of using layers:

- Organizing logical structure
- Collaboration and coupling direction

What problems does using layers address?

- Source code change
- Application logic, reusability
- Business logic, reusability
- Coupling and cohesion



Model View Separation Principle

Guideline: Model-View Separation Principle

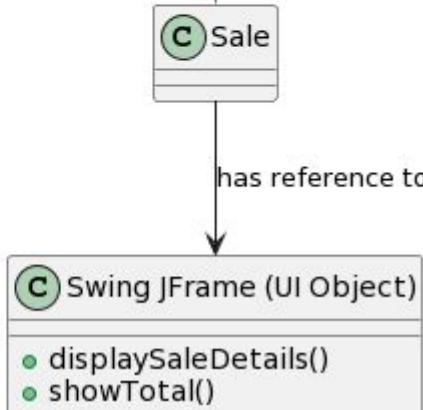
This principle has at least two parts:

1. Do not connect or couple non-UI objects directly to UI objects. For example, don't let a *Sale* software object (a non-UI "domain" object) have a reference to a Java Swing *JFrame* window object. Why? Because the windows are related to a particular application, while (ideally) the non-windowing objects may be reused in new applications or attached to a new interface.
2. Do not put application logic (such as a tax calculation) in the UI object methods. UI objects should only initialize UI elements, receive UI events (such as a mouse click on a button), and delegate requests for application logic on to non-UI objects (such as domain objects).

Principle 1: Examples

C	Sale (Domain Object)
•	recordSale()
•	calculateTotal()

Not recommended:
The domain object is tied
directly to a specific UI framework.



```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Sale {

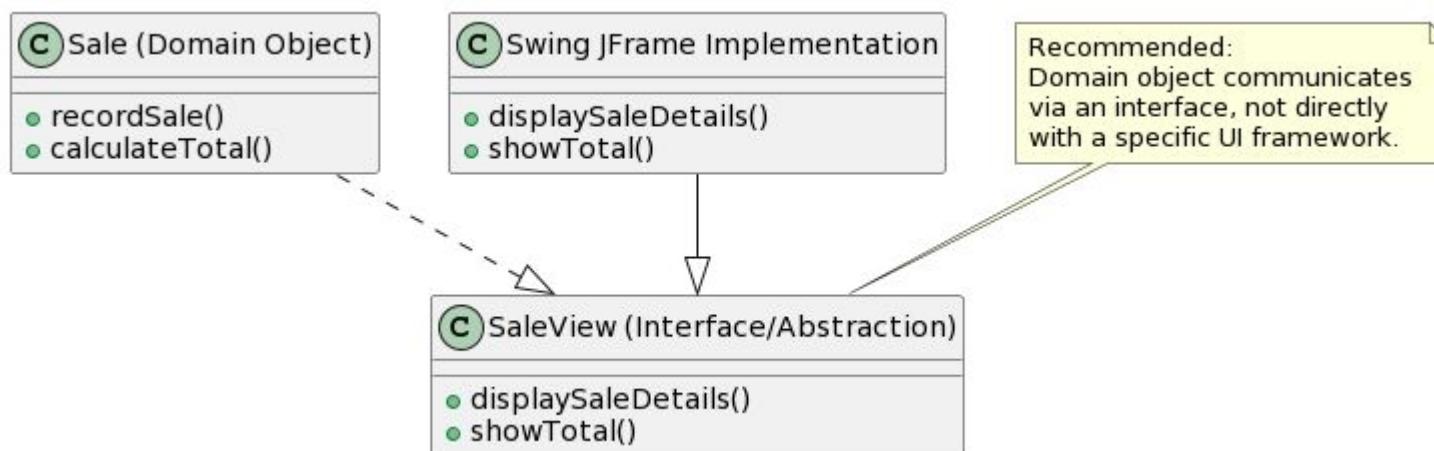
    private SwingJFrame uiObject; // Direct reference

    public Sale(SwingJFrame uiObject) {
        this.uiObject = uiObject;
    }

    public void recordSale() {
        // Implementation of recording a sale
        // ...
        // Possibly update the UI directly (not recommended)
        uiObject.displaySaleDetails();
    }

    public void calculateTotal() {
        // Implementation of calculating the total
        // ...
        // Possibly update the UI directly (not recommended)
        uiObject.showTotal();
    }
}
```

Principle 1: Examples



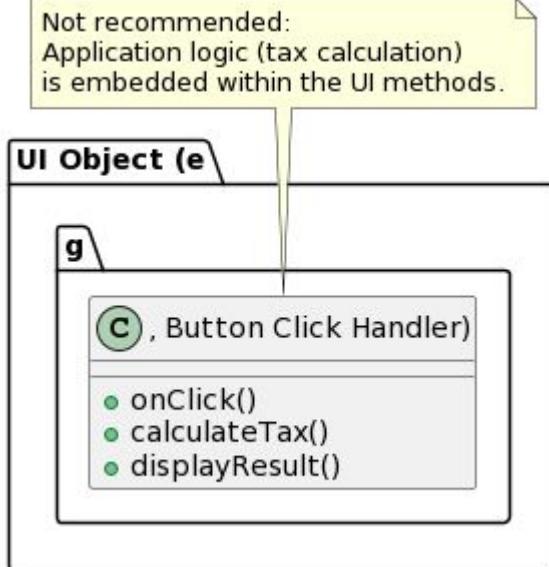
Model View Separation Principle

Guideline: Model-View Separation Principle

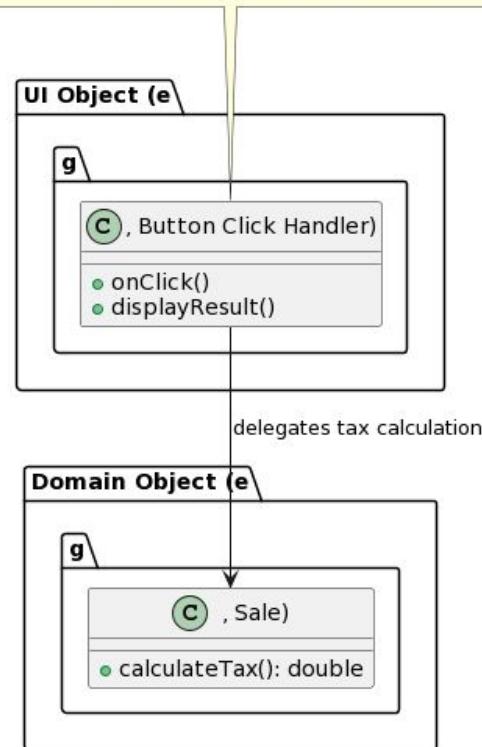
This principle has at least two parts:

1. Do not connect or couple non-UI objects directly to UI objects. For example, don't let a *Sale* software object (a non-UI "domain" object) have a reference to a Java Swing *JFrame* window object. Why? Because the windows are related to a particular application, while (ideally) the non-windowing objects may be reused in new applications or attached to a new interface.
2. Do not put application logic (such as a tax calculation) in the UI object methods. UI objects should only initialize UI elements, receive UI events (such as a mouse click on a button), and delegate requests for application logic on to non-UI objects (such as domain objects).

Principle 2: Examples

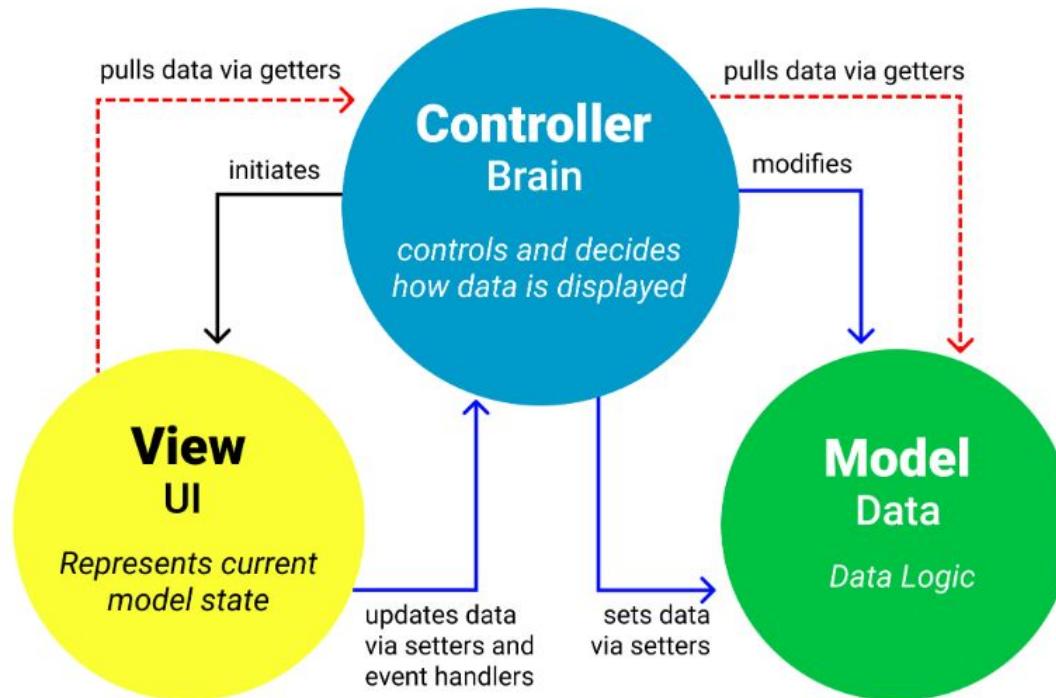


Recommended:
UI events trigger actions in the UI object,
which then delegates application logic to domain objects.



Model View Controller

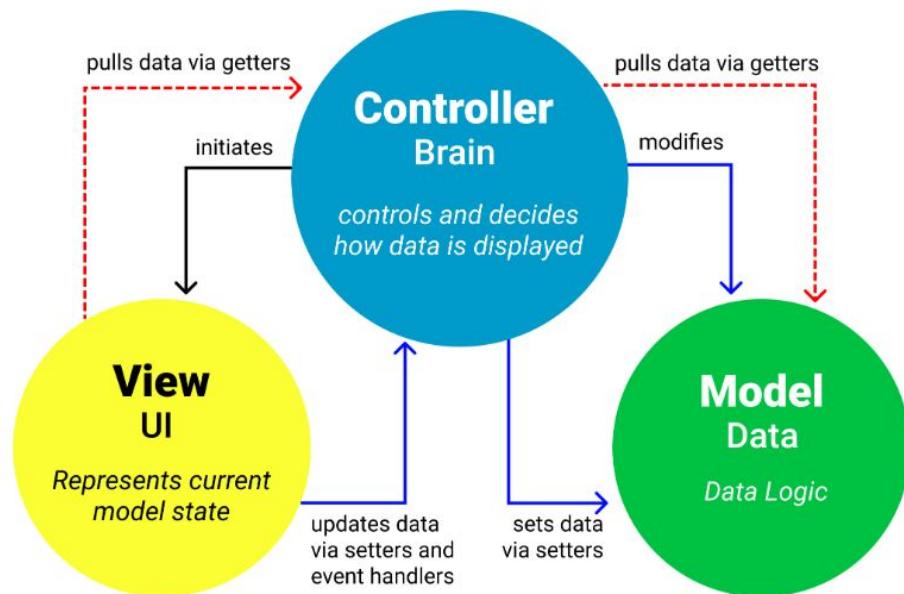
MVC Architecture Pattern



Model View Controller

- Objects are allocated a role, (model, view or controller), with a well-defined responsibility.
- Achieve low coupling and high cohesion.
- Achieve a design that is:
 - easy to understand
 - easy to test
 - easy to maintain

MVC Architecture Pattern



MVC and Robustness Diagram - "BCE-Diagram"

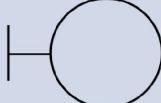
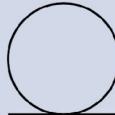
Robustness diagram also known as:

BCE Model

- **Boundary (MVC Equivalent: View)**
 - It can be broader than just a UI – it could represent **any external interface**. In a web application, the Boundary could be the HTML pages or RESTful API endpoints through which users or other systems interact with our application.
- **Controller (MVC Equivalent: Controller)**
 - In a shopping cart application, when a user decides to check out, the Controller would handle this request, calculate the total cost, and decide what to display next.
- **Entity (MVC Equivalent: Model)**
 - In a blog application, an Entity might represent a blog post, storing the title, content, and author, and providing methods to save or retrieve a post from a database.

Depending on the context, developers might find one pattern more intuitive or applicable than the other.

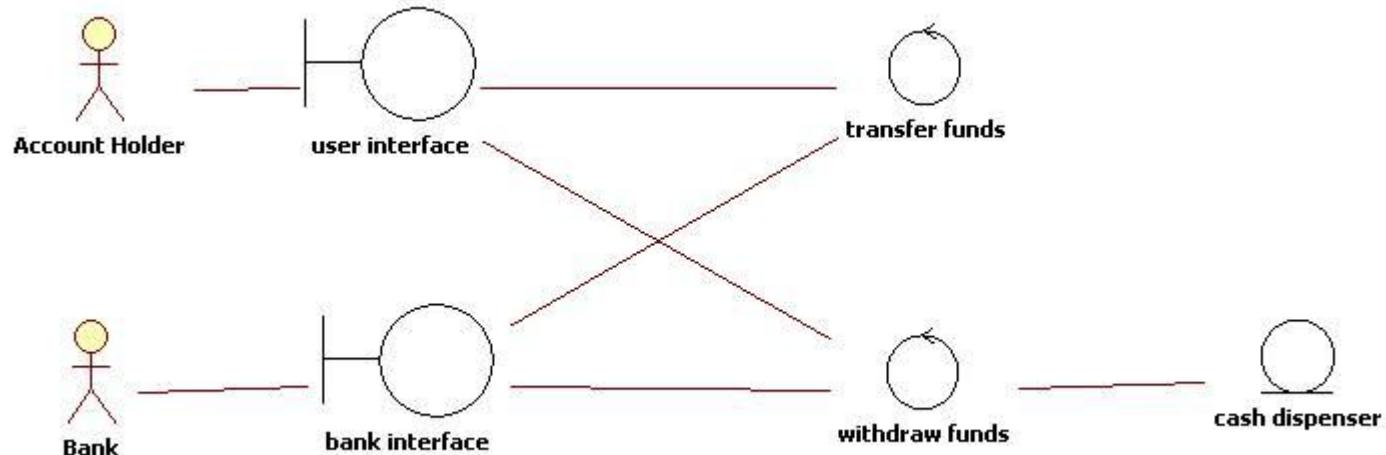
MVC and Robustness Diagram - "BCE-Diagram"

Stereotype	Symbol	Betydning
<<boundary>>		En klasse som håndterer interaktion/kommunikation med systemets omgivelser
<<control>>		En klasse der indkapsler use-case specifik opførsel – fordeler ansvar på systemets andre objekter og fastlægger sekvens i handlinger
<<entity>>		En klasse der modellerer det systemet handler om. Har persistens (hukommelse)

BCE /Robustness diagram

"Variant of the class diagram"

- Not included in the UML specification.
- Introduced to help bridge the gap between high-level analysis models (use case diagrams) and detailed design models (class diagrams, sequence diagrams).



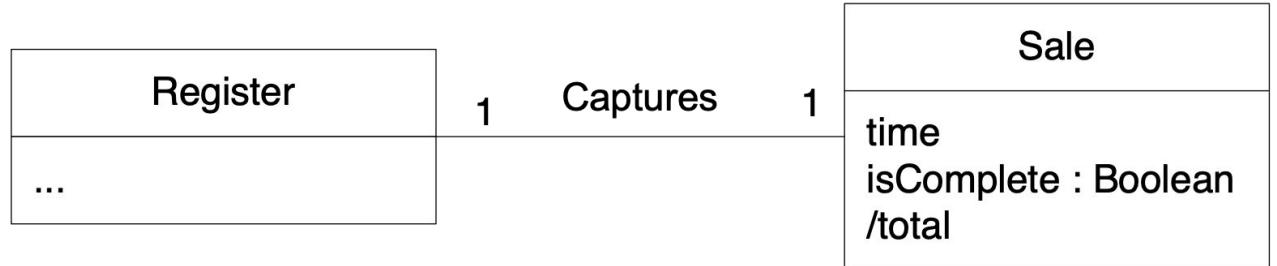
Class diagrams

- Class diagrams are used for structure/static modeling
 - Provide a visual representation of the static structure of a system
 - It's like looking at a blueprint of a building, showing the layout but not the activities inside.
 - The instance of class leads to objects
- Analysis:
 - Conceptual model / domain model
 - Class name and possibly attributes
- Design:
 - Software model / design model
 - Class name, attributes and operations
- A design model can contain 10 to 100 times more classes than the analysis model
 - Classes that do not exist in the domain
 - Classes for design patterns
 - Controllers, adapters...

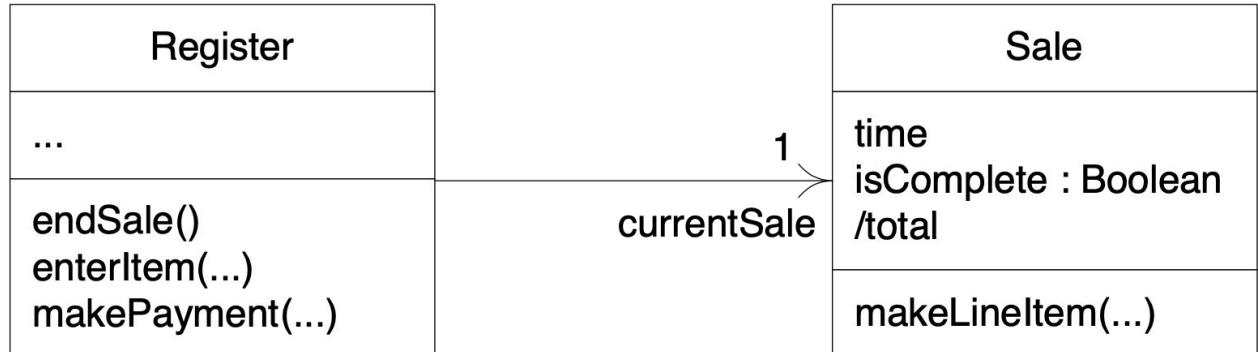
Class Diagrams: Domain vs Design



Domain Model
conceptual perspective

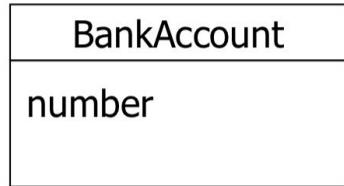


Design Model
DCD; software perspective

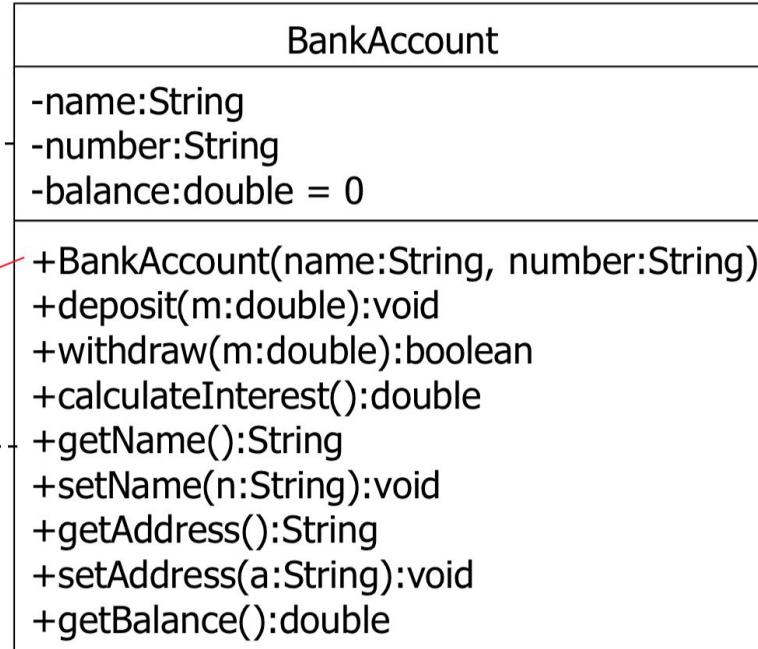


Analysis vs. Design classes

Analyse:



Design:



«trace»

constructor

«trace»

eller

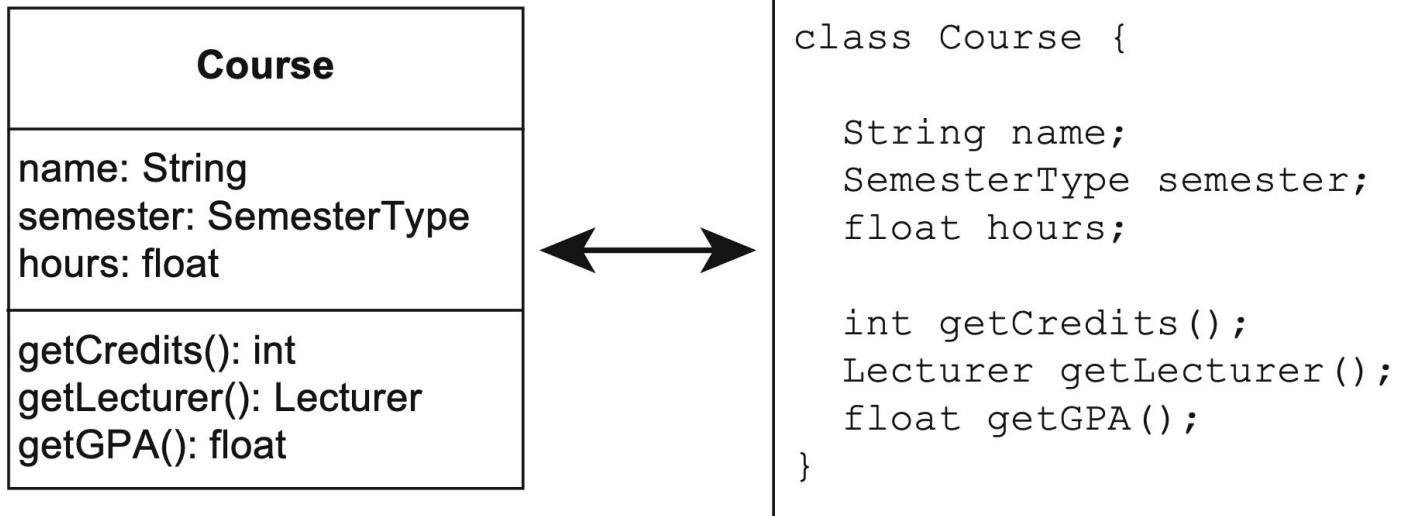
Analyse:



Arlow & Neustadt

Design and Implementation

- Class definition in UML vs Java
- Note syntax differences



Perspectives of Class Diagram

The perspective chosen is influenced by the development stage. Each stage emphasizes different perspectives: Conceptual, Specification, Implementation.

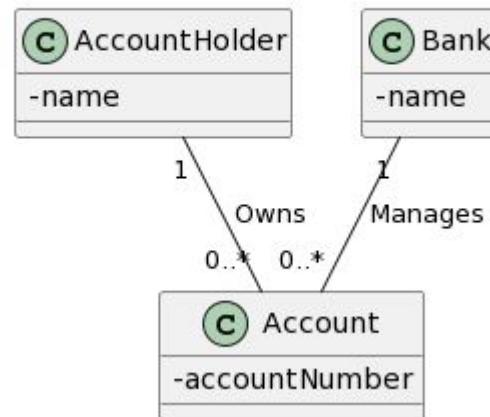
Domain Model Development. Focus mainly on the *Conceptual Perspective*. Understand the high-level domain without delving into specifics.

Analysis Model Perspectives. Understand the high-level concepts and relationships (*conceptual*). Begin to outline specifics and detailed functionalities (*specification*). Lay groundwork for design by understanding what the system should do.

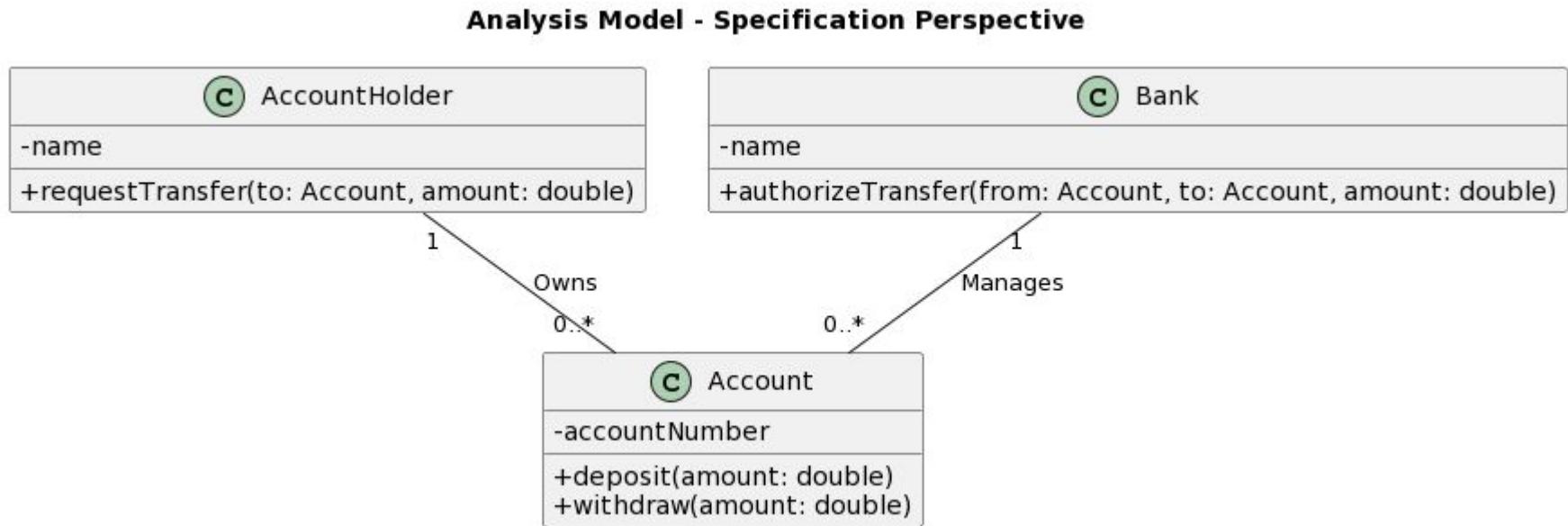
Design Model Development. Start with a heavy emphasis on detailed interactions, processes, and functionalities (*specification*). Evolve into the *Implementation Perspective*. Focus on concrete details, classes, methods, and data structures. Prepares for actual coding and system development.

Domain Model Development - Example

Domain Model - Conceptual Perspective

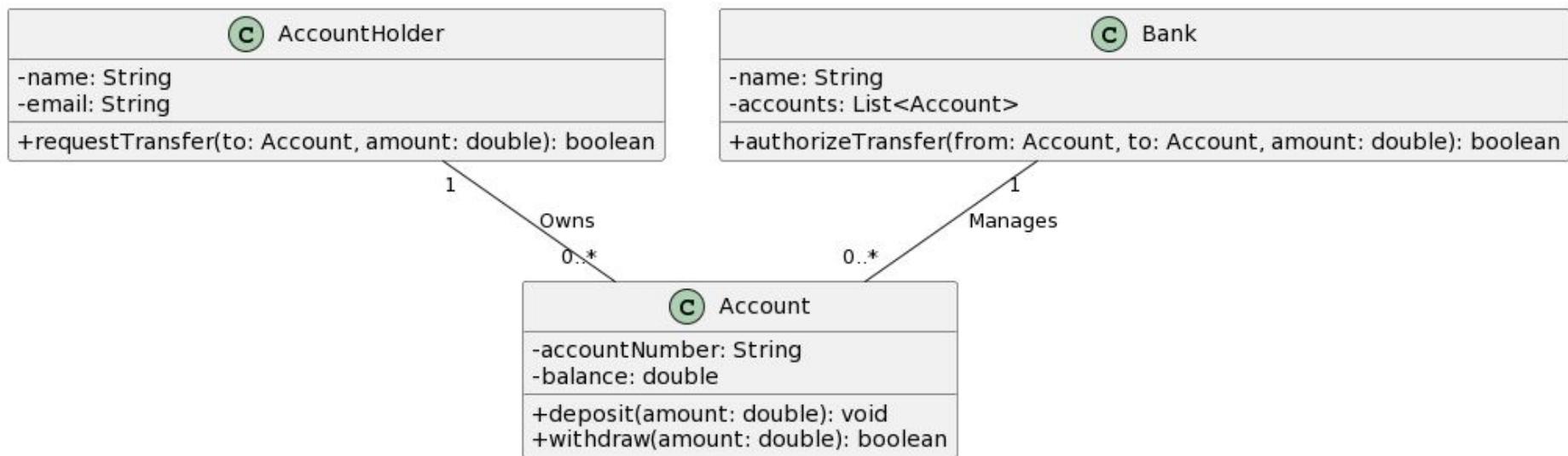


Analysis Model Perspectives - Example



Design Model Development - Example

Design Model - Implementation Perspective



Varying degrees of detail

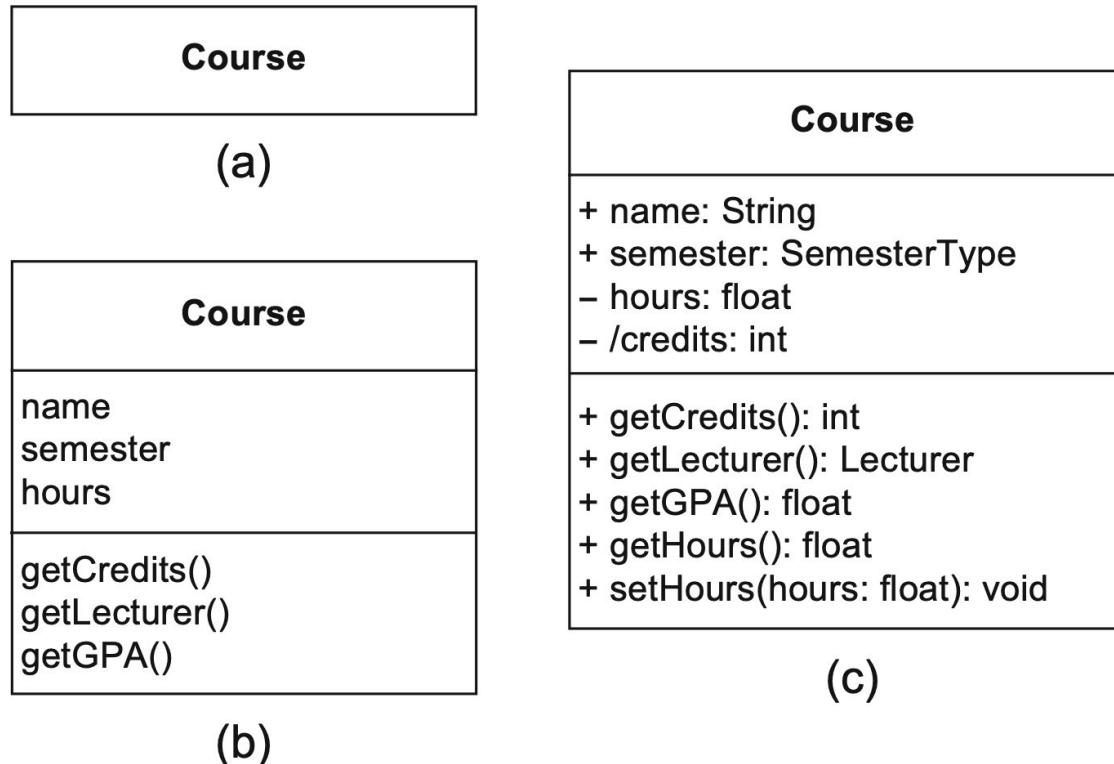
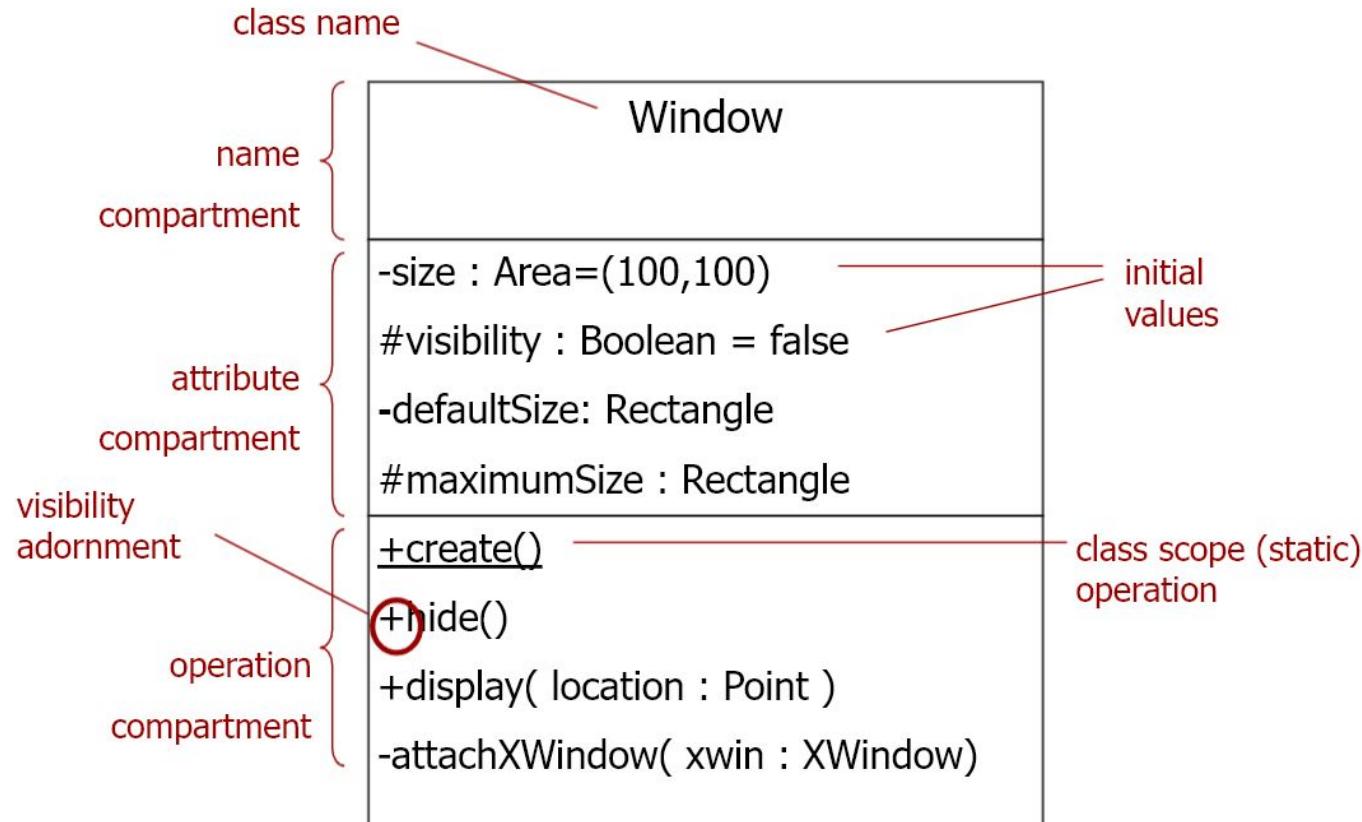


Fig.4.4 UML @ Classroom; Seidl, Scholz, Huemer, Kappel

Design class diagrams



UML ⇄ Java

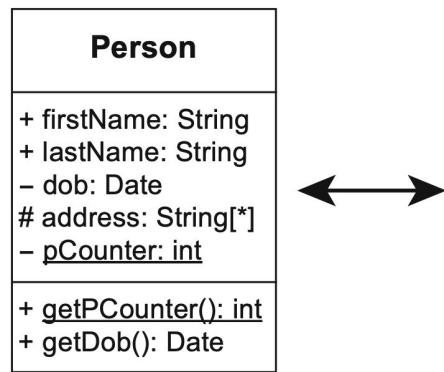
Person	
+ firstName:	String
+ lastName:	String
- dob:	Date
# address:	String[*]
- pCounter:	int
+ getPCounter():	int
+ getDob():	Date



```
class Person {  
  
    public String firstName;  
    public String lastName;  
    private Date dob;  
    protected String[] address;  
    private static int pCounter;  
  
    public static int getPCounter() {...}  
    public Date getDob() {...}  
}
```

UML ⇔ Java

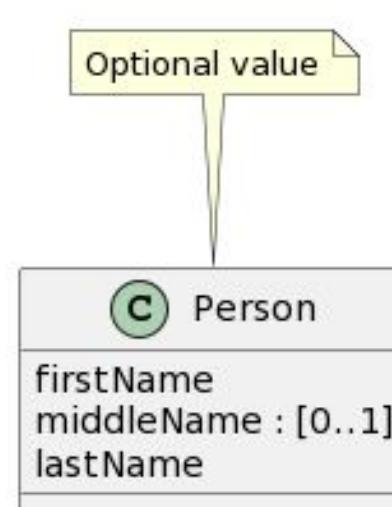
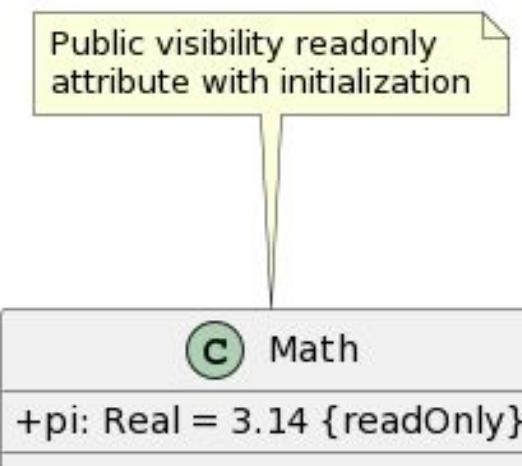
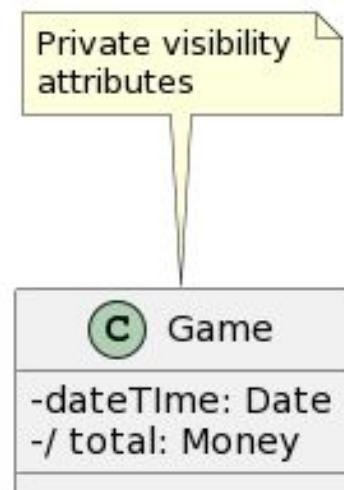
- Visibility
 - UML: +,#,- ;
 - Java: Public, Protected, Private
- Type
 - UML: name : type ;
 - Java: type name
- Class variables vs Object variables
 - UML: underline
 - Java: static



```
class Person {  
  
    public String firstName;  
    public String lastName;  
    private Date dob;  
    protected String[] address;  
    private static int pCounter;  
  
    public static int getPCounter() {...}  
    public Date getDob() {...}  
}
```

Attributes

- Name, Type, possibly multiplicity, default value, visibility
 - visibility name : type multiplicity = default {property string}
- Examples:
 - - passengerList : String [0..*] = checkedInList {ordered}
 - - age : int
 - + GRAVITY : double = 9.81 {readOnly}



The symbol “/”

A derived attribute, which can be derived from other attributes in the model, and it does not need to be stored explicitly

```
public class Game {  
    private Date dateTime;  
    private int teamAScore;  
    private int teamBScore;  
  
    // ... constructor, other methods ...  
  
    public Money getTotal() {  
        return teamAScore + teamBScore; //  
    }  
}
```

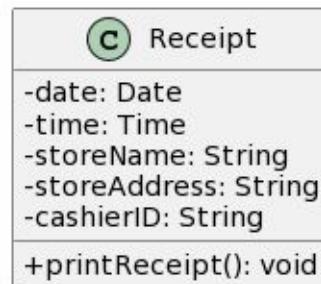
Attributes

When to show attributes?

Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.

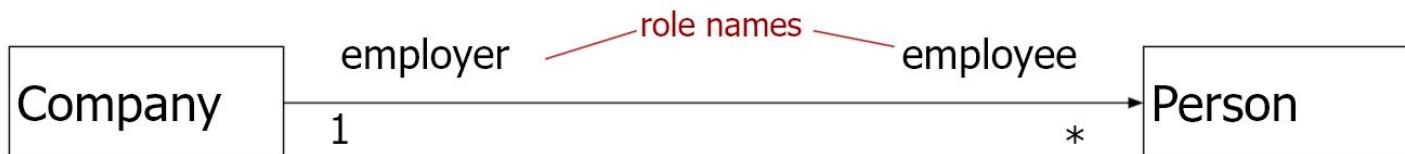
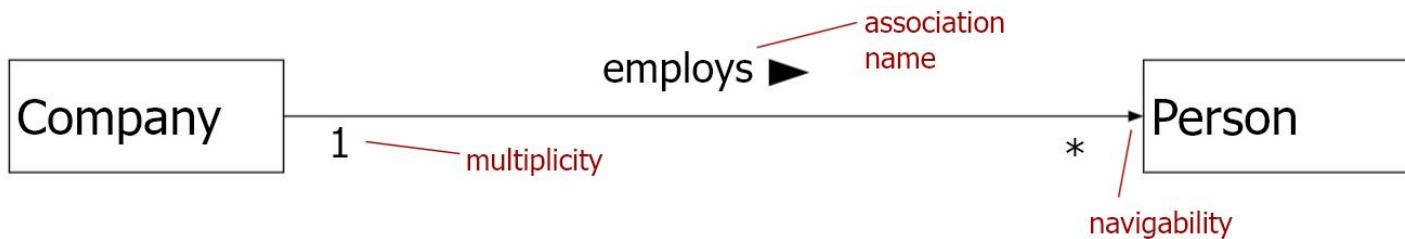
For example, a receipt (which reports the information of a sale) in the Process Sale use case normally includes a date and time, the store name and address, and the cashier ID, among many other things.

Class Diagram for Receipt in Process Sale Use Case



Associations

- Relationships between classes
- Specifies a form of communication between classes
 - Typically by sending messages (method calls) between the classes.
- Represents a reference to an object.



Associations: Multiplicity

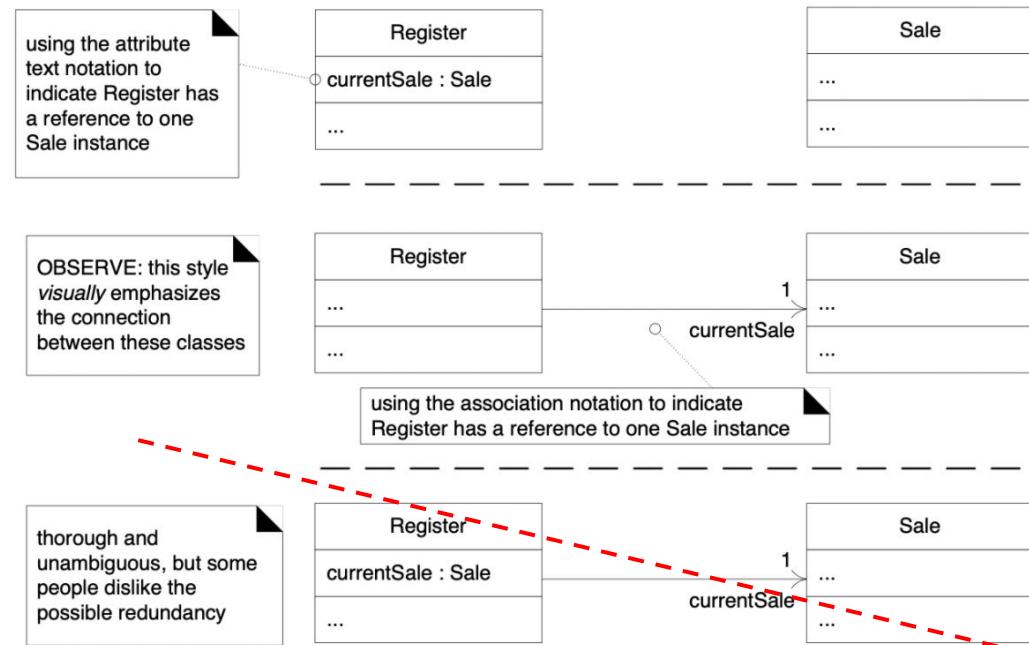
-



multiplicity syntax: minimum..maximum	
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..6	1 to 6

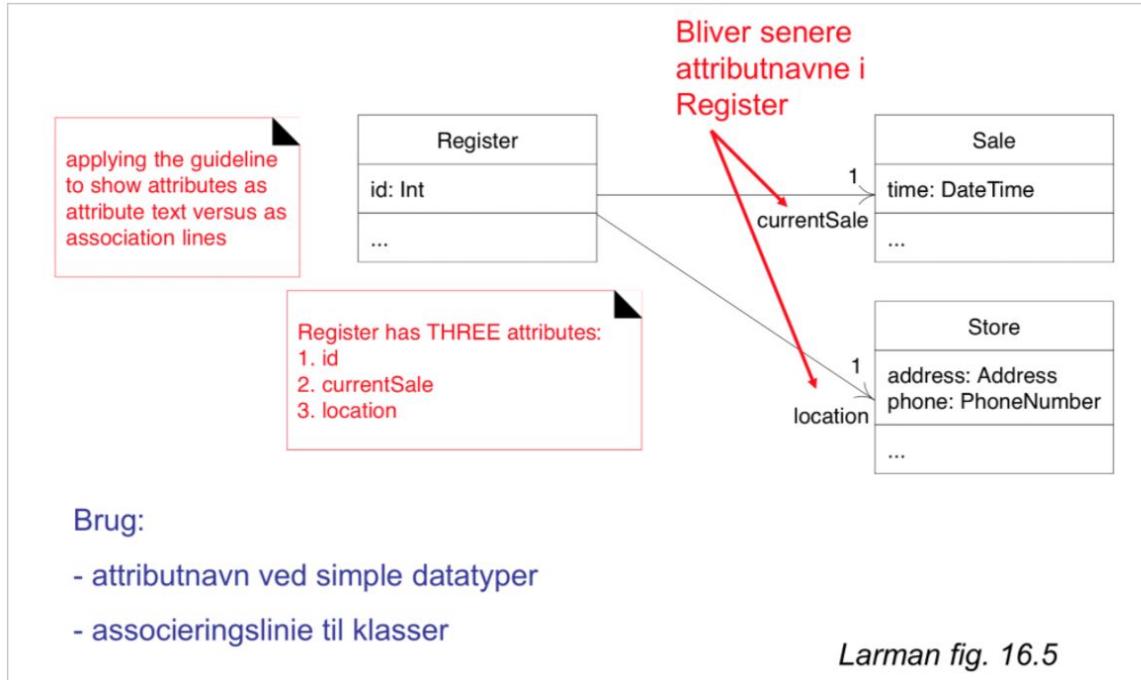
Attributes and Associations

- Basically the same
 - Arrows can mean several things
- Visually different
- 3: (I don't like redundant information)



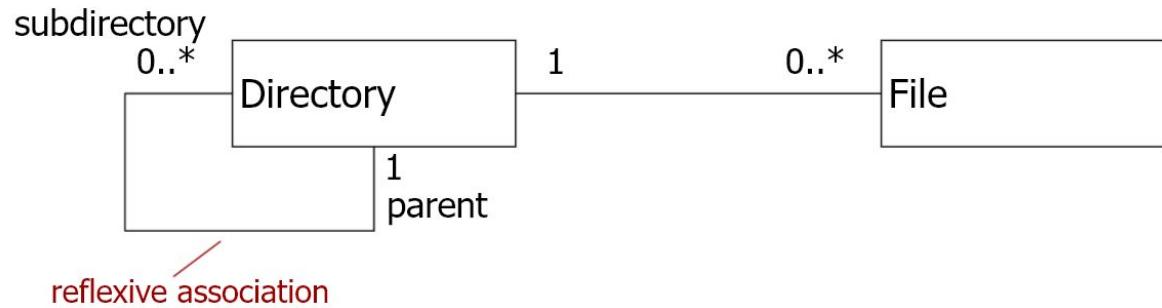
Rule of thumb - Associations

- Simple types: Attribute
- Classes: Associations (or Attributes)



Reflexive association

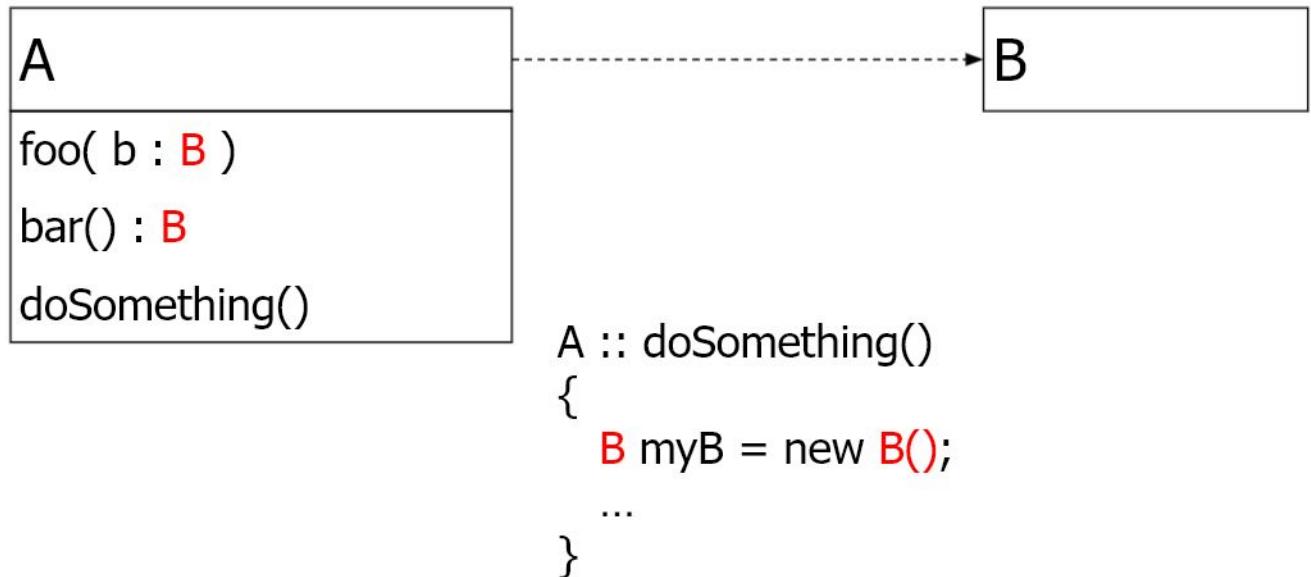
- Briefcase
 - with Subfolder
 - with subfolder



A concept may have an association to itself, this is known as a **reflexive association**.

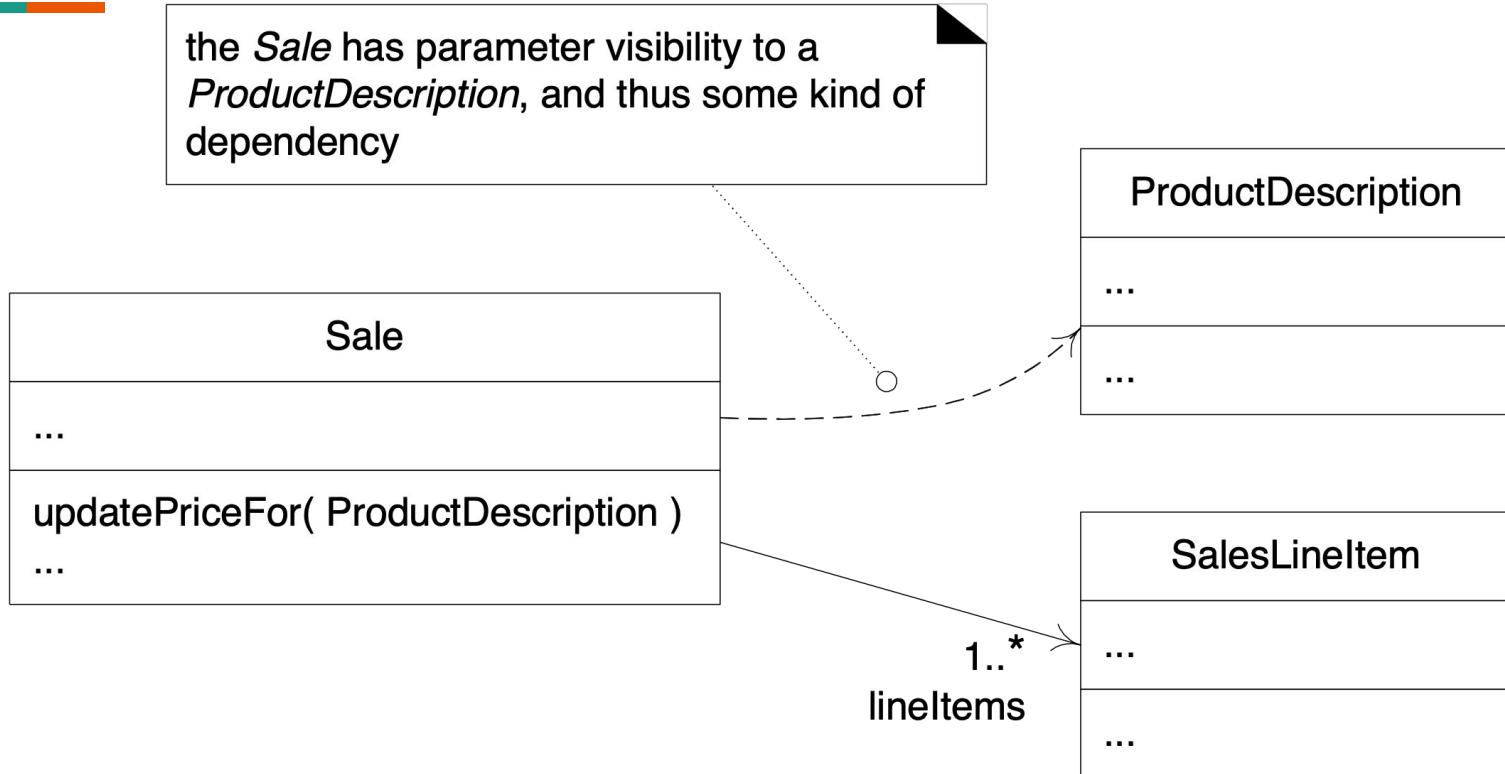
Dependency / dependence

- "Light association"
- Temporary reference
- Change in one class affects another

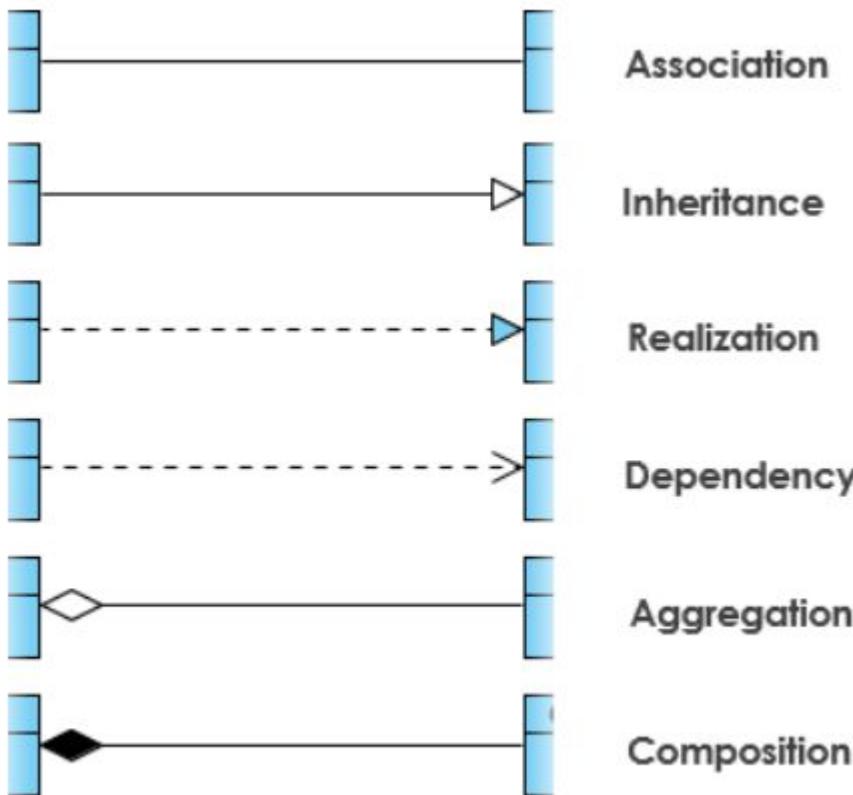


Dependency

the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency



Types of Relationships



Aggregation and Composition

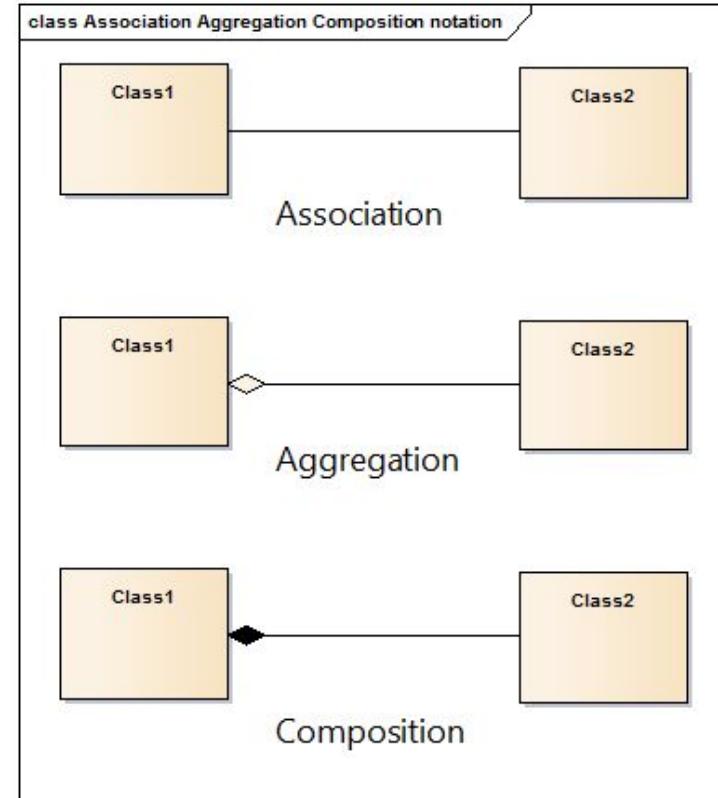
- Models “whole – part” relationship.
- Two forms:
 - Aggregation (shared)
 - Composition
- Aggregation - "parts" exist independently of the "whole"
 - Eg: printers and computers
- Composition is a stronger form of aggregation – “parts” do not exist outside of the “whole”
 - E: Brick house
- NB: Rarely necessary to distinguish in Java
 - In C++, composition can be achieved by declaring objects (not pointers or reference) directly within classes

```
class Engine { /*...*/};
```

```
class Car {
```

```
    Engine myEngine;
```

```
};
```



Aggregation

LabClass can have students

StudyProgram can have Courses



(a)



(b)

Composition

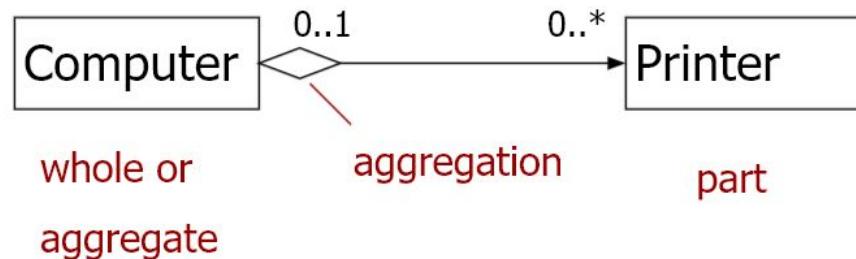
Strong coupling

Below the object belongs 0 or 1 parent object

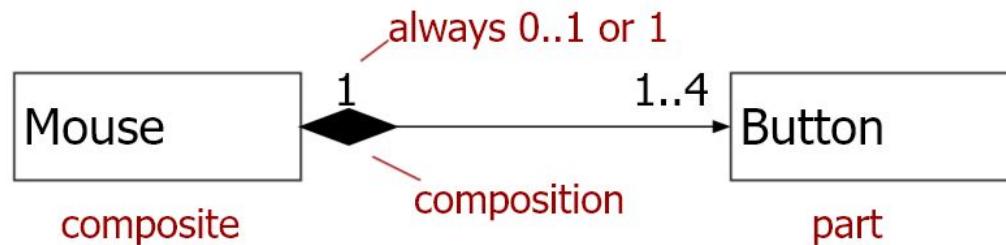


Aggregation vs Composition

aggregation is a whole–part relationship

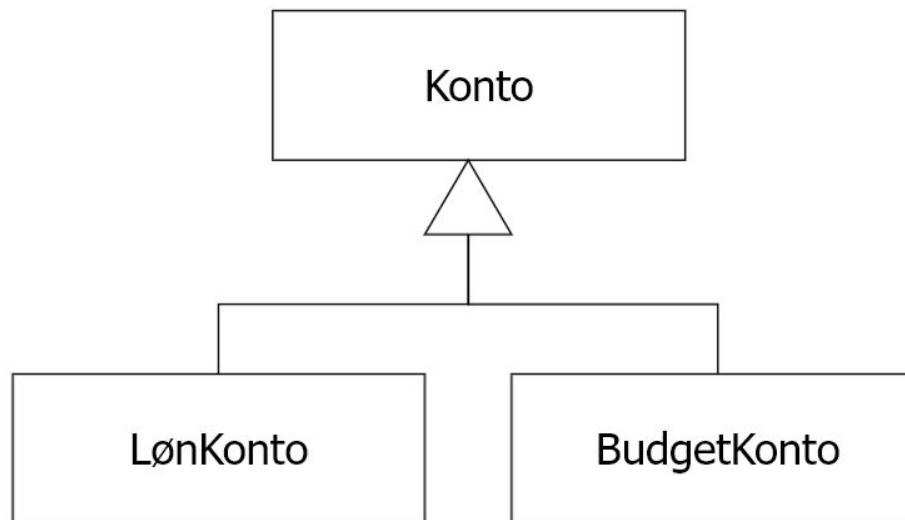


composition is a strong form of aggregation



Generalization - Inheritance

- Properties from superclass are inherited to subclasses
- Subclasses can add their own specialization



Abstract classes

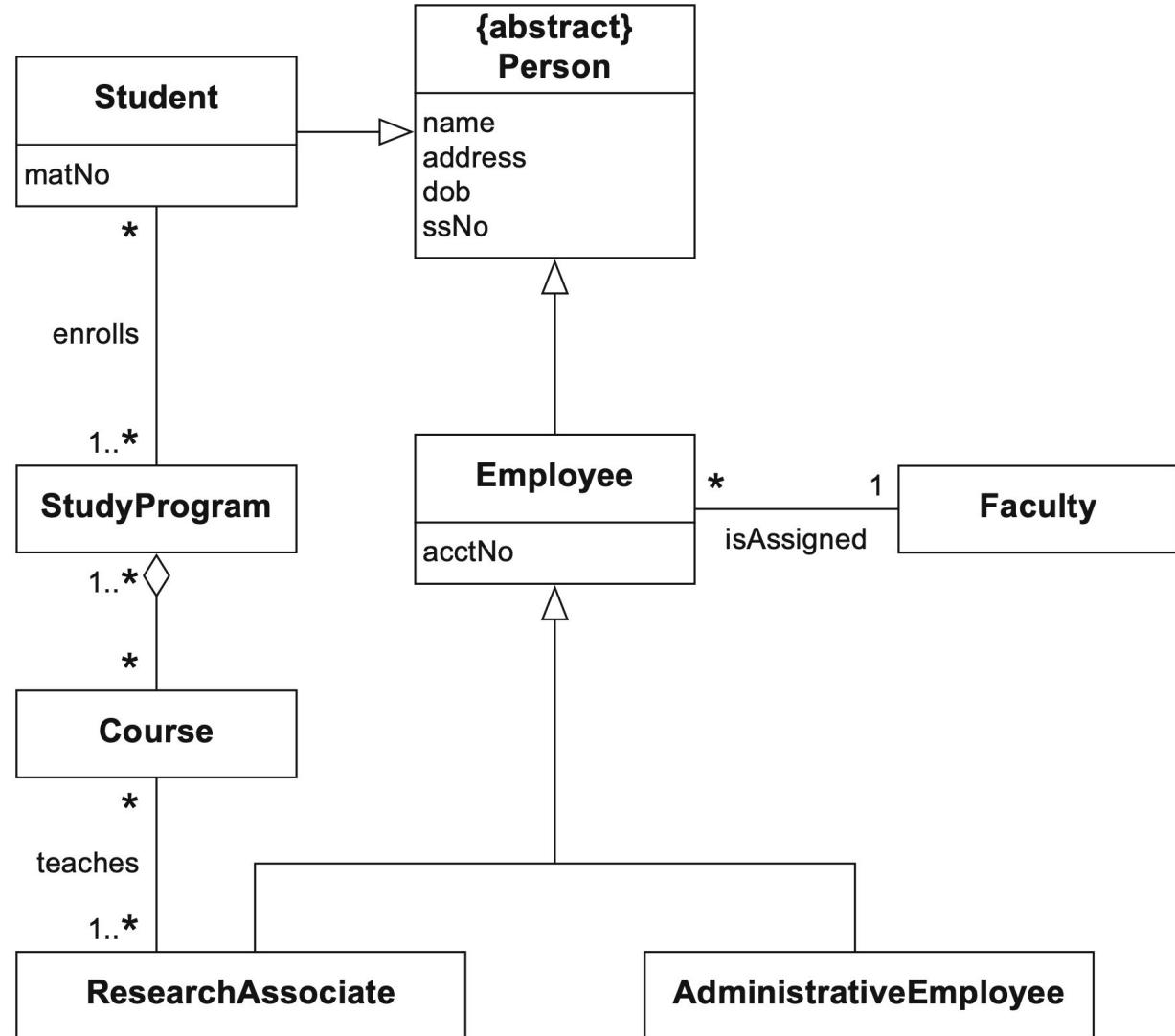
Cannot be instantiated

Person

{abstract}
Person

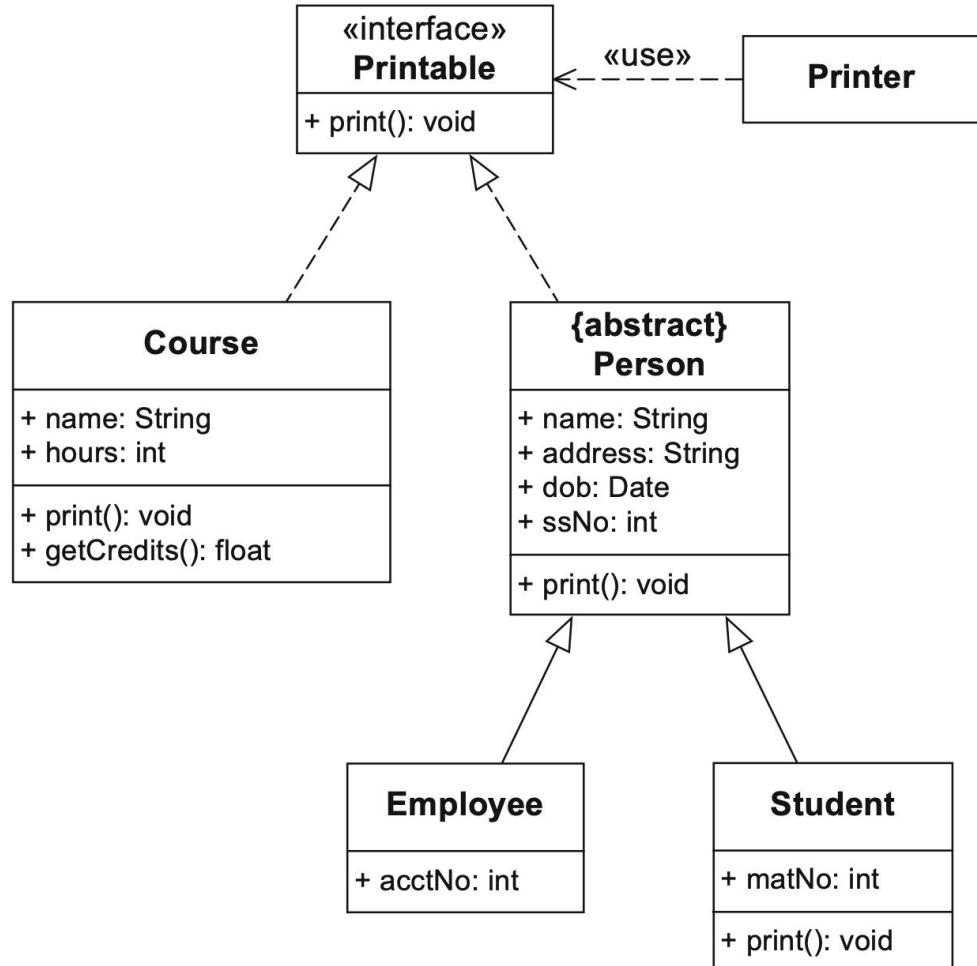
Inheritance

NB: Person cannot be instantiated



Interface

- Contract on functionality
- In family with abstract class
- Classes implement the interface
- Classes use the interface



Examples

<https://www.uml-diagrams.org/class-diagrams-overview.html>

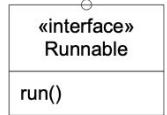
Class diagram syntax



3 common compartments

1. classifier name
2. attributes
3. operations

an interface shown with a keyword



interface implementation and subclassing

```

classDiagram
    SuperclassFoo
    or
    SuperClassFoo { abstract }

    - classOrStaticAttribute : Int
    + publicAttribute : String
    - privateAttribute
    assumedPrivateAttribute
    isInitializedAttribute : Bool = true
    aCollection : VeggieBurger [ * ]
    attributeMayLegallyBeNull : String [ 0..1 ]
    finalConstantAttribute : Int = 5 { readOnly }
    /derivedAttribute

    + classOrStaticMethod()
    + publicMethod()
    assumedPublicMethod()
    - privateMethod()
    # protectedMethod()
    ~ packageVisibleMethod()
    <<constructor>> SuperclassFoo( Long )
    methodWithParms(parm1 : String, parm2 : Float)
    methodReturnsSomething() : VeggieBurger
    methodThrowsException() { exception IOException }
    abstractMethod()
    abstractMethod2() { abstract } // alternate
    finalMethod() { leaf } // no override in subclass
    synchronizedMethod() { guarded }
  
```

```

classDiagram
    SubclassFoo
    or
    SubclassFoo { }

    ...
    run()
    ...
  
```

- ellipsis “...” means there may be elements, but not shown
- a *blank* compartment officially means “unknown” but as a convention will be used to mean “no members”

officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common

```

classDiagram
    java.awt::Font
    or
    java.awt.Font

    plain : Int = 0 { readOnly }
    bold : Int = 1 { readOnly }
    name : String
    style : Int = 0
    ...

    getFont(name : String) : Font
    getName() : String
    ...
  
```

dependency

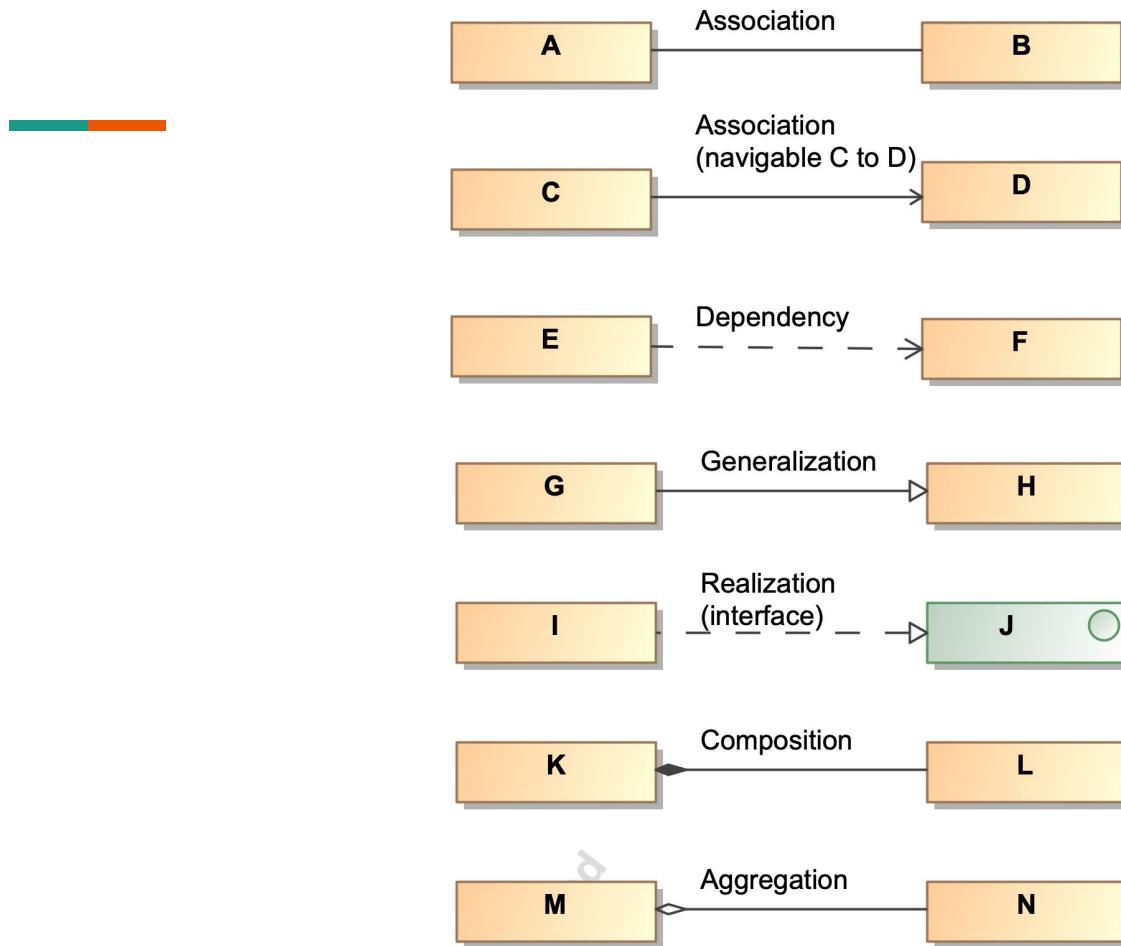
```

classDiagram
    Fruit
    ...
  
```

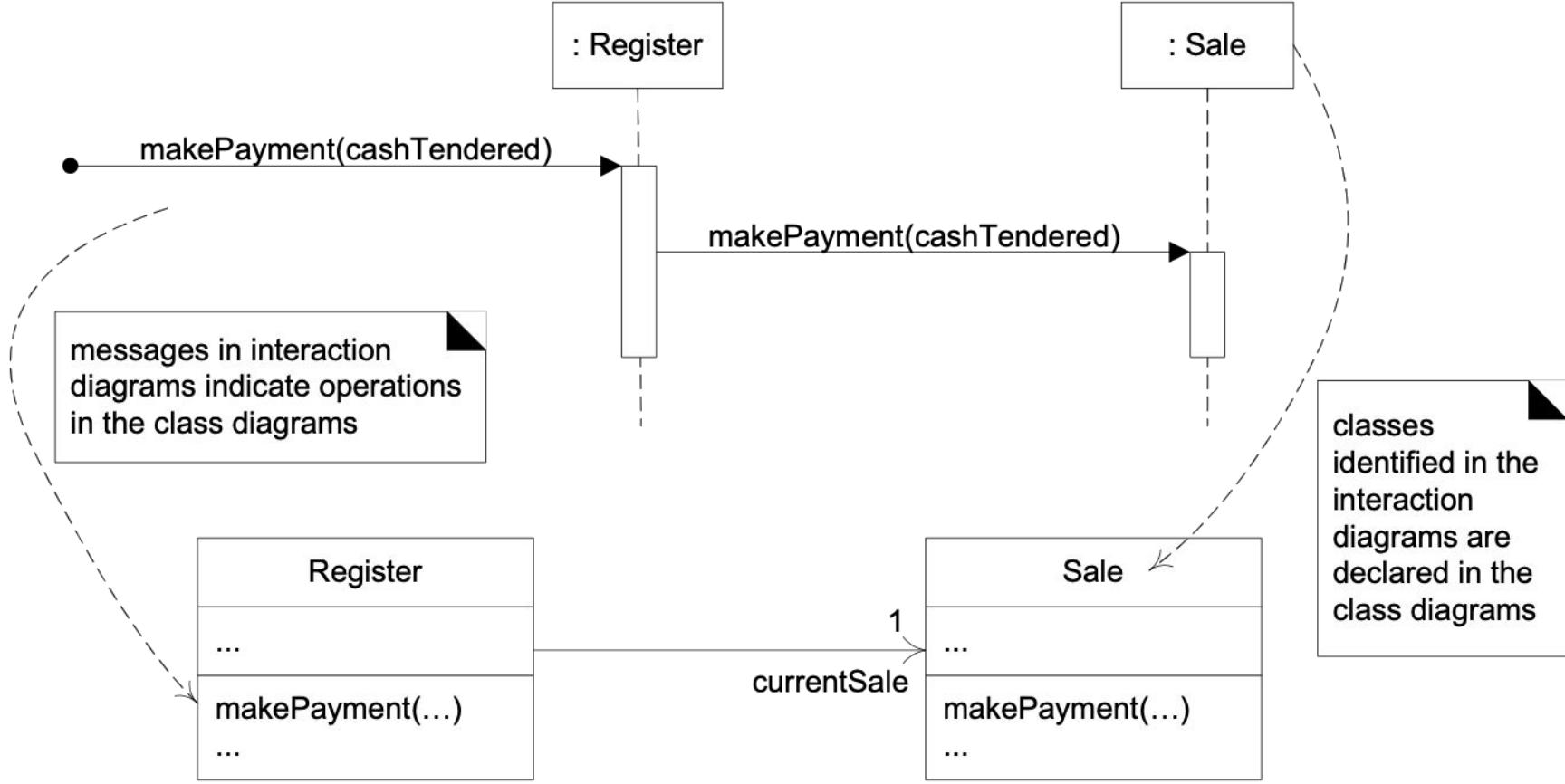
```

classDiagram
    PurchaseOrder
    ...
  
```

association with multiplicities



Classes in sequence diagrams



More about Design patterns and GRASP

Design pattern

- Design solution to common problem
- Generalizable

GRASP

- General Responsibility Assignment Software Patterns
- Classes have responsibilities

GoF

- Gang of Four
- Another set of software patterns

GRASP

- *Creator*
- *Information Expert*
- Low Coupling
- High Cohesion
- Controller
- (Polymorphism)
- (Pure Fabrication)
- (Indirection)
- (Protected Variations)

GRASP: Creator

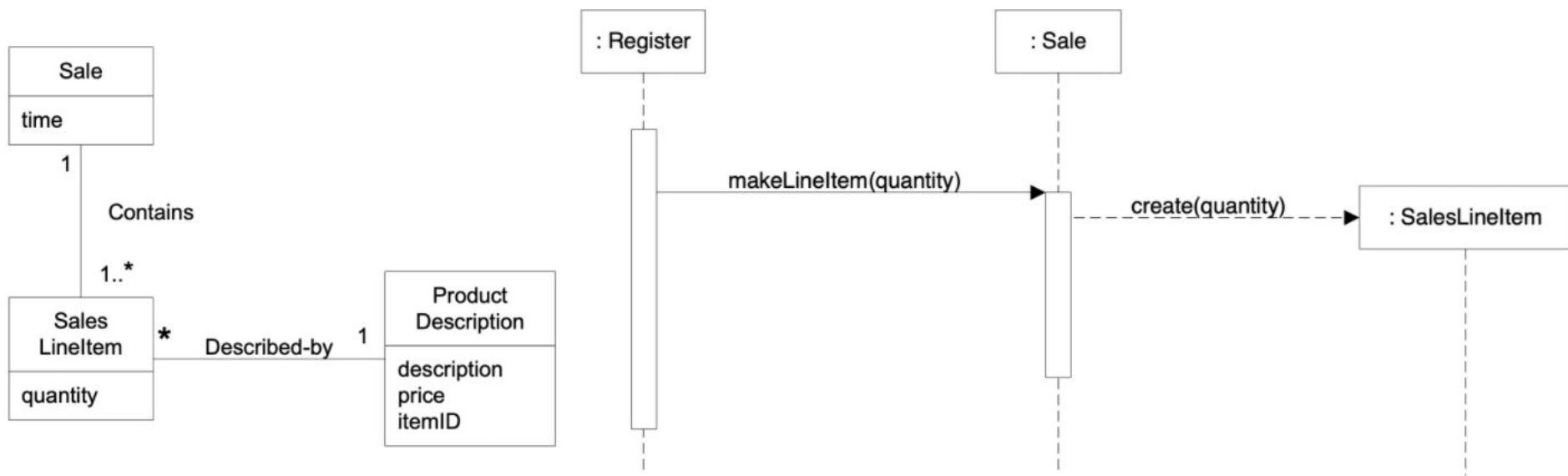
Who is responsible for creating a class?

B is the creator of A if:

- B aggregates A
- B contains A
- B uses A
- B has initializing data for A

GRASP: Creator

Who creates SalesLineItem ?



Larman, Fig.17.12

Larman, Fig.17.13

GRASP: Information Expert

Principle for assigning responsibility to objects

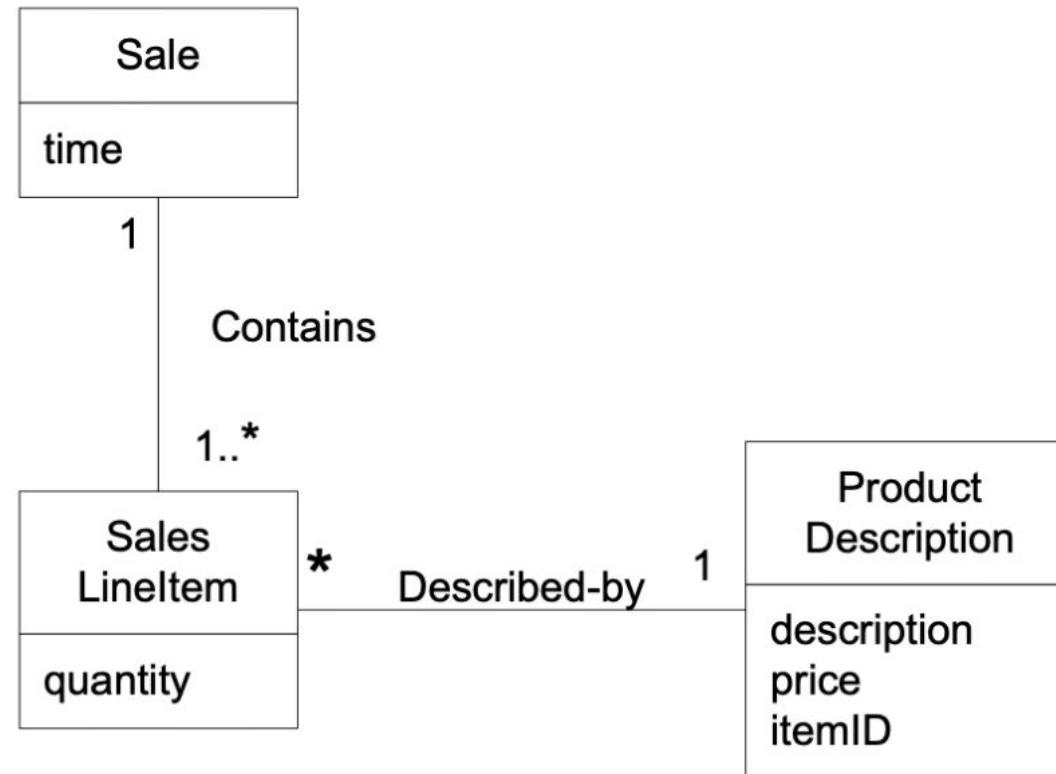
- The class has the necessary information to fulfill the responsibility

If there are relevant classes in the design model, this is used

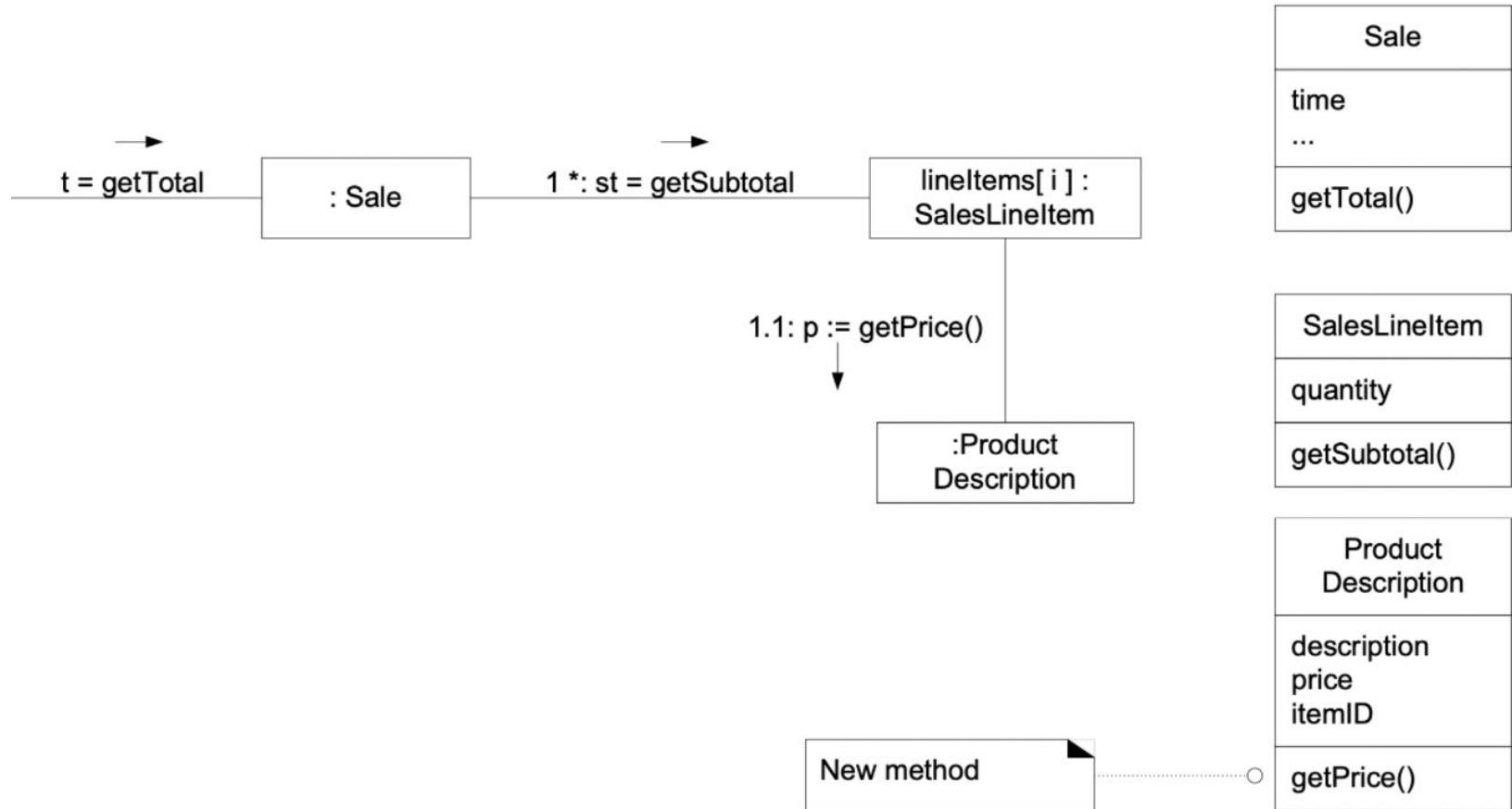
- Otherwise, the domain model is used

GRASP: Information Expert

- Who is responsible for Total?



GRASP: Information Expert



GRASP: Low Coupling

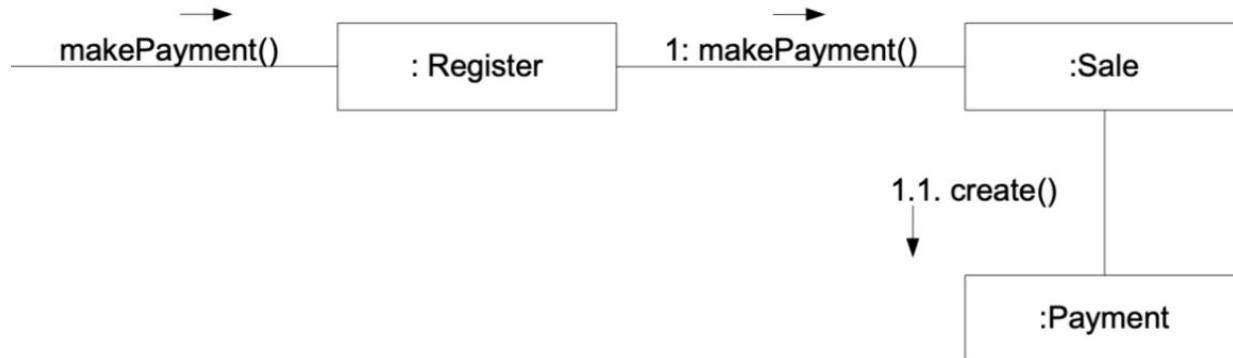
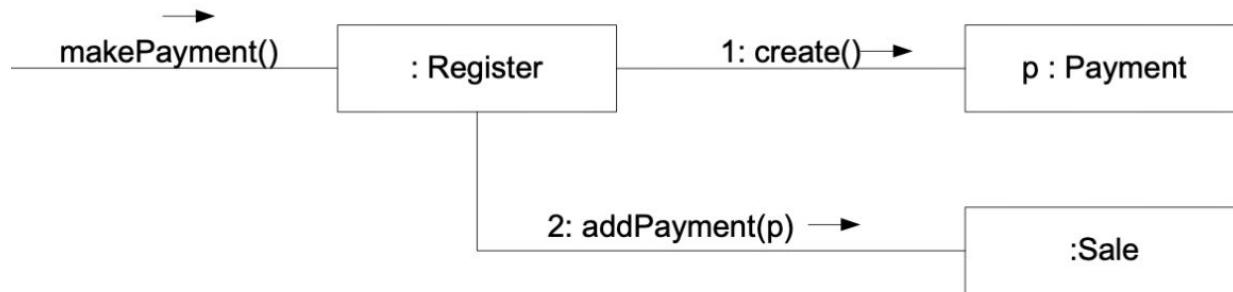
How to ensure low dependency between classes and thus e.g.

- isolation of changes
- easier to understand the design
- easier to recycle

Assign responsibility so coupling is low.

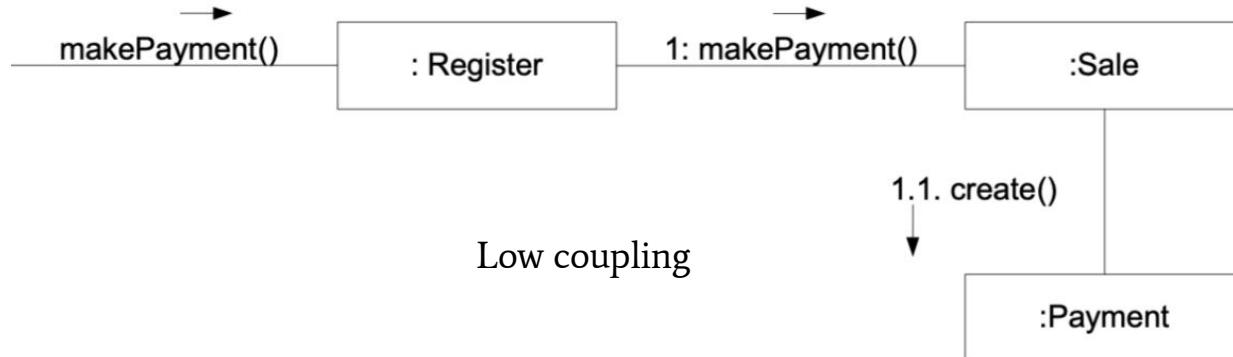
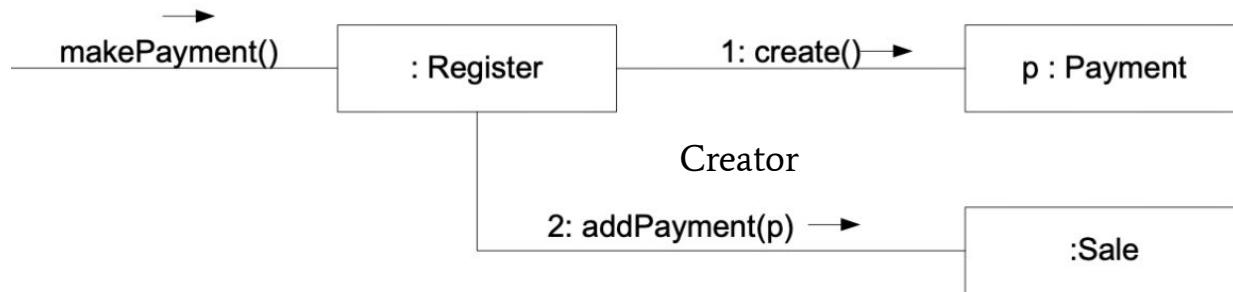
GRASP: Low Coupling

- Which class should create a "Payment" object?



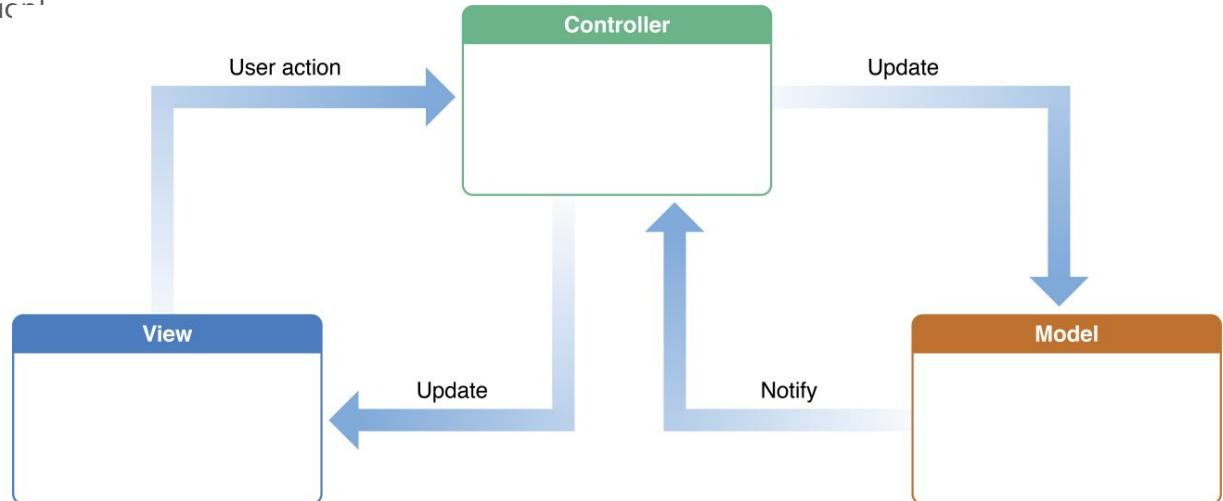
Opposition between Creator and Low Coupling...

- Which class should create a "Payment" object?



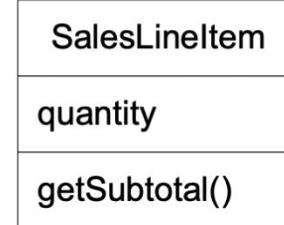
GRASP: Controller

- Responsible for handling input events?
- Responsible for communicating changes in the model layer to the view layer?
- → Which classes should be Controllers.
 - Typically a controller per use case scenario
 - Use case controllers
- Beware
 - Controller Bloat - Alt for store controllers
 - Lack of cohesion



GRASP: High Cohesion

- Uniform area of responsibility
- Supports Low Coupling
 - If there are several areas of responsibility in a class, several links quickly arise
- Better understanding
- Easier maintenance
- Easier Tests
- Easier recycling



Development methods 62531

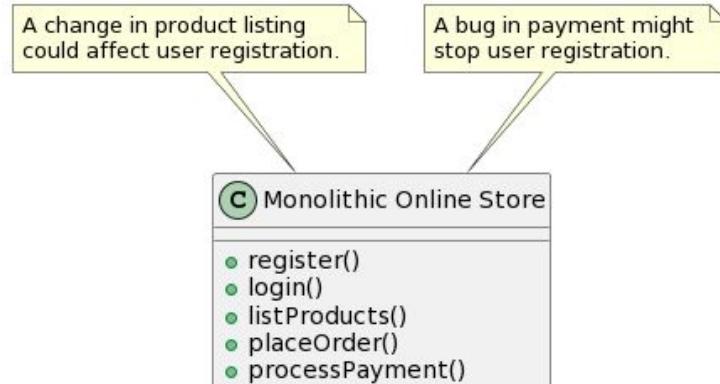
Lecture 8 - More GRASP and Design
Lei You, Assistant Professor - leiyo@dtu.dk

What have we learned?

- Architecture
- Design class diagrams
 - UML class notation
 - UML class association
- GRASP (sneak peak)
 - General Responsibility Assignment Software Patterns

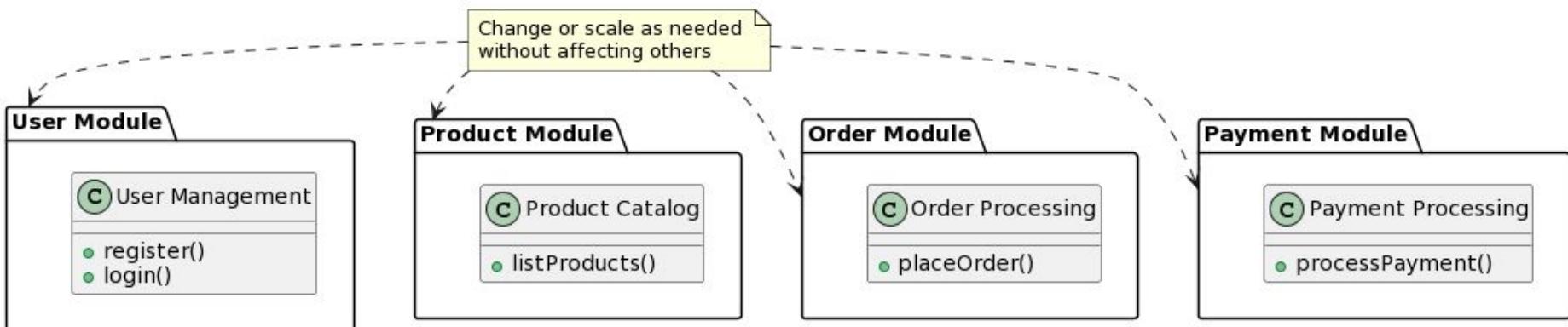
Separation of Concerns

- Why modularity?
 - Division of development
 - Readability
 - Easier fault isolation
 - Easier tests
 - Recycling
 - Easier replacement
- Example:



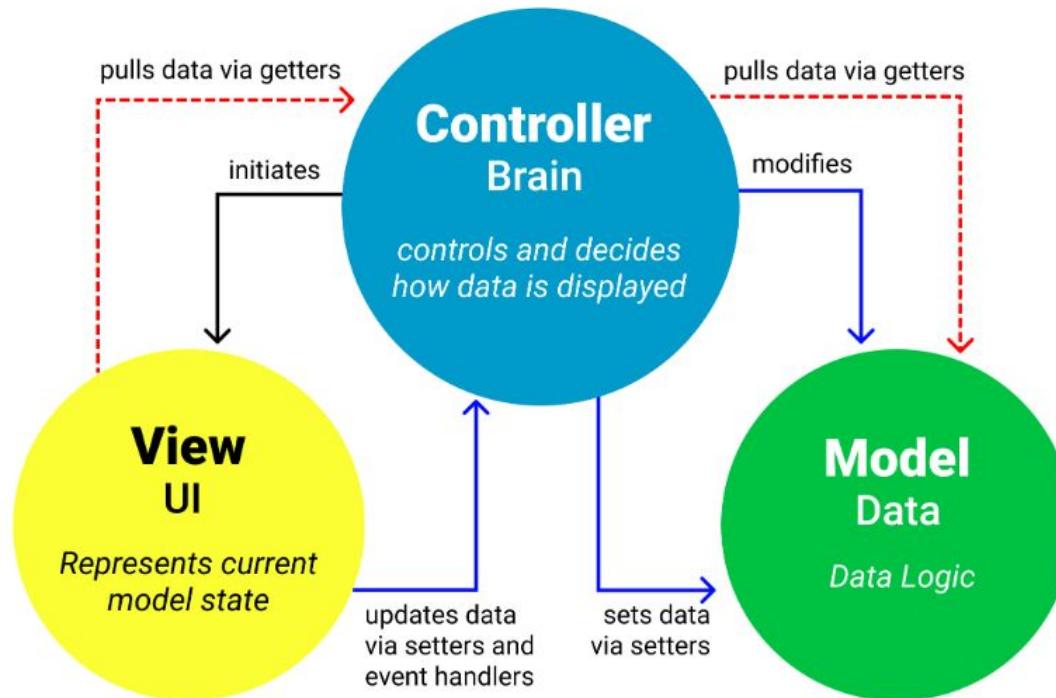
GRASP: Low Coupling and High Cohesion

- Low Coupling
 - A class should be as independent as possible
 - A class should be associated with only the few classes necessary for it to fulfill its scope of responsibility
- High cohesion
 - A class must have a well-defined area of responsibility
 - A class must have a set of operations that support the scope of responsibility



Model View Controller

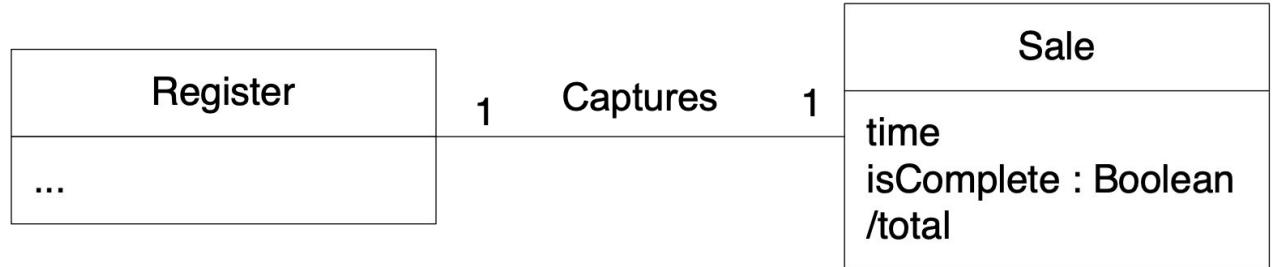
MVC Architecture Pattern



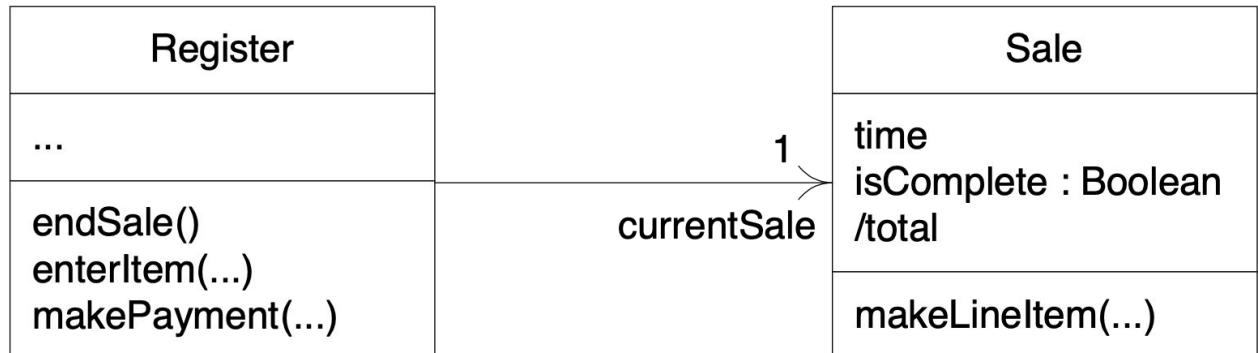
Class Diagrams: Domain vs Design

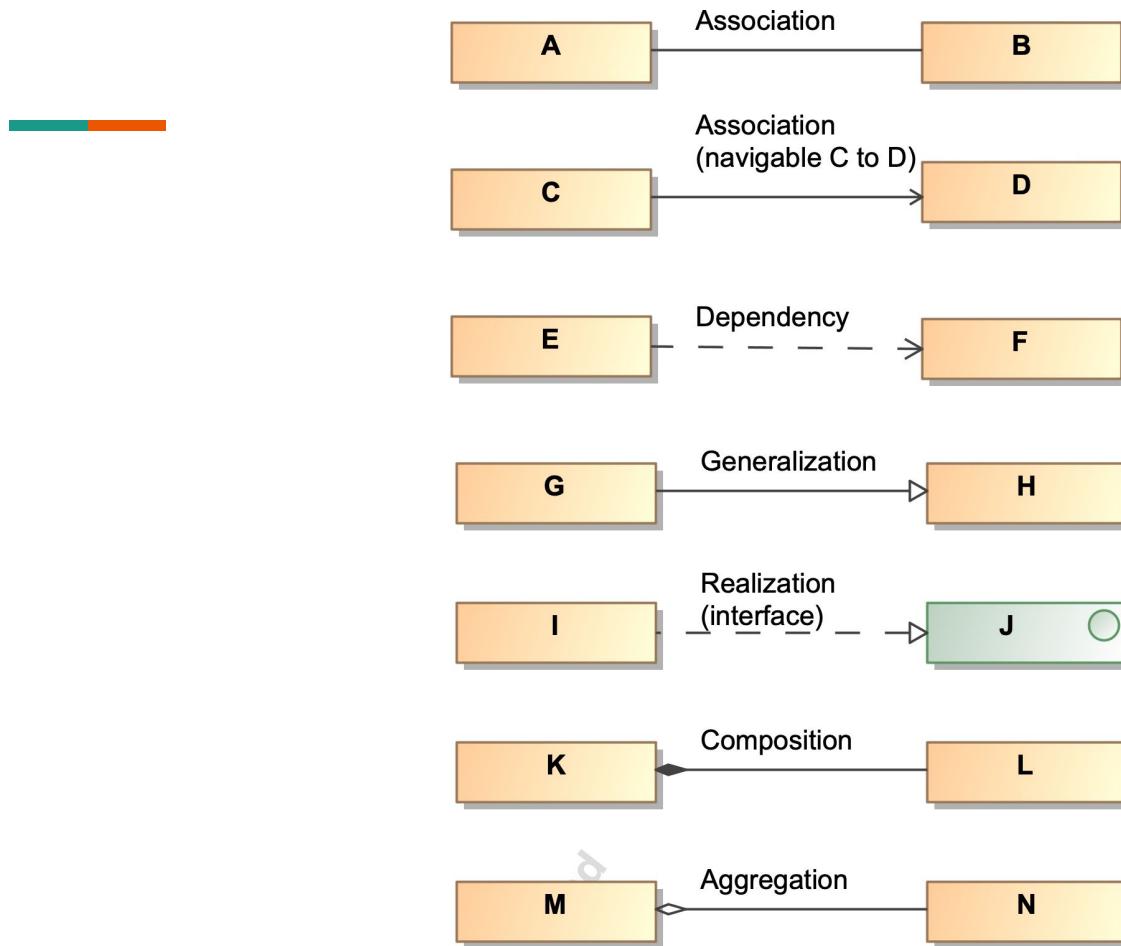


Domain Model
conceptual perspective



Design Model
DCD; software perspective





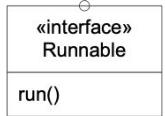
Class diagram syntax



3 common compartments

1. classifier name
2. attributes
3. operations

an interface shown with a keyword



interface implementation and subclassing

```

classDiagram
    SuperclassFoo
    or
    SuperClassFoo { abstract }

    - classOrStaticAttribute : Int
    + publicAttribute : String
    - privateAttribute
    assumedPrivateAttribute
    isInitializedAttribute : Bool = true
    aCollection : VeggieBurger [ * ]
    attributeMayLegallyBeNull : String [ 0..1 ]
    finalConstantAttribute : Int = 5 { readOnly }
    /derivedAttribute

    + classOrStaticMethod()
    + publicMethod()
    assumedPublicMethod()
    - privateMethod()
    # protectedMethod()
    ~ packageVisibleMethod()
    <<constructor>> SuperclassFoo( Long )
    methodWithParms(parm1 : String, parm2 : Float)
    methodReturnsSomething() : VeggieBurger
    methodThrowsException() { exception IOException }
    abstractMethod()
    abstractMethod2() { abstract } // alternate
    finalMethod() { leaf } // no override in subclass
    synchronizedMethod() { guarded }
  
```

```

classDiagram
    SubclassFoo
    or
    SubclassFoo { }

    ...
    run()
    ...
  
```

- ellipsis “...” means there may be elements, but not shown
- a *blank* compartment officially means “unknown” but as a convention will be used to mean “no members”

officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common

```

classDiagram
    java.awt::Font
    or
    java.awt.Font

    plain : Int = 0 { readOnly }
    bold : Int = 1 { readOnly }
    name : String
    style : Int = 0
    ...

    getFont(name : String) : Font
    getName() : String
    ...
  
```

dependency

```

classDiagram
    Fruit
    ...
    ...
  
```

```

classDiagram
    PurchaseOrder
    ...
    ...
  
```

association with multiplicities

Today's program

- Feedback on evaluation
 - www.evaluering.dtu.dk
- GRASP
 - General Responsibility Assignment Software Patterns
 - Classes have responsibilities
- Design Pattern
 - Design solution to common problem
 - Generalizable
- GoF
 - Gang of Four
 - Another set of software patterns
- Use Case Realization (Self Reading)

GRASP - 9 principles in total

- *Creator*
- *Information Expert*
- **Low Coupling**
- **High Cohesion**
- Controller
- (Polymorphism)
- (Pure Fabrication)
- (Indirection)
- (Protected Variations)

GRASP: Creator

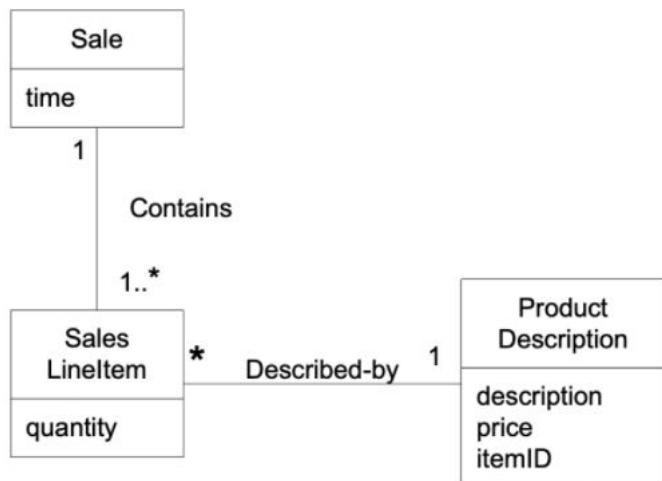
Who is responsible for creating a class?

B is the creator of A if:

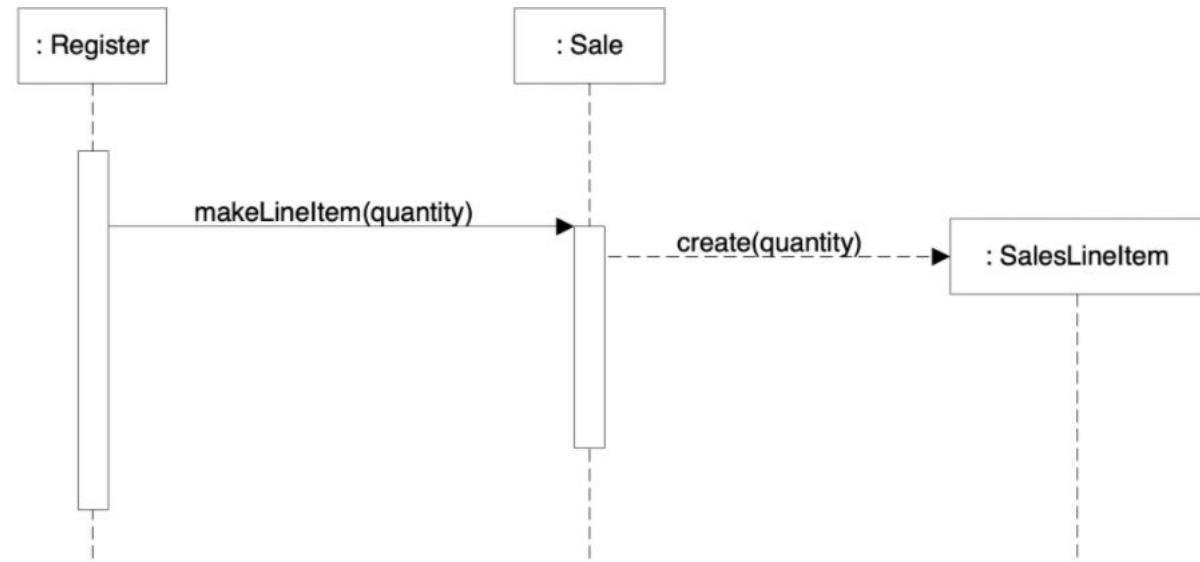
- B aggregates A
- B contains A
- B uses A
- B has initializing data for A

GRASP: Creator

Who creates SalesLineItem ?



Larman, Fig.17.12



Larman, Fig.17.13

GRASP: Information Expert

Principle for assigning responsibility to objects

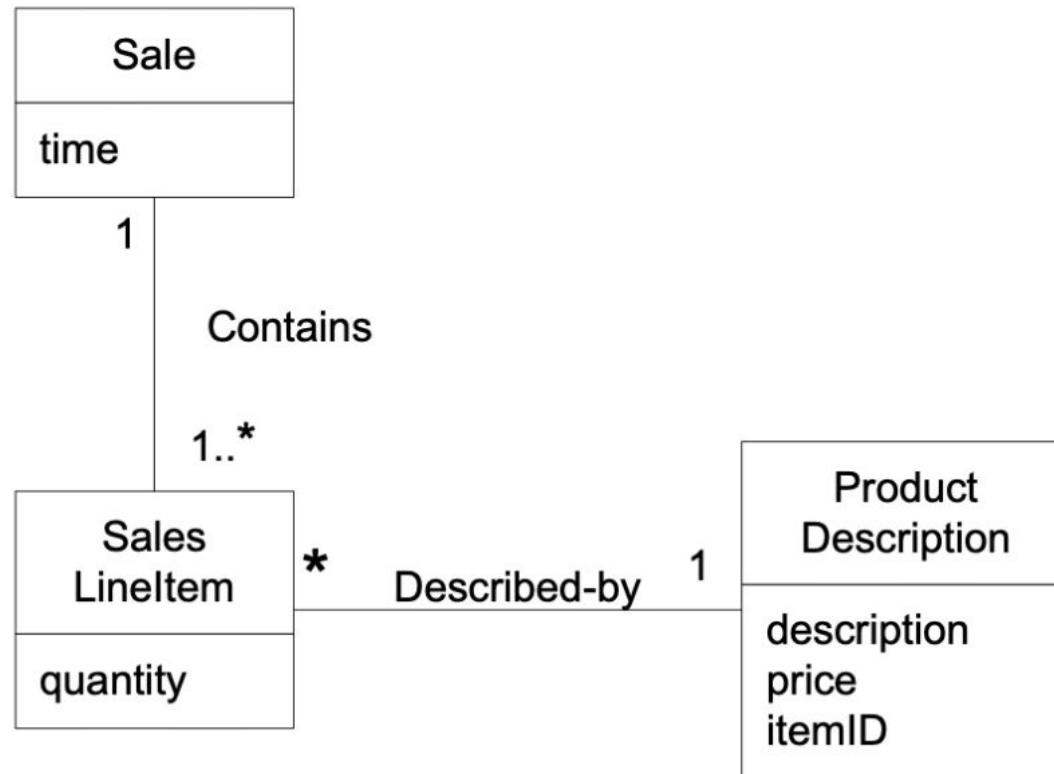
- The class has the necessary information to fulfill the responsibility

If there are relevant classes in the design model, this is used

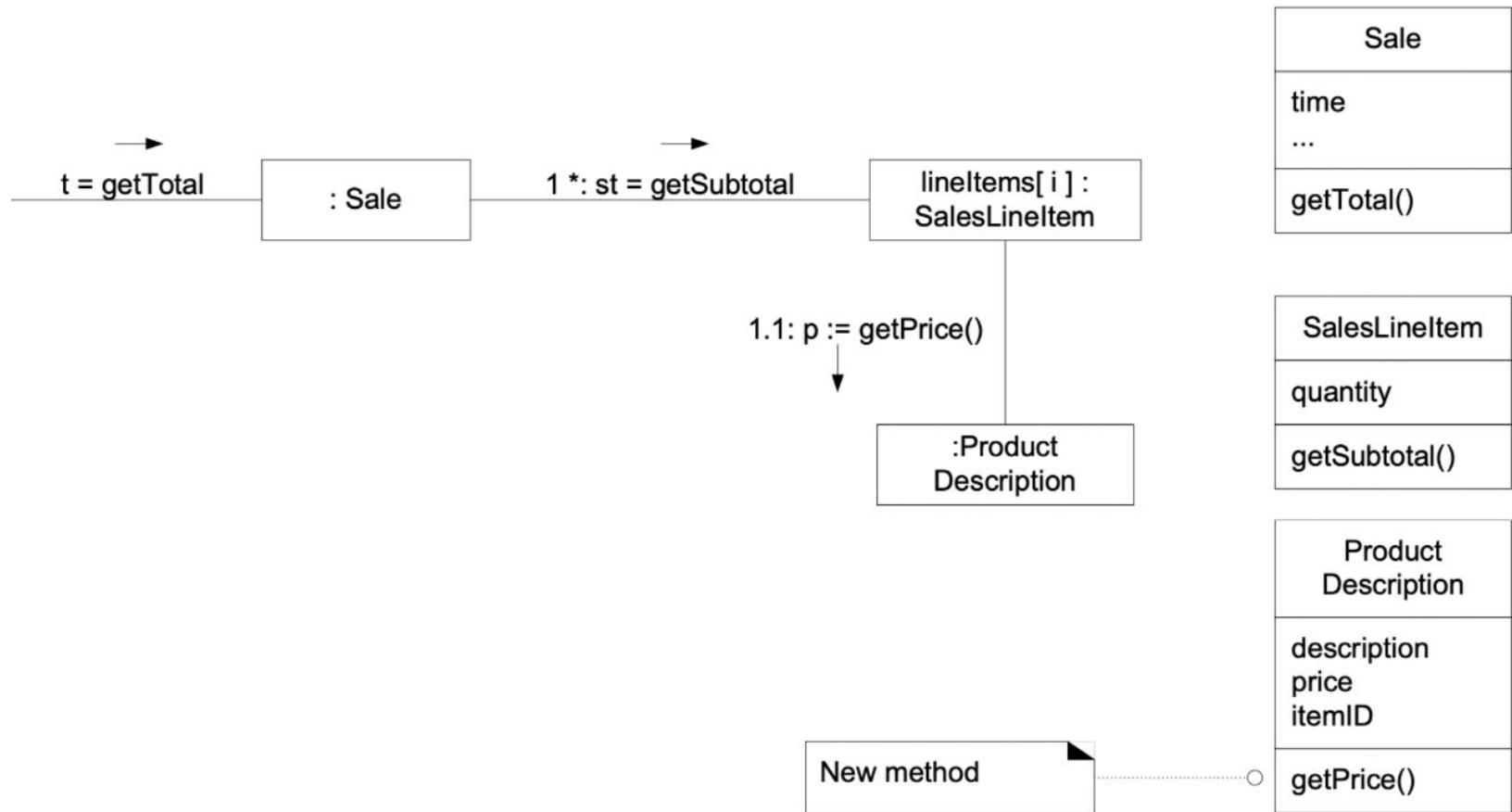
- Otherwise, the domain model is used

GRASP: Information Expert

- Who is responsible for Total?



GRASP: Information Expert



GRASP: Low Coupling

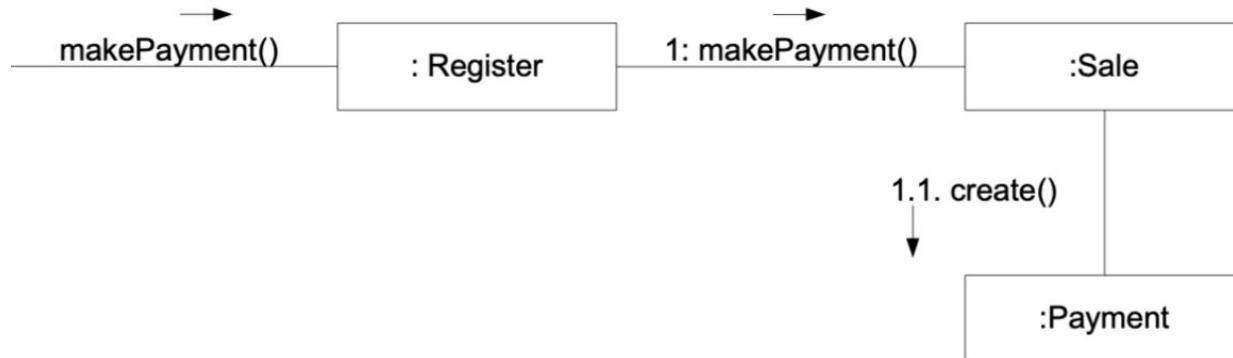
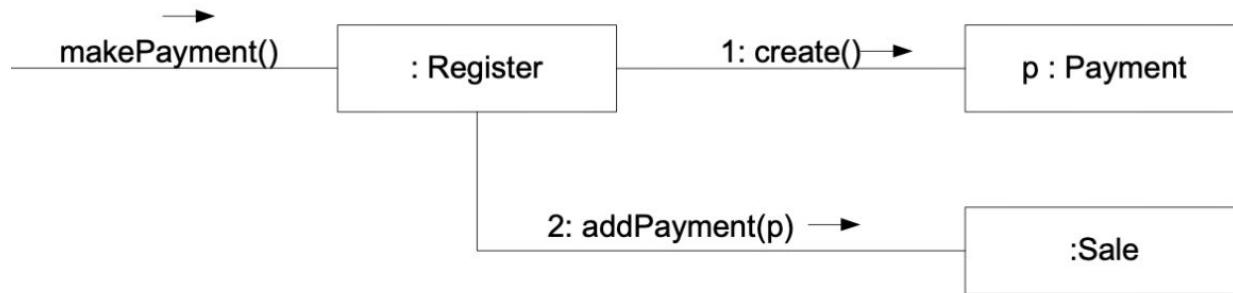
How to ensure low dependency between classes and thus e.g.

- isolation of changes
- easier to understand the design
- easier to recycle

Assign responsibility so coupling is low.

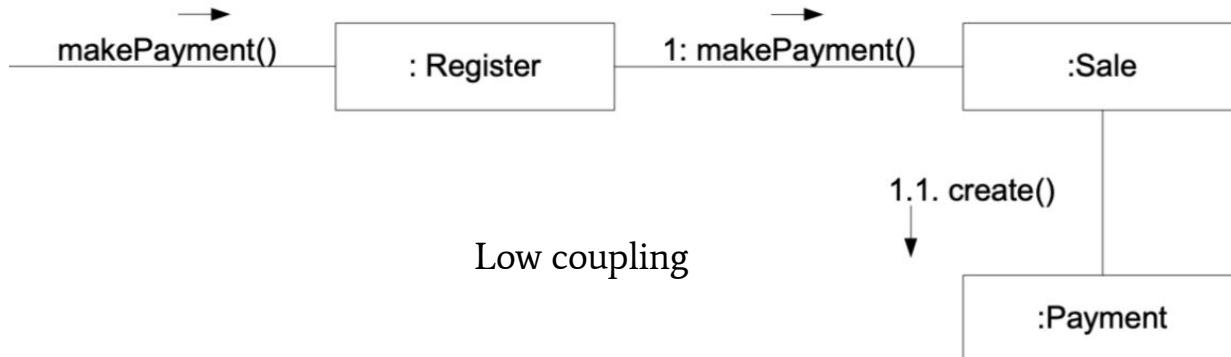
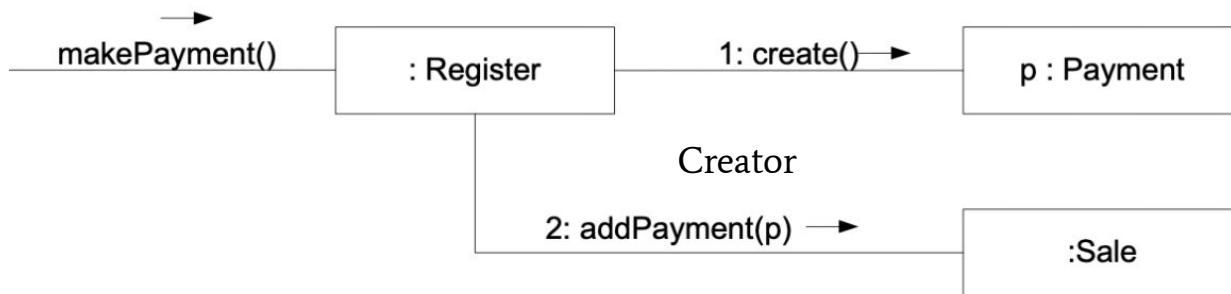
GRASP: Low Coupling

- Which class should create a "Payment" object?



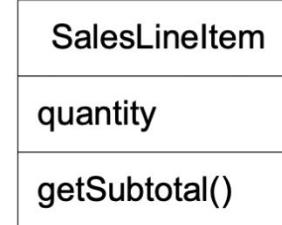
Opposition between Creator and Low Coupling...

- Which class should create a "Payment" object?



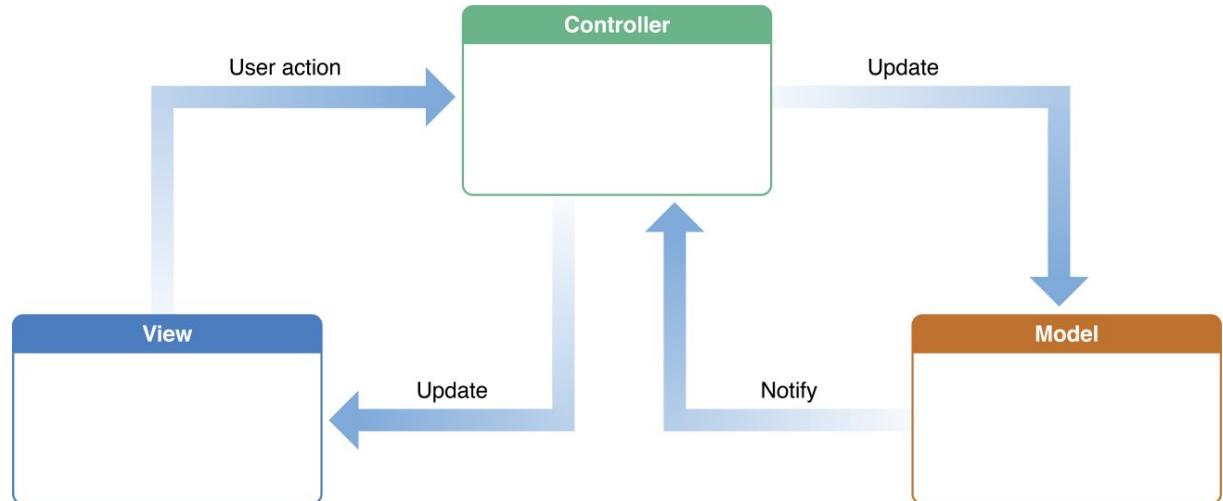
GRASP: High Cohesion

- Uniform area of responsibility
- Supports Low Coupling
 - If there are several areas of responsibility in a class, several links quickly arise
- Better understanding
- Easier maintenance
- Easier Tests
- Easier recycling

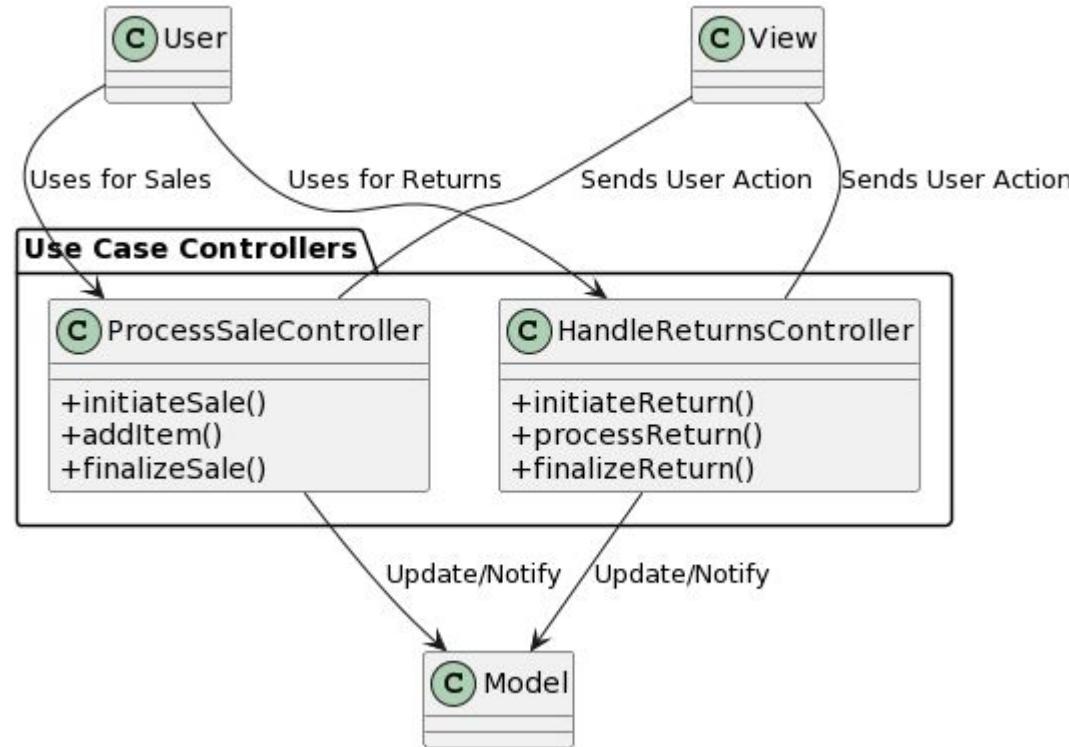


GRASP: Controller

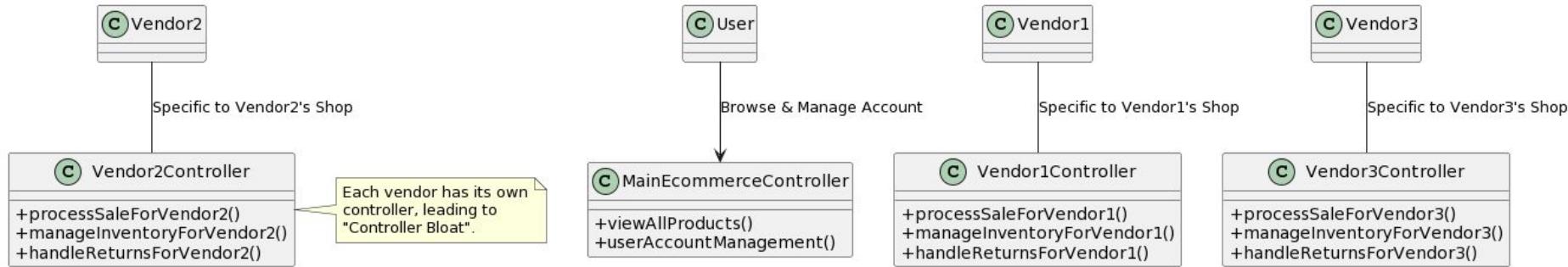
- Responsible for handling input events
- Responsible for communicating changes in the model layer to the view layer
- → Which classes should be Controllers.
 - Typically a controller per use case scenario
 - Use case controllers



GRASP: Controller - Use Case Controllers



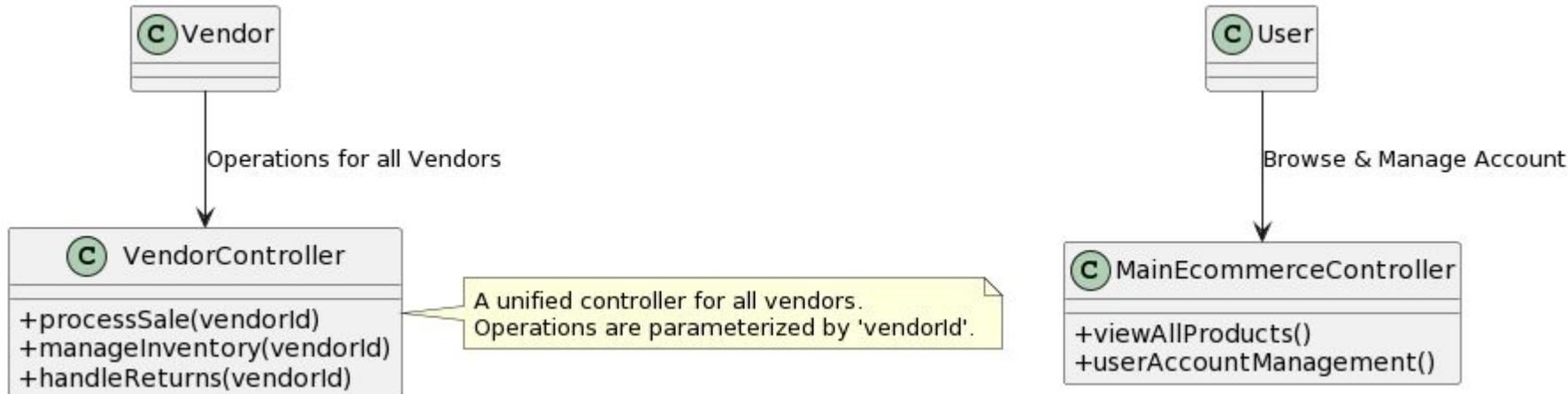
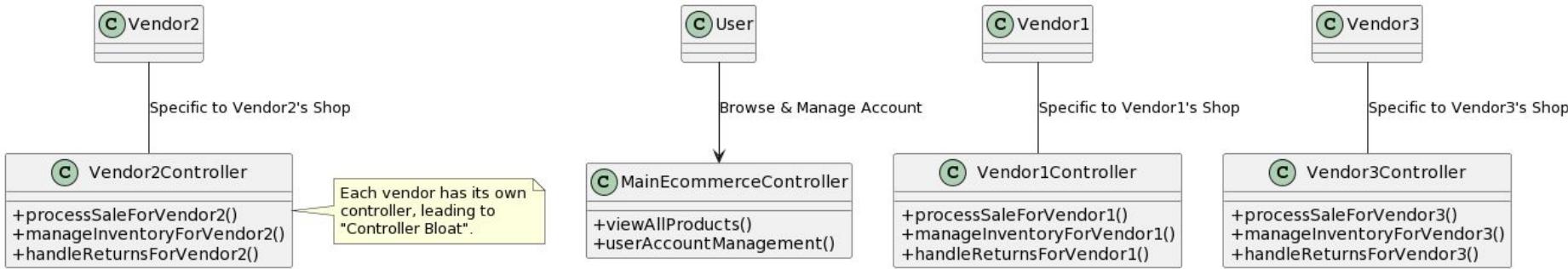
GRASP: Controller



Assign 'Controller responsibility' to:

- System class
- Class representing the use case scenario
 - eg: "ProcessSaleHandler"
 - (Not included in the Domain model - invented for the occasion)
- Beware
 - Controller Bloat - Alt for store controllers
 - Lack of cohesion!

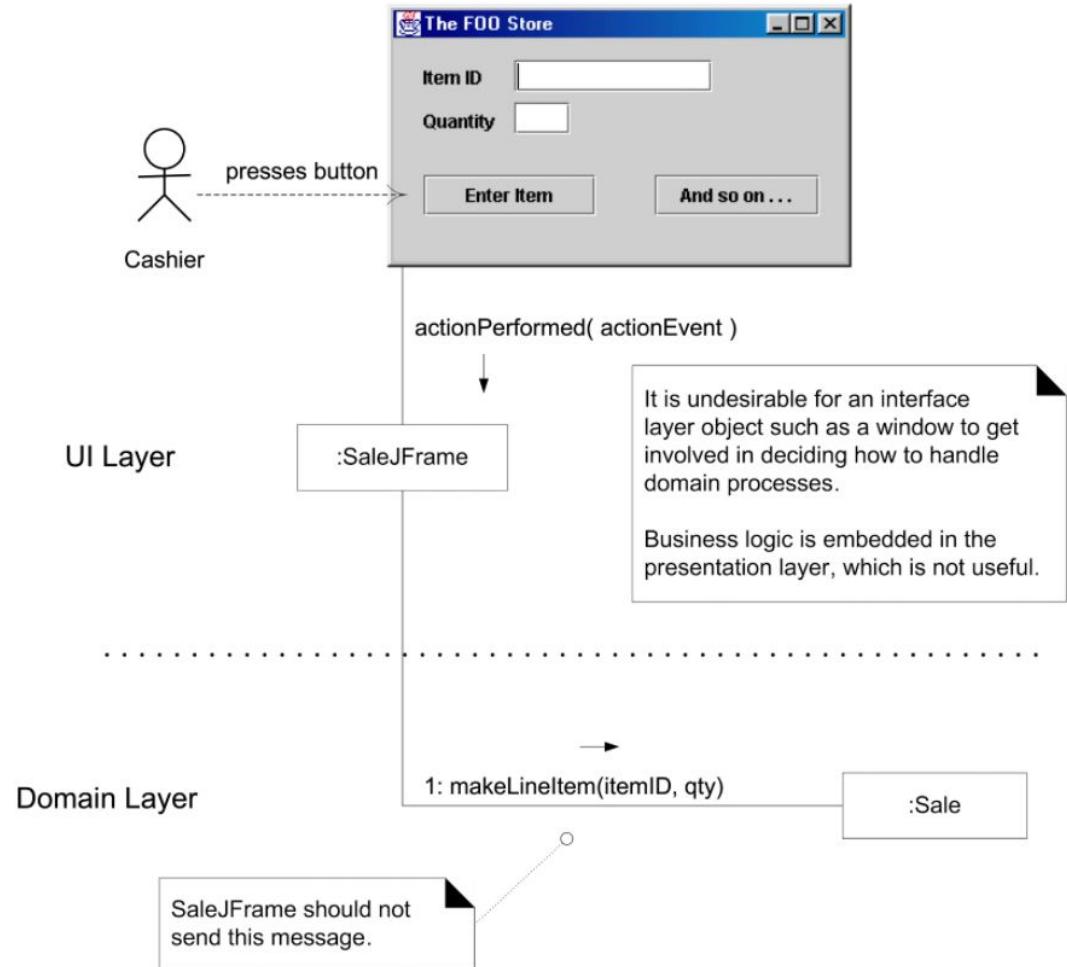
GRASP: Controller



Bad design

What's wrong here?

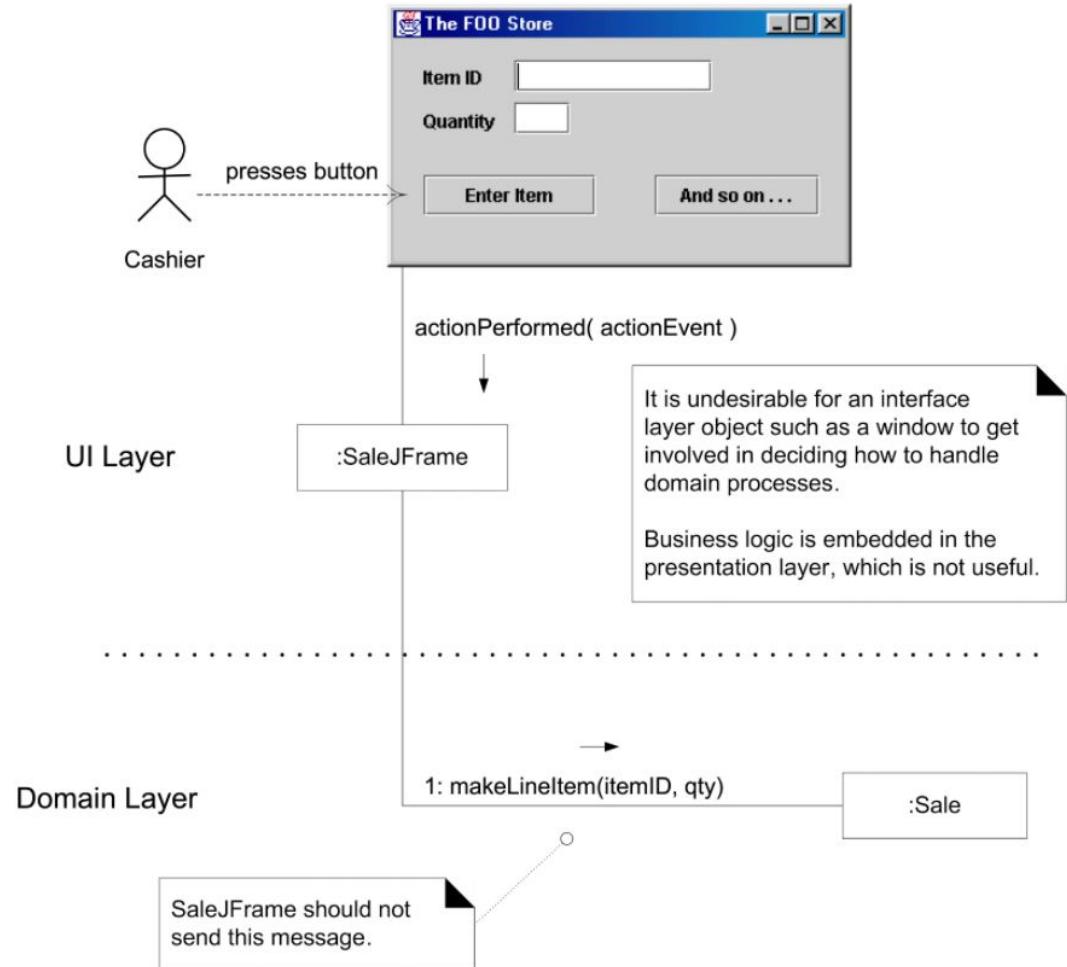
Which concepts are we breaking with?



Bad design

Bad separations of concerns (SoC)

Low cohesion



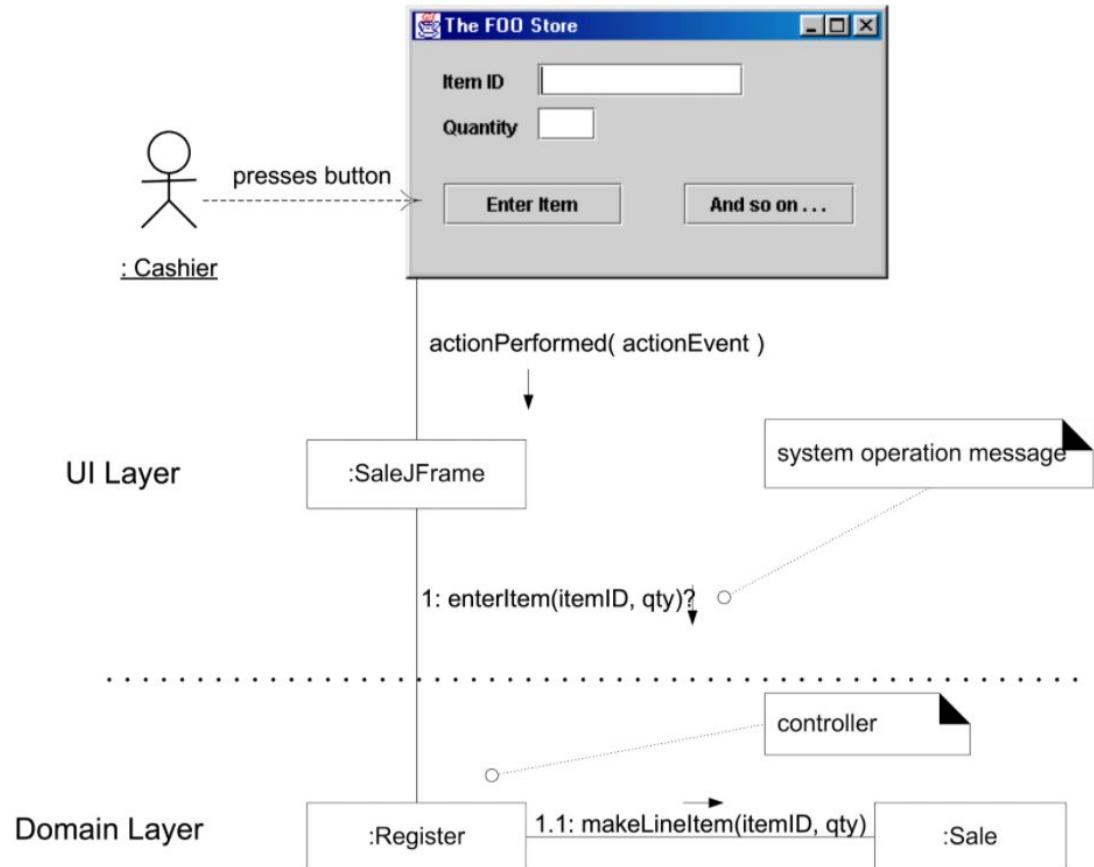
Bad design

High Cohesion

UI layer takes care of UI message

Domain layer handles Business logic

Correct MVC



GRASP - Polymorphism

"Many Shapes" - Greek words 'poly' (many) and 'morph' (shape)

"When several types of objects can do something uniform"

Ex.

Polymorphism with Inheritance

```
abstract class Animal {  
    abstract String talk();  
}  
  
class Cat extends Animal {  
    String talk() {  
        return "Meow!";  
    }  
}  
  
class Dog extends Animal {  
    String talk() {  
        return "Woof!";  
    }  
}  
  
static void letsHear(final Animal a) {  
    println(a.talk());  
}  
  
static void main(String[] args) {  
    letsHear(new Cat());  
    letsHear(new Dog());  
}
```

Polymorphism (with Inheritance)

Subtyping and Interfaces - Run time polymorphism - classic polymorphism

- Objects are treated as if they were their supertype
 - letsHear(Animal a);
 - letsHear(new Cat());
- Declared type != run-time type

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

Polymorphism (Parametric)

Parametric Polymorphism (Java Generics) -

- Actually the same concept, but in different packaging
- In the 2nd semester

```
1 public class Box<T> {  
2     private T t;  
3  
4     public void set(T t) {  
5         this.t = t;  
6     }  
7  
8     public T get() {  
9         return t;  
10    }  
11 }
```

```
1 Box<Integer> integerBox = new Box<Integer>();  
2 Box<String> stringBox = new Box<String>();
```

Polymorphism

Controversial! - People argue for hours online about polymorphism

Method Overloading - *Compile-time* "polymorphism"

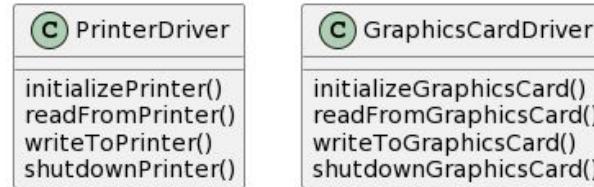
- 2 methods with the same name but different signature
 - Actually two completely different methods
 - pseudo-polymorphism (but practical ;)

```
class Dog {  
    public void bark() {  
        System.out.println("woof ");  
    }  
  
    //overloading method  
    public void bark(int num) {  
        for(int i=0; i<num; i++)  
            System.out.println("woof ");  
    }  
}
```

Polymorphism - Real-World Use Cases

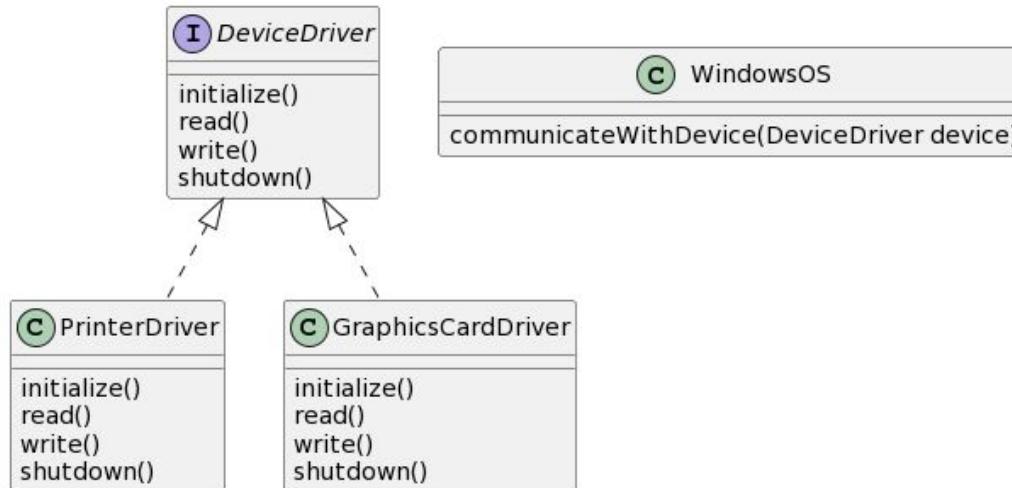
Windows supports a wide variety of hardware devices like printers, graphics cards, audio devices, and more. The OS needs a standardized way to communicate with all these varied hardware components without knowing the specifics of each.

Imagine if Windows had to have a unique way of communicating with each specific model of every hardware device. This would require the operating system to be updated constantly as new hardware models are released, and the codebase would become immensely complex and unmanageable.



Polymorphism - Real-World Use Cases

In the diagram, **WindowsOS** uses the **DeviceDriver interface** to communicate with devices. **PrinterDriver** and **GraphicsCardDriver** are examples of specific drivers that implement this interface. This design allows new devices to be easily supported: a manufacturer simply writes a new driver that implements the **DeviceDriver** interface.



Polymorphism implementation

- Inheritance
 - Same method signature, but different implementation
 - Different child classes can have the same method name (inherited from the parent class) but with a distinct implementation.
- Interfaces
 - A contract that classes can implement. It specifies a set of methods that the implementing class must define.
 - Classes implementing the same interface can be used interchangeably, making the system modular and extensible.
- (Polymorphic methods)
 - Having multiple methods in the same class with the same name but different parameters.
 - The correct method to be called is determined at compile-time based on the method signature.

Inheritance, interfaces: When to use which?

Polymorphism: Inheritance vs. Interfaces

Inheritance Polymorphism (Subtype Polymorphism)

- Nature
 - Derived from the parent-child relationship between classes.
 - Child class inherits properties and methods from the parent class.
- Key characteristics
 - Most object-oriented languages like Java and C# allow a class to inherit from only one parent class.
 - Child class can override or extend methods inherited from the parent class.
- When to use?
 - When there's a clear "is-a" relationship. E.g., a Dog is an Animal.
 - When you want to reuse and extend the base functionality of the parent class.

Polymorphism: Inheritance vs. Interfaces

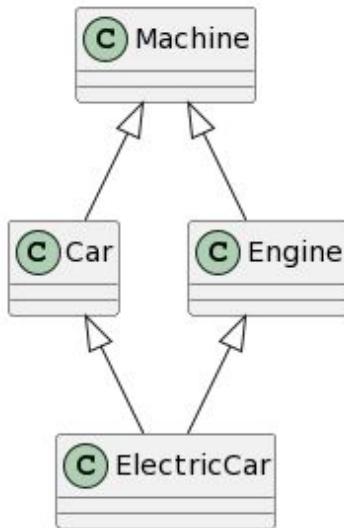
Interfaces Polymorphism

- Nature
 - Defines a contract (set of methods) that implementing classes must adhere to.
 - No implementation details; only method declarations.
- Key characteristics
 - A class can implement multiple interfaces, offering more flexibility in design.
 - Classes implementing an interface are contractually obligated to provide an implementation for all of its declared methods.
- When to use?
 - When various classes need to adhere to a specific contract but don't share a common parent-child relationship
 - When you want to ensure certain classes have specific methods, regardless of their position in the class hierarchy.
 - When you're working in a language that doesn't support multiple inheritance (like Java), interfaces can provide a workaround to implement functionality from multiple sources.

Why not multiple inheritance?

```
class Machine:  
    def __init__(self, color, brand):  
        self.color = color  
        self.brand = brand  
  
    def start(self):  
        print("Machine started")  
  
    def stop(self):  
        print("Machine stopped")
```

```
class Car(Machine):  
    def __init__(self, color, brand, model):  
        super().__init__(color, brand)  
        self.model = model  
        super().start()  
  
    def start(self):  
        print("Car started")  
  
    def display(self):  
        print(f"This is a {self.color} {self.brand} {self.model}")
```



```
class Engine(Machine):  
    def set_power(self, power):  
        self.power = power  
  
    def start(self):  
        print("Engine Started")  
  
class ElectricCar(Car, Engine):  
    pass
```

Question: What's the output after executing this line below?

```
tesla = ElectricCar("White", "Tesla", "Model S")
```

Why not multiple inheritance?

Method Resolution Order (MRO) is a technique used primarily in object-oriented programming languages to determine the order in which base classes are searched when calling a method on a derived class.

Why do we need MRO?

In multiple inheritance scenarios, there can be ambiguities about which method to call if more than one parent class has a method with the same name. MRO provides a clear, **linear order** in which classes are checked, ensuring that a method is determined unambiguously.



```
print(ElectricCar.mro())
```

```
[<class '__main__.ElectricCar'>, <class '__main__.Car'>, <class '__main__.Engine'>, <class '__main__.Machine'>, <class 'object'>]
```

Why not multiple inheritance?



```
print(ElectricCar.mro())
```

```
[<class '__main__.ElectricCar'>, <class '__main__.Car'>, <class '__main__.Engine'>, <class '__main__.Machine'>, <class 'object'>]
```

So, what happened?

1. The constructor of Car is called (since ElectricCar doesn't have its own constructor).
2. Within the Car constructor, super().start() is invoked.
3. Based on the MRO, the next class that has a start method after Car is Engine, so the start method of Engine is executed, leading to "Engine Started" being printed.

(If you were to remove the start method from Engine, then the start method from Machine would be called, printing "Machine started".)

GRASP: Pure Fabrication

Introducing a class that does not exist in the problem domain. Namely, the class isn't necessarily a direct reflection of real-world objects or concepts but is introduced for technical reasons.

- To achieve low coupling and high cohesion

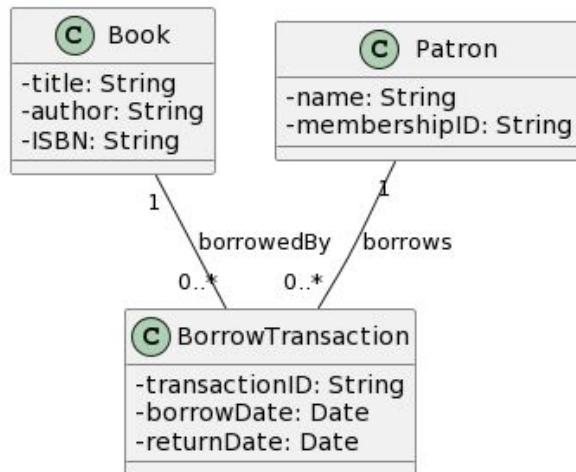
GoF Design patterns are pure fabrications

- Many of these patterns can be seen as "pure fabrications". That is, they don't always directly represent concepts from the problem domain but are introduced to solve specific software design challenges.

GRASP: Pure Fabrication

Imagine you're building a system for a library. In the real world, the library has books and patrons. Patrons borrow books. Now, in terms of software, we'd typically have a Book class and a Patron class.

We introduce a BorrowTransaction class which does not exist in the real world but helps track the relationship between books and patrons efficiently in our system.



BorrowTransaction is our **pure fabrication**. It doesn't have a direct counterpart in the real-world.

This design ensures that our system remains modular (**low coupling** between Book and Patron) and each class has a focused responsibility (**high cohesion**).

GRASP: Indirection

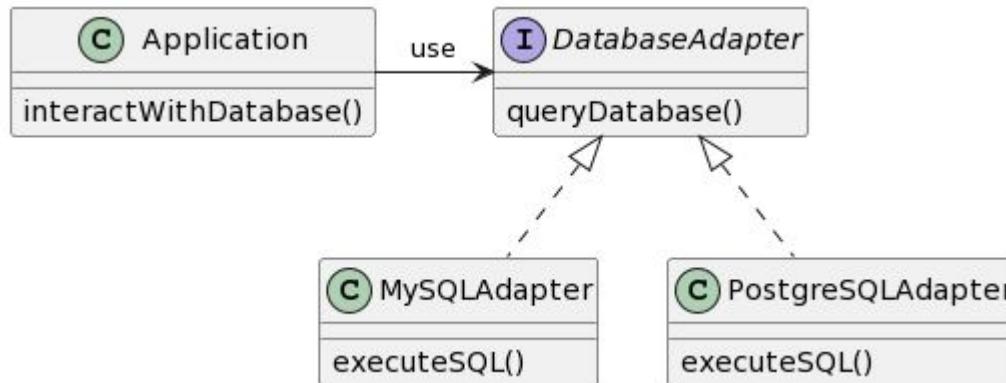
Introduction of an "intermediary object"

- To introduce lower coupling between objects.
- Ex. database adapter
 - Makes it potentially easier to switch databases
 - Creates possibly easier interaction
- Ex. dice cup
 - Both pure fabrication and indirection.

GRASP: Indirection

This diagram depicts the AppLogic class of the application using a Adapter class to interact with databases.

The Adapter abstracts the specifics of database interaction, allowing for easier switching between different databases like MySQL and PostgreSQL.



GRASP: Indirection

The Player class uses the DiceCup to roll the Dice. The DiceCup contains the Dice and indirectly influences the randomness and manner of the roll.

Encapsulation: The DiceCup encapsulates the action of rolling dice. This means that all the complexities, rules, and logic associated with rolling dice are encapsulated within the DiceCup.

Decoupling: By introducing the DiceCup, the game's main logic is decoupled from the specifics of how dice are rolled.

Flexibility: If in the future, there's a need to introduce new dice with different characteristics (e.g., 20-sided dice or dice with special symbols), the DiceCup can easily accommodate these changes.

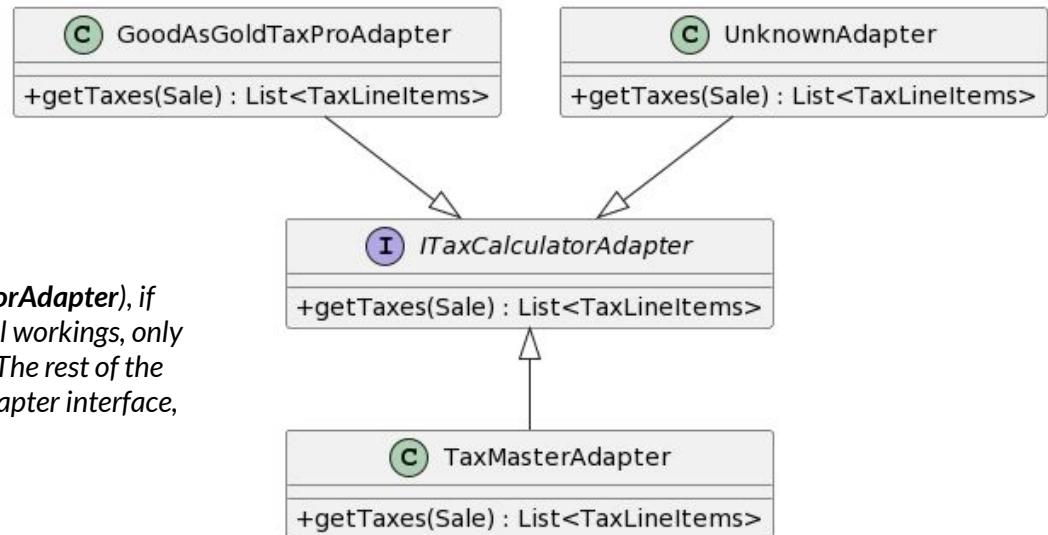
Consistency: If multiple games or parts of the game require dice-rolling, using the DiceCup ensures consistency in the dice-rolling process.



GRASP: Protected Variations

Protect objects from changes in other objects

- If unforeseen changes could happen
- Introduce stable interface
- Ex. Adapter
 - Tax adapter



*By having a common interface (**ITaxCalculatorAdapter**), if any of these tax systems change their internal workings, only the respective adapter needs to be modified. The rest of the system, which relies on the **ITaxCalculatorAdapter** interface, remains unaffected by such changes.*

Revisit Object Oriented Analysis and Design (OOAD)

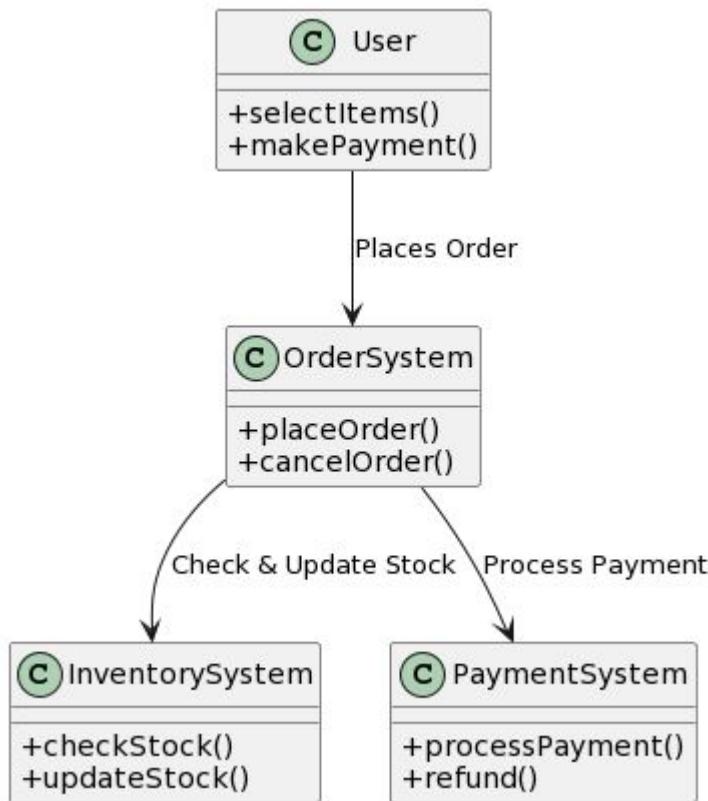
“The assignment of responsibilities and design of collaborations are very important and creative steps during design, both while diagramming and while coding”

- Larman s322

Key take aways:

- **Responsibility Assignment:** Properly assigning responsibilities ensures a cohesive and modular design, which makes maintenance and extension easier.
- **Collaboration Design:** This involves determining how different objects will interact with each other. Effective collaboration design can enhance system performance and reduce complexities.
- **Balance Between Diagramming and Coding:** While visual representations aid in conceptual understanding, it is equally vital to consider the practical aspects during the coding phase.
- **Importance of Creativity:** OOAD is not just a systematic process, but also an art. Approaching design problems creatively can lead to innovative solutions.

Revisit Object Oriented Analysis and Design (OOAD)



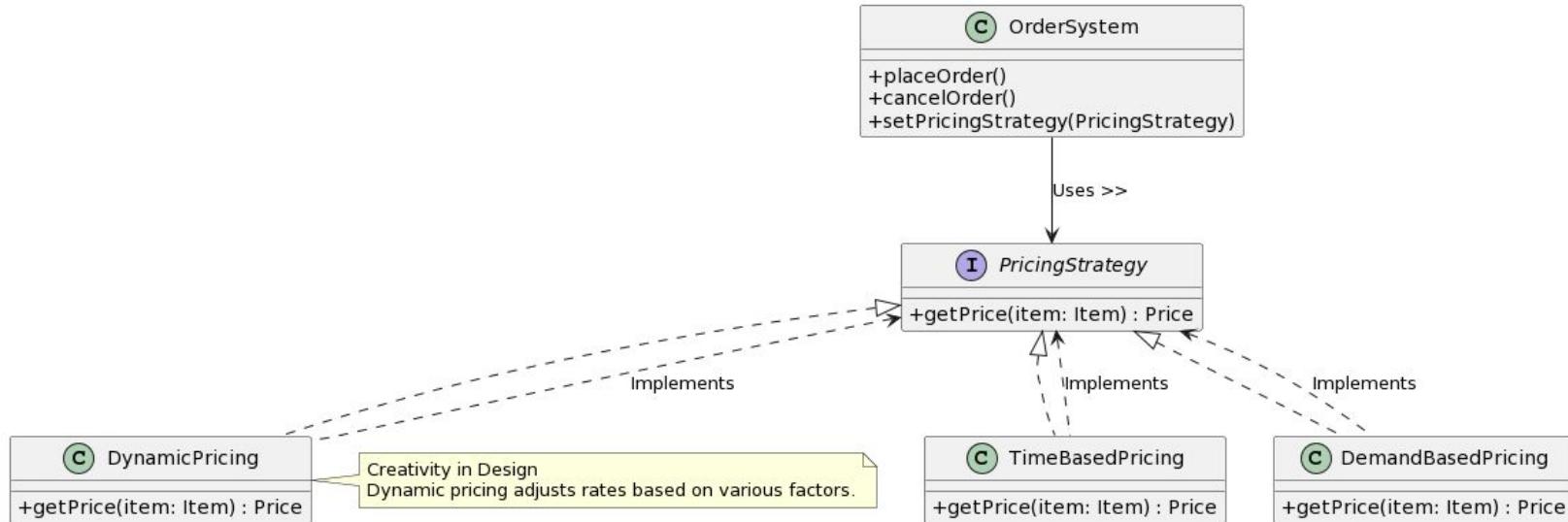
Consider a scenario where the OrderSystem wants to introduce a **feature that offers dynamic pricing** based on demand, time of day, or other factors.

Traditional OOAD might direct us towards a monolithic PricingModule inside the OrderSystem.

But, using a **creative design**, we can use a strategy pattern to allow for flexible, interchangeable pricing strategies.

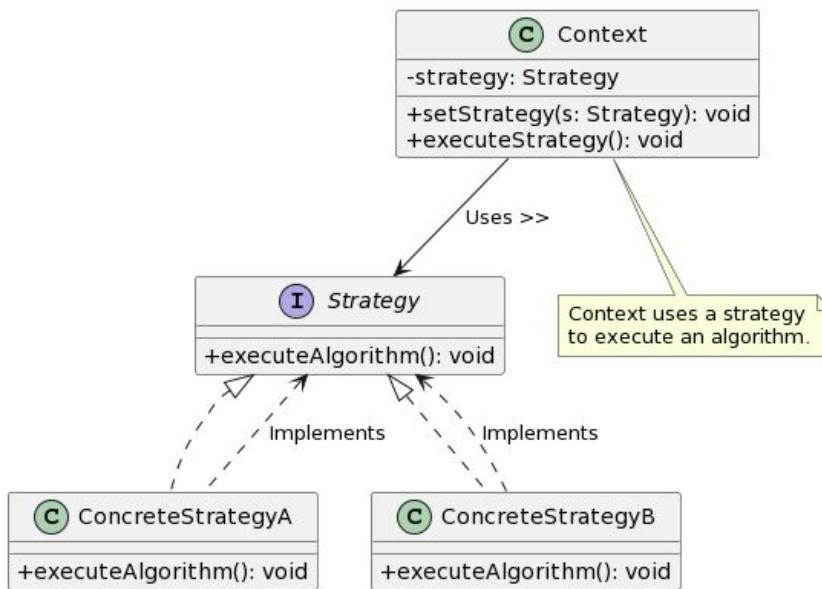
OOAD- Being Creative

- The OrderSystem doesn't directly decide the price. Instead, it uses a **strategy pattern** via the PricingStrategy interface.
- Different pricing strategies can be plugged in. This allows for **maximum flexibility and adaptability**.
- This design choice allows for quick adaptations to market changes without altering the core OrderSystem.



Design Patterns - “Strategy Pattern”

- Proven design solutions for common problems.
- Sufficiently abstract for general use.
- Strategy Pattern as an example:
 - The strategy pattern is a design pattern that allows you to **define a family of algorithms**, encapsulate each one, and make them interchangeable.



Much more about design patterns in:

- 02369 Software Processes and Patterns
- Now 'Sneak preview'

Design Patterns - “Strategy Pattern”

```
// Strategy Interface
public interface Strategy {
    void executeAlgorithm();
}

// Concrete Strategy A
public class ConcreteStrategyA implements Strategy {
    @Override
    public void executeAlgorithm() {
        System.out.println("Executing Algorithm A");
    }
}

// Concrete Strategy B
public class ConcreteStrategyB implements Strategy {
    @Override
    public void executeAlgorithm() {
        System.out.println("Executing Algorithm B");
    }
}
```

```
// Context class that uses the strategy
public class Context {
    private Strategy strategy;

    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy() {
        if(strategy != null) {
            strategy.executeAlgorithm();
        } else {
            System.out.println("Strategy not set!");
        }
    }
}
```

Design Patterns - “Strategy Pattern”

```
// Demo
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context();

        // Set strategy to ConcreteStrategyA and execute
        context.setStrategy(new ConcreteStrategyA());
        context.executeStrategy();

        // Change strategy to ConcreteStrategyB and execute
        context.setStrategy(new ConcreteStrategyB());
        context.executeStrategy();
    }
}
```

Design Patterns - “Factory Pattern”

The Factory Pattern is a creational design pattern that provides an **interface for creating objects**, but **allows subclasses to alter the type of objects** that will be created. It's about *using inheritance and subclasses to produce objects that fit a general contract*.



A normal factory produces goods

A software factory produces objects.

And not just that – it does so without specifying the exact class of the object to be created.

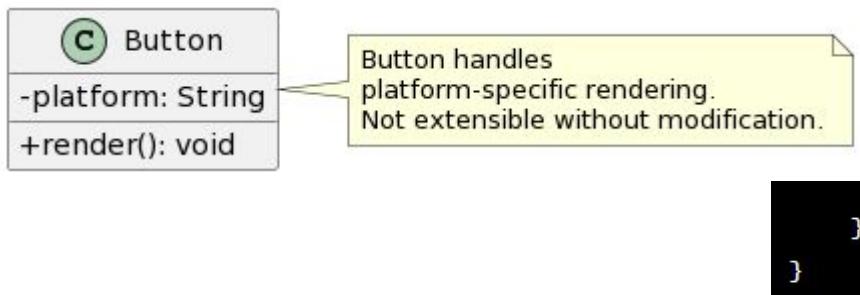
To accomplish this, objects are created by calling a factory method instead of calling a constructor.

Design Patterns - “Factory Pattern”

Example: UI Elements Factory

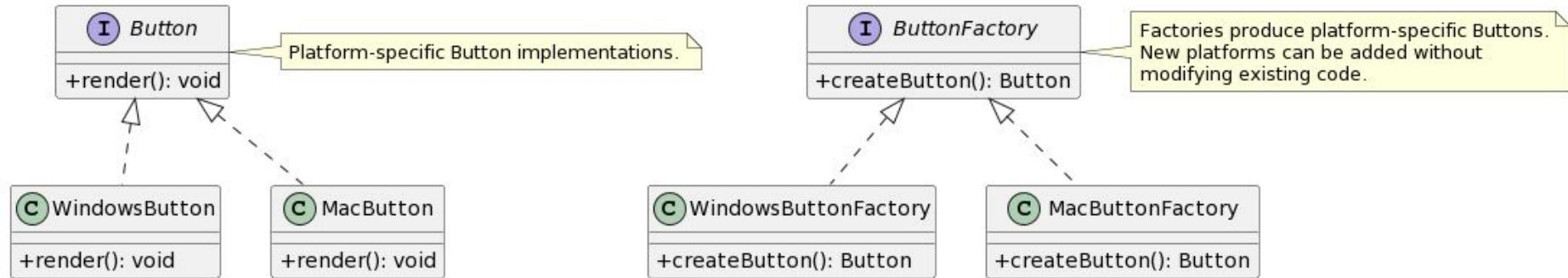
Imagine you're developing a cross-platform UI toolkit, and you need a way to create platform-specific UI elements like buttons.

Without the Factory Pattern:



```
public class Button {  
    private String platform;  
  
    public Button(String platform) {  
        this.platform = platform;  
    }  
  
    public void render() {  
        if("Windows".equals(platform)) {  
            System.out.println("Rendering a Windows-style button");  
        } else if("Mac".equals(platform)) {  
            System.out.println("Rendering a Mac-style button");  
        } else {  
            System.out.println("Rendering a default button");  
        }  
    }  
}
```

Design Patterns - “Factory Pattern”



ButtonFactory provides an interface to create platform-specific buttons, and each platform implements its own factory.

Design Patterns - “Factory Pattern”

```
// Button interface
public interface Button {
    void render();
}

// Concrete classes for each platform
public class WindowsButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering a Windows-style button");
    }
}

public class MacButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering a Mac-style button");
    }
}
```

```
// Factory method interface
public interface ButtonFactory {
    Button createButton();
}

// Concrete factories for each platform
public class WindowsButtonFactory implements ButtonFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}

public class MacButtonFactory implements ButtonFactory {
    @Override
    public Button createButton() {
        return new MacButton();
    }
}
```

Design Patterns - “Factory Pattern”

```
Button button = buttonFactory.createButton();
button.setup(); // Setup the button resources (platform-specific)
button.render(); // Render the button (platform-specific)
```

Highlight on the Mandatory of Using Factory Method:

Decoupling: The Factory Method pattern decouples the client code (which needs the object) from the actual type of object that's being created. This separation allows for more modular and scalable code.

Flexibility: When adding support for a new platform, you simply create a new factory without modifying existing factories or UI elements.

Consistency: The pattern enforces a consistent way of creating objects, making the codebase easier to understand.

Centralization of Creation Logic: If there's a change in the object creation process or initialization, it only needs to be made in one place (the factory), rather than everywhere an object is created.

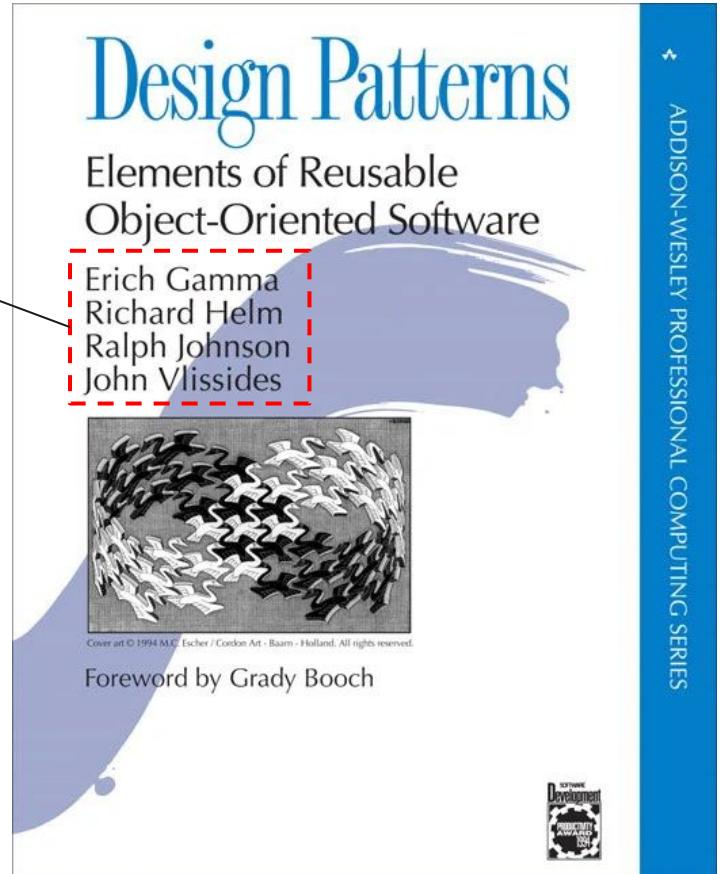
Gang of Four patterns

23 patterns in 3 categories

- Creational
- Structural
- Behavioral

More on that later (02369 Software Processes and Patterns)

Summary: https://en.wikipedia.org/wiki/Design_Patterns



GoF Creational Patterns

Purpose: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Why: Direct instantiation of objects exposes the details of the object's representation and implementation. By using these patterns, a system can become independent of how its objects are created, composed, and represented.

- Abstract factory
 - groups object factories that have a common theme.
- Builder
 - constructs complex objects by separating construction and representation.
- ***Factory method***
 - creates objects without specifying the exact class to create.
- Prototype
 - creates objects by cloning an existing object.
- Singleton
 - restricts object creation for a class to only one instance.

GoF Structural Patterns

Purpose: These patterns are concerned with how classes and objects are composed to form larger structures.

Why: These patterns simplify structure by identifying a simple way to realize relationships between entities.

- Adapter
 - allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- Bridge
 - decouples an abstraction from its implementation so that the two can vary independently.
- Composite
 - composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator (to be introduced)**
 - dynamically adds/overrides behaviour in an existing method of an object.
- Facade
 - provides a simplified interface to a large body of code.
- Flyweight
 - reduces the cost of creating and manipulating a large number of similar objects.
- Proxy
 - provides a placeholder for another object to control access, reduce cost, and reduce complexity.

GoF Behavioural Patterns

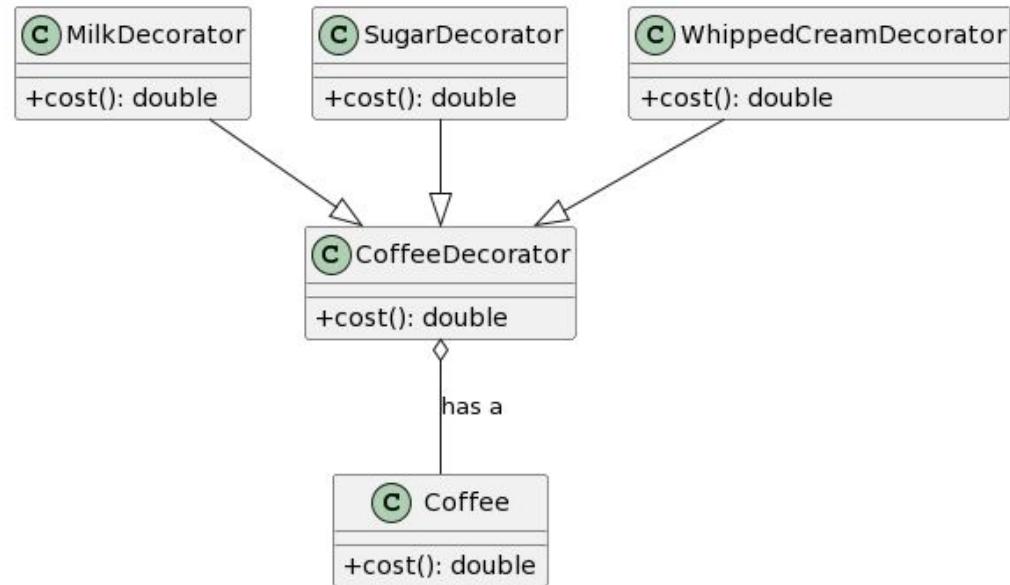
Purpose: These patterns characterize the ways in which objects or classes interact and distribute responsibilities.

Why: These patterns increase the system's flexibility in terms of behaviors and the processes they represent. They help ensure that entities in a system behave together harmoniously.

- Chain of responsibility
 - delegates commands to a chain of processing objects.
- Command
 - creates objects that encapsulate actions and parameters.
- Interpreter
 - implements a specialized language.
- Iterator
 - accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator
 - allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento
 - provides the ability to restore an object to its previous state (undo).
- ***Observer (to be introduced)***
 - is a publish/subscribe pattern, which allows a number of observer objects to see an event.
- State
 - allows an object to alter its behavior when its internal state changes.
- Strategy
 - allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method
 - defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- Visitor
 - separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Ex. Decorator Pattern

Imagine a simple scenario where we have a Coffee class, and we want to be able to dynamically add ingredients (like milk, sugar, whipped cream) to it.



Ex. Decorator Pattern

```
// Component
public interface Coffee {
    double cost();
}

// Concrete Component
public class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 2.00; // Base price of plain coffee
    }
}

// Abstract Decorator
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public double cost() {
        return coffee.cost();
    }
}
```

```
// Concrete Decorators
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return super.cost() + 0.50; // Price of milk
    }
}

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return super.cost() + 0.25; // Price of sugar
    }
}

public class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return super.cost() + 0.75; // Price of whipped cream
    }
}
```

Ex. Decorator Pattern

```
Coffee myCoffee = new SimpleCoffee();
myCoffee = new MilkDecorator(myCoffee);
myCoffee = new SugarDecorator(myCoffee);

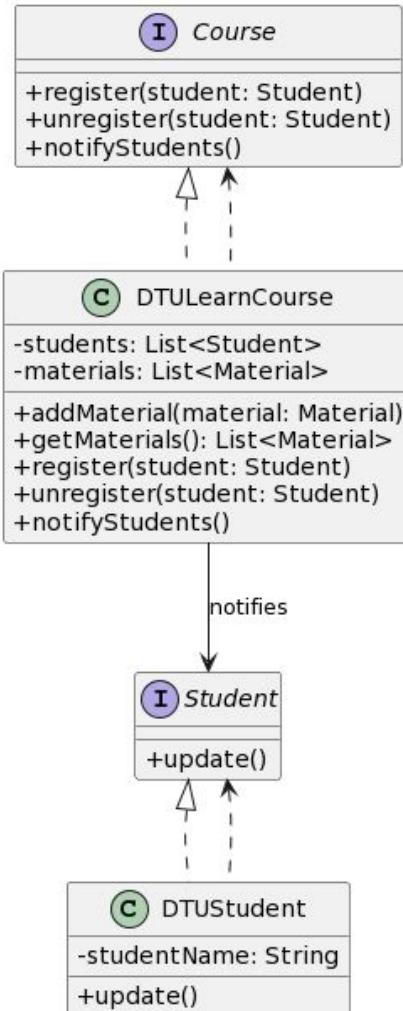
// This will output 2.75
// (2.00 for coffee + 0.50 for milk + 0.25 for sugar)
System.out.println(myCoffee.cost());
```



Eks: Observer pattern

- The slide introduces the "Observer pattern," a design pattern commonly used in programming, especially for systems that require one-to-many communication between objects.
- A commonly used pattern
 - For 'normal' graphical user interfaces

Whenever there's a new material added to the course by “addMaterial()”, all registered students should be notified.



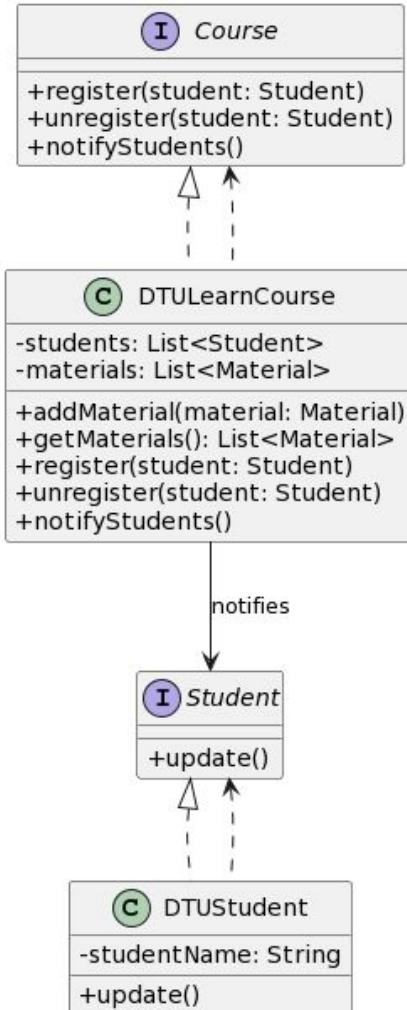
Eks: Observer pattern

Course: This is an interface representing any course on the DTU Learning platform. It defines methods to register, unregister, and notify students.

Student: This is an interface representing a student who can be notified of updates in a course.

DTULearnCourse: This is a concrete course in the DTU Learning platform. Whenever new material is added, it notifies all its registered students.

DTUStudent: Represents a student in the DTU system. They implement the update() method to take necessary actions (like fetching the new material) when notified.



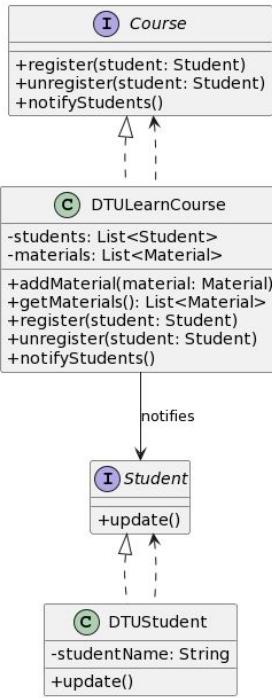
Eks: Observer pattern

```
// Observer Interface
interface Student {
    void update();
}

// ConcreteObserver
class DTUStudent implements Student {
    private String studentName;

    public DTUStudent(String name) {
        this.studentName = name;
    }

    @Override
    public void update() {
        System.out.println(studentName +
            " has been notified of new material!");
    }
}
```



```
// Subject Interface
interface Course {
    void register(Student student);
    void unregister(Student student);
    void notifyStudents();
}

// ConcreteSubject
class DTULearnCourse implements Course {
    private List<Student> students = new ArrayList<>();
    private List<String> materials = new ArrayList<>();

    // ... omitted code

    @Override
    public void notifyStudents() {
        for (Student student : students) {
            student.update();
        }
    }

    public void addMaterial(String material) {
        materials.add(material);
        notifyStudents();
    }
}
```

Eks: Observer pattern

```
public class observerPatternDemo {  
    public static void main(String[] args) {  
        DTULearnCourse course = new DTULearnCourse();  
  
        DTUStudent alice = new DTUStudent("Alice");  
        DTUStudent bob = new DTUStudent("Bob");  
  
        course.register(alice);  
        course.register(bob);  
  
        course.addMaterial("New Lecture Slides");  
    }  
}
```

Design patterns

- Good to know
- Criticism: Solutions to problems that OO Programming introduces
- Some patterns are unnecessary in other languages
 - Functional languages
 - Languages where types and functions are also variables

Use Case Realization

“A use case realization describes how a particular use case is realised within the Design Model, in terms of collaborating objects”

- Larman, p322

Use Case → Design model

Development methods 62531

Lecture 9 - Design to code. Inheritance and Polymorphism
Lei You, Assistant Professor

leiyo@dtu.dk

What have we learned?

- GRASP
 - General Responsibility Assignment Software Patterns
 - Classes have responsibilities
- Design Pattern
 - Design solution to common problem
 - Generalizable
- GoF
 - Gang of Four
 - Another set of software patterns

GRASP - 9 principles in total

- *Creator*
- *Information Expert*
- **Low Coupling**
- **High Cohesion**
- Controller
- (Polymorphism)
- (Pure Fabrication)
- (Indirection)
- (Protected Variations)

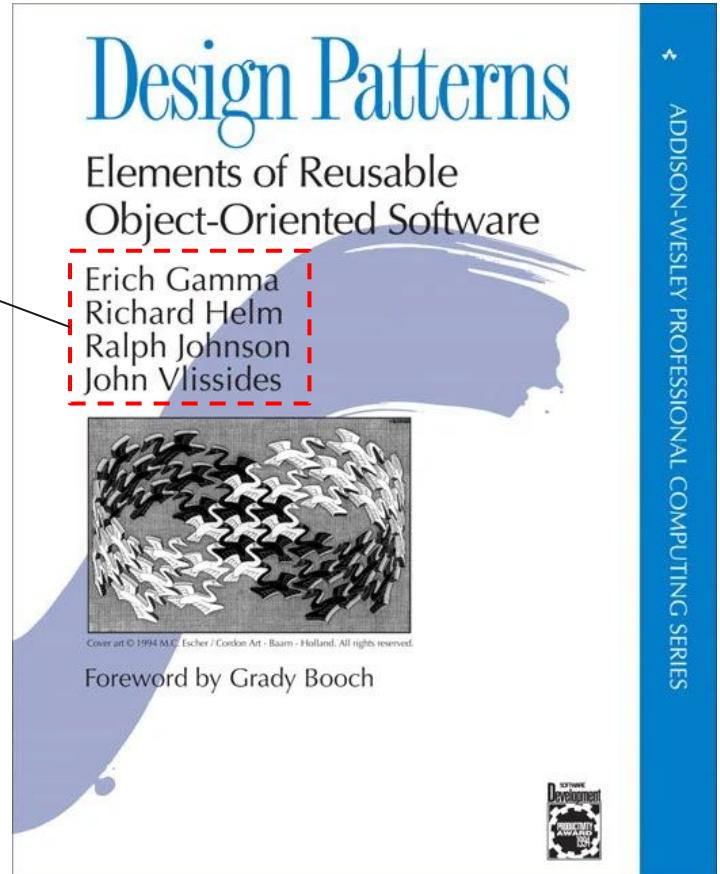
Gang of Four patterns

23 patterns in 3 categories

- Creational
- Structural
- Behavioral

More on that later (02369 Software Processes and Patterns)

Summary: https://en.wikipedia.org/wiki/Design_Patterns



Use Case Realization

“A use case realization describes how a particular use case is realised within the Design Model, in terms of collaborating objects”

- Larman, p322

Use Case → Design model

Today's program

- Object knowledge of each other
- Design to code
- Inheritance and polymorphism

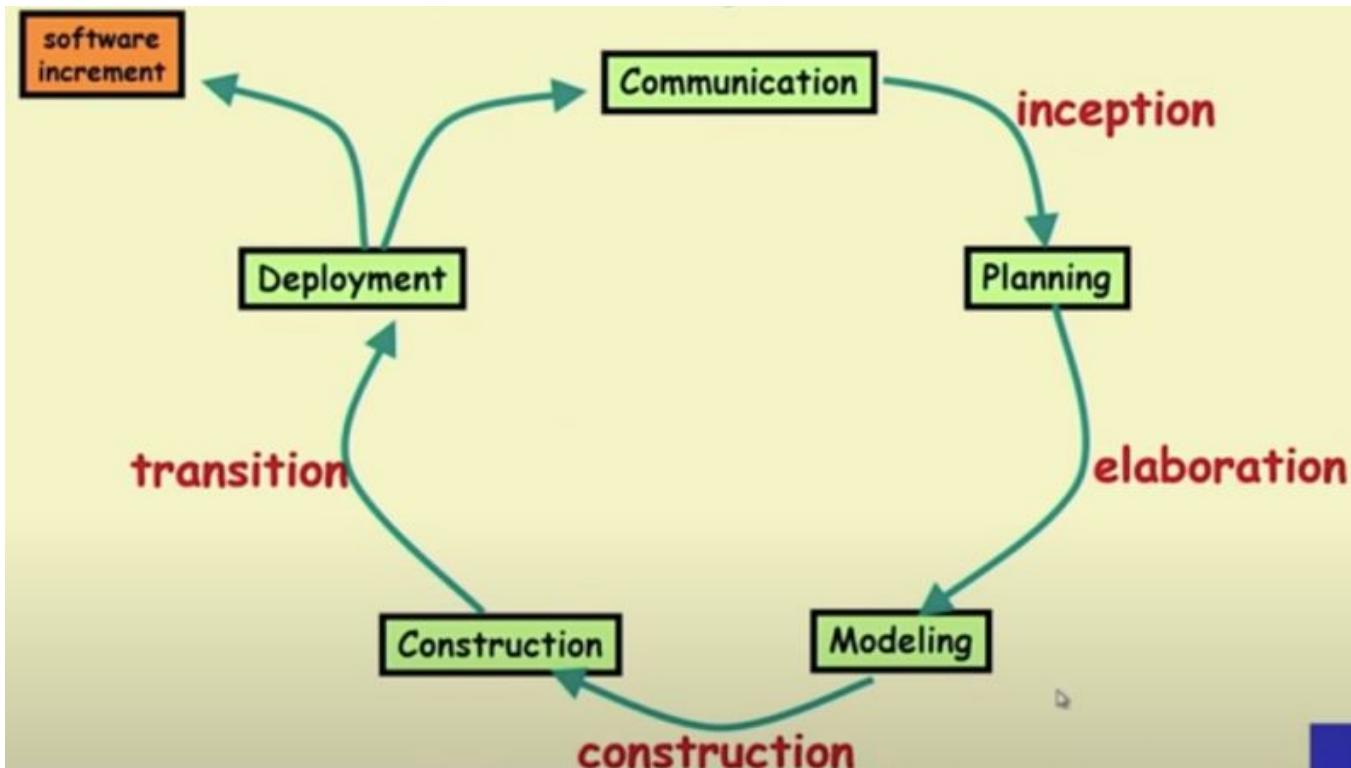
Revisit the Unified Process (UP)

Goal: To deliver high-quality software that meets user needs while managing risks and changes effectively.

Key Characteristics:

1. **Iterative and Incremental:** UP divides the development timeline into multiple "iterations". Each iteration results in an increment, a partial version of the product.
2. **Use-case Driven:** System functionality is primarily described using use-cases. They guide the design, implementation, and testing stages.
3. **Architecture-centric:** UP emphasizes a robust system architecture. This architecture serves as a foundation upon which the rest of the system is built.
4. **Risk-focused:** At each iteration, high-risk elements are identified and tackled first.

Revisit the Unified Process (UP)



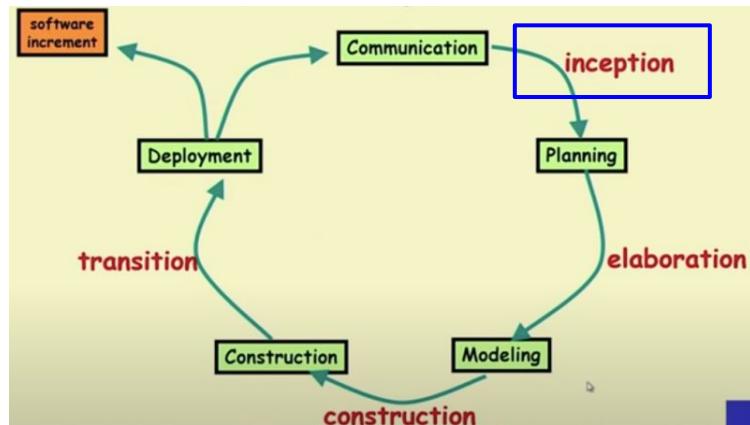
UP Ex. - Online Bookstore Application

Inception Phase

Objective: Define the project's scope and determine its feasibility.

Activities & Artifacts:

- Gather initial **requirements** from stakeholders.
- Develop a vision document outlining the **main objectives**: "Create an online platform where users can browse, search for, and purchase books."
- Identify **potential risks**: For example, "Integration with payment gateways" or "Scalability to handle thousands of users."
- Rough cost and time **estimates**.
- **Decision** on whether to proceed with the project.



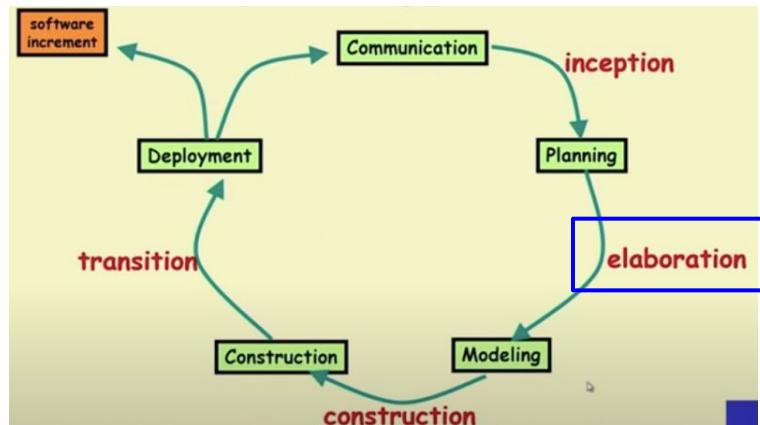
UP Ex. - Online Bookstore Application

Elaboration Phase

Objective: Establish a solid understanding of the problem domain and lay down the architectural foundation.

Activities & Artifacts:

- **Use-case Modeling:** Identify primary actors (Users, Admins, Suppliers) and their interactions with the system.
 - Example Use-Case: "User searches for a book", "User makes a purchase", "Admin adds a new book".
- **Architectural Design:** Decide on key components of the system, like the database, user interface, and integration modules.
- Address critical **risks:** Test the integration with a mock payment gateway to ensure seamless transactions.
- Revise **estimates** based on a deeper understanding of requirements.



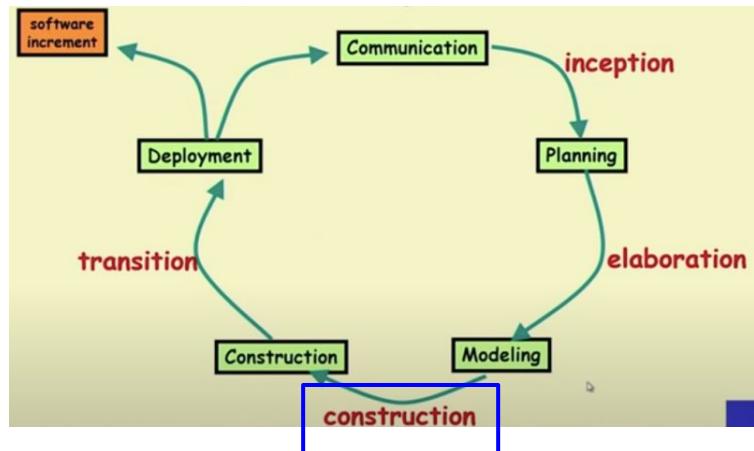
UP Ex. - Online Bookstore Application

Construction Phase

Objective: Incremental development of the system.

Activities & Artifacts:

- Development is divided into **iterations**, each focusing on a set of use-cases or features.
- Iteration 1: Implement "User registration and login" and "Basic book search".
 - At the end of this iteration, a working prototype allows users to register, log in, and search for books.
- Iteration 2: Add "Advanced search filters", "User reviews", and "Purchase books".
- Automated **testing** ensures that new features don't break existing functionality.
- Periodic reviews with stakeholders gather feedback and ensure alignment with requirements.



UP Ex. - Online Bookstore Application

Transition Phase

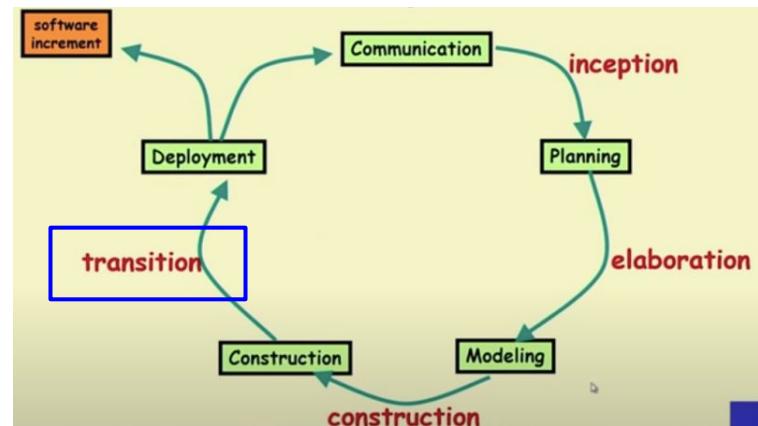
Objective: Get the system ready for release and deployment.

Activities & Artifacts:

- **Beta Testing:** A select group of users is given access to test the platform.
- Bug Fixing: Address any issues or glitches reported.
- **Deployment:** Set up servers, databases, and deploy the application for general access.
- Training sessions for admins to manage the platform.
- User **documentation** is made available.

Beta testing involves external users, in contrast to the “in-house” alpha one that only involves the dev team

User feedback is gathered



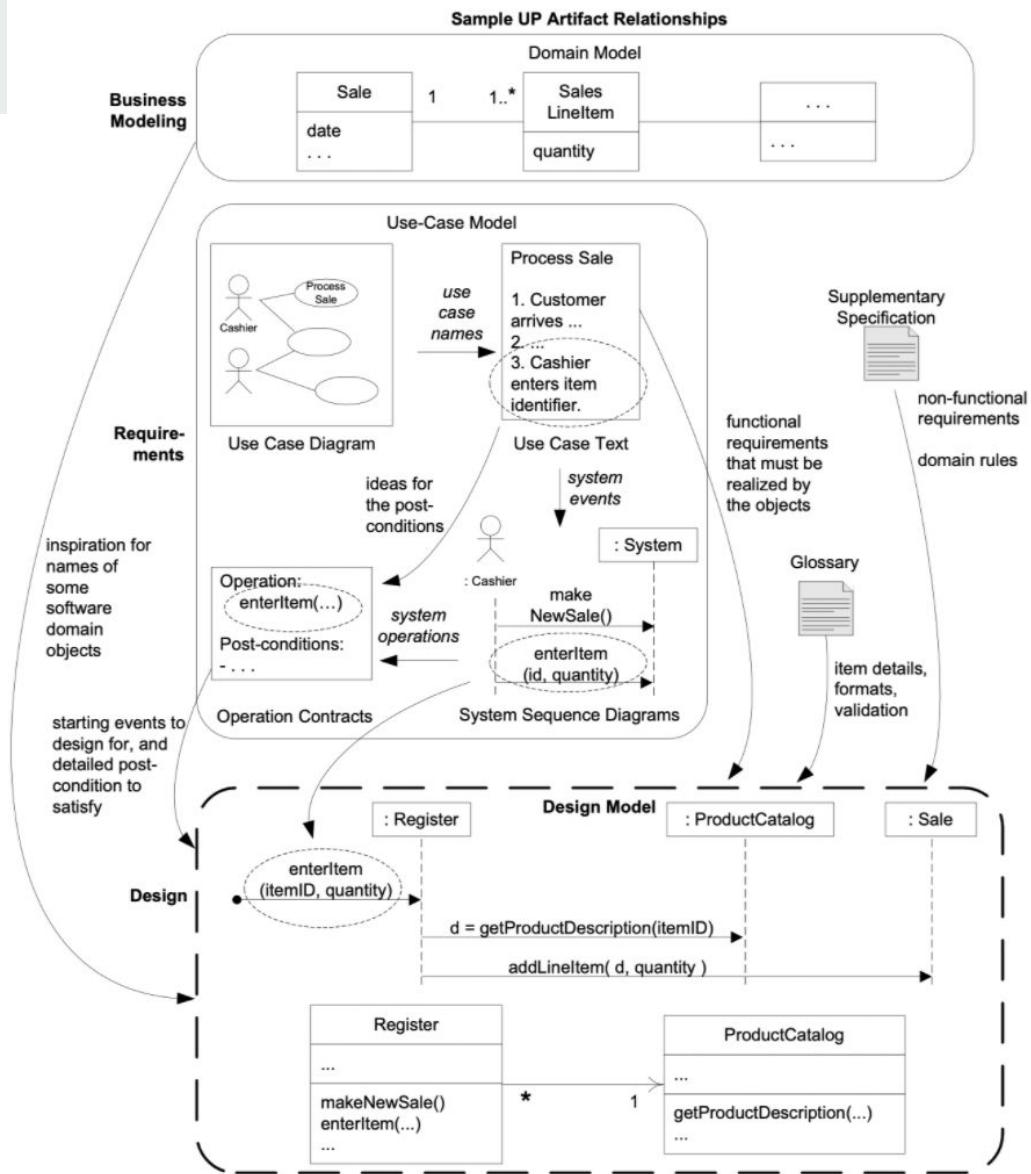
Important Artifacts

Functional requirements - like “specifications”

- A user should be able to register an account.
- The system should be able to generate monthly sales reports.
- Users must be able to reset their passwords.

Non-functional requirements - like “criteria”

- The system should have 99.9% uptime.
- Response time for user queries should not exceed 2 seconds.
- Data backups should occur daily.
- The system should be able to handle 10,000 concurrent users.



Object knowledge/reference

In order for object A to call a method from object B,

object B must be "visible" to object A.

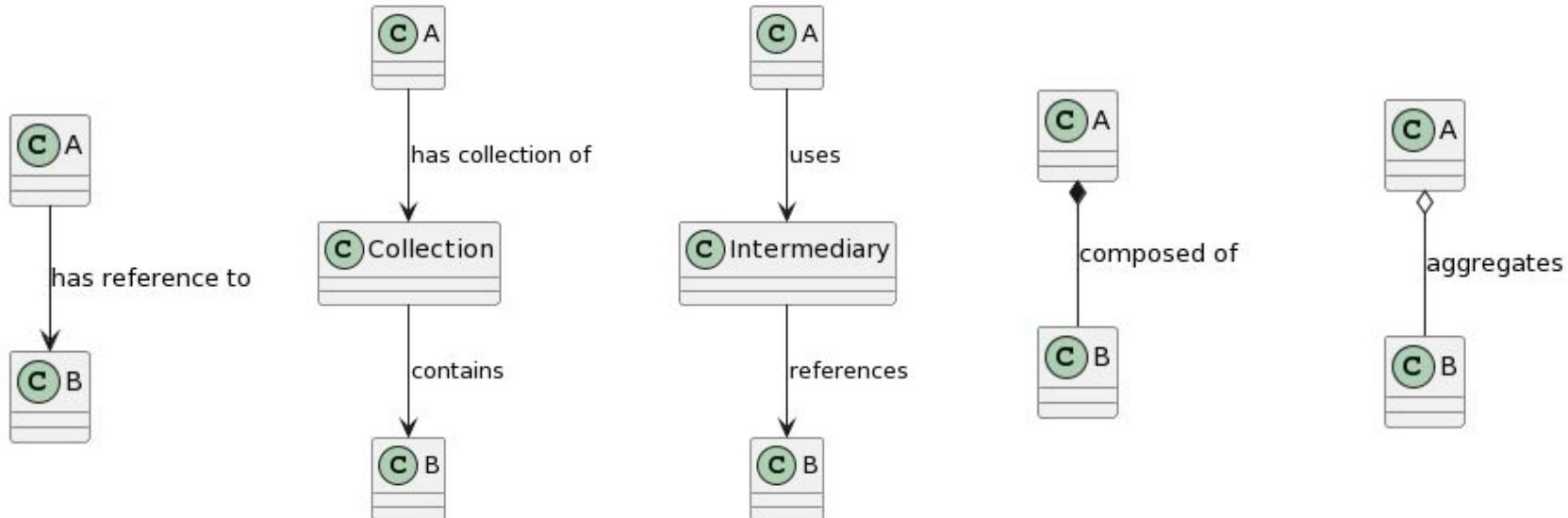
(NOTE. the object is called “a” and the class is called A - ':' is read in UML as 'of the type')



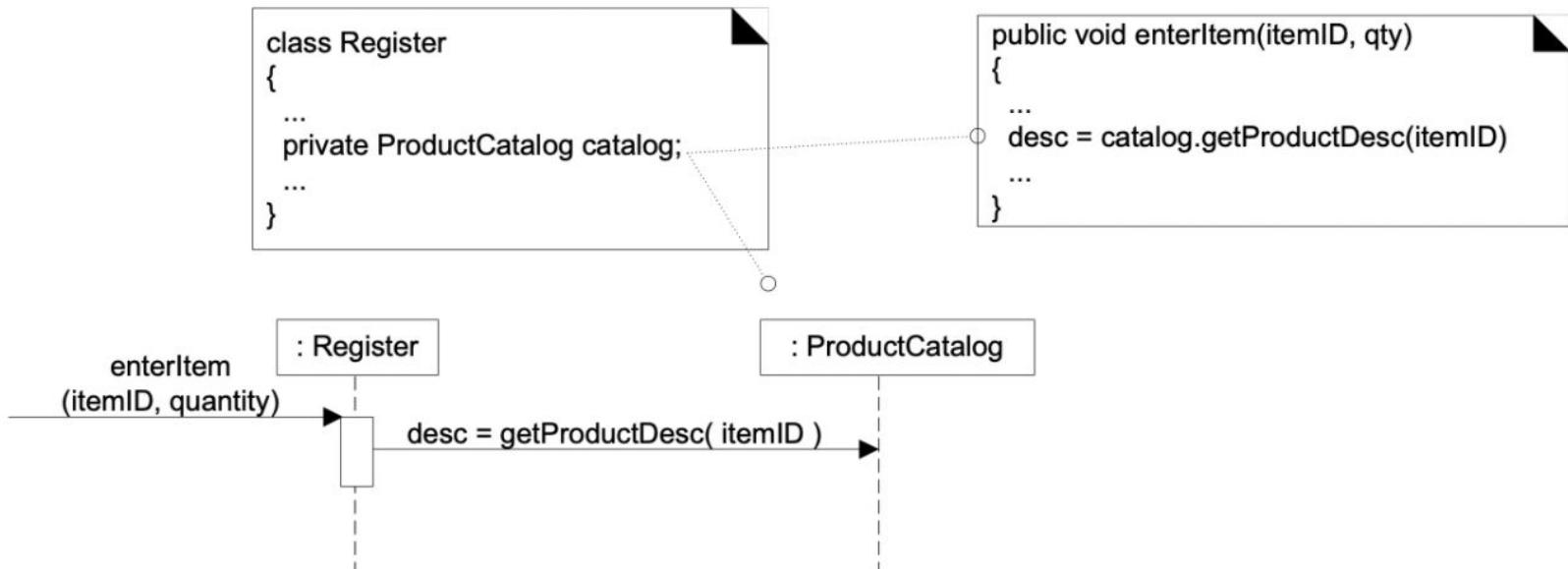
Object knowledge/reference

A must have a reference to B

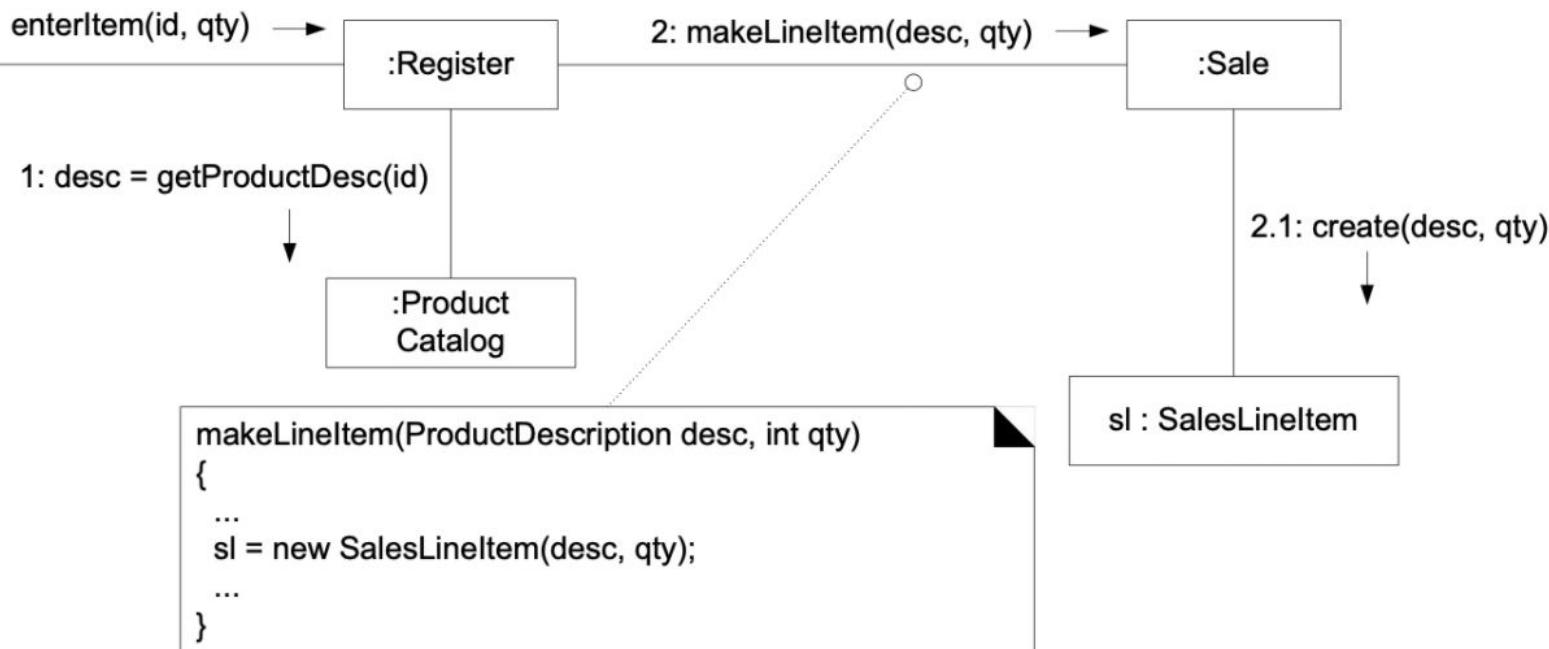
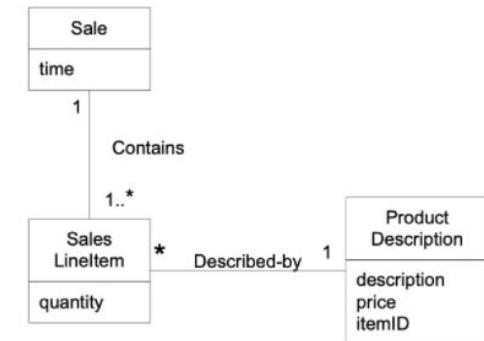
- Different ways...



Attribute visibility

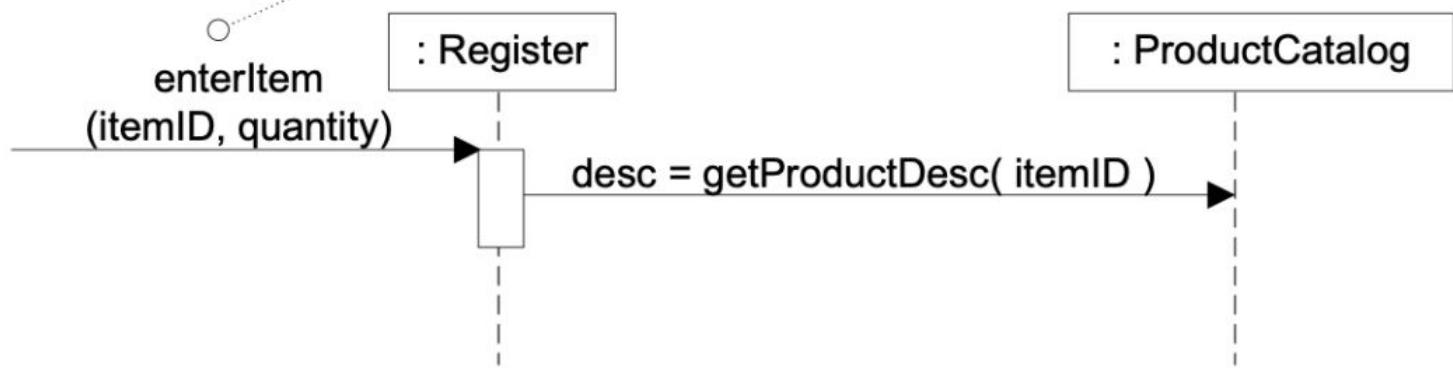


Parameter visibility



Local visibility

```
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductDescription desc = catalog.getProductDes(id);
...
}
```



Global visibility

- Static variables belong to a class (rather than a specific instance)
- Singleton pattern
 - A design pattern ensuring that a class has only one instance throughout the execution of a program and provides a way to access its single instance.

Singleton

- singleton : Singleton
- Singleton()
- + getInstance() : Singleton

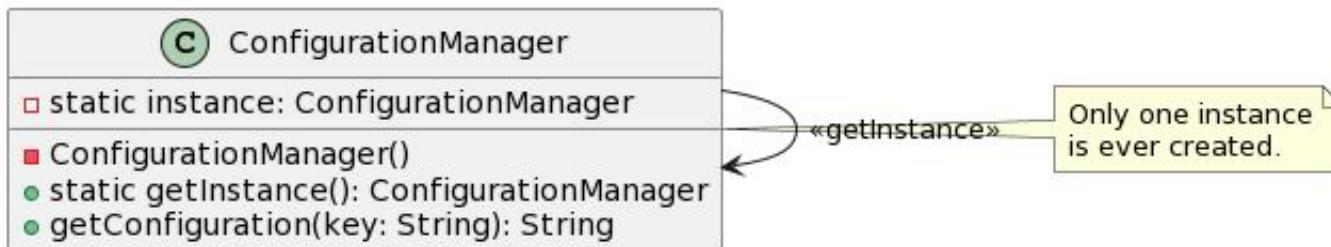
GoF - Singleton Pattern Ex 1.

Consider the example of a **configuration manager** in a software application. Often, applications have a set of configurations (e.g., server settings, UI themes) that need to be accessed globally and remain consistent across the app.

If you don't use the Singleton pattern, you might instantiate the configuration manager multiple times. This could lead to:

- **Inconsistencies:** Different instances might have different configurations, leading to unpredictable behavior.
- **Resource waste:** Multiple instances consume more memory.

GoF - Singleton Pattern Ex 1.



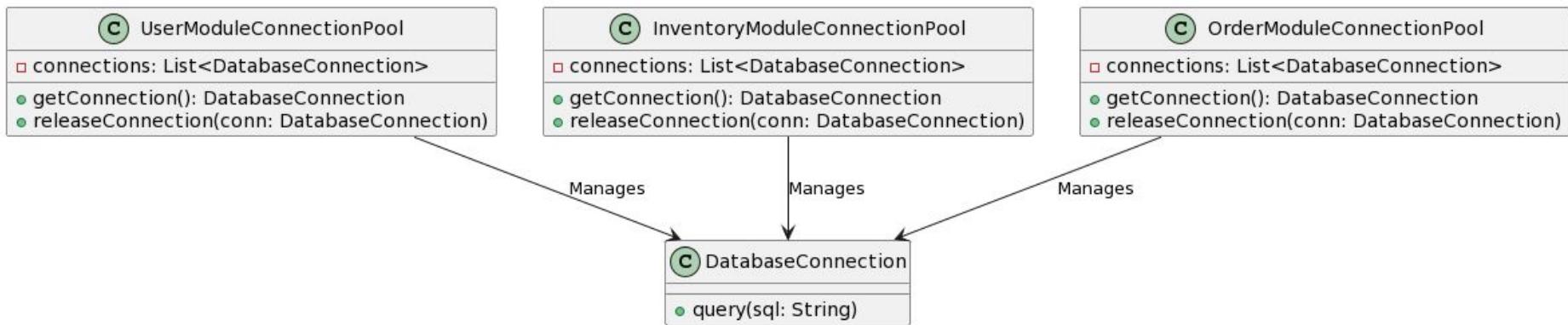
Whenever you need to access configurations, you'd call

`ConfigurationManager.getInstance().getConfiguration(key).`

This ensures consistency and avoids unnecessary resource usage.

GoF - Singleton Pattern Ex 2.

Modern applications often connect to databases to perform various operations. Creating and tearing down these connections frequently can be resource-intensive and slow.



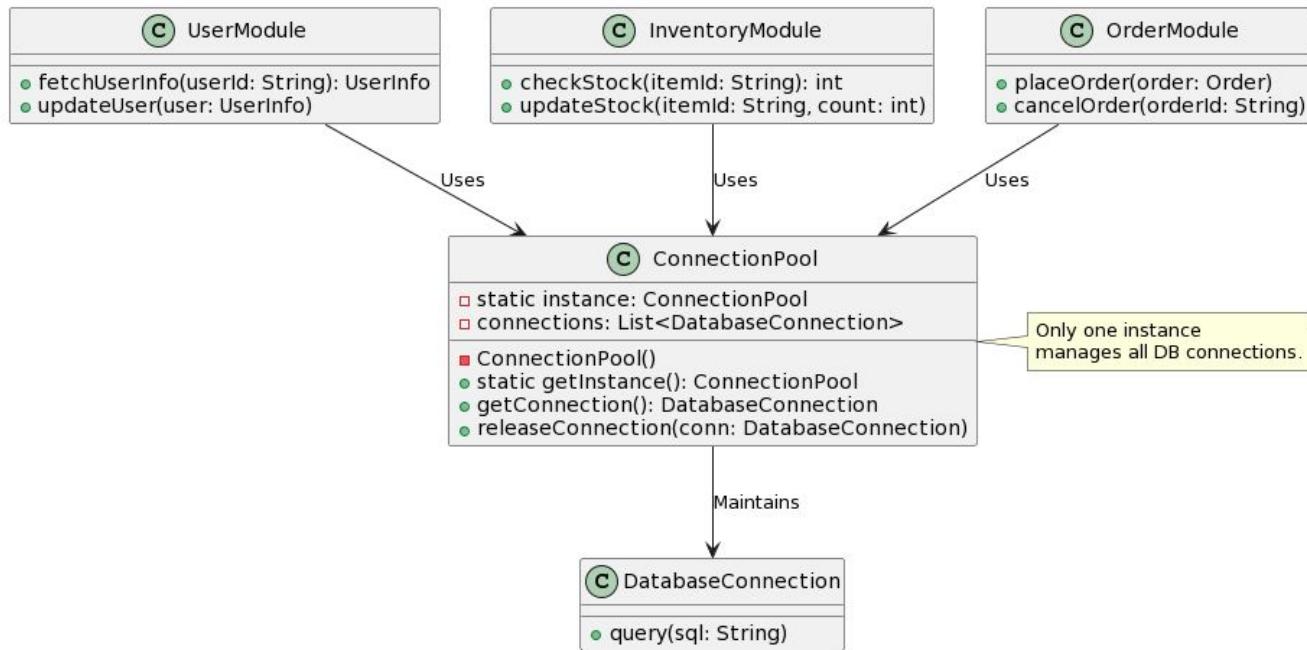
Resource Overload: Multiple pools multiply the number of active connections.

Conflicting Configurations: Each pool might be configured differently leading to inconsistent data operations.

Wasted Resources: Idle connections in one pool can't be used by another pool, leading to wastage.

GoF - Singleton Pattern Ex 2.

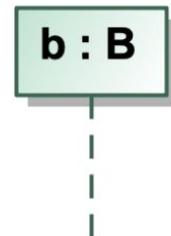
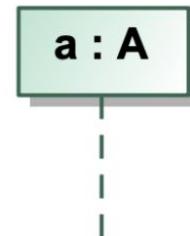
To mitigate this, applications use a **connection pool** where a set of database connections are maintained, allowing the app to reuse them.



Object knowledge/reference

- Attributes
 - B is an attribute of A
- Parameters
 - B is a parameter to a Method in A
- Local variables
 - B is a local variable in a method in A
- Global variable
 - B is a global variable

How to code it in java?



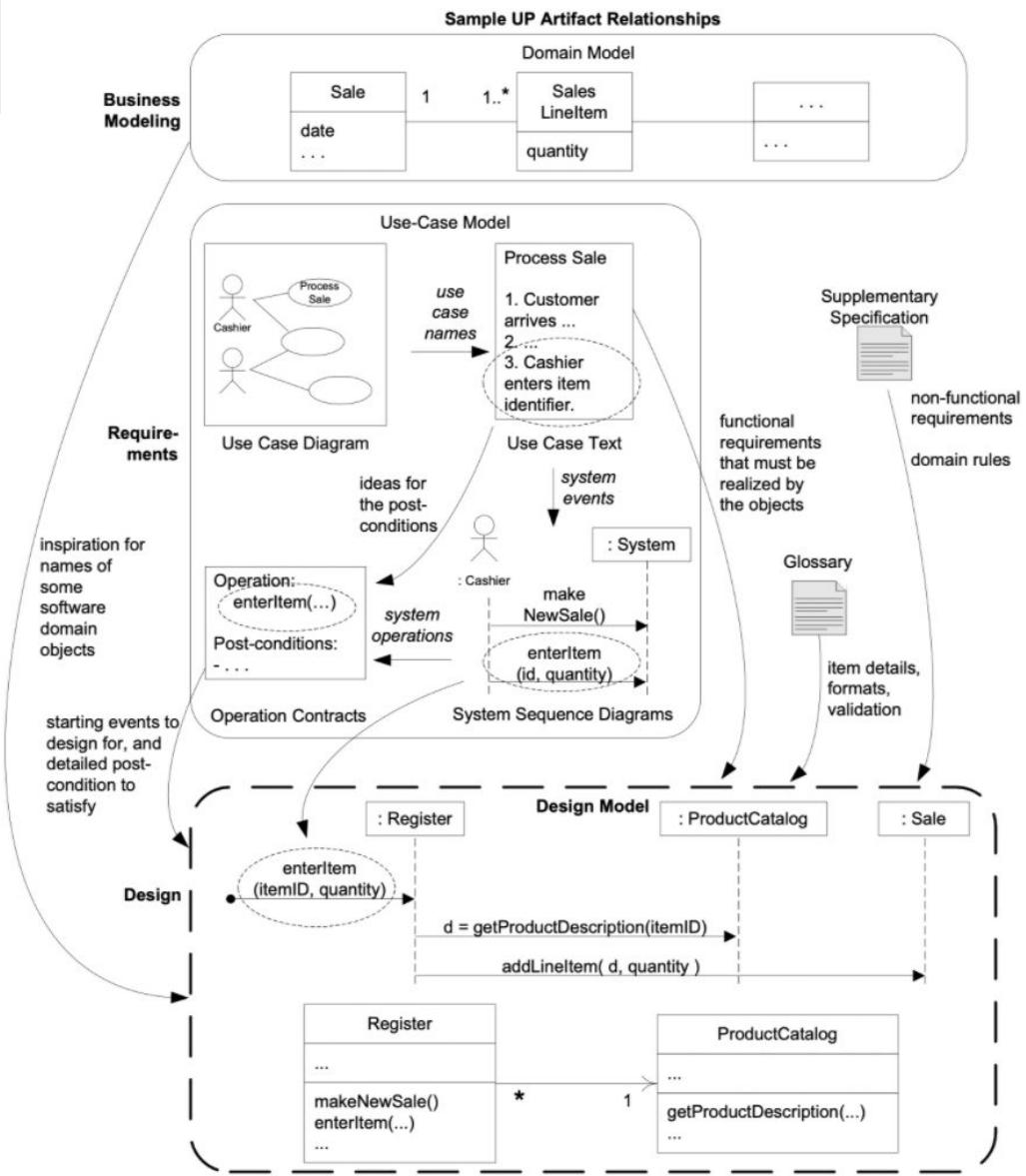
Object knowledge/reference

- Attributes
 - B is an attribute of A
- Parameters
 - B is a parameter to a Method in A
- Local variables
 - B is a local variable in a method in A
- Global variable
 - B is a global variable

How to code it in java?

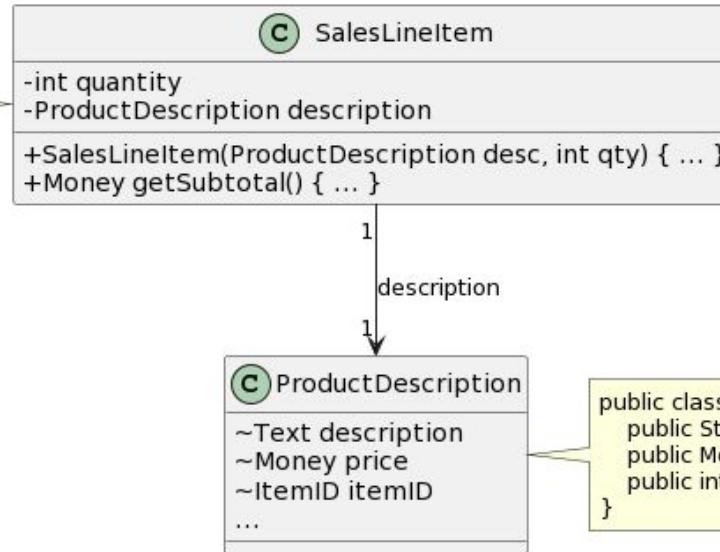
```
public class A {  
    private B bAttribute; //Attribut  
  
    public void doSomething() {  
        bAttribute.doBStuff();  
    }  
  
    public void doSomething(B bParameter) {  
        bParameter.doBStuff();  
    }  
  
    public void doSomethingElse(){  
        B b = new B();  
        b.doBStuff();  
    }  
  
    public void doGlobalBstuff(){  
        B.getInstance().doBStuff();  
    }  
}
```

Design to code



Design for Code - Classes and Attributes

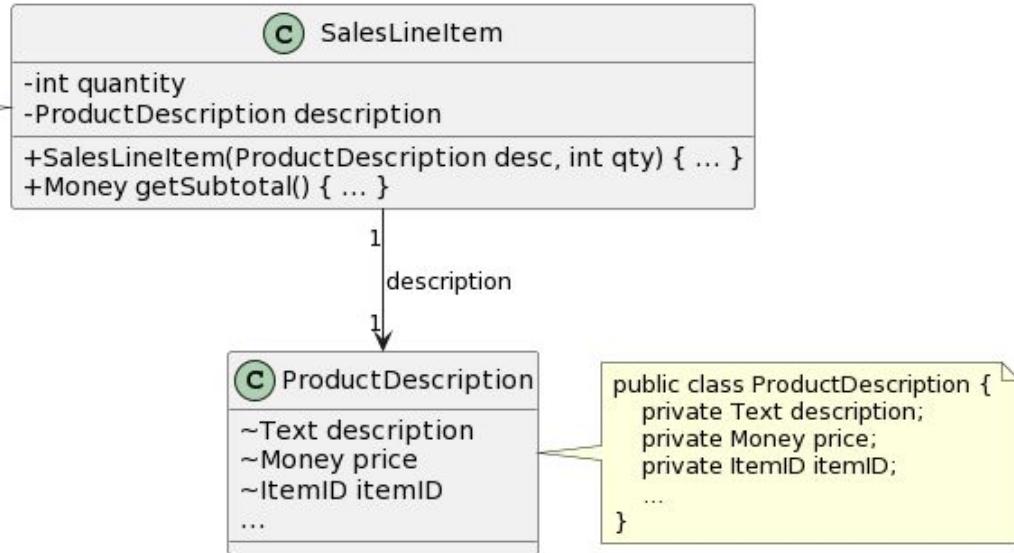
```
public class SalesLineItem {  
    public int quantity;  
    public ProductDescription description;  
  
    public Money getSubtotal() {  
        return description.price.multiply(quantity);  
    }  
}
```



Is it a correct design? Why or why not?

Design for Code - Classes and Attributes

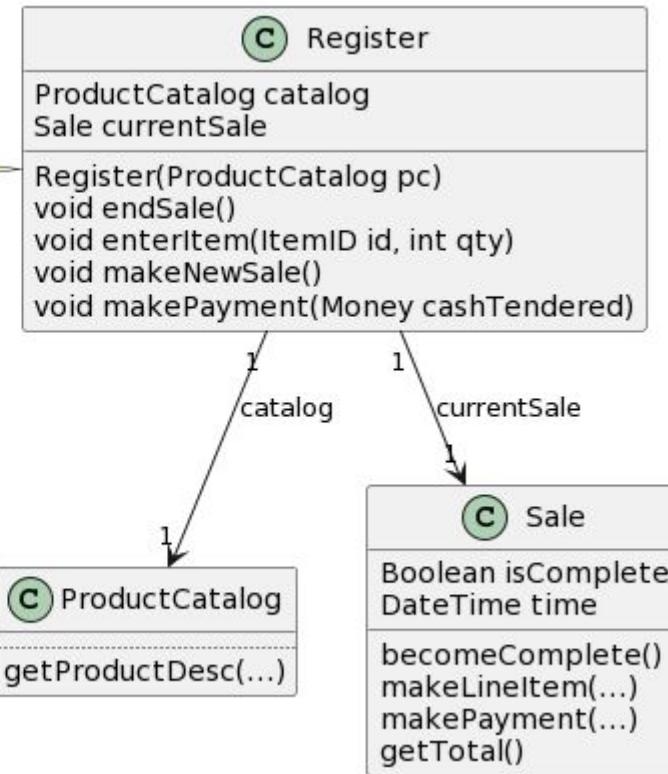
```
public class SalesLineItem {  
    private int quantity;  
    private ProductDescription description;  
  
    public SalesLineItem(ProductDescription desc, int qty) { ... }  
    public Money getSubtotal() { ... }  
}
```



- Consistency with the diagram
- Initialization and encapsulation
- Comments for clarity

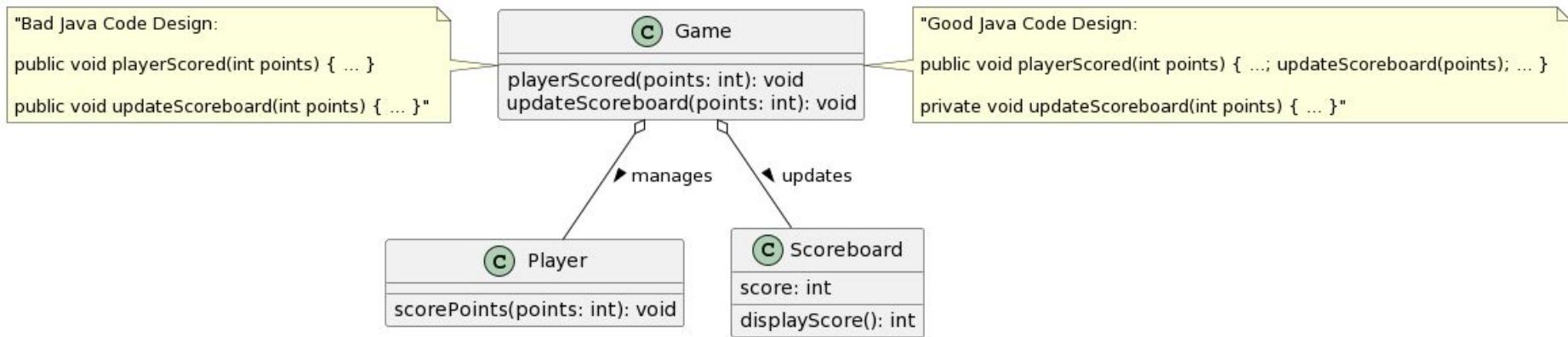
Design for Code - Classes and Attributes

```
public class Register {  
    private ProductCatalog catalog;  
    private Sale currentSale;  
  
    public Register(ProductCatalog pc) {...}  
    public void endSale() {...}  
    public void enterItem(ItemID id, int qty) {...}  
    public void makeNewSale() {...}  
    public void makePayment(Money cashTendered) {...}  
}
```



Design for Code - Classes and Attributes

Sometimes, it is not obvious whether a method or property should be private or public based on the class diagram.

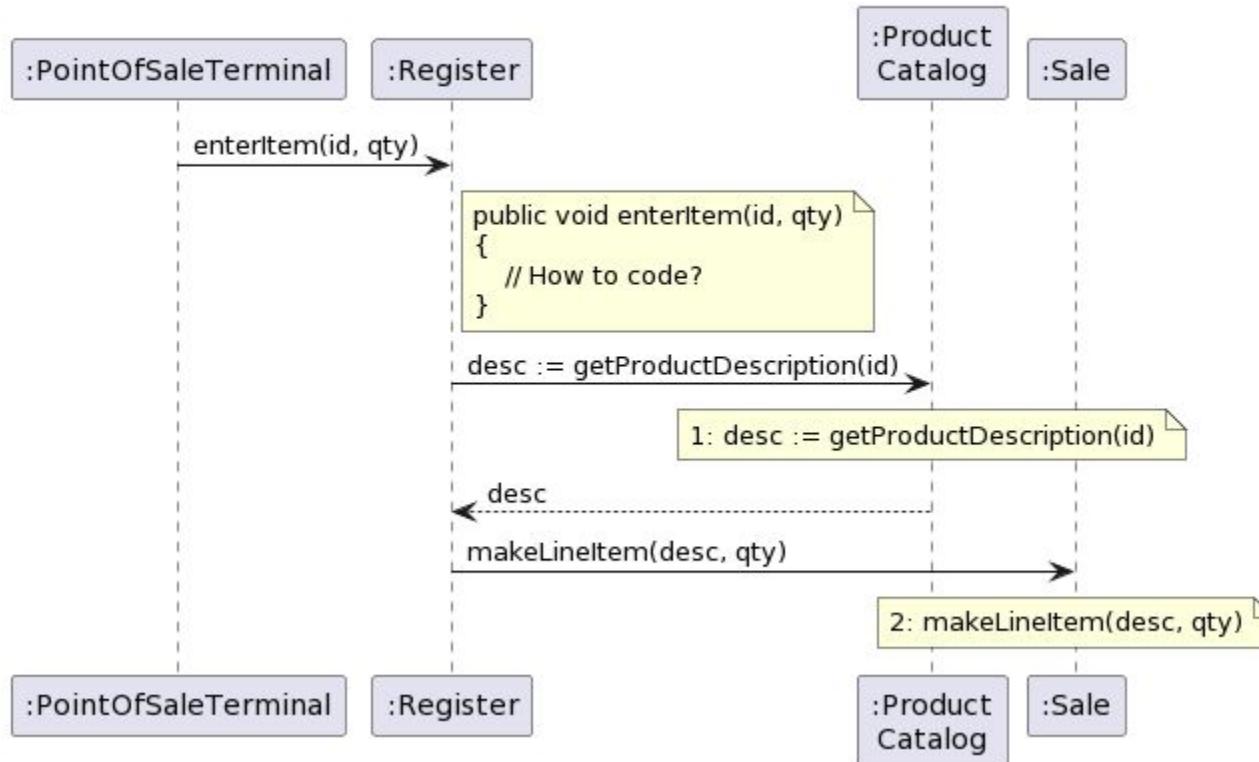


Consider the Purpose of the Class.

Default to Least Access: If in doubt, default to making the method or property private or protected.

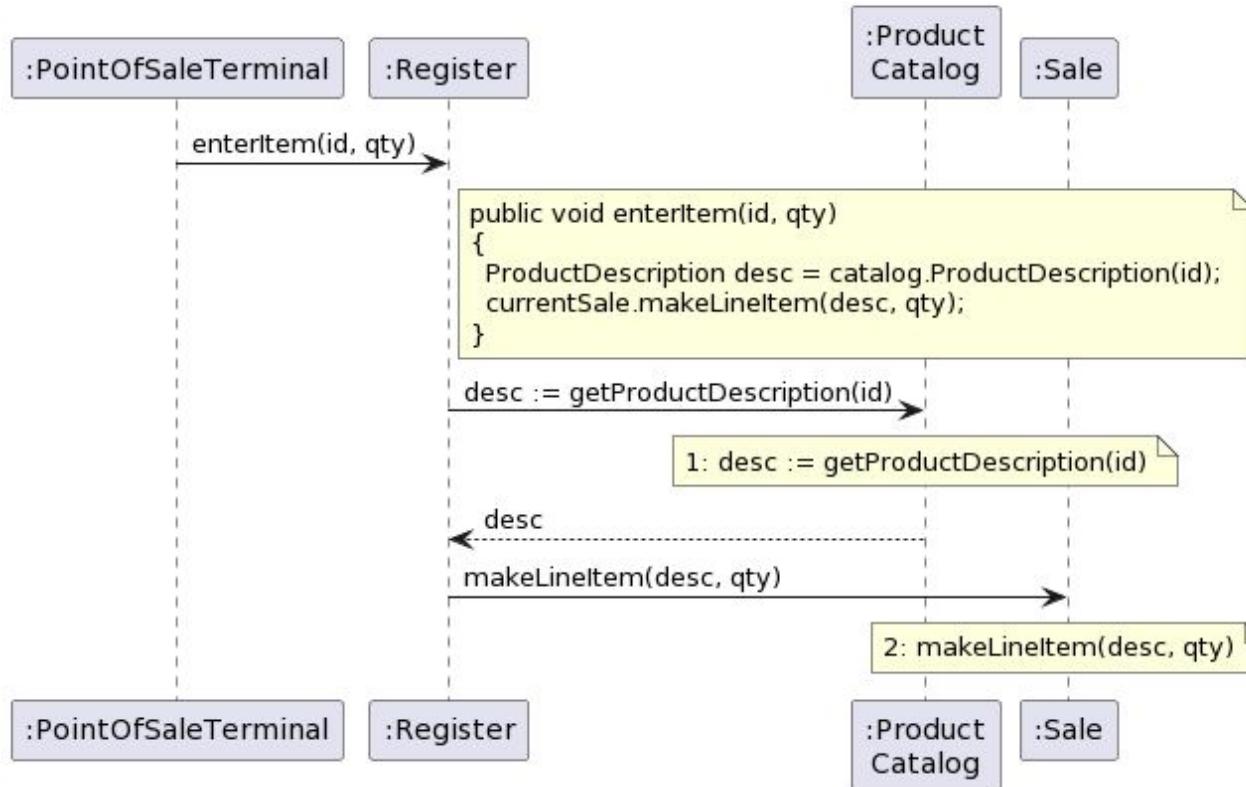
Design to Code - Methods

Design to Code - Methods

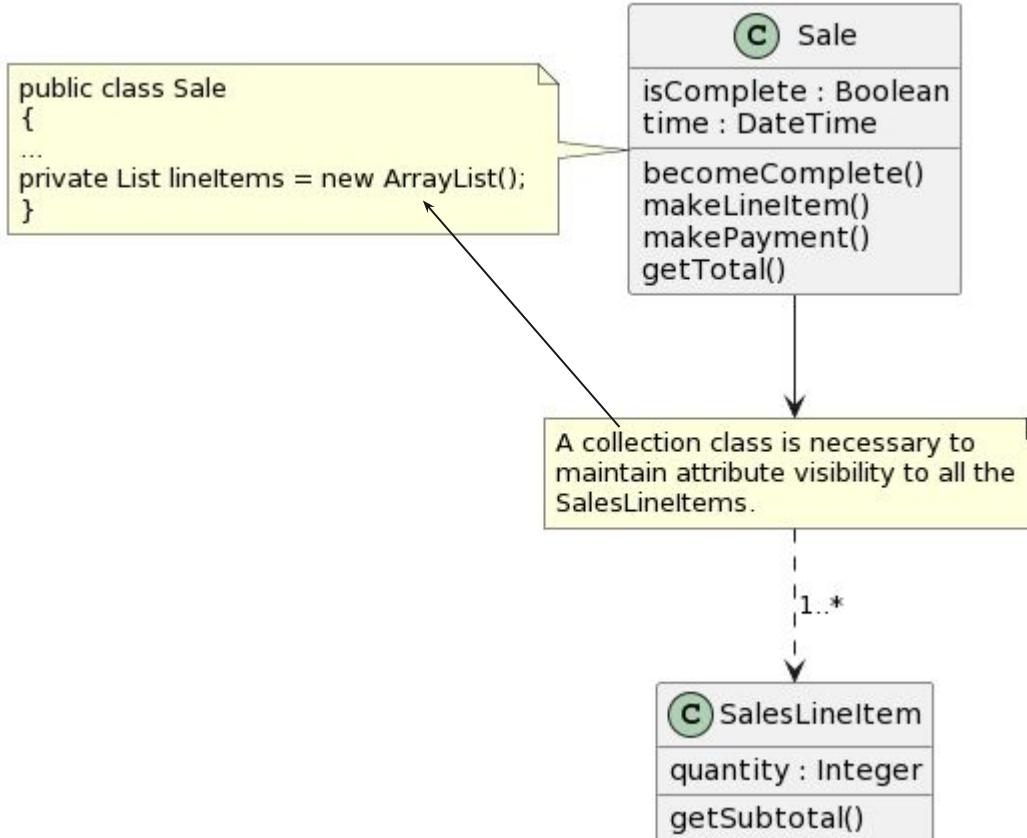


Design to Code - Methods

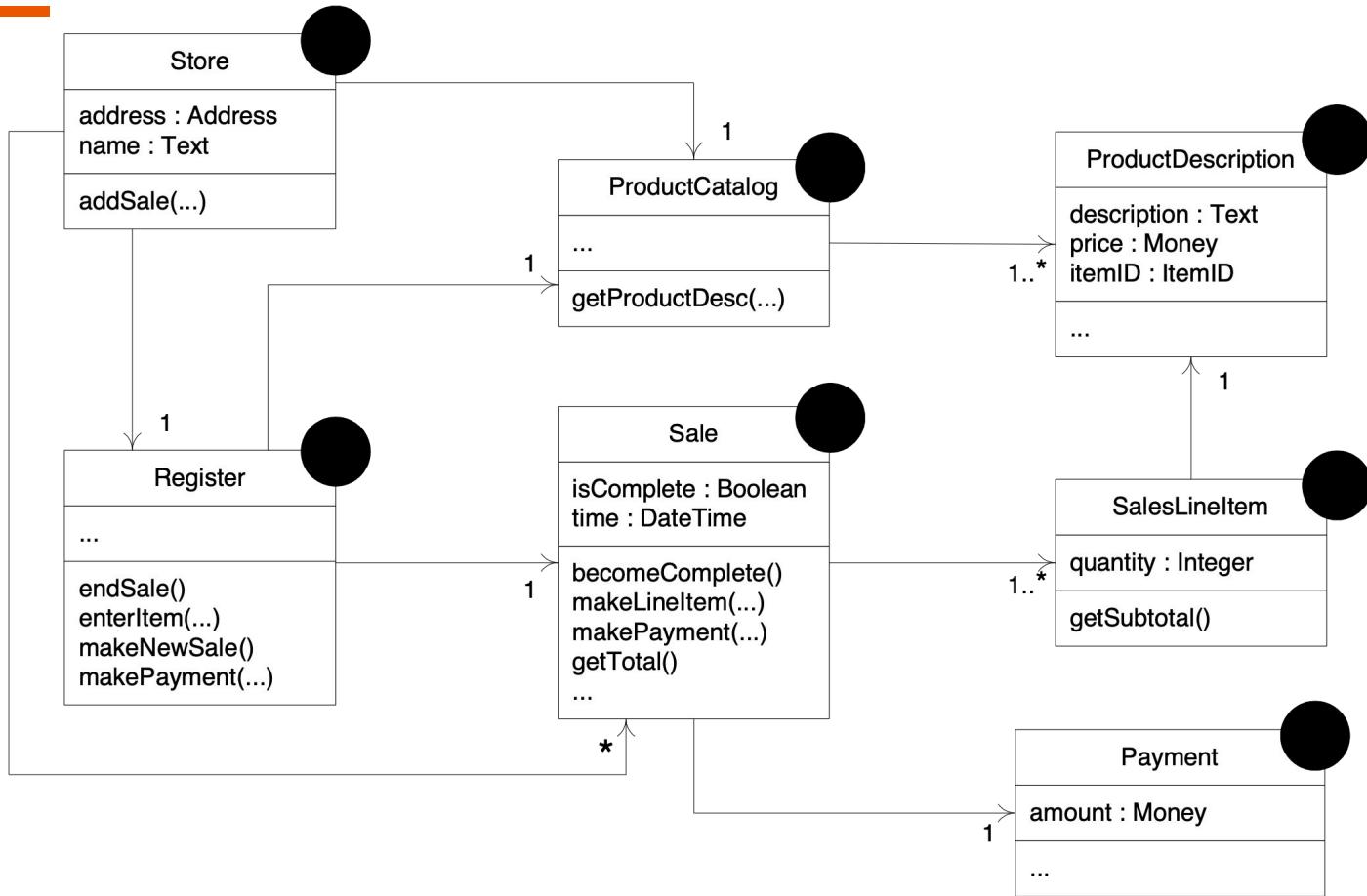
Design to Code - Methods



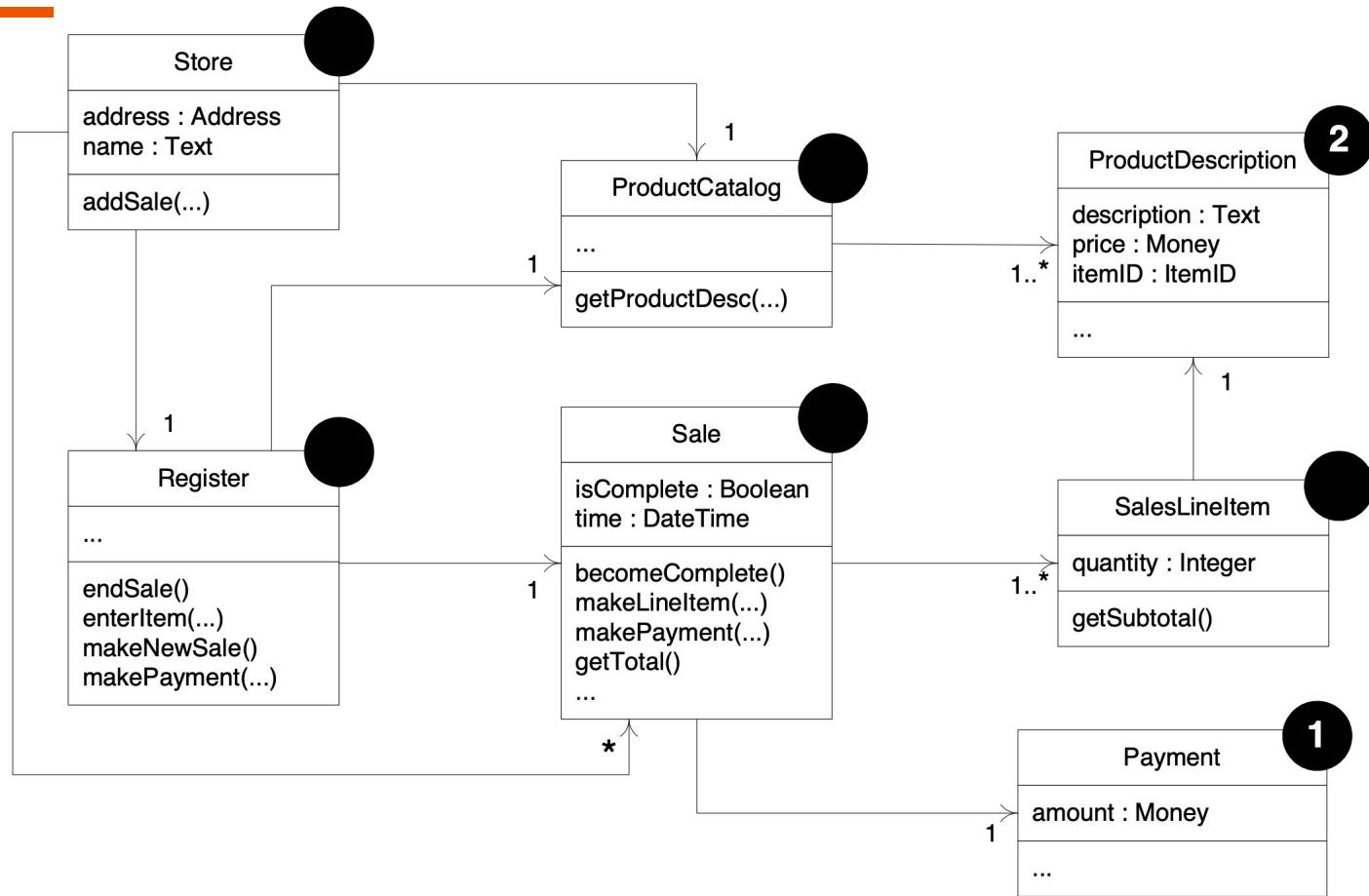
Design for code - Collection classes (..* - relations)



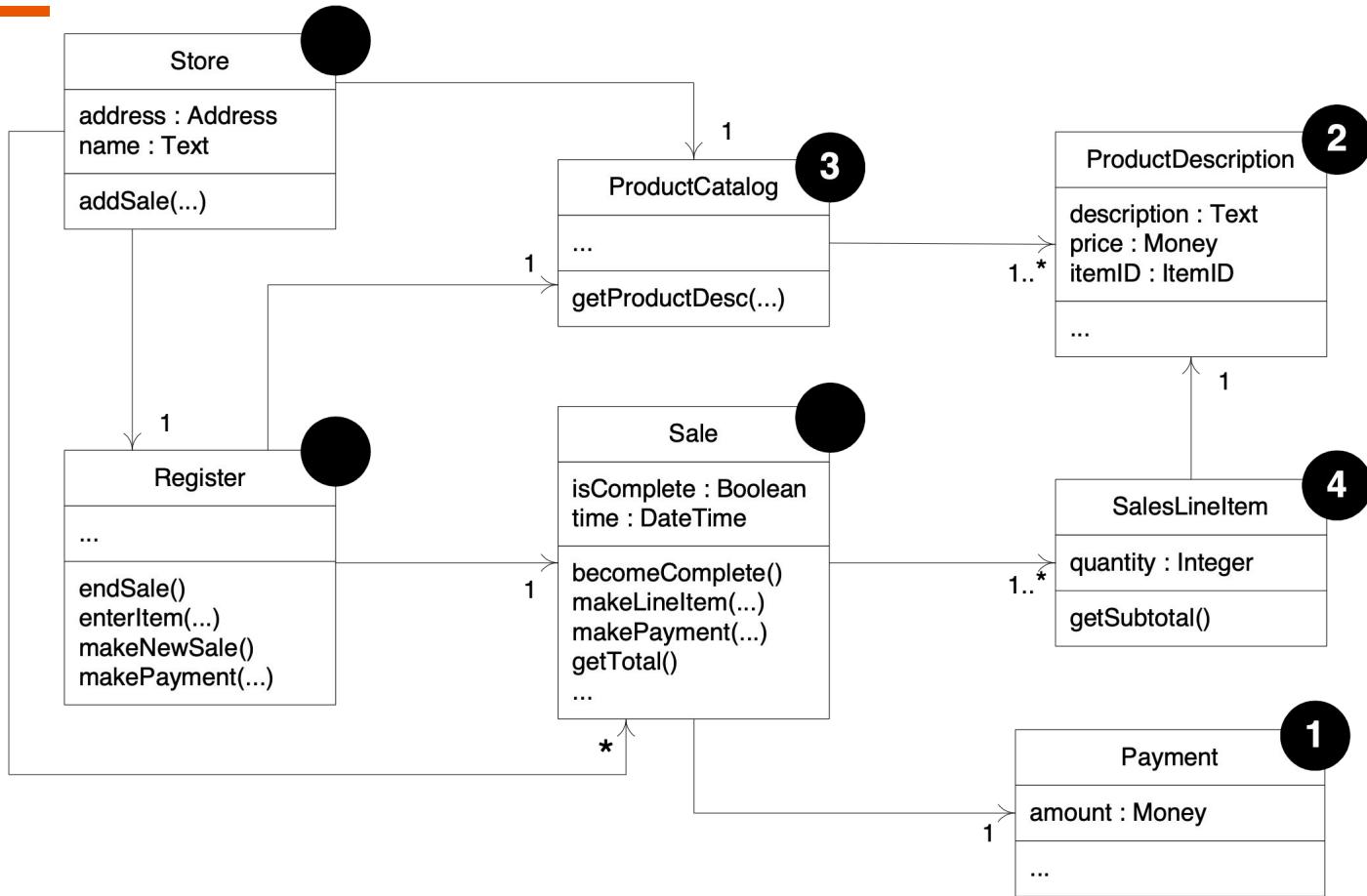
Order of implementation



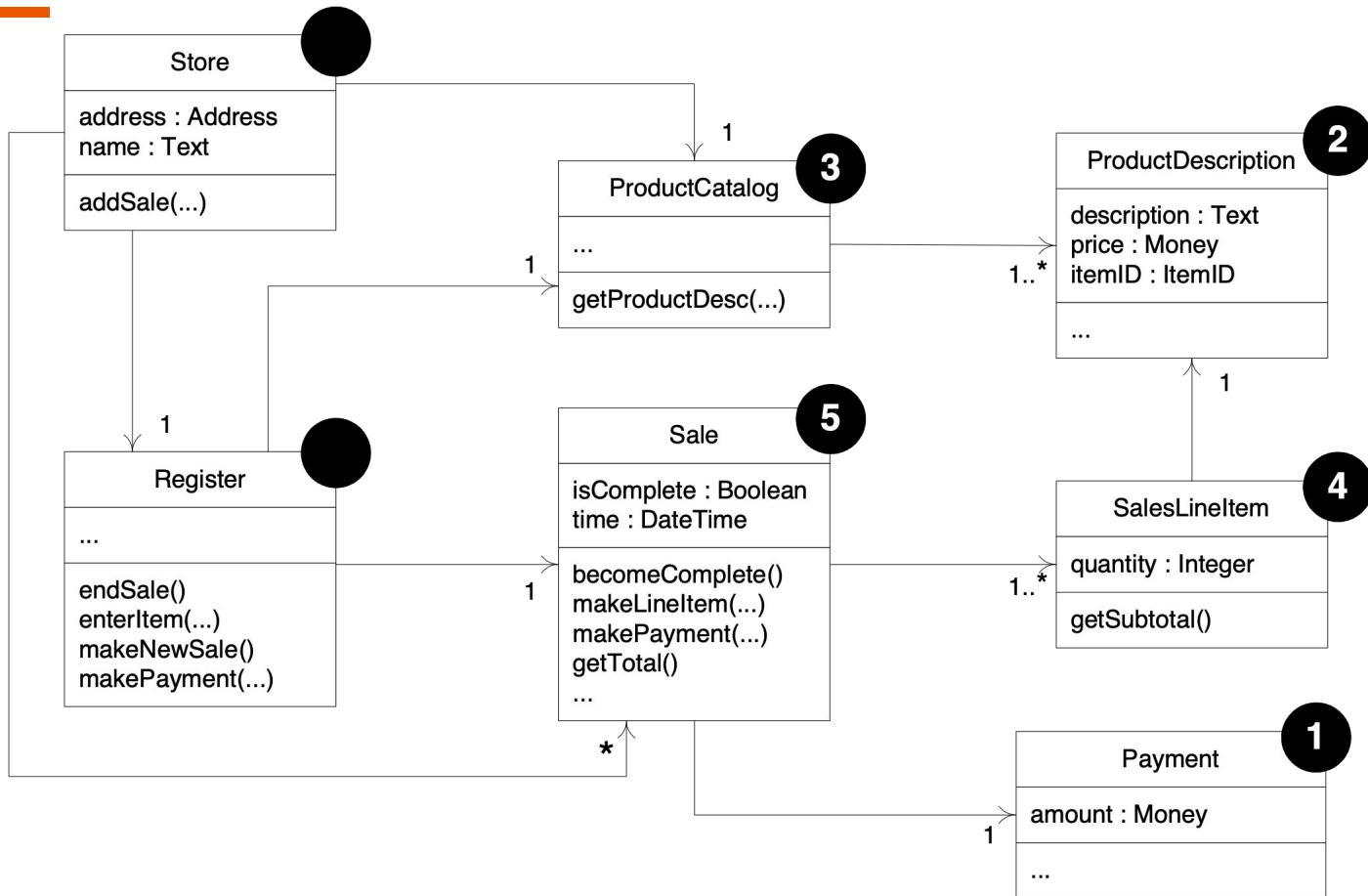
Order of implementation



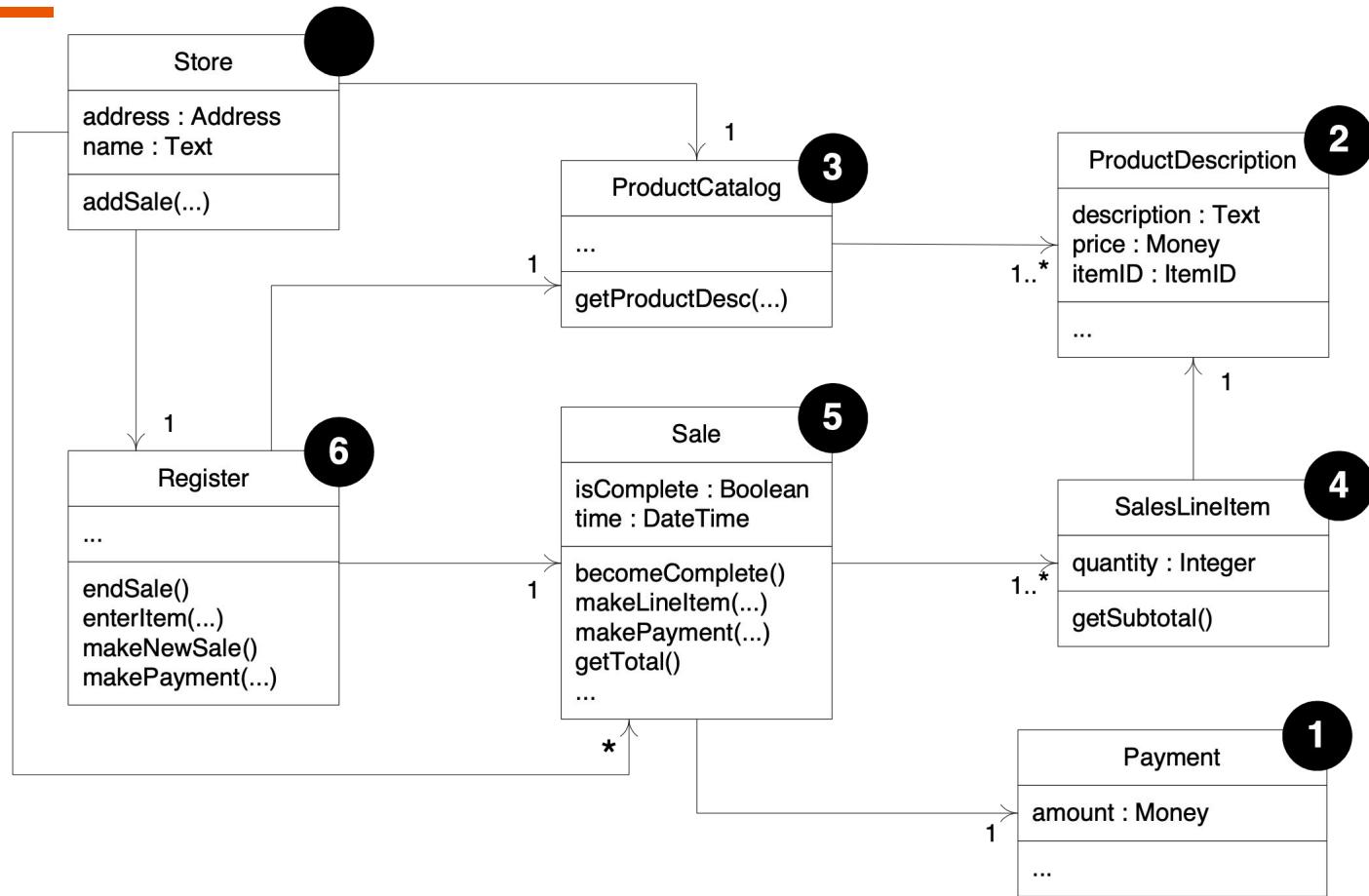
Order of implementation



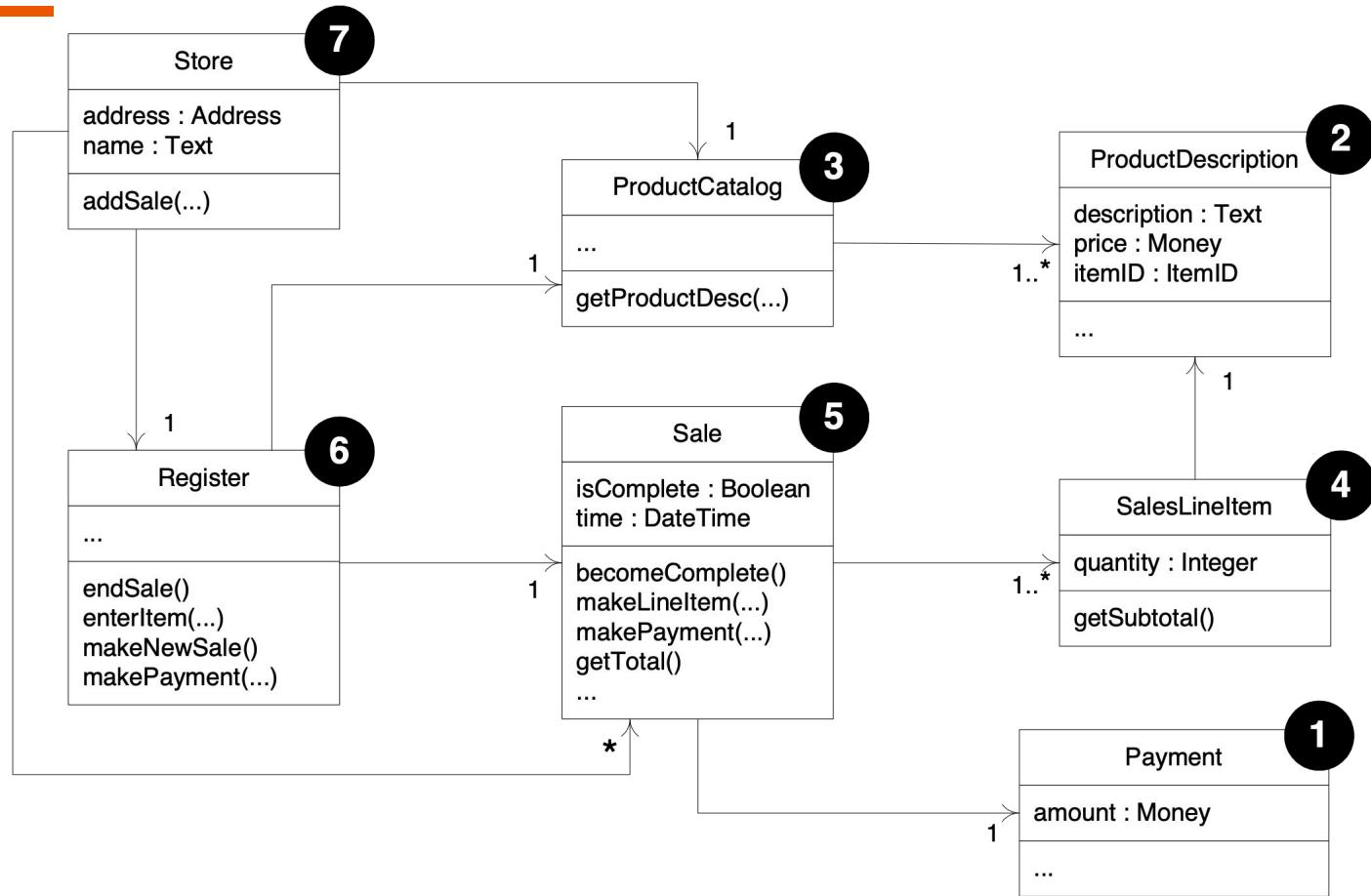
Order of implementation



Order of implementation



Order of implementation



Design to Code - Example

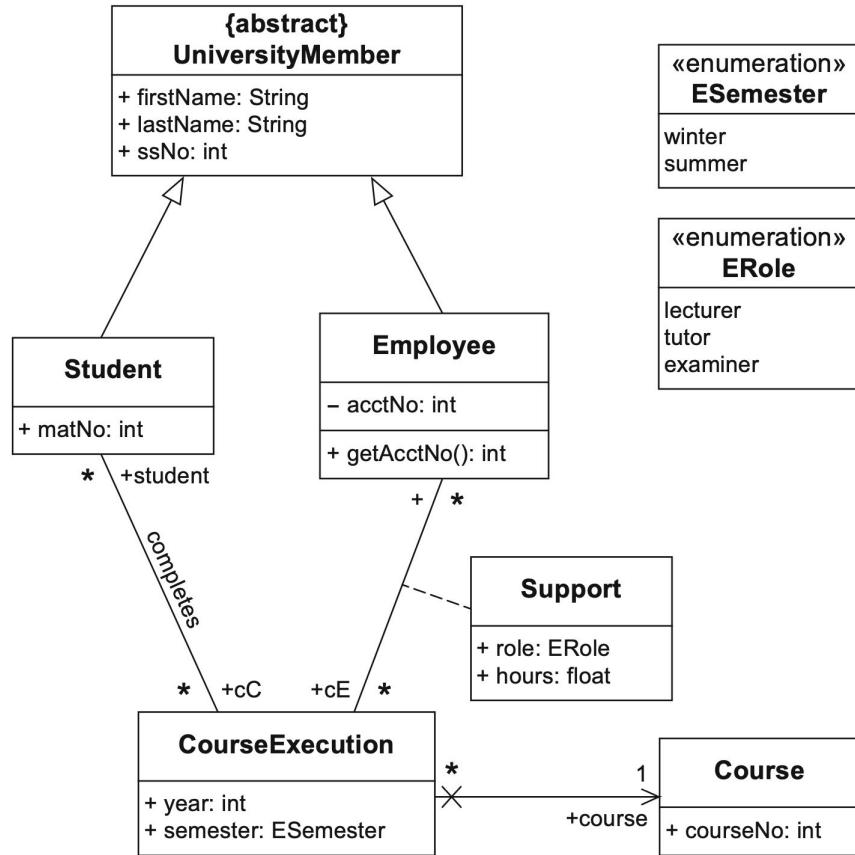


Figure 4.34
Class diagram from which code is to be generated

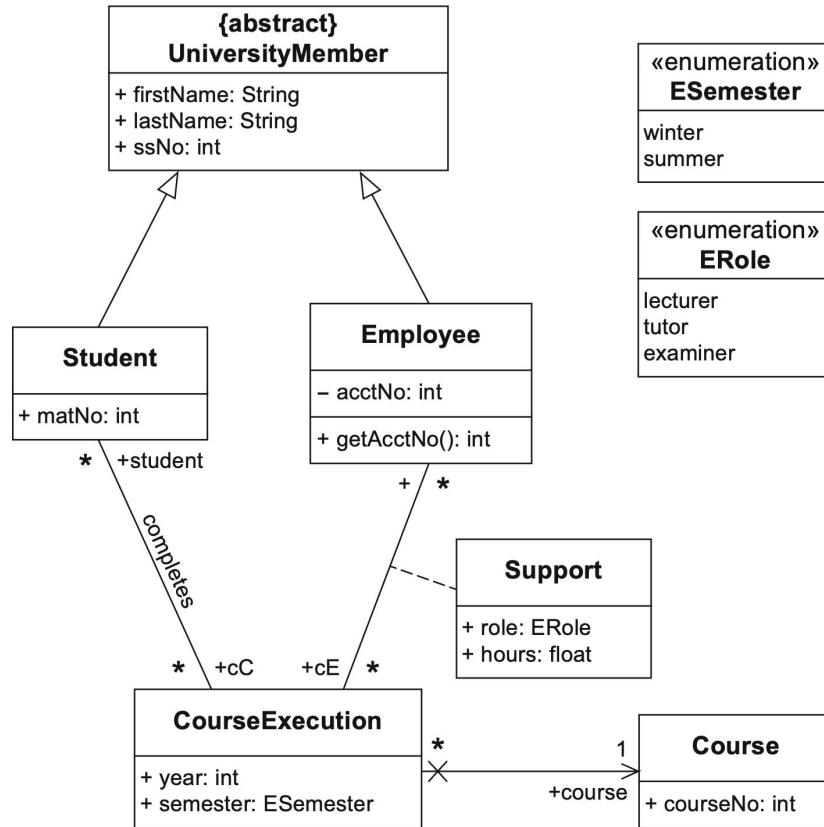


Figure 4.34
Class diagram from which
code is to be generated

```

class Course {
    public int courseNo;
}

Enumeration ESemester {
    winter;
    summer;
}

Enumeration ERole {
    lecturer;
    tutor;
    examiner;
}
  
```

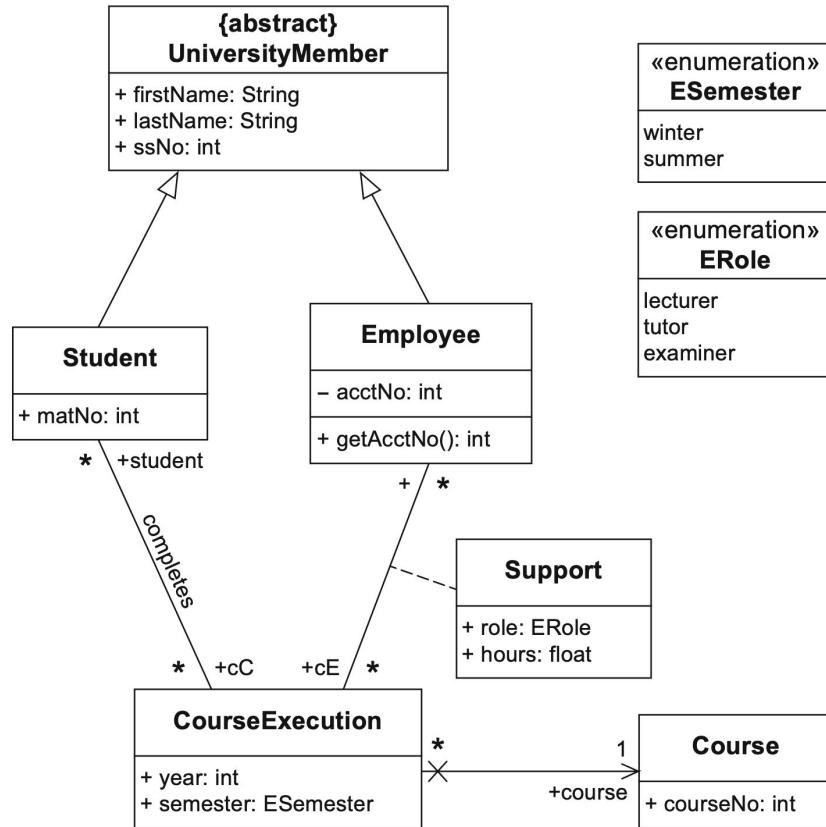


Figure 4.34
Class diagram from which
code is to be generated

```

abstract class UniversityMember {
    public String firstName;
    public String lastName;
    public int ssNo;
}

class Student extends UniversityMember {
    public int matNo;
    public CourseExecution [] cC; // completed c.
}

class Employee extends UniversityMember {
    private int acctNo;
    public CourseExecution [] cE; // supported c.
    public int getAcctNo { return acctNo; }
}
  
```

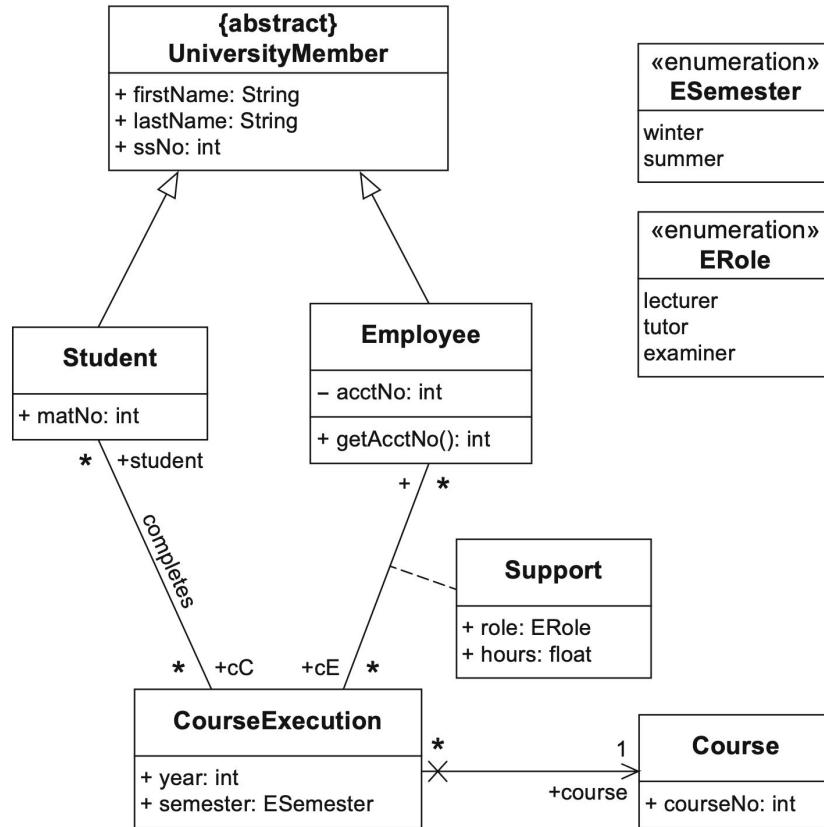


Figure 4.34
Class diagram from which
code is to be generated

```

class CourseExecution {
    public int year;
    public ESemester semester;
    public Student [] student;
    public Course course;
    public Hashtable support;
        // Key: employee
        // Value: (role, hours)
}
  
```

Figure 4.35
Java code that can be generated automatically from
Fig. 4.34

```
abstract class UniversityMember {
    public String firstName;
    public String lastName;
    public int ssNo;
}

class Student extends UniversityMember {
    public int matNo;
    public CourseExecution [] cC; // completed c.
}

class Employee extends UniversityMember {
    private int acctNo;
    public CourseExecution [] cE; // supported c.
    public int getAcctNo { return acctNo; }
}

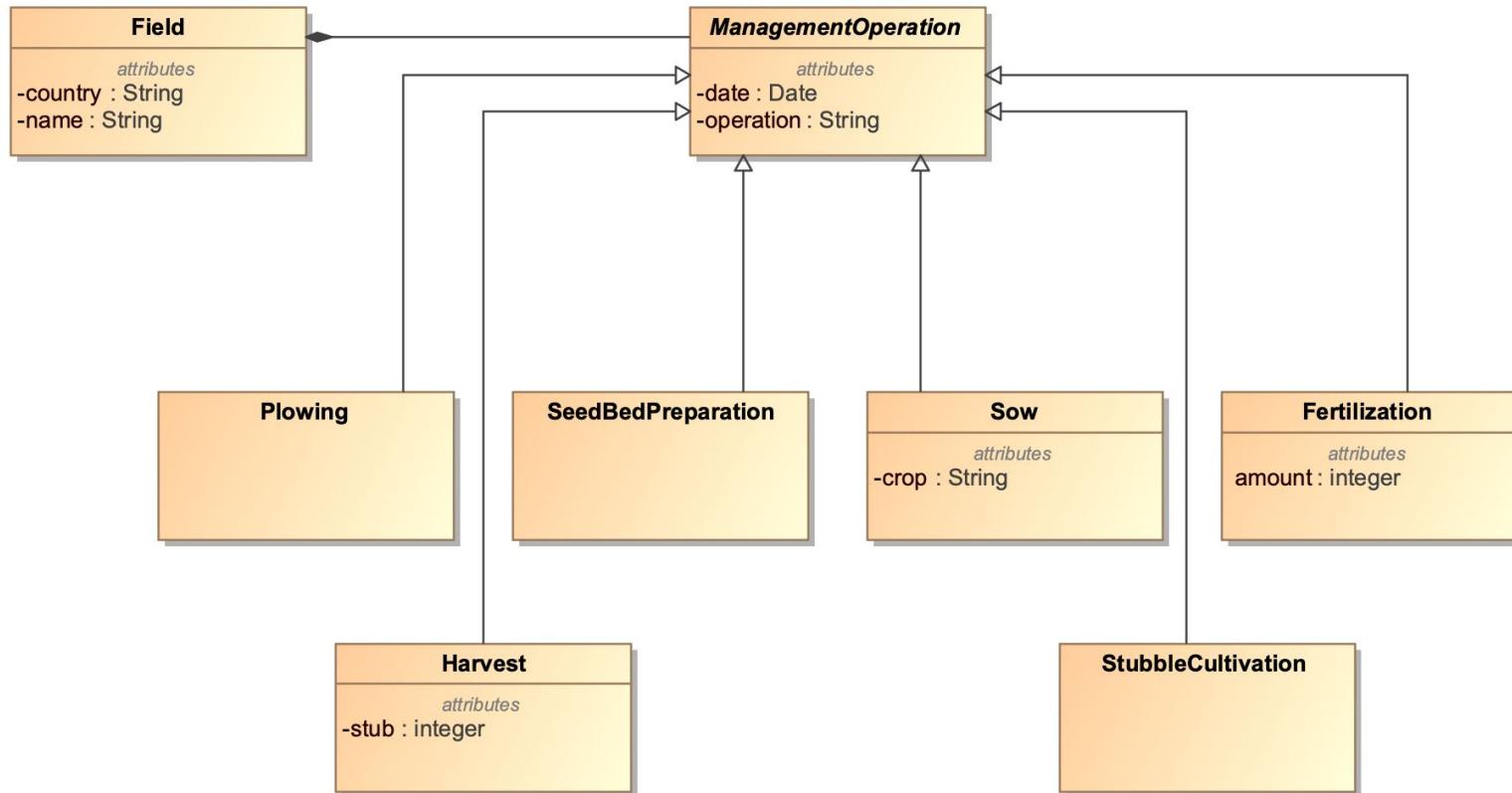
class CourseExecution {
    public int year;
    public ESemester semester;
    public Student [] student;
    public Course course;
    public Hashtable support;
        // Key: employee
        // Value: (role, hours)
}

class Course {
    public int courseNo;
}

Enumeration ESemester {
    winter;
    summer;
}

Enumeration ERole {
    lecturer;
    tutor;
    examiner;
}
```

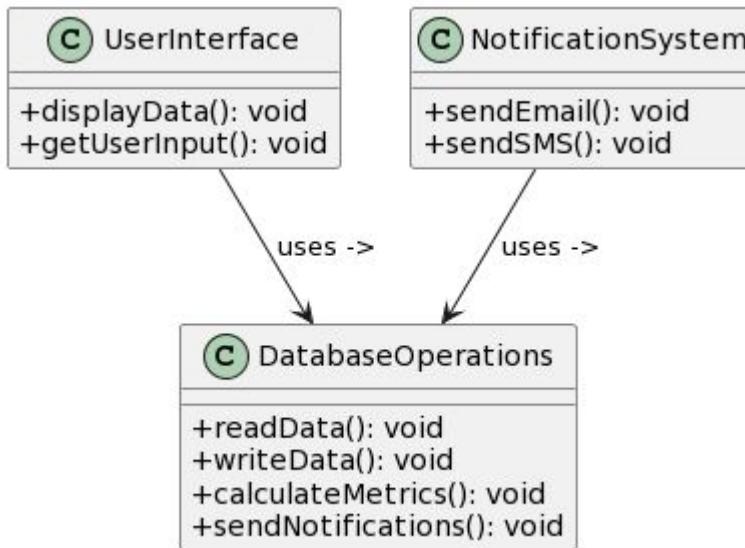
Design for Code - Legacy



Design for Code - Legacy

- Define more specialized classes (subclasses) from existing classes (superclass)
- Better design:
 - easier to understand, maintain, test
 - less code duplication (DRY: Don't repeat yourself)

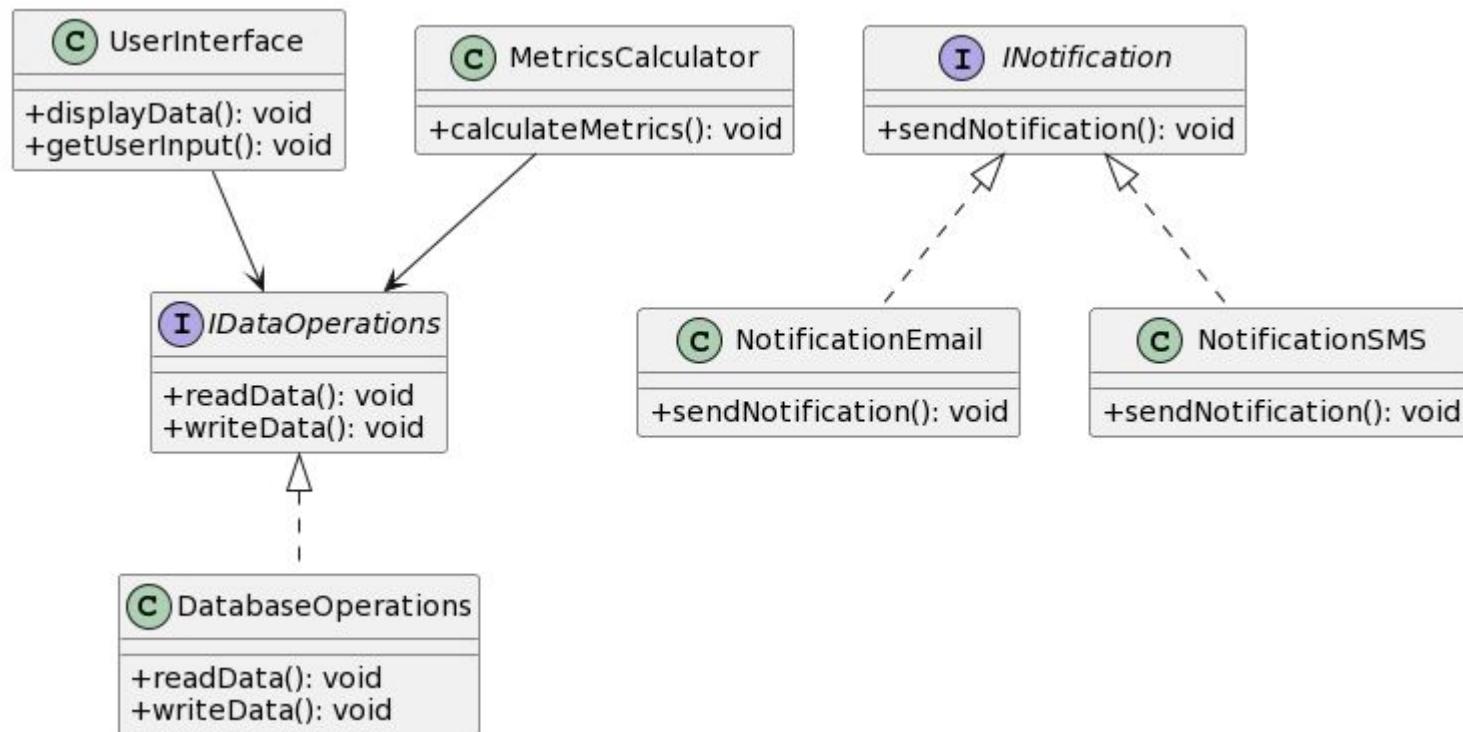
Counterexample



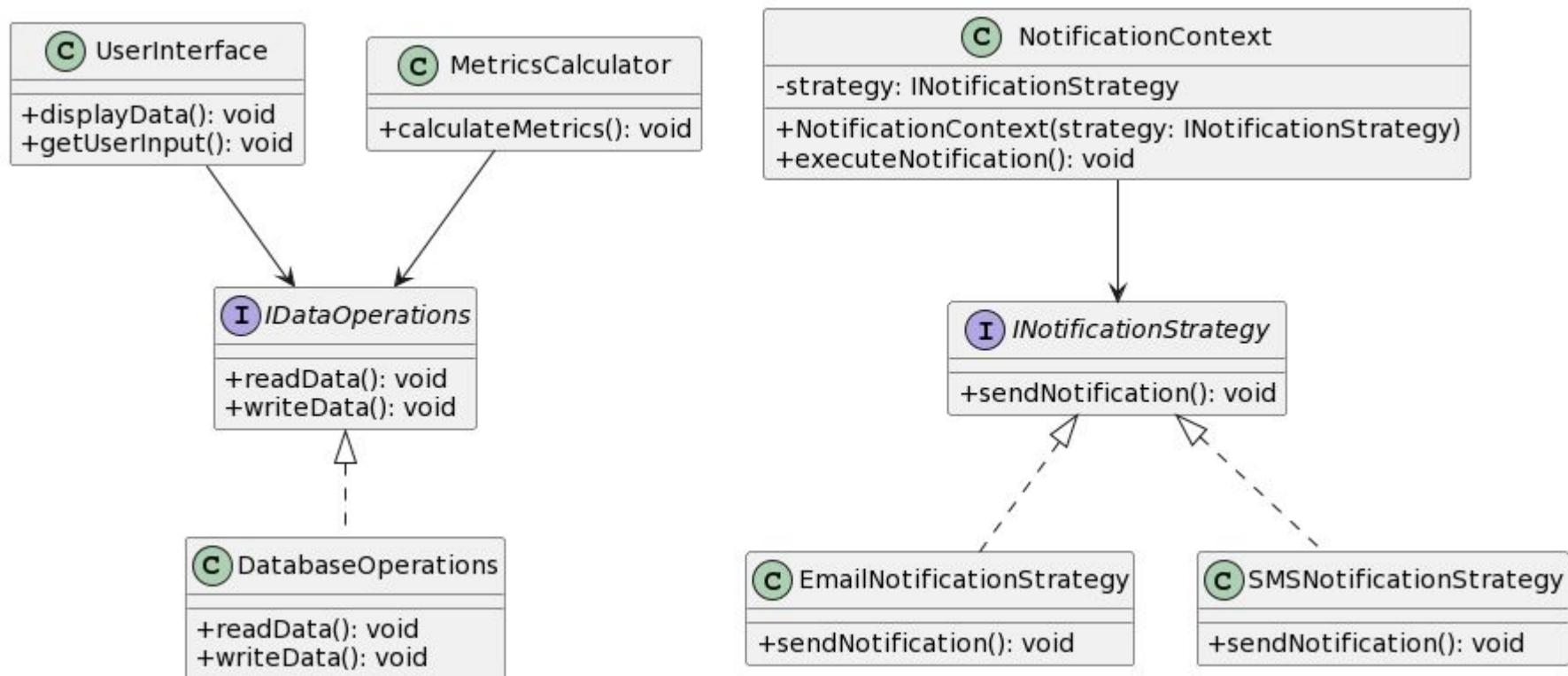
Tight Coupling: The classes are tightly coupled. UserInterface directly depends on the methods inside DatabaseOperations, and so does NotificationSystem.

Lack of Single Responsibility: The DatabaseOperations class handles database interactions, calculations, and also sending notifications.

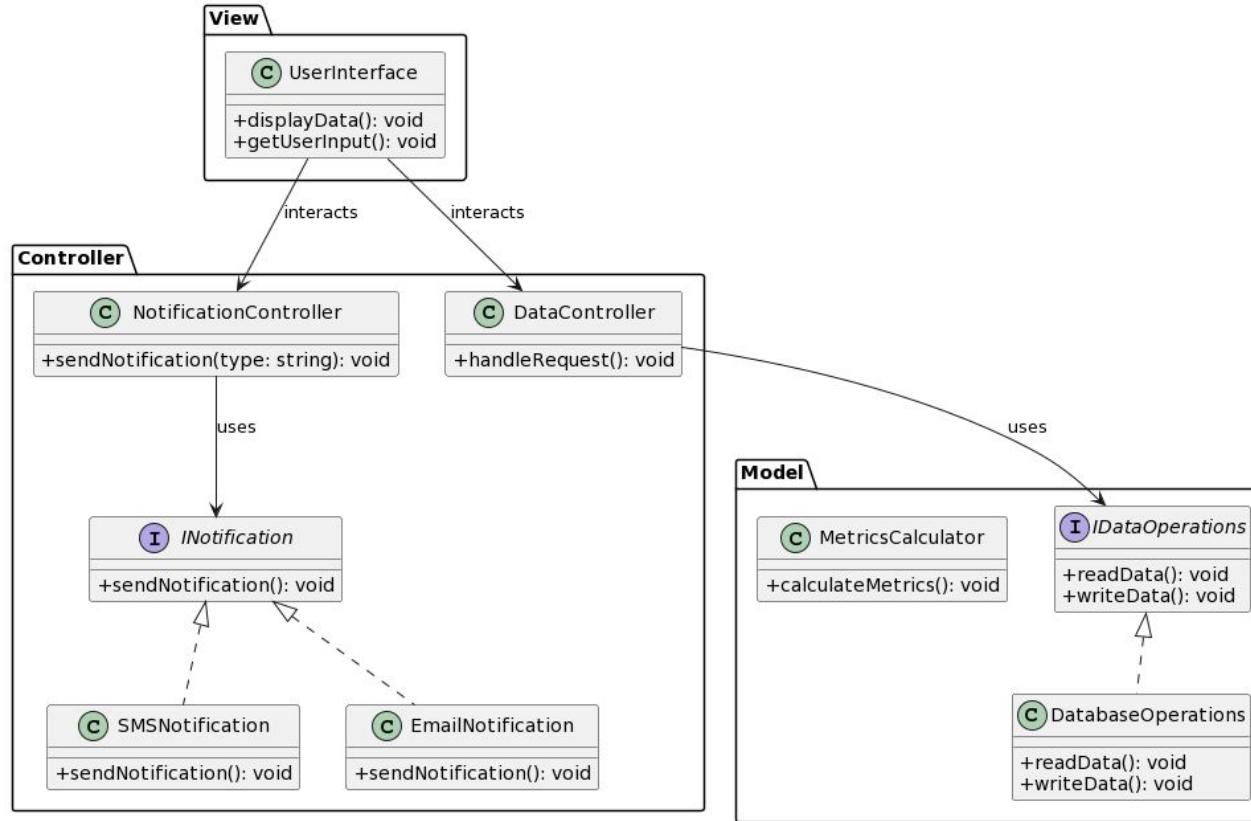
Better Design 1 - GRASP: Indirection



Better Design 2 - GoF Strategy Pattern



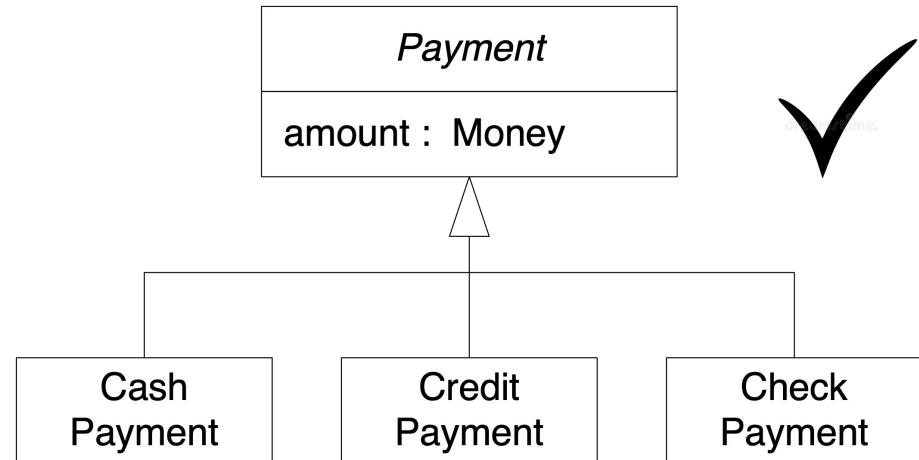
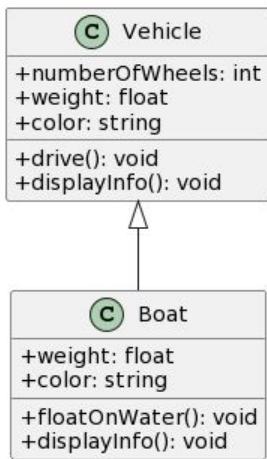
Better Design 3 - MVC



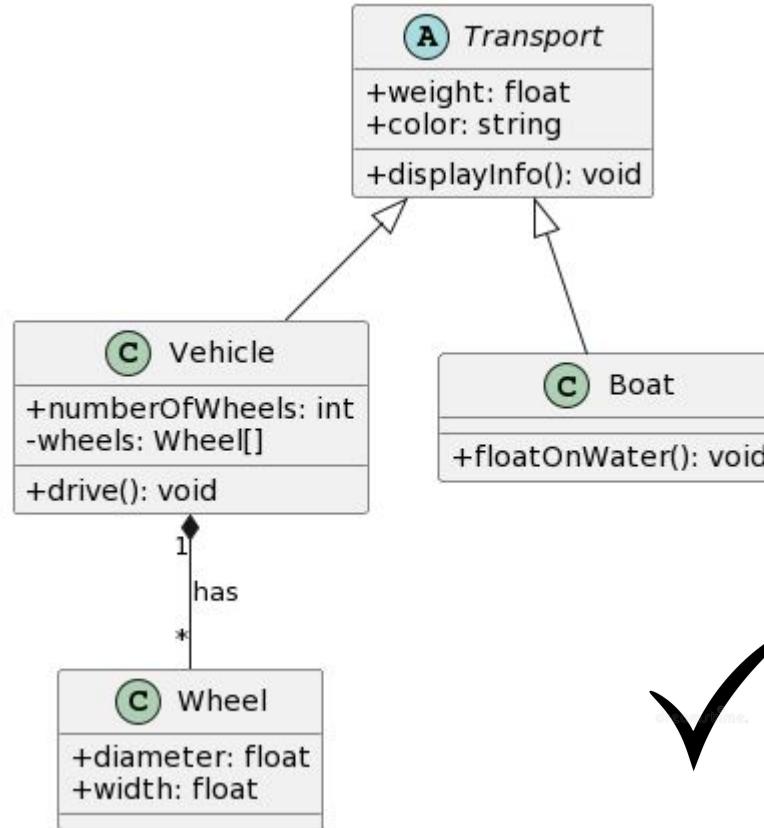
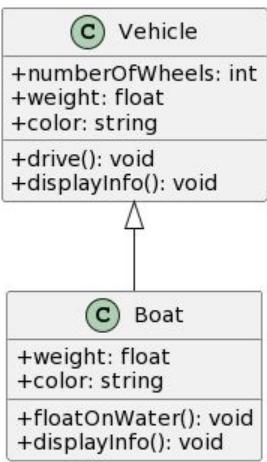
Inheritance

Instructions for using subclass:

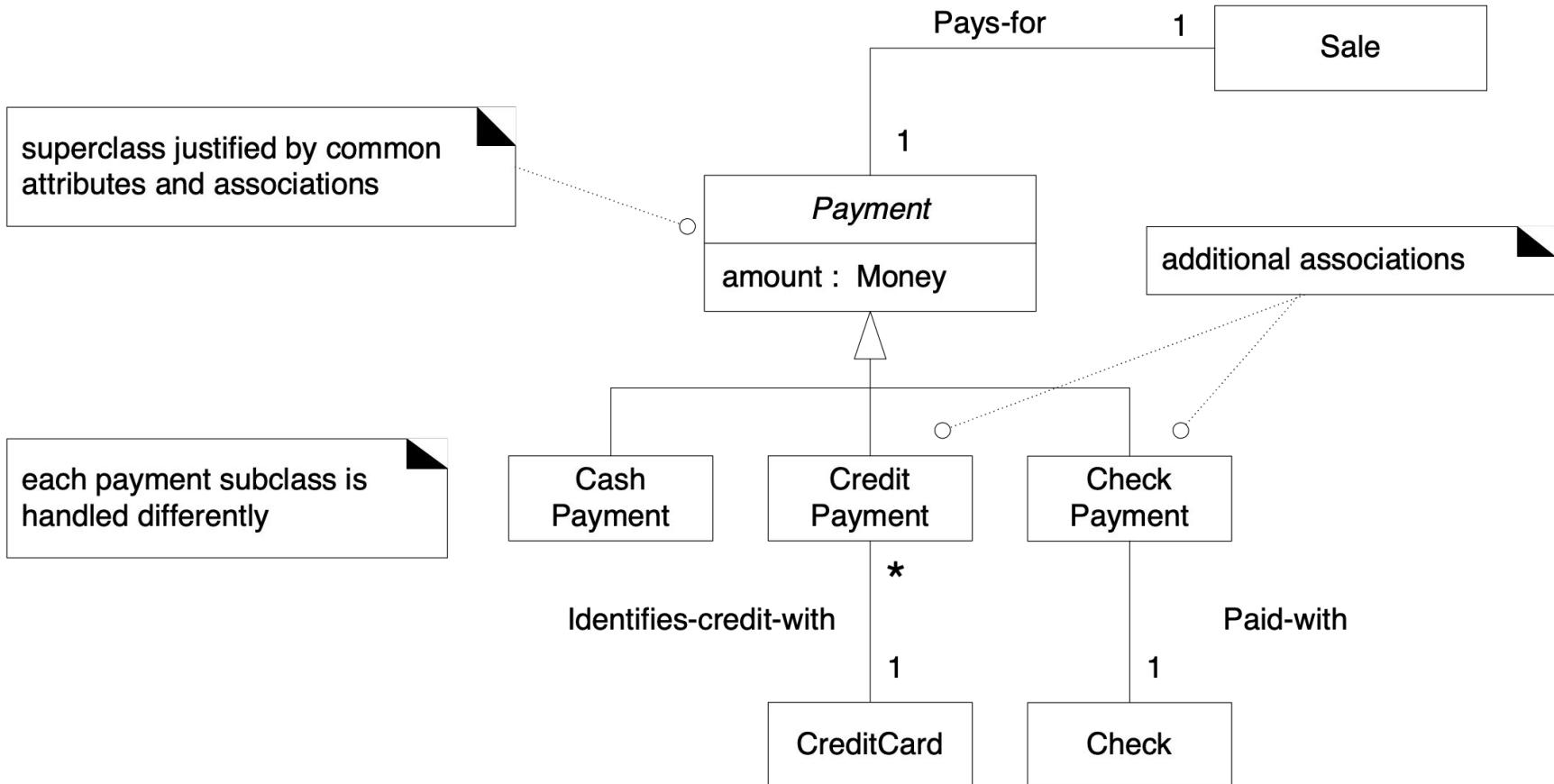
- 100% rule (attributes, associations)
 - If all fields are inherited.
- “is a” (“is”) rule
 - (Cucumber is a vegetable)



Inheritance

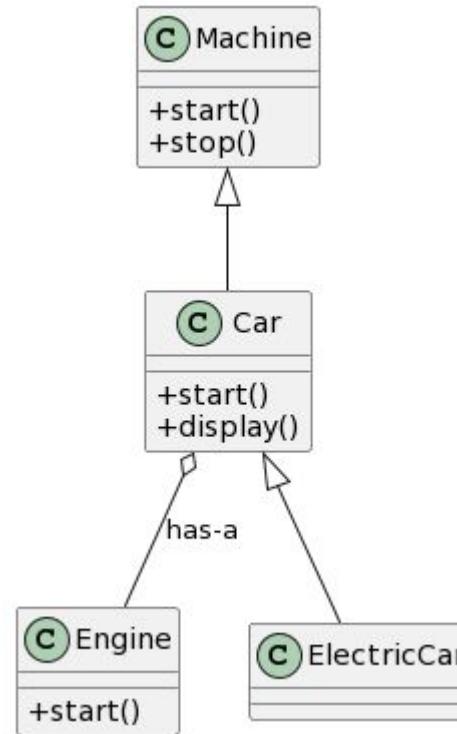
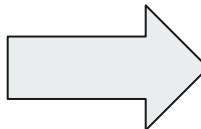
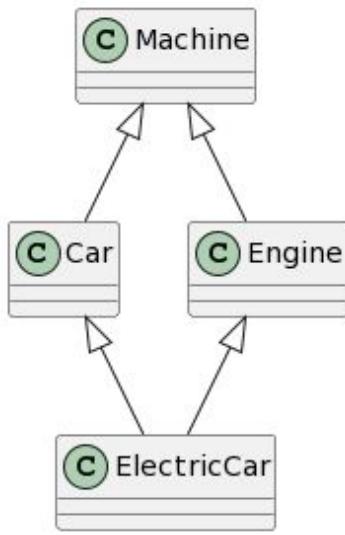


Inheritance - common and difference



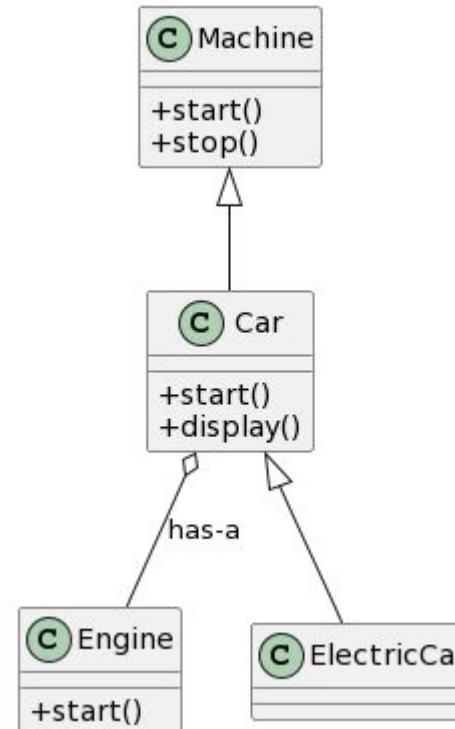
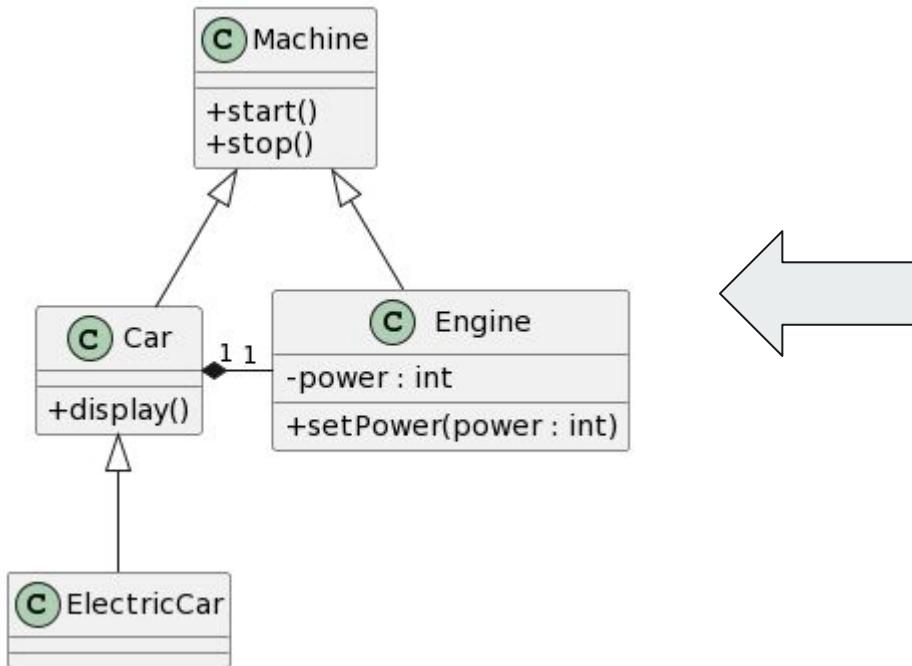
“Has a” or “Is a”?

Using “has a” to avoid multiple inheritance



“Has a” or “Is a”?

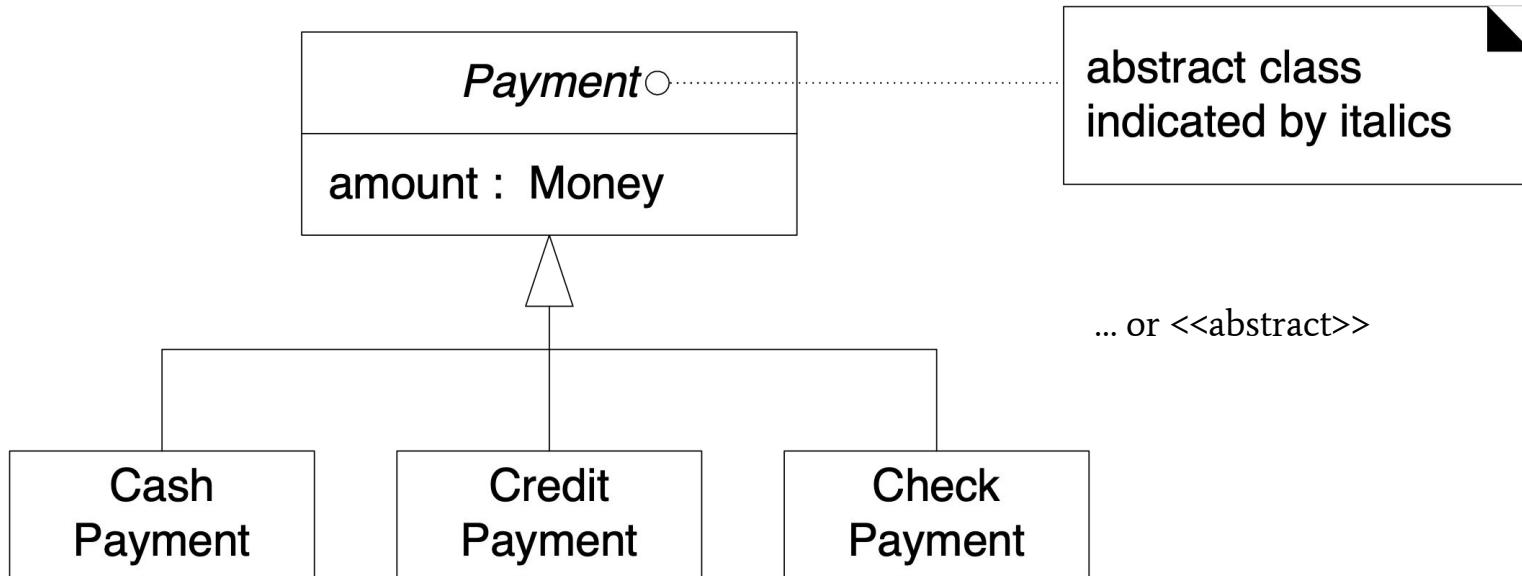
Using “has a” to avoid multiple inheritance



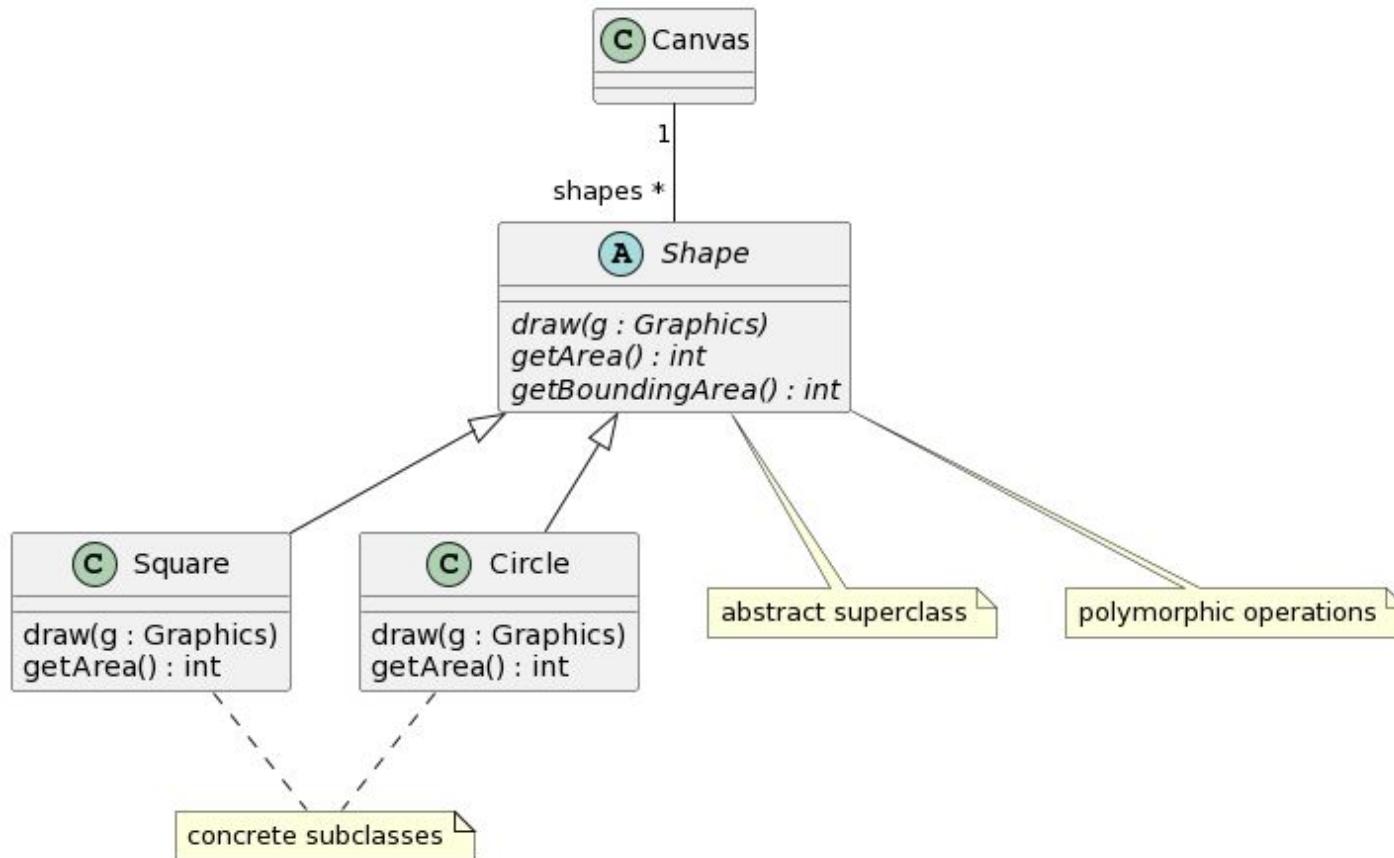
Abstract classes

- A class can be defined as abstract.
- A method can be abstract - ie without implementation.
- A class that contains abstract methods is abstract. (Why?)
- An abstract class cannot be instantiated.

Abstract classes



Polymorphism



Polymorphic method

A polymorphic method call has "many forms"

- Subclasses implement the method differently

In this case:

shape.draw() is implemented differently -
dependent subclass

```
// Define an abstract Shape class with an abstract draw() method
abstract class Shape {
    public abstract void draw();
}

// Implement the Square subclass of Shape
class Square extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}

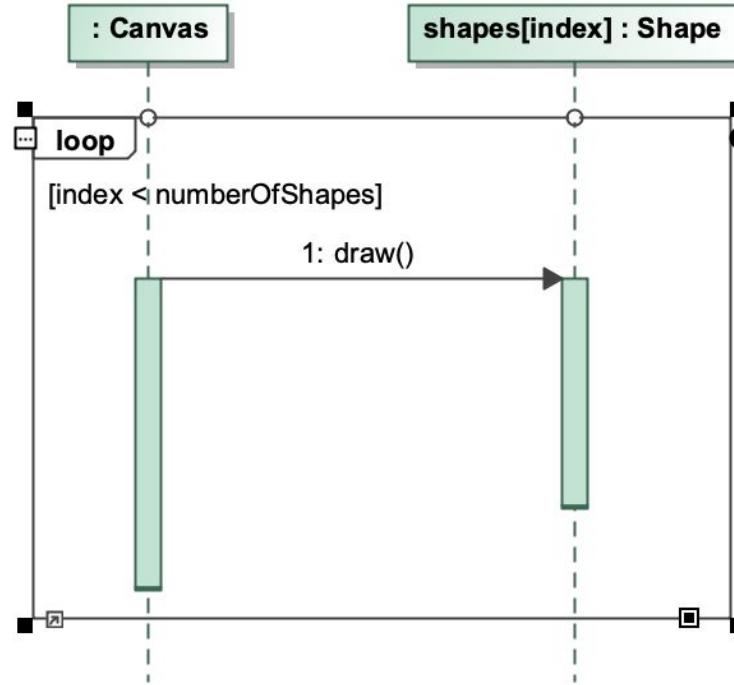
// Implement the Circle subclass of Shape
class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a Square and Circle object
        Shape square = new Square();
        Shape circle = new Circle();

        // Call the draw() method on each object
        square.draw(); // Outputs: Drawing a square
        circle.draw(); // Outputs: Drawing a circle
    }
}
```

Polymorphism

Ex.: Iteration over a list of shapes



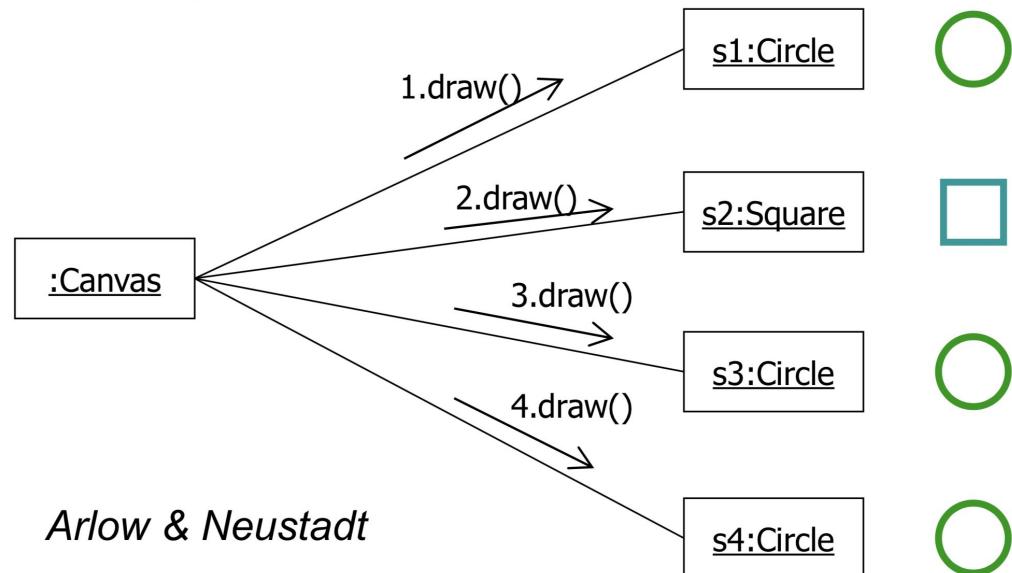
Various shapes can be drawn using a single method call, and the correct drawing method for each shape type is automatically determined and executed.

Polymorphism

Each subclass of objects has its own implementation of the draw() method

When the draw() method is called, each object type uses its own implementation

Objektdiagram:



Development methods 62531

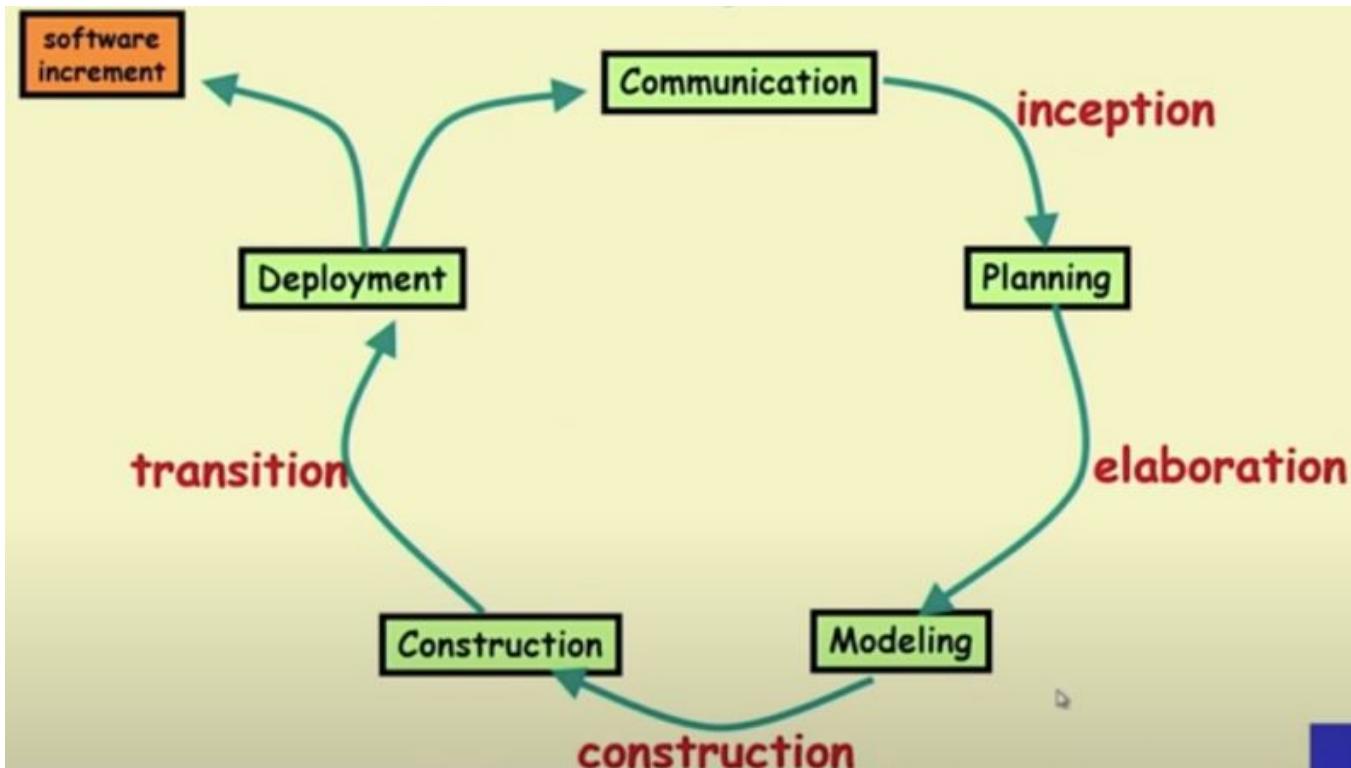
Lecture 10 - State Diagrams, Software Quality, Review Technique
Lei You, Assistant Professor

leiyo@dtu.dk

What have we learned?

- Architecture
 - Layered? MVC? ---> Separations of Concerns
- Class diagrams
 - Domain model
 - Design model
- GRASP
 - General Responsibility Assignment Software Patterns
 - 9 Principles in total
- Design Pattern and GoF
 - Design solution to common problem
 - Summarized from experiences
 - Some concrete patterns from GoF
- From Design to Code

Revisit the Unified Process (UP)



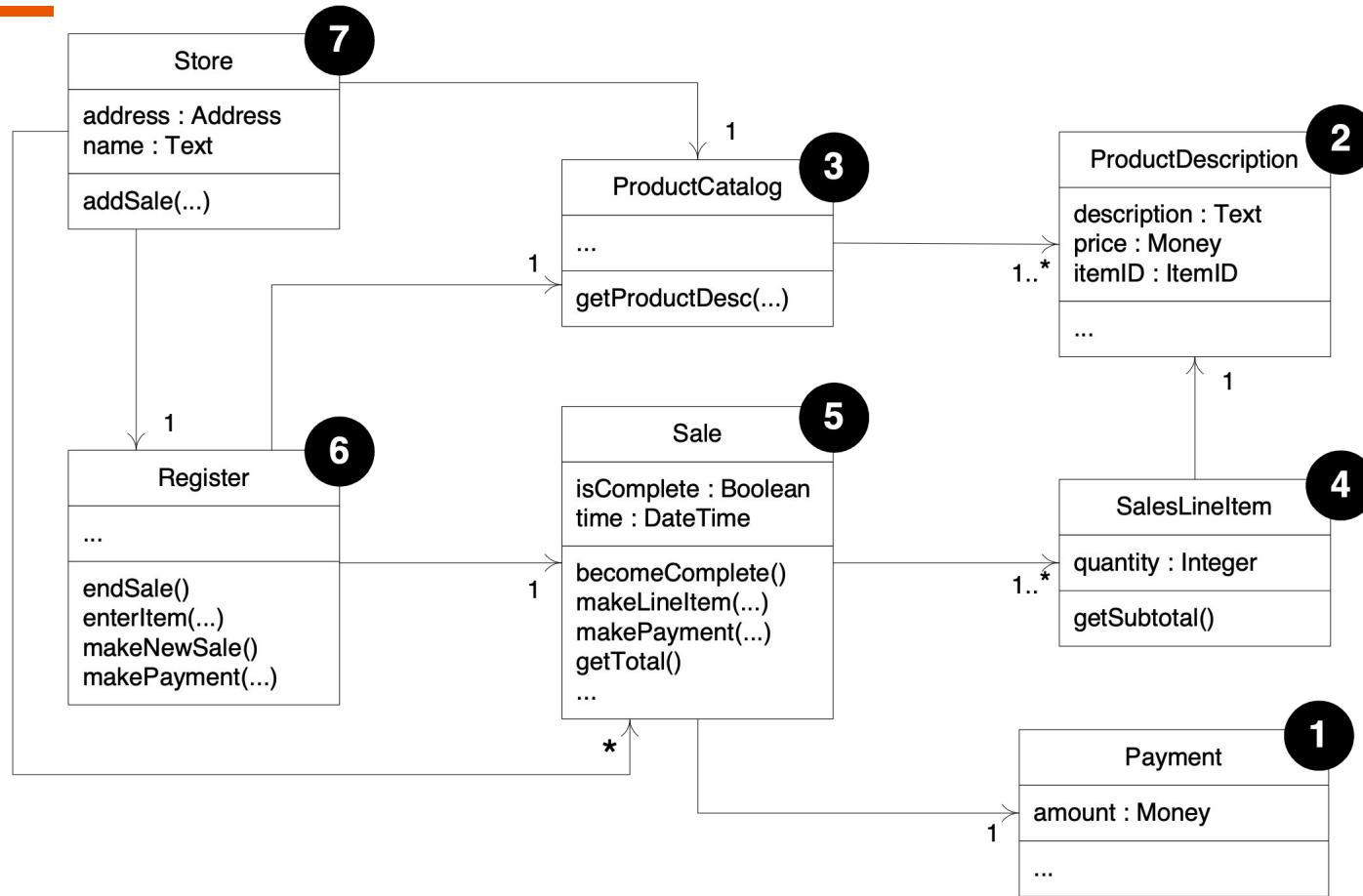
Object Knowledge / Visibility

- Attributes
 - B is an attribute of A
- Parameters
 - B is a parameter to a Method in A
- Local variables
 - B is a local variable in a method in A
- Global variable
 - B is a global variable

How to code it in java?

```
public class A {  
    private B bAttribute; //Attribut  
  
    public void doSomething() {  
        bAttribute.doBStuff();  
    }  
  
    public void doSomething(B bParameter) {  
        bParameter.doBStuff();  
    }  
  
    public void doSomethingElse(){  
        B b = new B();  
        b.doBStuff();  
    }  
  
    public void doGlobalBstuff(){  
        B.getInstance().doBStuff();  
    }  
}
```

Design to Code: Order of implementation



Design to Code - Example

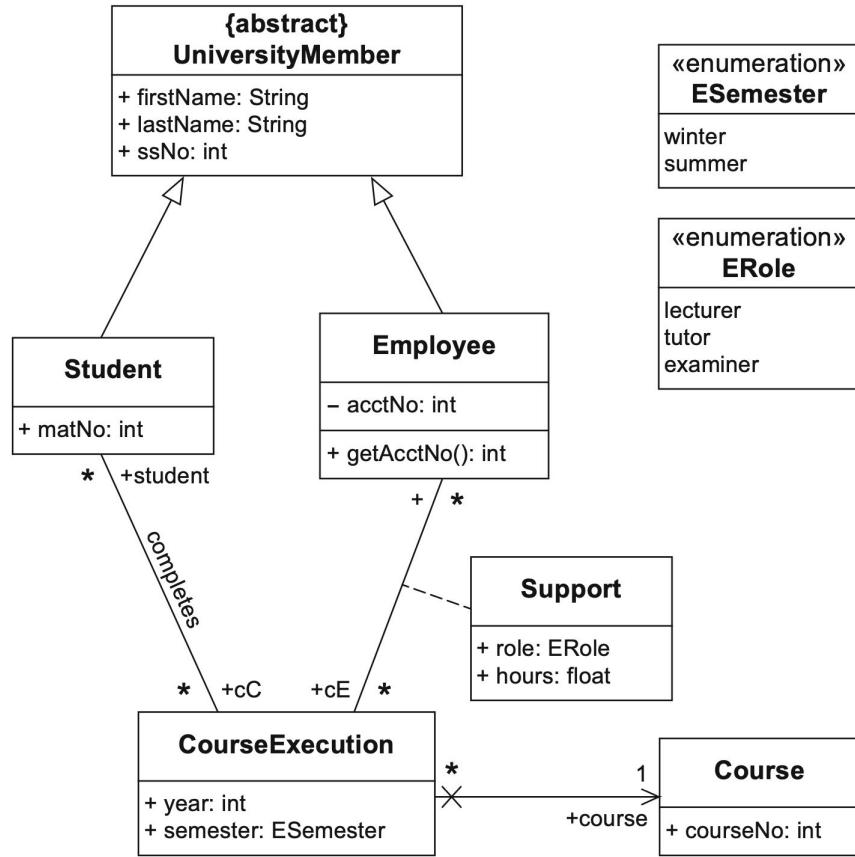
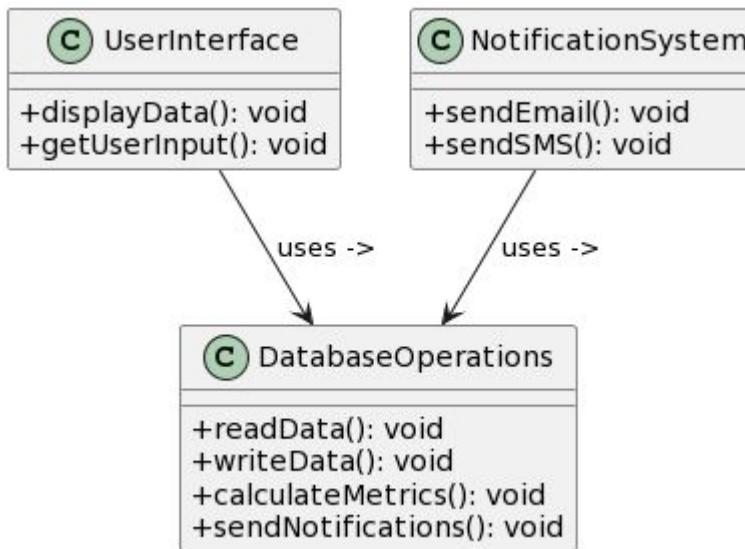


Figure 4.34
Class diagram from which code is to be generated

Design for Legacy Code - Bad Example

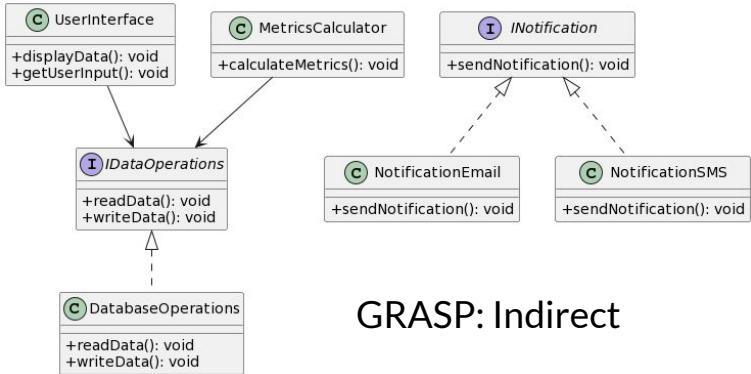


Tight Coupling: The classes are tightly coupled. UserInterface directly depends on the methods inside DatabaseOperations, and so does NotificationSystem.

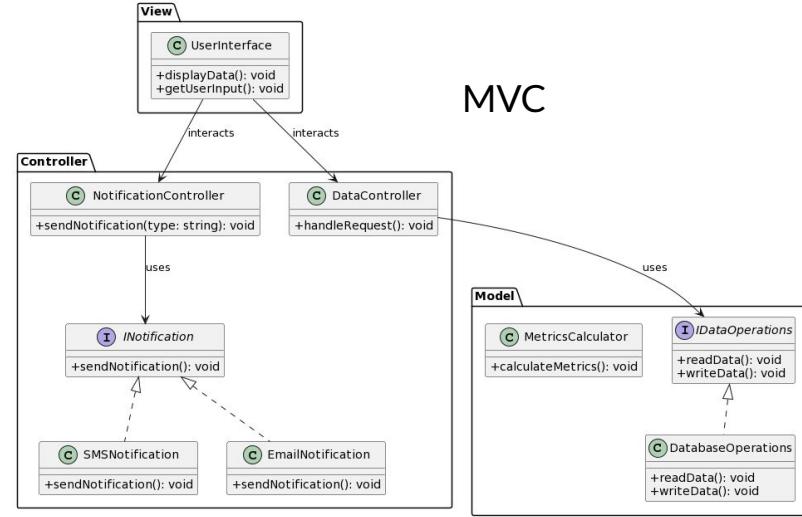
Lack of Single Responsibility: The DatabaseOperations class handles database interactions, calculations, and also sending notifications.

Key point: If such code exists as legacy, then “design to code” process for new functionality is difficult

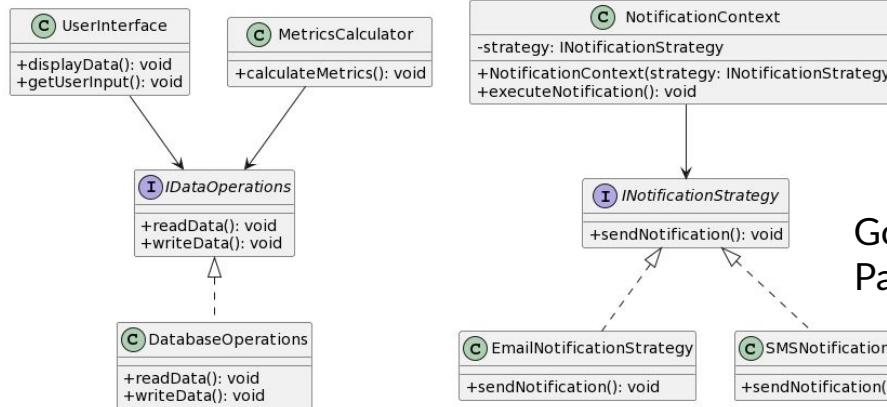
Better Designs



GRASP: Indirect



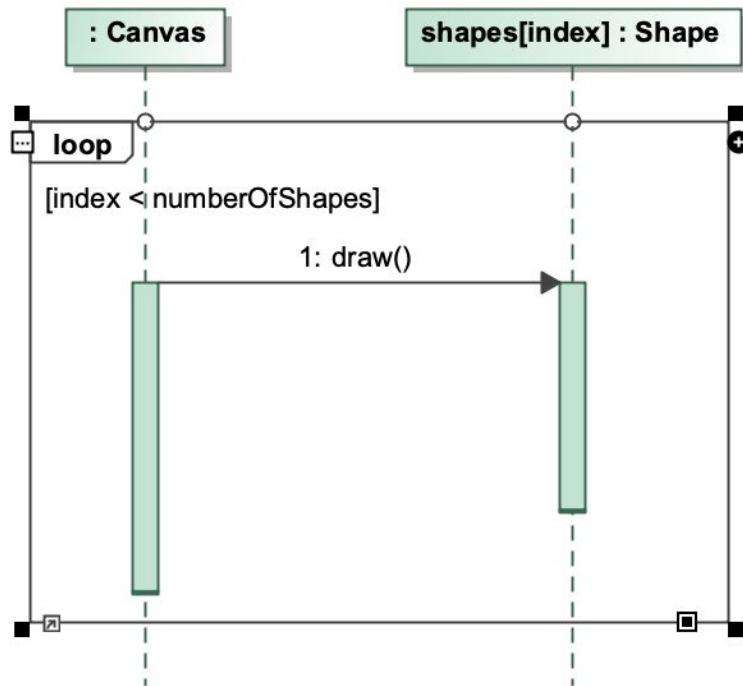
MVC



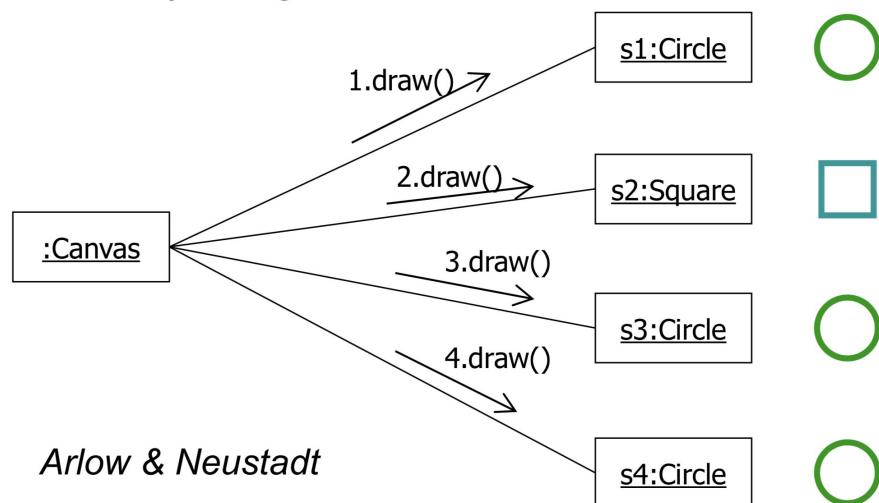
GoF: Strategy Pattern

Design to code: Why polymorphism helps?

Ex.: Iteration over a list of shapes



Objektdiagram:



Arlow & Neustadt

Today's program

- State diagrams
- Software quality
 - Understand quality
 - Methodologies of ensuring good quality
- Review technique

State Diagrams

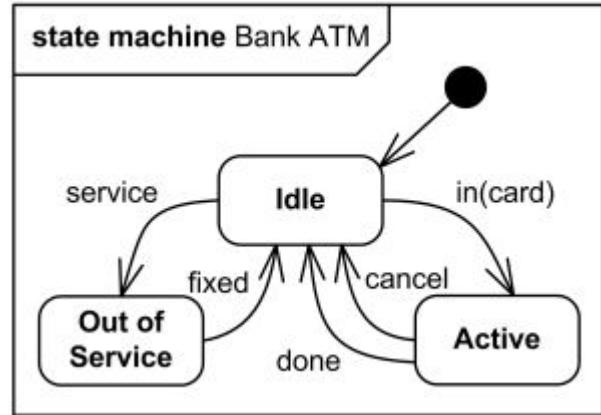


State diagram

Goal: Model the behavior of systems that respond to events by changing from one state to another.

Why important?

- Complexity management: Ease complex logic and operations
- Communications
- Parallel development: Compatibility between components.



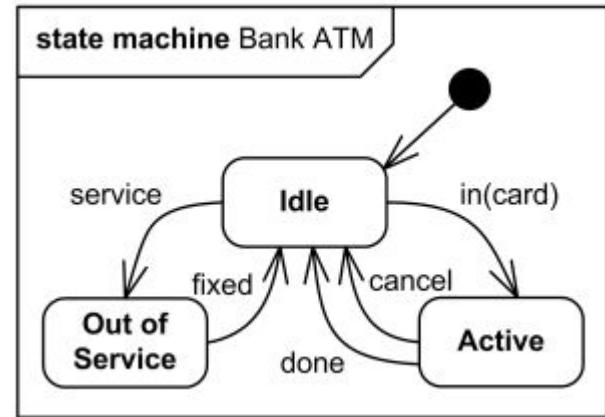
State diagram

States: Define what these states represent in the context of an ATM. For instance:

- Idle: The ATM is not currently servicing a customer and is ready for a new operation.
- Active: The ATM is currently engaged in a transaction with a customer.
- Out of Service: The ATM is not operational and cannot accept any transactions.

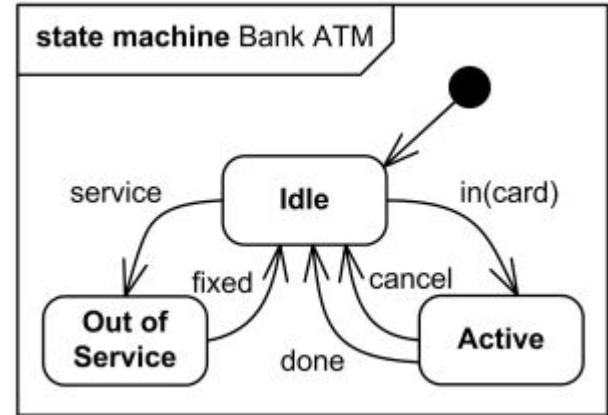
Transitions: Explain what causes the transitions between states:

- service: When the ATM starts up or finishes maintenance, it goes into the Idle state.
- in(card): When a card is inserted, the ATM transitions from Idle to Active.
- cancel: If a cancel request is made, or the session ends, the ATM returns to Idle.
- fixed/done: Once servicing is complete, the ATM moves from Out of Service to Idle.



Object state

- An object can be anything
 - User
 - Car
 - Microwave oven
 - Application
 - e.g. Navigation, Login mode
 - Order
 - Delivery
 - Examination
- Whitebox: The condition is made up of
 - The states of the object's attributes
- Textual:
 - File open
 - Logged in \Leftrightarrow Logged out



Object state

- An object changes state if its attributes change state
 - Requires an event
 - UML:
 - method call
 - message
 - hours
 - trigger (condition)
- An object typically has a number of different states
- In a given state, an object can
 - Perform an activity
 - Wait
 - Fulfill a condition



State

Object state

- Conditional on attributes
- Login ok
 - Conditional on "logintoken != null"
- Name found
 - name != null
- Package received
 - package != null

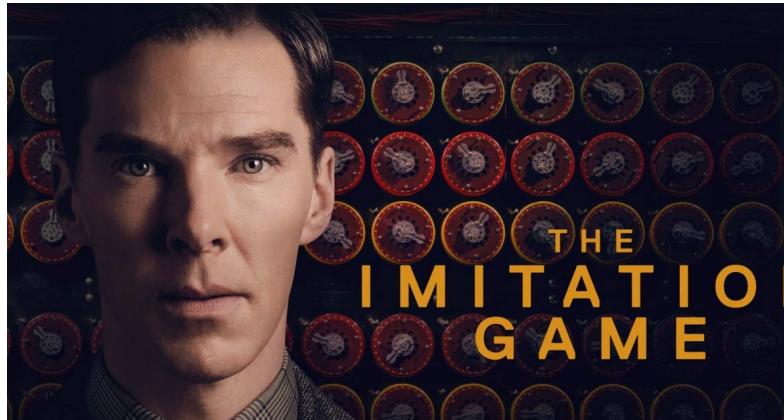
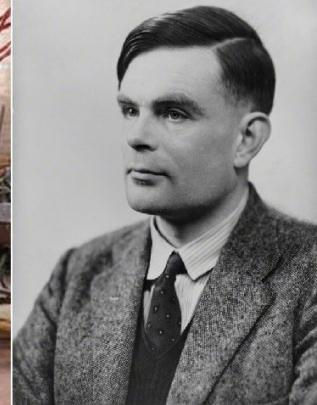
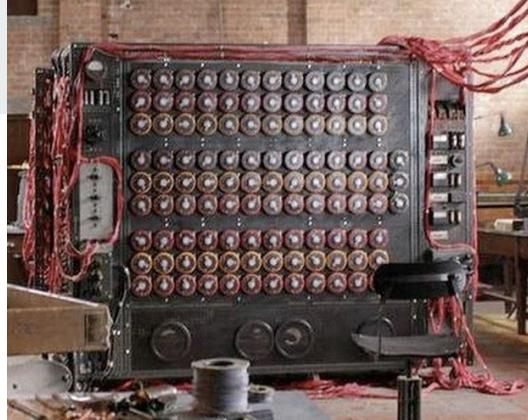
Name found

Package received

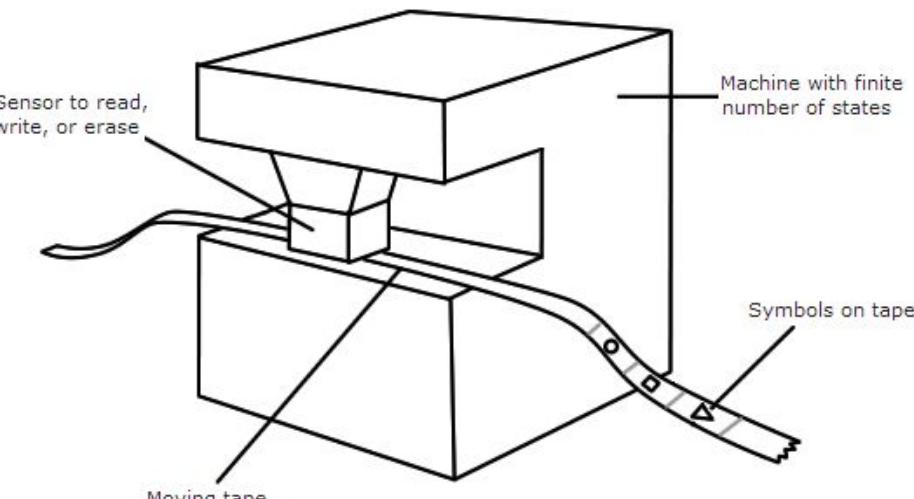
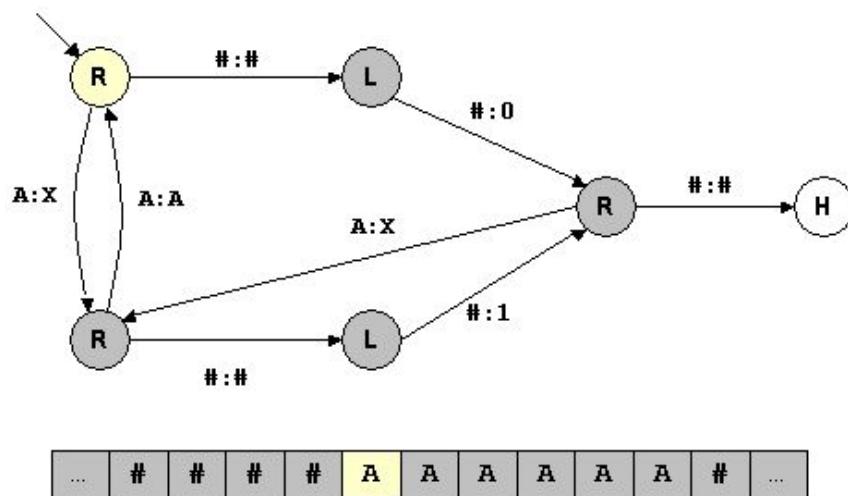
Login OK

State machines

- Shows the states of an object during its lifetime
- Events that the object can respond to cause
 - Possible results
 - Possible transition
- Finite State Machine
 - Finite number of states and transitions
- Deterministic
 - All events result in one specific transition
- Non-deterministic
 - Some events may result in several different transitions
- Software:
 - Deterministic FSMs



Turing machine (A Deterministic FSM)

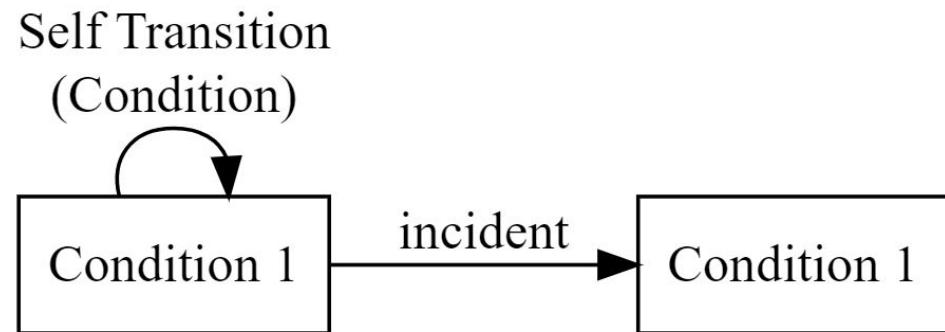


A Turing Machine

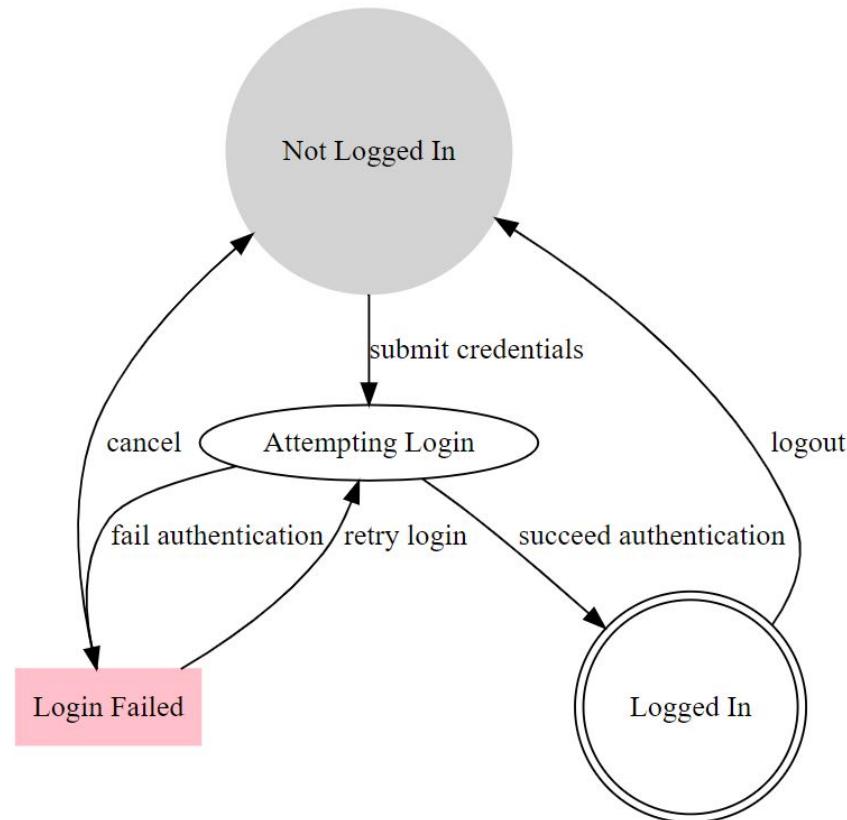
Change of state

An object can change from one state to another.

- Transition
 - Conditional on a trigger
 - Without trigger
- Possibly. Same condition



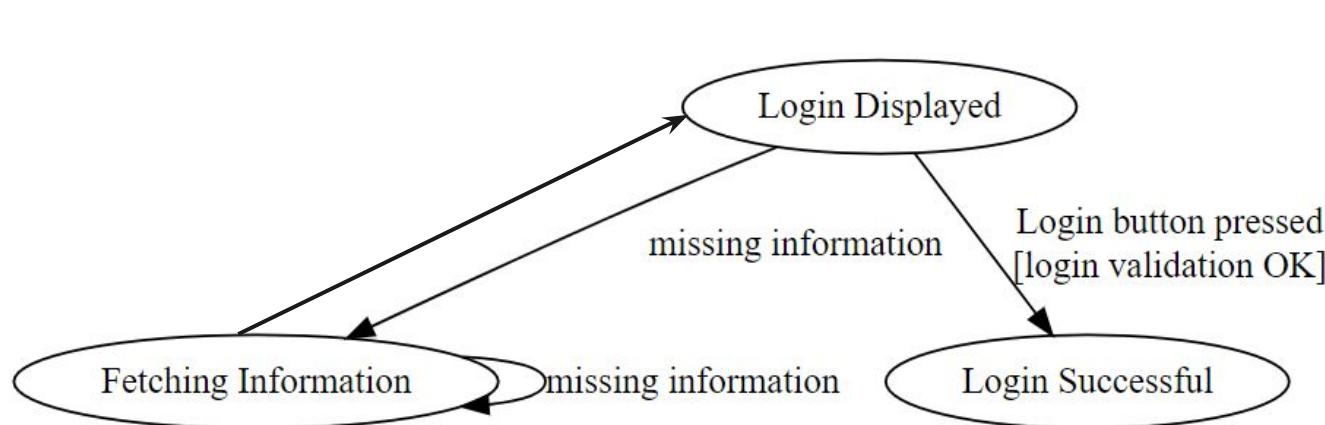
Example



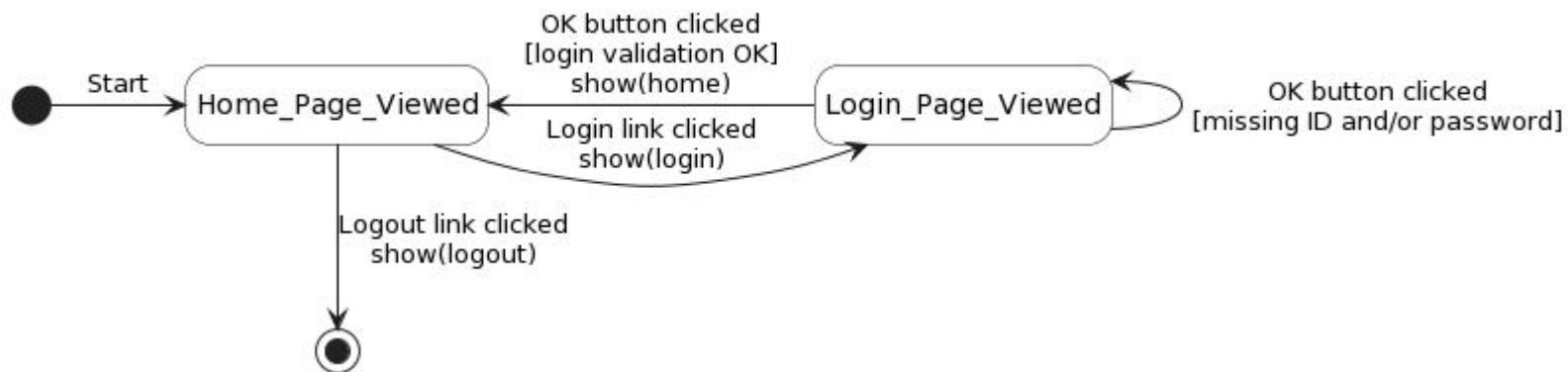
Transitions - Guard condition

Guard condition

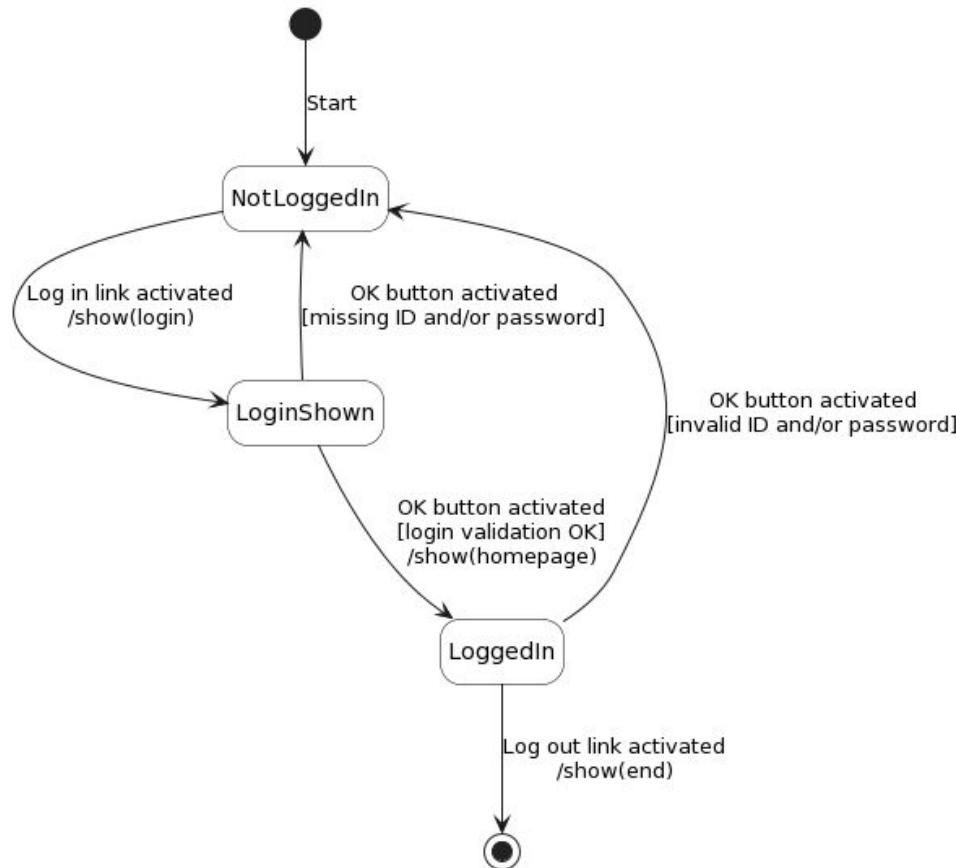
- Boolean expressions that must evaluate to true for the transition to occur.
- Event [Condition]
 - Login [Password correct]
- [Guard]



Start state and end state

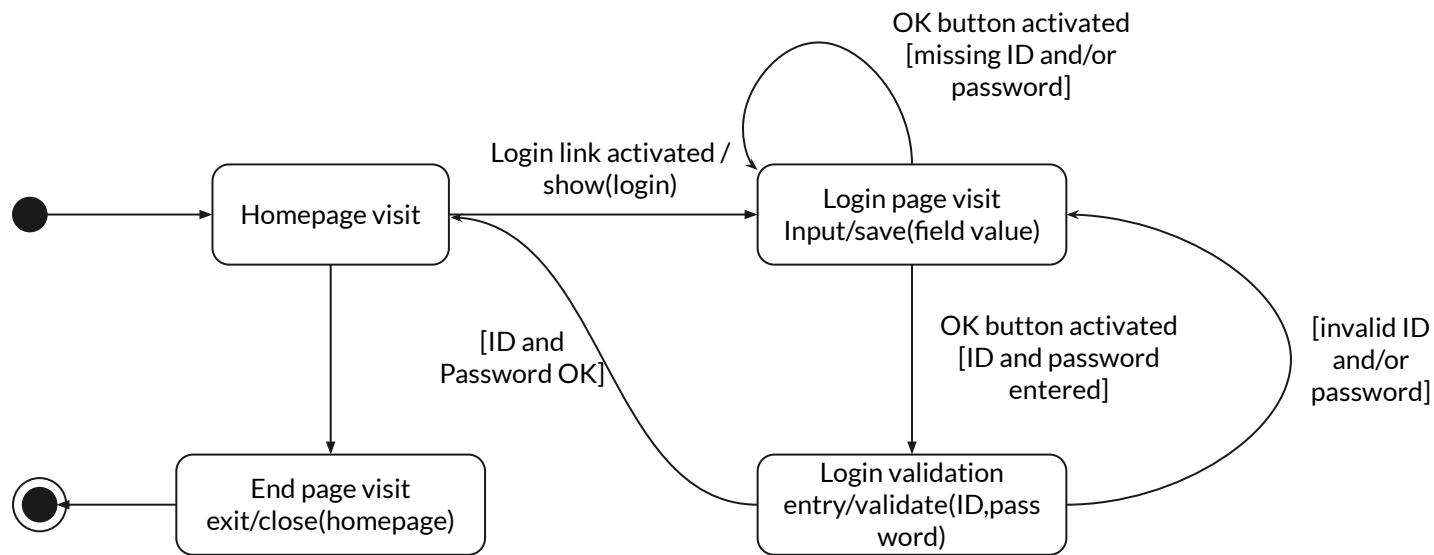


Start state and end state



Actions that do not change state

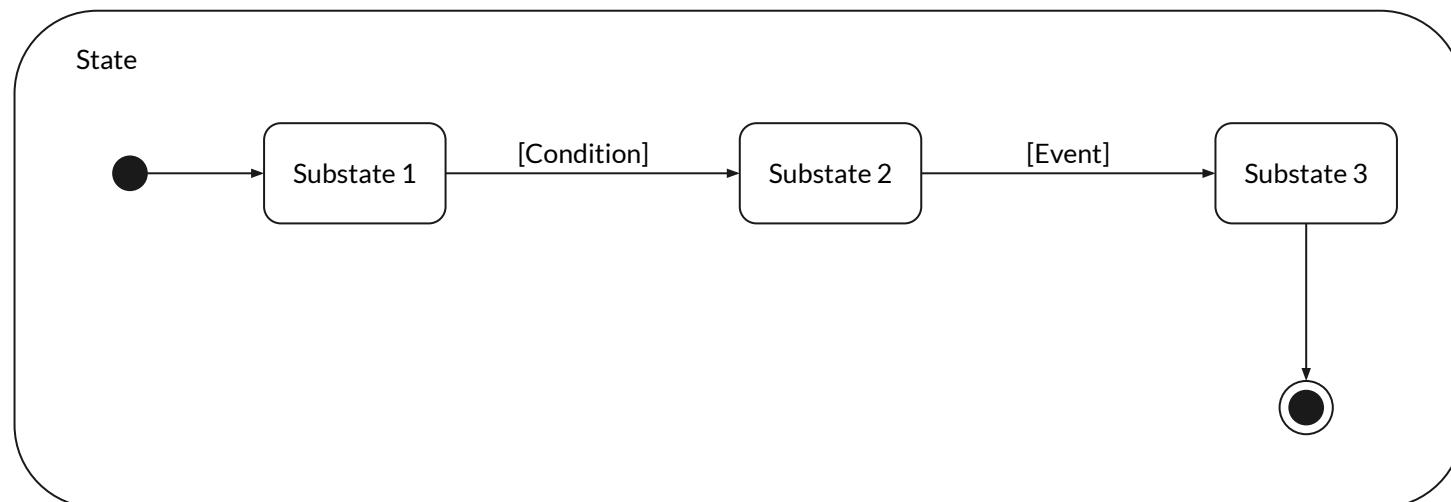
- Entry - Action at the start of a state change
- Exit - Action at the end of a state change
- Wait activity - Performs continuously during the state



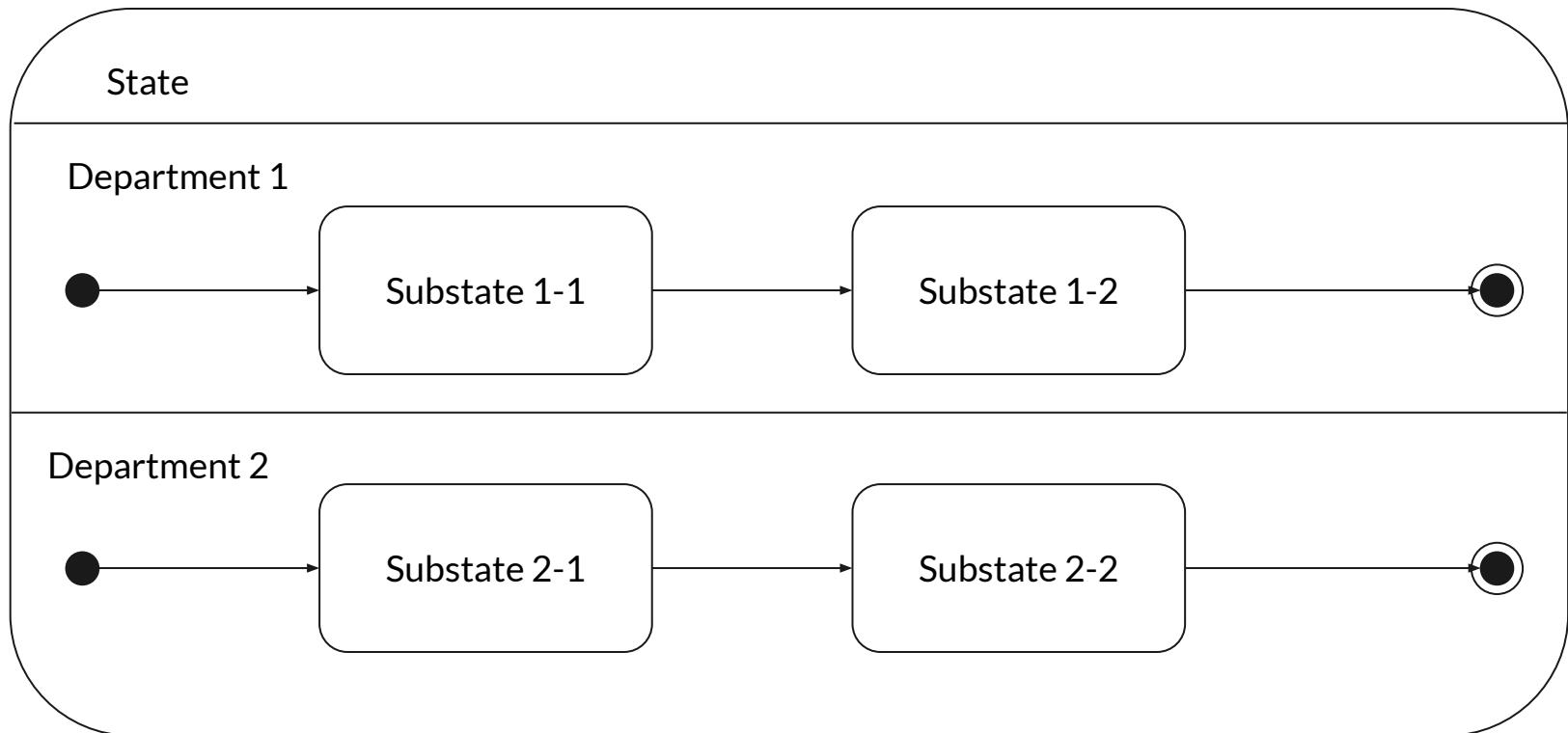
Composite modes

States with substates i

- Sequential
 - one substate at a time
- Simultaneous
 - Multiple simultaneous submodes



Concurrent substates



When State Diagrams?

- Analysis of the domain
 - Describe existing processes and protocols
- The design process
 - What modes can the software be in?
 - eg Ready, waiting for response, error
- Testing
 - Test of possible states

The “Therac-25 incident”

The Therac-25 was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in the 1980s.

Patients were given massive overdoses of radiation, due to failure of the **software-controlled** system.

Accidents with the Therac-25 were caused by a combination of software bugs, design flaws, and inadequate safety mechanisms.



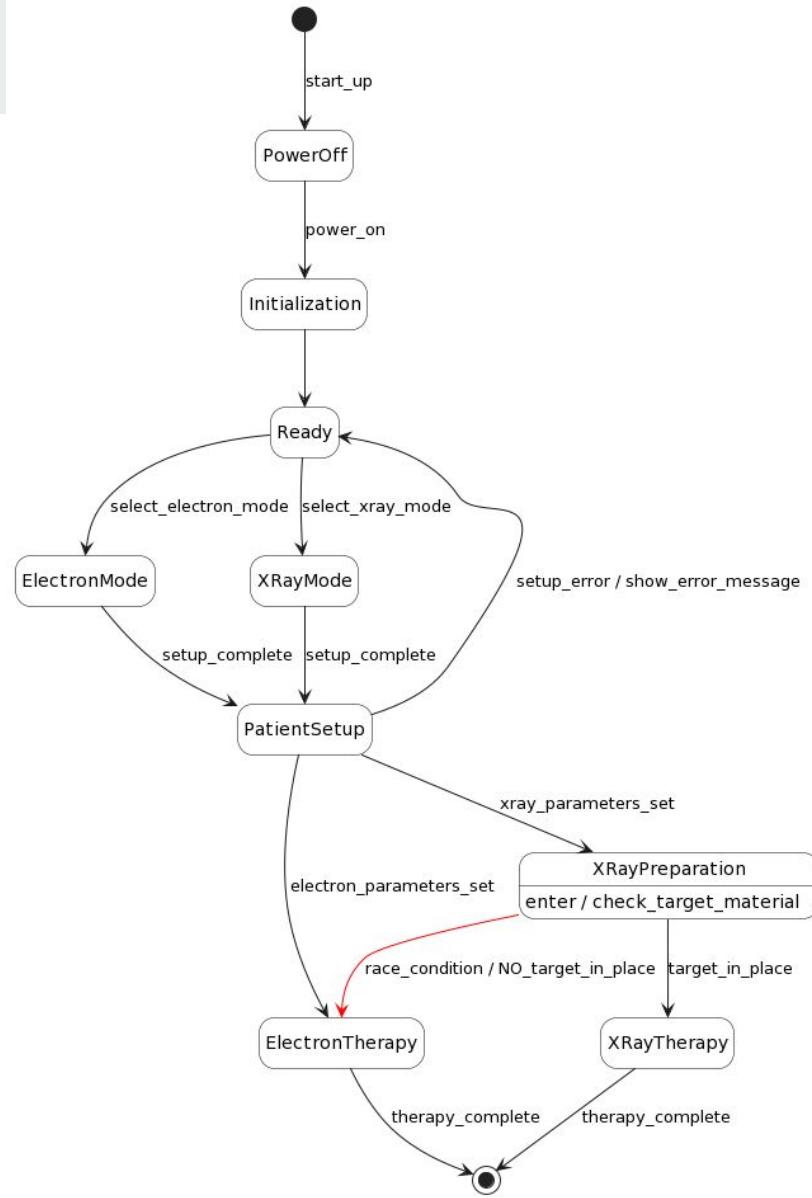
The “Therac-25 incident”

The red transition represents the critical bug.

Normally, the transition from XRayPreparation to XRayTherapy should only occur when the target material is confirmed to be in place.

A rapid sequence of “mode switching” could trigger the race condition, leading the Therac-25 to incorrectly indicate it was safe for low-power electron therapy when it was actually set for high-power X-ray mode without proper shielding, resulting in dangerous radiation overdoses.

A well-designed state diagram could have helped prevent the Therac-25 tragedy – all states and transitions were clearly defined and that no unsafe transitions were possible.



The “Corrupted Blood” Incident





The “Corrupted Blood Incident”

"Corrupted Blood" was a virtual plague that affected players

Only affect characters within a specific dungeon called
Zul'Gurub

What happened?

The plague was highly contagious and would drain the health of players, potentially causing their characters' death if not cured in time.

The plague can be carried out of dungeon by pets or minions that players could summon and dismiss at will.

The virtual world experienced an epidemic as the disease quickly spread unchecked through densely populated areas, leading to virtual chaos and countless character deaths.

Hakkar, generates plague



Player, cannot carry plague out



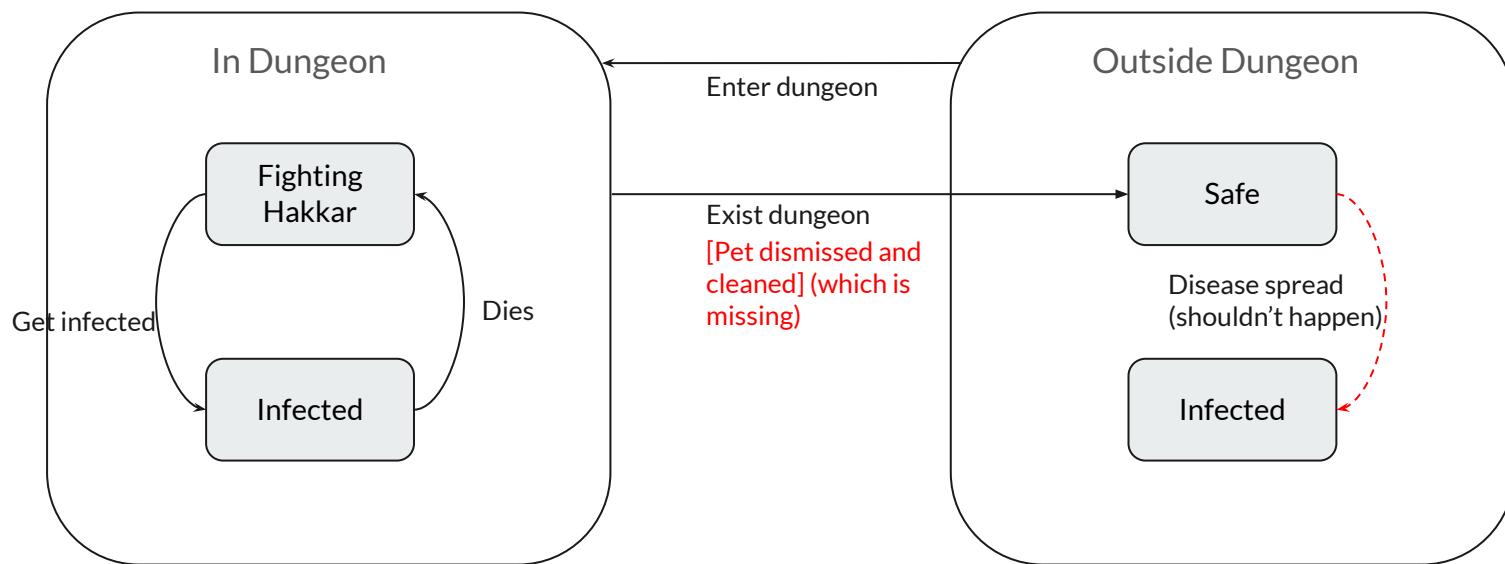
Minion, carries plague out



The “Corrupted Blood Incident”

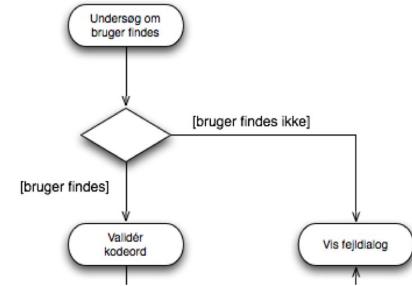
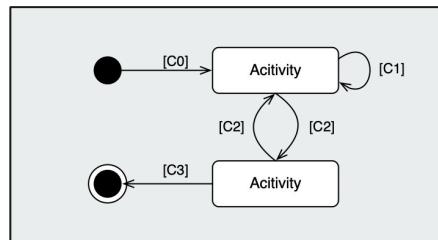
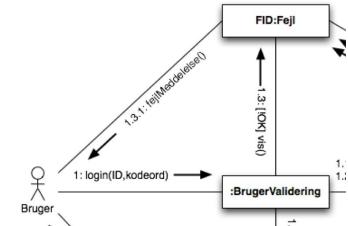
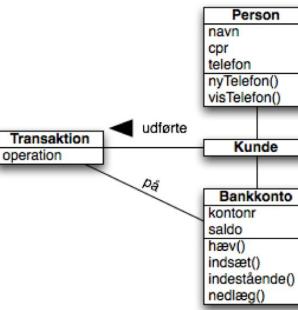
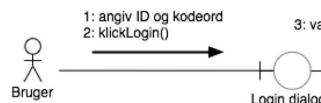
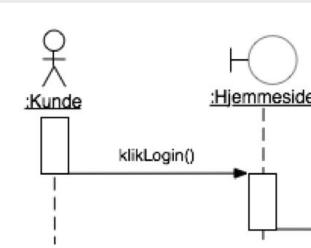
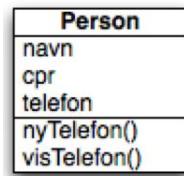
Lack of State Diagram and Consequences:

In the design of the Corrupted Blood ability, it appears that the designers did not fully anticipate the state in which pets could carry the disease out of the dungeon and did not consider the transition between the dungeon state and the outside world state.



Design elements

- Sequence diagram
 - Messages between objects
- (Collaboration/Communication Diagram)
 - Cooperation between objects
- Class diagram
 - Structure of classes
- Activity chart
 - Algorithms
- State diagram
 - Transitions between states



Software Quality

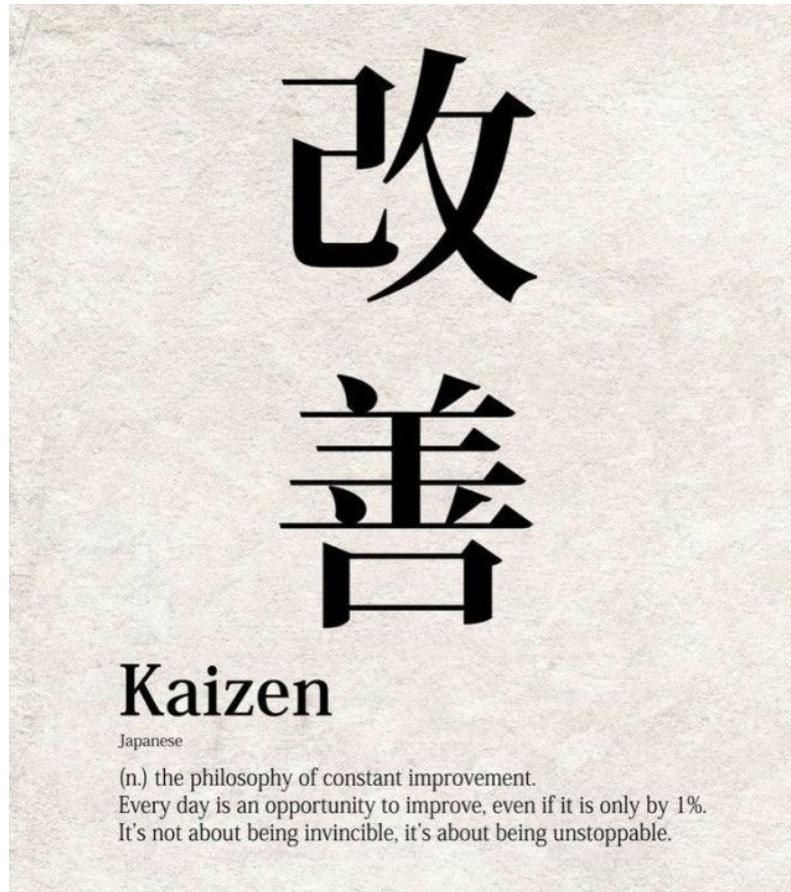


What is quality?



How to achieve quality

- Audit of potential and current suppliers
 - Supplier qualifications
- Development of specification
 - Measurable requirements
- Test
- Inspection
- Continuous Improvement
 - Kaizen
 - Continuous improvement of the value chain

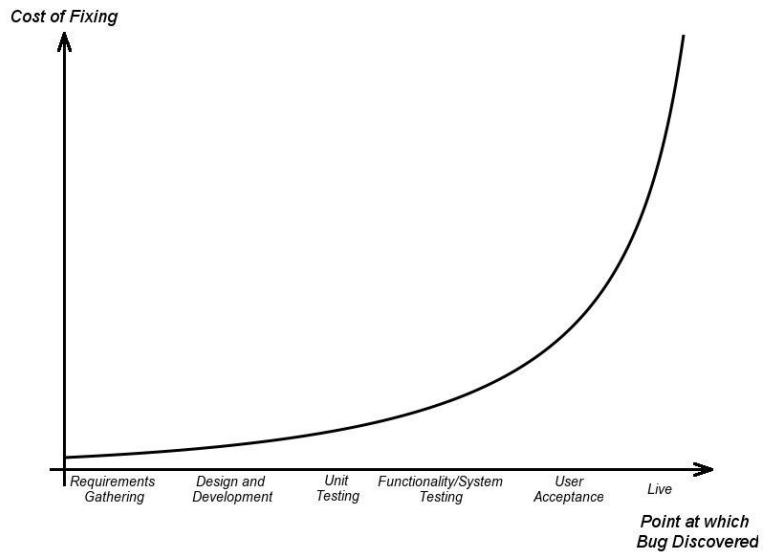
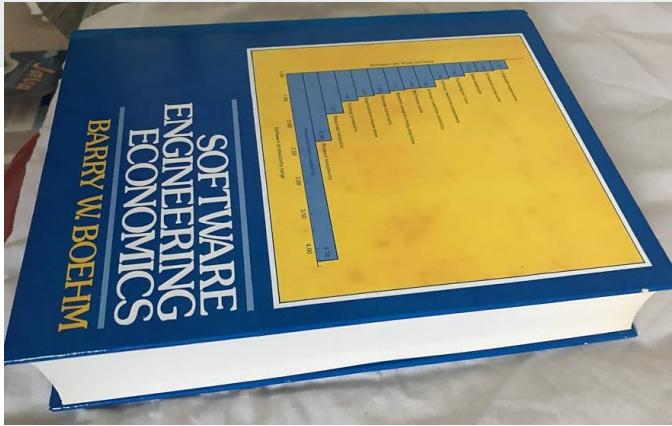


Software quality

- Meets Requirements
 - (If the requirements are good)
 - Utilities
- Usable
 - Usability
- Well structured
 - Easy to maintain
- Meets standards
 - Programming, Process, Documentation

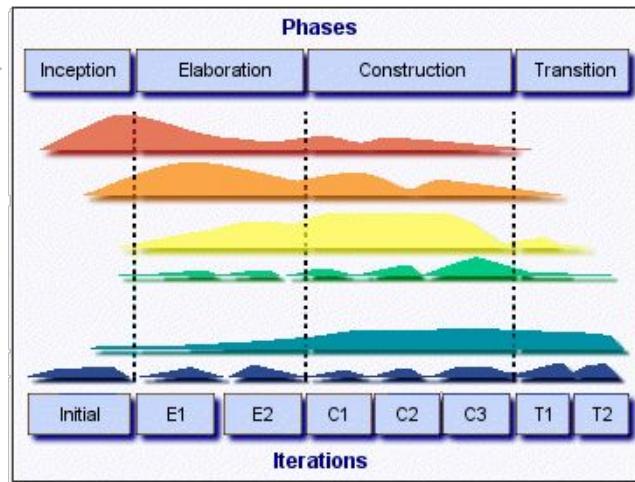
Test

"1 hour of analysis can save 200 hours of code"

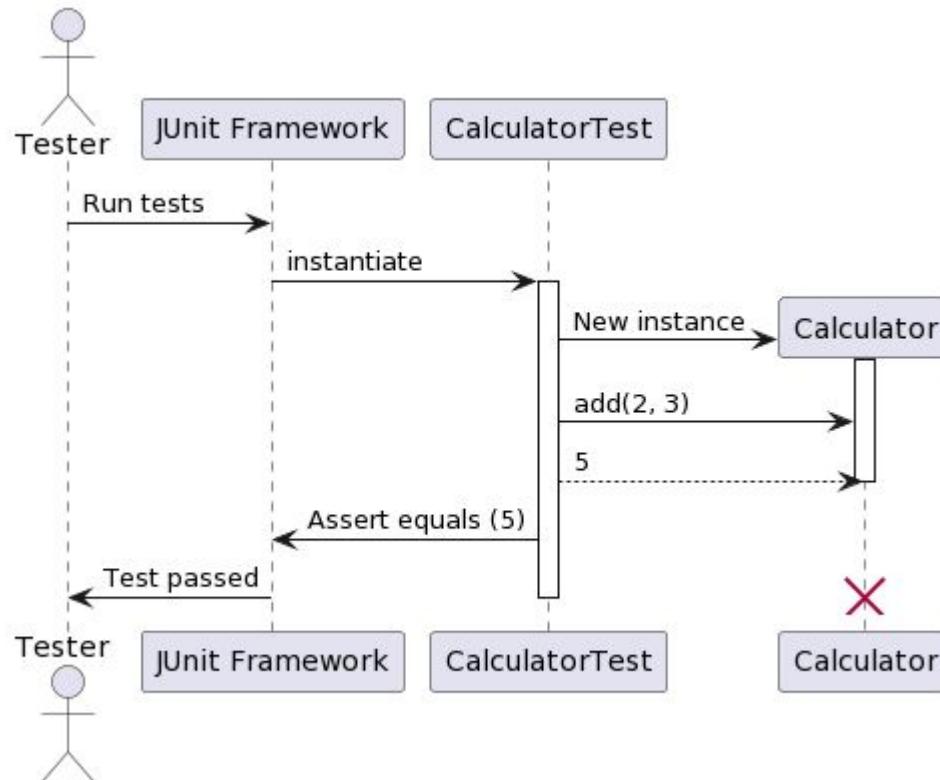
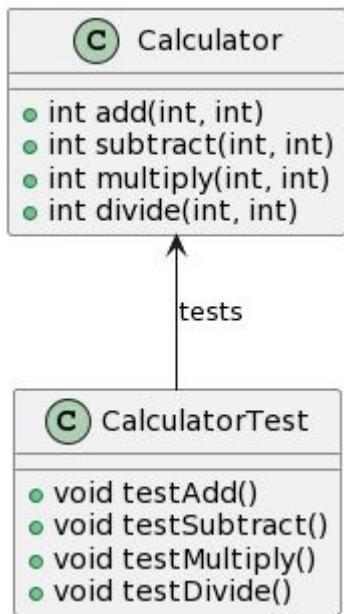


Disciplines

- Requirements
- Analysis & Design
- Implementation
- Test
- Configuration & Change Management
- Project Management



Test (using JUnit)



Test (using JUnit)

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public int divide(int a, int b) {  
        if (b == 0) {  
            throw new IllegalArgumentException("Divide cannot be 0.");  
        }  
        return a / b;  
    }  
}
```

```
import static org.junit.Assert.*;  
import org.junit.Before;  
import org.junit.Test;  
  
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setUp() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdd() {  
        assertEquals("10 + 5 must be 15", 15, calculator.add(10, 5));  
    }  
  
    @Test  
    public void testSubtract() {  
        assertEquals("10 - 5 must be 5", 5, calculator.subtract(10, 5));  
    }  
  
    @Test  
    public void testMultiply() {  
        assertEquals("10 * 5 must be 50", 50, calculator.multiply(10, 5));  
    }  
  
    @Test  
    public void testDivide() {  
        assertEquals("10 / 5 must be 2", 2, calculator.divide(10, 5));  
    }  
  
    @Test(expected = IllegalArgumentException.class)  
    public void testDivideByZero() {  
        calculator.divide(10, 0);  
    }  
}
```

Test

Can you write test for this class?

```
public class FactorialCalculator {

    public static long calculateFactorial(int number) {
        long result = 1;
        for (int i = 2; i <= number; i++) {
            result *= i;
        }
        return result;
    }
}
```

```
import static org.junit.Assert.*;
import org.junit.Test;

public class FactorialCalculatorTest {

    @Test
    public void whenCalculatingFactorialOfZero_thenOneShouldBeReturned() {
        assertEquals(1, FactorialCalculator.calculateFactorial(0));
    }

    @Test
    public void whenCalculatingFactorialOfOne_thenOneShouldBeReturned() {
        assertEquals(1, FactorialCalculator.calculateFactorial(1));
    }

    @Test
    public void whenCalculatingFactorialOfFive_thenOneHundredTwentyShouldBeReturned() {
        assertEquals(120, FactorialCalculator.calculateFactorial(5));
    }

    @Test
    public void whenCalculatingFactorialOfLargeNumber_thenCorrectResultShouldBeReturned() {
        assertEquals(2432902008176640000L, FactorialCalculator.calculateFactorial(20));
    }

    @Test(expected = IllegalArgumentException.class)
    public void whenCalculatingFactorialOfNegativeNumber_thenExceptionShouldBeThrown() {
        FactorialCalculator.calculateFactorial(-1);
    }

    // Optionally add more tests for numbers that may cause overflow to ensure that the method handles them correctly
    // if the method is supposed to deal with integer overflow issues
}
```

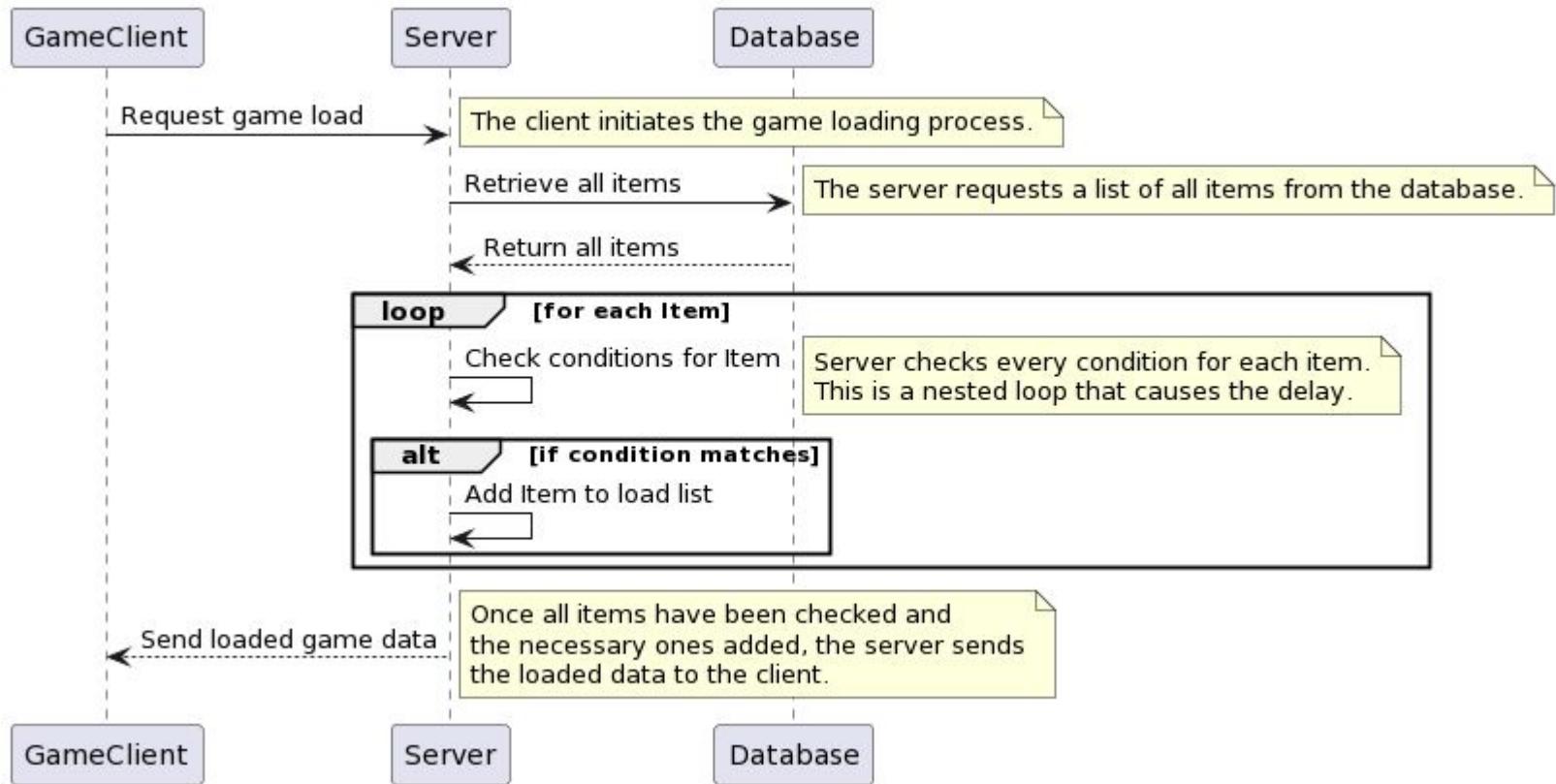
GTA V - loading screen bug



GTA V - loading screen bug



GTA V - loading screen bug



GTA V - loading screen bug

Pseudo code of what RockStar did

```
function loadGame() {
    items = getItemsFromDatabase()
    itemsToLoad = []

    for each item in items {
        for each condition in conditions {
            if (checkCondition(item, condition)) {
                itemsToLoad.add(item)
            }
        }
    }
    continueLoadingGame(itemsToLoad)
}
```

Length of items and conditions is 10

$$10 * 10 / 2 = 50$$

GTA V - loading screen bug

Pseudo code of what RockStar did

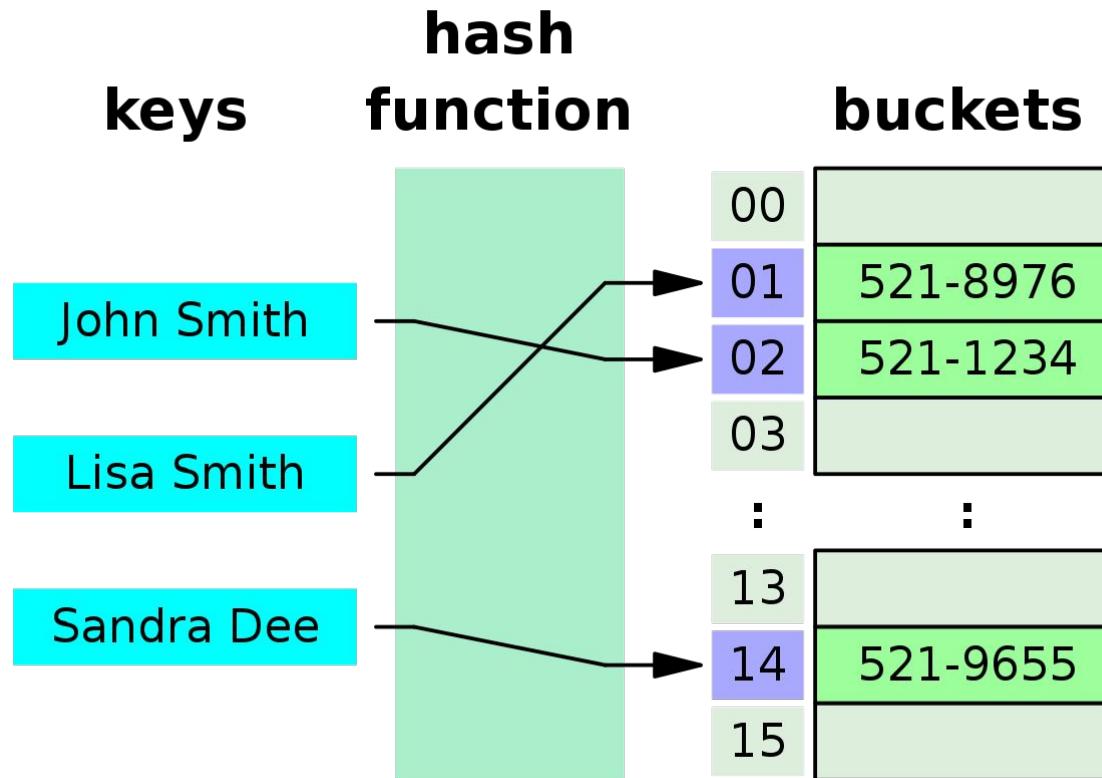
```
function loadGame() {
    items = getItemsFromDatabase()
    itemsToLoad = []

    for each item in items {
        for each condition in conditions {
            if (checkCondition(item, condition)) {
                itemsToLoad.add(item)
            }
        }
    }
    continueLoadingGame(itemsToLoad)
}
```

Length of items and conditions is 63000

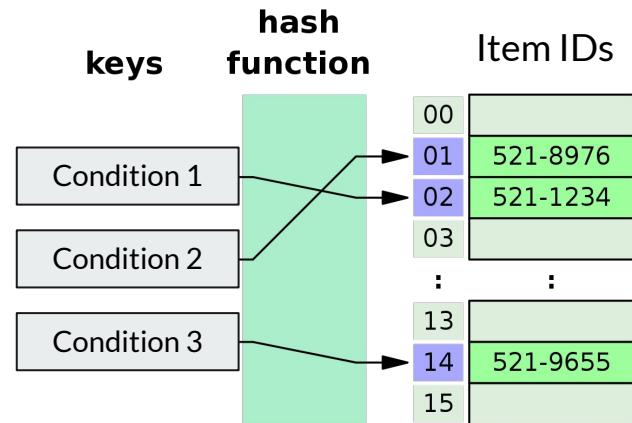
$63000 * 63000 / 2 \approx 1.98$ Billions

GTA V - loading screen bug



GTA V - loading screen bug

```
function loadGame() {  
    items = getItemsFromDatabase()  
    conditionsHashMap = createConditionsHashMap(conditions)  
    itemsToLoad = []  
  
    for each item in items {  
        if (conditionsMap.containsKey(item.conditionId)) {  
            itemsToLoad.add(item)  
        }  
    }  
    continueLoadingGame(itemsToLoad)  
}
```



Length of items and conditions is 63000

Executions: 63000

GTA V - unit test

```
public class GameLoaderTest {

    private GameLoader gameLoader;
    private List<Item> items;
    private List<Condition> conditions;

    @Before
    public void setUp() {
        gameLoader = new GameLoader();
        items = new ArrayList<>();
        conditions = new ArrayList<>();

        // Assuming we have 1000 items and 1000 conditions, with some overlap
        for (int i = 0; i < 1000; i++) {
            items.add(new Item(i));
            conditions.add(new Condition((int)(Math.random() * 1500)));
        }
    }

    @Test
    public void testLoadItemsWithHashMapIsFaster() {
        long startTime = System.nanoTime();
        List<Item> loadedItems = gameLoader.loadItems(items, conditions);
        long duration = System.nanoTime() - startTime;

        // Loading should be no more than 5 minutes (300 seconds)
        Assert.assertTrue(duration < 300);
    }
}
```

Verification and Validation

- Test, Inspection and Review
 - Part of verification and validation
- Validation
 - Checking that the product meets the need
- Verification
 - Checking that the product meets the requirements
- **Validation and Verification must match**
 - If requirements and needs do not match, you have a problem!

Verification and Validation

- Validation
 - Are we building the **right product**?
- Verification
 - Are we building the **product right**?

Use Case: A mobile app development company has been contracted to create "TrailGuide," an application that provides users with detailed maps, trail difficulty ratings, weather conditions, and points of interest for hiking trails across the country.

Can you give examples?

- What would the development team do for validation?
- What would the development team do for verification?

Verification and Validation

Validation Example:

- Conduct user interviews with avid hikers to understand what features they value most in a hiking app.
- Develop user profile based on the feedback to represent the needs of the target audience.
- Create a prototype and conduct usability testing with a group of hikers to see if the app meets their needs and is the right product.
- Gather feedback on the prototype to validate that the product's concept and design are aligned with user expectations and solve the right problem.

Verification and Validation

Verification Example:

- Perform code reviews and static analysis to ensure the application adheres to the specified architectural and design requirements.
- Implement continuous integration with automated tests to verify that each feature works as intended after any changes.
- Carry out performance testing to verify that the app meets the non-functional requirements, such as loading maps quickly even with limited internet connectivity.
- Conduct thorough testing, including unit testing, integration testing, and system testing, to ensure all parts of the app work together correctly and meet the technical specifications.

Coverage of needs

- Requirements
 - Domain model
 - Use cases
 - Requirement specification
- Pretotyping
- Prototyping
- Feedback and iterative development

Pretotyping

Would people be interested in it? Can we build it?

Will people use it as expected? Will it work as expected?

Will people continue to use it? How cheaply can we build it?

Will people pay for it? How fast can we make it?

Review Techniques

Standards for software review

IEEE Standard (1028-2008)

3.1 anomaly: Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences.

NOTE—Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.

3.2 audit: An independent examination of a software product, software process, or set of software processes performed by a third party to assess compliance with specifications, standards, contractual agreements, or other criteria.

3.3 inspection: A visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications.

NOTE—Inspections are peer examinations led by impartial facilitators who are trained in inspection techniques. Determination of remedial or investigative action for an anomaly is a mandatory element of a software inspection, although the solution should not be determined in the inspection meeting.

3.4 management review: A systematic evaluation of a software product or process performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches used to achieve fitness for purpose.

3.5 review: A process or meeting during which a software product, set of software products, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors or other interested parties for examination, comment or approval.

3.6 software product: (A) A complete set of computer programs, procedures, and associated documentation and data. (B) One or more of the individual items in (A).

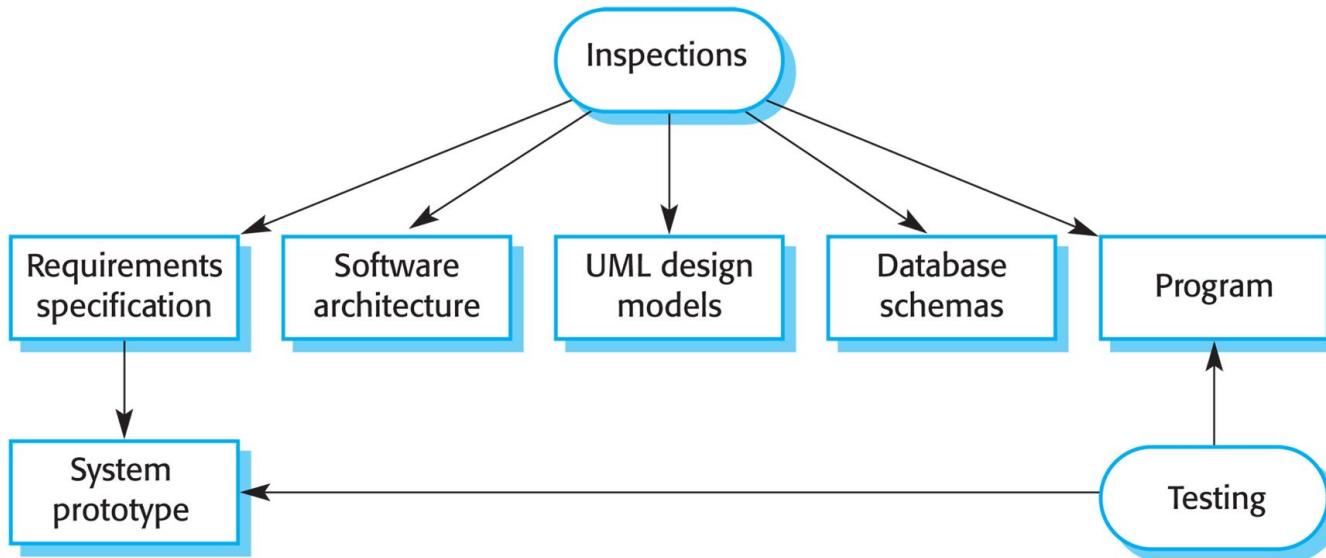
3.7 technical review: A systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.

NOTE—Technical reviews may also provide recommendations of alternatives and examination of various alternatives.

3.8 walk-through: A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems.

Reviews and Inspections

- Quality Assurance
 - Quality of Deliverables - Requirements, Software, documentation, etc.



Reviews and Inspections

- Informal
- Formal
 - Well-defined procedure
 - [1028-2008 - IEEE Standard for Software Reviews and Audits](#)

Technical review

A **systematic evaluation** of a **software product** by a team of **qualified personnel** that examines the **suitability of the software product** for its intended use and identifies **discrepancies** from specifications and standards.

- Systematic
- Qualified team
- Suitability for intended use
- Discrepancies

Walkthrough

A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems.

Table A.1—Comparison of review types

Characteristic	Management review	Technical review	Inspection	Walk-through	Audit
Objective	Monitor progress; set, confirm, or change objectives; change the allocation of resources	Evaluate conformance to specifications and plans; assess change integrity	Find anomalies; verify resolution; verify product quality	Find anomalies; examine alternatives; improve product; forum for learning	Independently evaluate conformance with objective standards and regulations
Decision-making	Management team charts course of action; decisions made at the meeting or as a result of recommendations	Review team requests management or technical leadership to act on recommendations	Review team chooses predefined product dispositions; anomalies should be removed	The team agrees on changes to be made by the author	Audited organization, initiator, acquirer, customer, or user
Change verification	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Responsibility of the audited organization
Recommended group size	Two or more people	Three or more people	Three to six people	Two to seven people	One to five people
Group attendance	Management, technical leadership, and documented attendance	Technical leadership and peer mix; documented attendance	Peers meet with documented attendance	Technical leadership and peer mix; documented attendance	Auditors; the audited organization may be called upon to provide evidence
Group leadership	Usually the responsible manager	Usually the lead engineer	Trained facilitator	Facilitator or author	Lead auditor

Review type



Review type

Table A.1—Comparison of review types (continued)

Characteristic	Management review	Technical review	Inspection	Walk-through	Audit
Data collection	As required by applicable policies, standards, or plans	Not a formal project requirement. May be done locally.	Required	Recommended	Not a formal project requirement. May be done locally.
Output	Management review documentation; including the specification of action items, with responsibilities and dates for resolution	Technical review documentation, including the specification of action items, with responsibilities and dates for resolution	Anomaly list, anomaly summary, inspection documentation	Anomaly list, action items, decisions, follow-up proposals	Formal audit report; observations, findings, deficiencies

Technical review Procedure

- Characteristics
 - Systematic
 - Qualified team
 - Suitability for intended use
 - Discrepancies/Deviations
- Products to review e.g.:
 - specifications
 - design description
 - test documentation
 - software development process descriptions
 - software architectural descriptions

Technical Review Procedure: Roles

- Decision maker
 - determines if objectives met
- Review leader
 - responsible for review
- Recorder
 - document anomalies, action items, decisions, and recommendations
- Technical reviewer
 - reviews and evaluates

Technical Review Procedure: Preparation

- Identify the review team
- Assign specific responsibilities to the review team members
- Schedule and announce the meeting place
- Distribute review materials to participants, allowing adequate time for their preparation
- Set a timetable for distribution of review material and return of comments

Technical Review Procedure: Examination

- Determine if the software product is:
 - complete
 - conforms to the regulations, standards, guidelines, plans, specifications, specifications, and procedures applicable to the project
 - suitable for its intended use
 - ready for the next activity
- Identify anomalies and decide their criticality
- Generate a list of action items, emphasizing risks
- Document the meeting

Technical Review Procedure: Output

- Review objectives and whether they were met
- A list of software product anomalies
- A list of unresolved system or hardware anomalies or specification action items
- Recommendations on how to dispose of unresolved issues and anomalies
- Whether the software product meets the applicable regulations, standards, guidelines, plans, specifications, and procedures without deviations

Quality management and agile development

- Based on good practice rather than formal documentation
 - check before check in
 - never break the build
 - fix problems when you see them

Where to put your Focus



Pareto Principle

- For many events, roughly 80% of the effects come from 20% of the causes
- Named after Italian economist Vilfredo Pareto
 - noticed that 80% of Italy's land was owned by 20% of the population
 - also noticed about 20% of the peapods in his garden contained 80% of the peas
- Microsoft noted that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in a given system would be eliminated.

- Other examples: e.g. 20% of code contains 80% of bugs
- (Last 20% takes 80% of the time)

Review: Requirements

Cohesive	The requirement addresses one and only one thing.
Complete	The requirement is fully stated in one place with no missing information.
Consistent	The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.
Correct	The requirement meets all or part of a business need as authoritatively stated by stakeholders.
Current	The requirement has not been made obsolete by the passage of time.
Externally Observable	The requirement specifies a characteristic of the product that is externally observable or experienced by the user.
Feasible	The requirement can be implemented within the constraints of the project.
Unambiguous	The requirement is concisely stated without recourse to technical jargon, acronyms or other esoteric verbiage. It is subject to one and only one interpretation.
Mandatory	The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated.
Verifiable	The implementation of the requirement can be determined through one of four possible methods: inspection, analysis, demonstration, or test.

Review: Analysis

- Domain models
 - meaningful?
 - sufficient?
 - errors?
 - inconsistencies?
 - syntax?

Review: Design

- Meaningful?
- Clear responsibilities?
- Cohesion and coupling?
- Sufficient?
- Errors?
- Inconsistencies?
- Syntax?

Development methods 62531

Lecture 11 - Review, Project management
Lei You, Assistant Professor

leiyo@dtu.dk

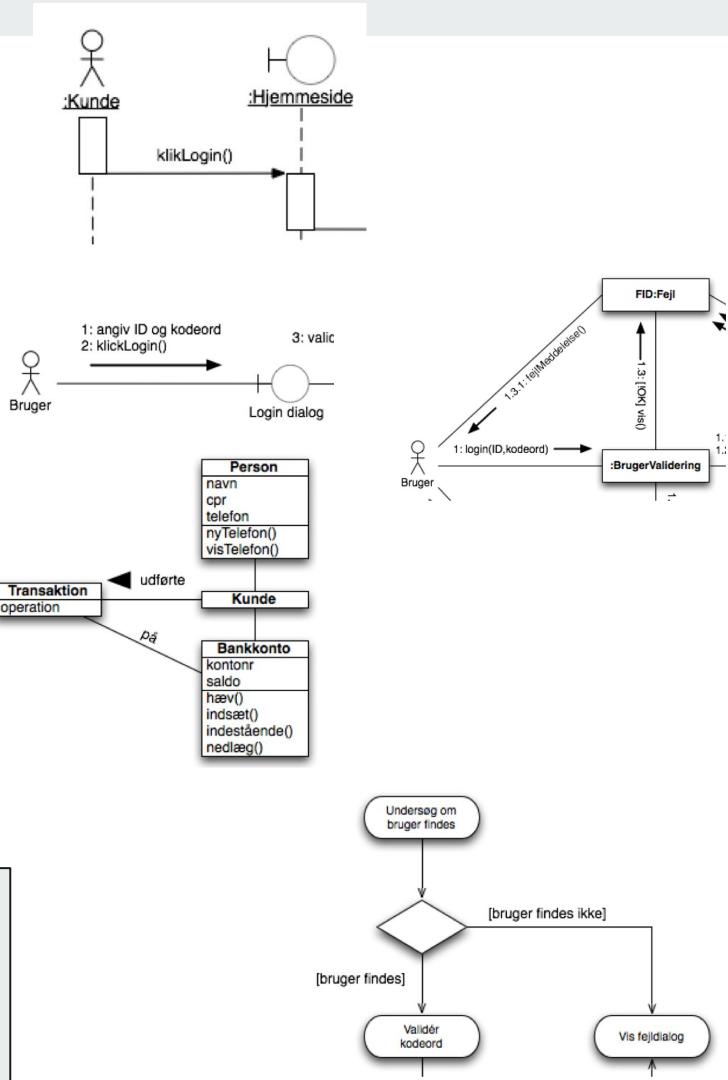
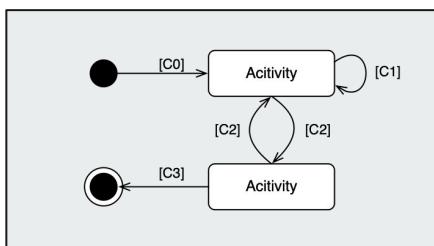
What have we learned?

- State Diagram: What? Why? Examples
- Software Quality: What?
 - Unit test: What? Why? How? Examples



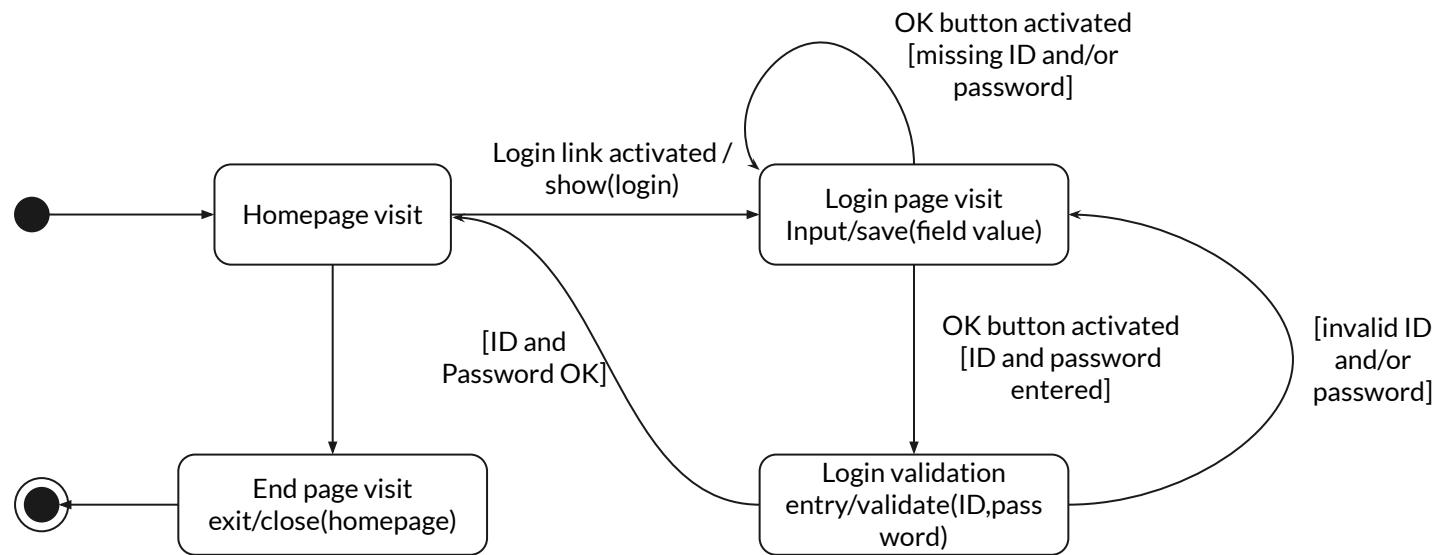
Design elements

- Sequence diagram
 - Messages between objects
- (Collaboration/Communication Diagram)
 - Cooperation between objects
- Class diagram
 - Structure of classes
- Activity chart
 - Algorithms
- State diagram
 - Transitions between states



State diagram

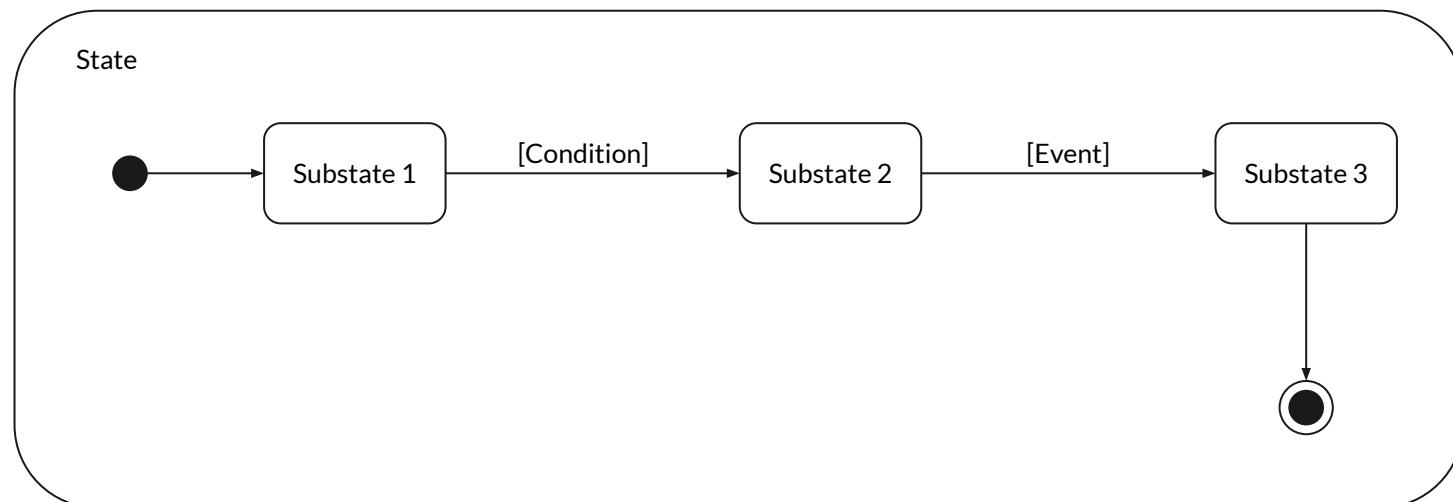
- Entry - Action at the start of a state change
- Exit - Action at the end of a state change
- Guard condition



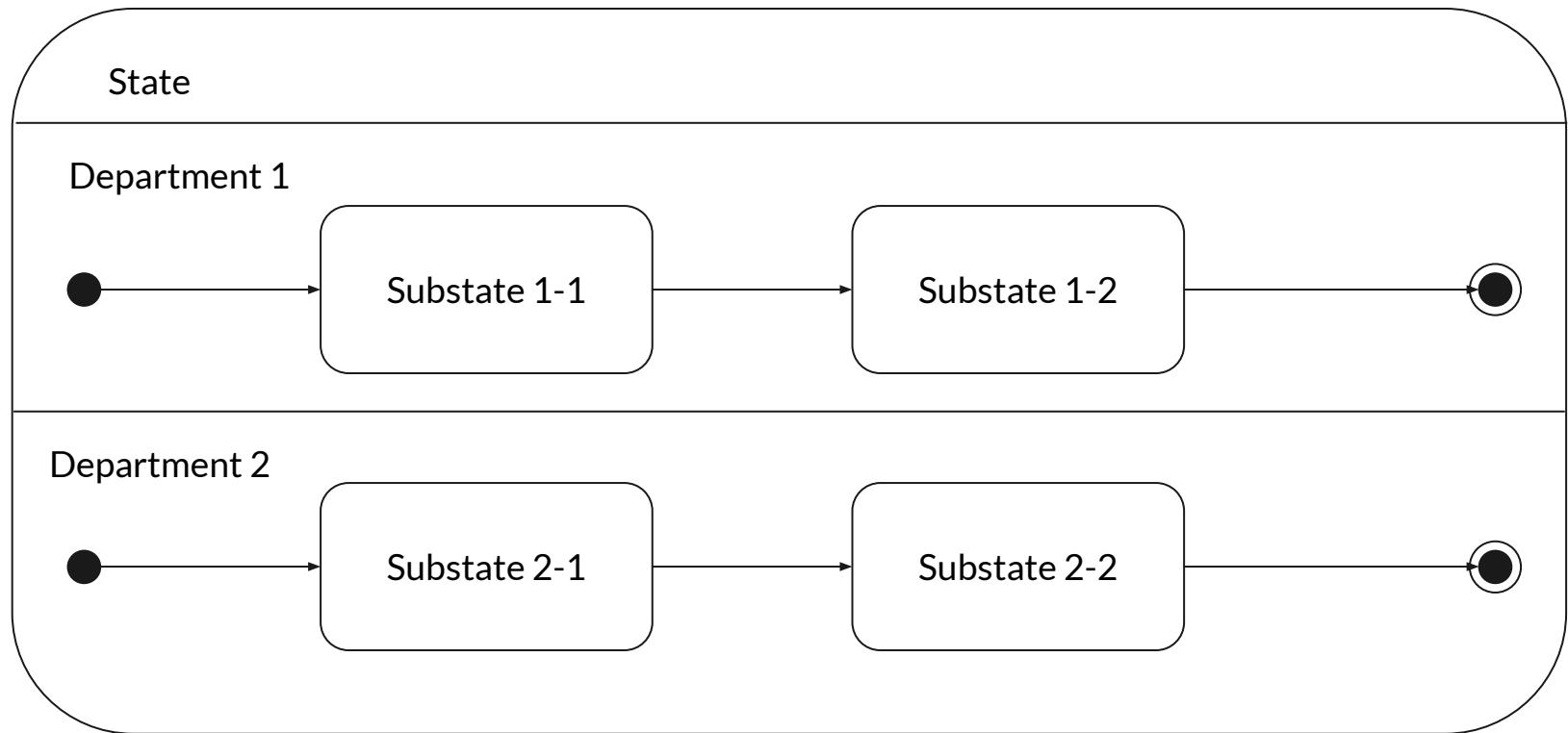
Composite modes

States with substates i

- Sequential
 - one substate at a time
- Simultaneous
 - Multiple simultaneous submodes



Concurrent substates



When State Diagrams?

- Analysis of the domain
 - Describe existing processes and protocols
- The design process
 - What modes can the software be in?
 - eg Ready, waiting for response, error
- Testing
 - Test of possible states

Examples:

- Therac-25 tragedy
- WoW Corrupted Blood

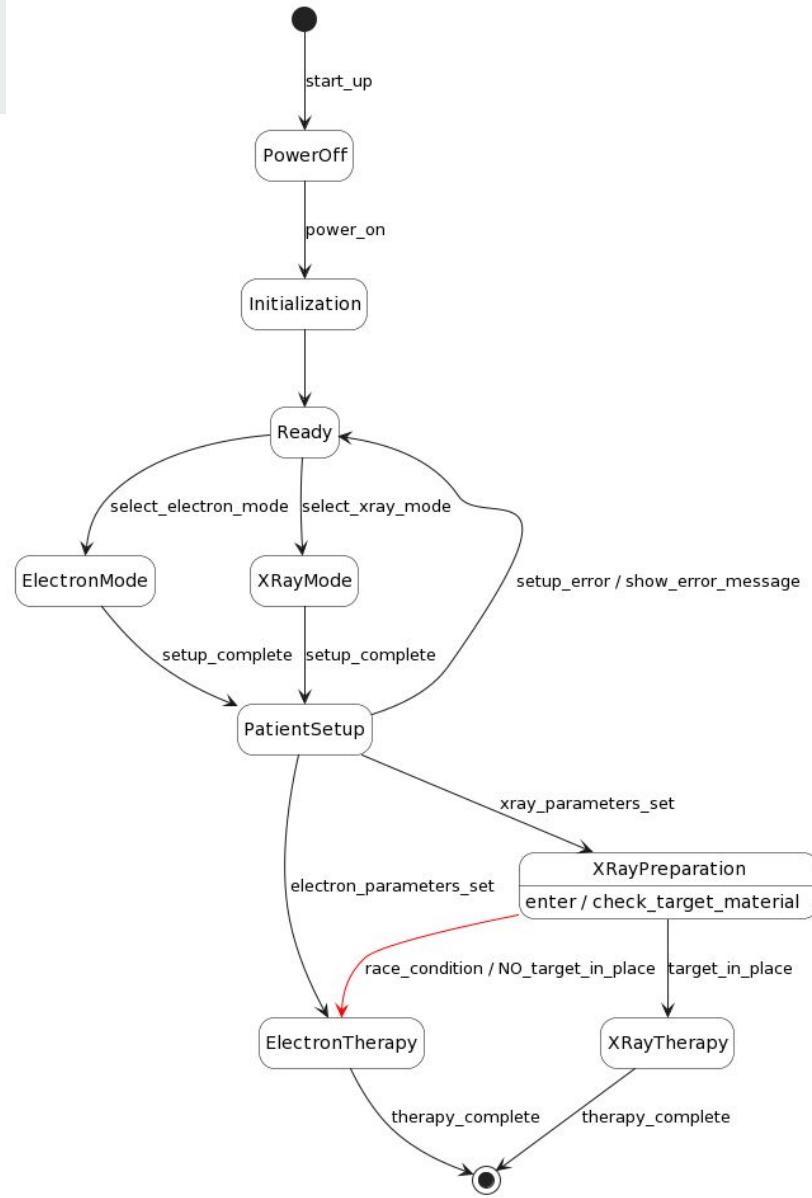
The “Therac-25 incident”

The red transition represents the critical bug.

Normally, the transition from XRayPreparation to XRayTherapy should only occur when the target material is confirmed to be in place.

A rapid sequence of “mode switching” could trigger the race condition, leading the Therac-25 to incorrectly indicate it was safe for low-power electron therapy when it was actually set for high-power X-ray mode without proper shielding, resulting in dangerous radiation overdoses.

A well-designed state diagram could have helped prevent the Therac-25 tragedy – all states and transitions were clearly defined and that no unsafe transitions were possible.

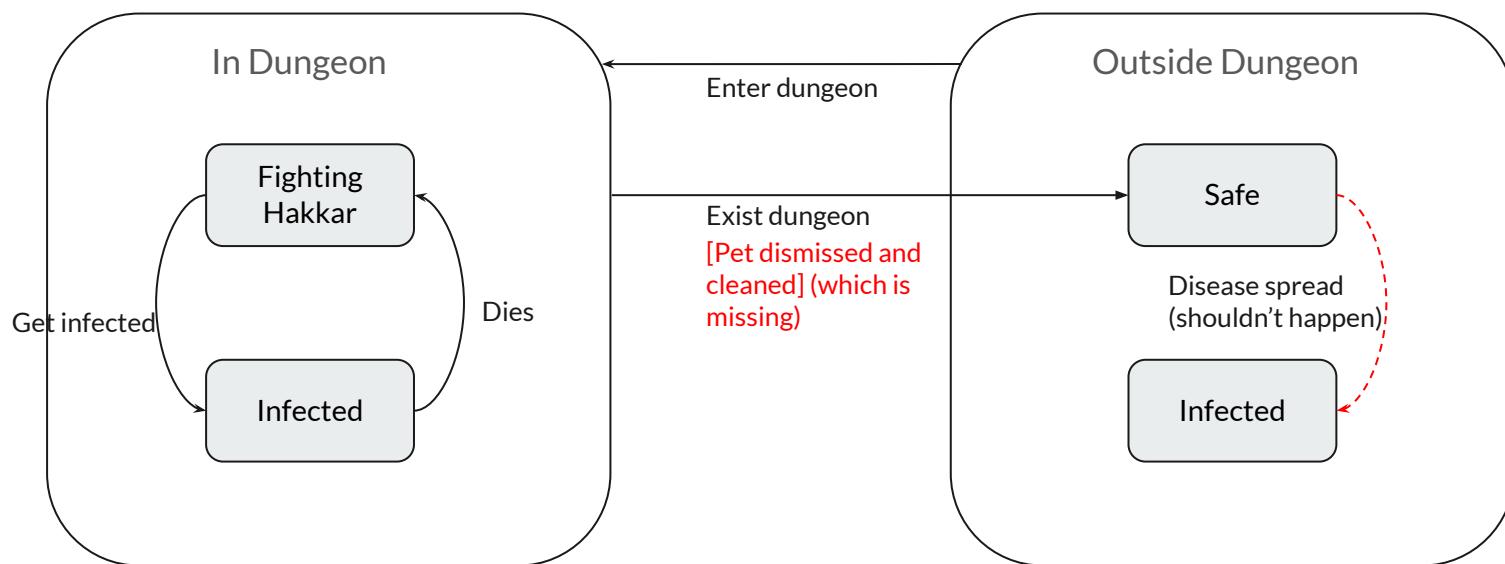




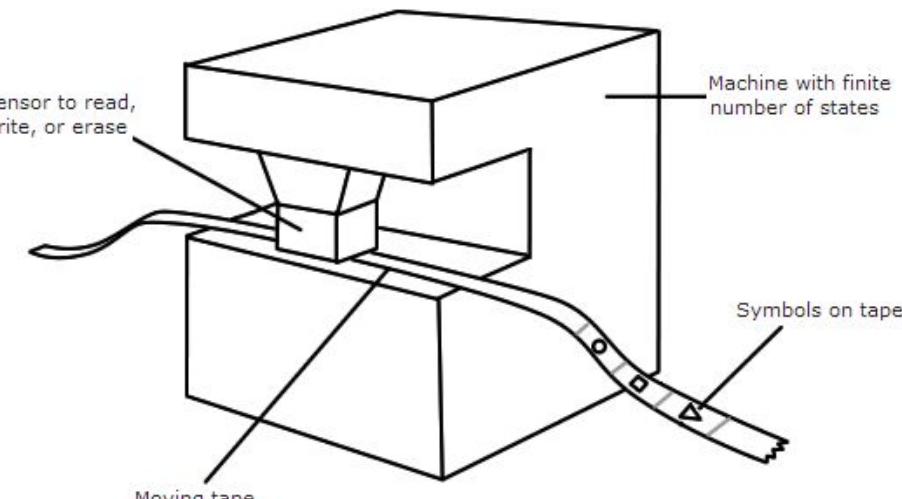
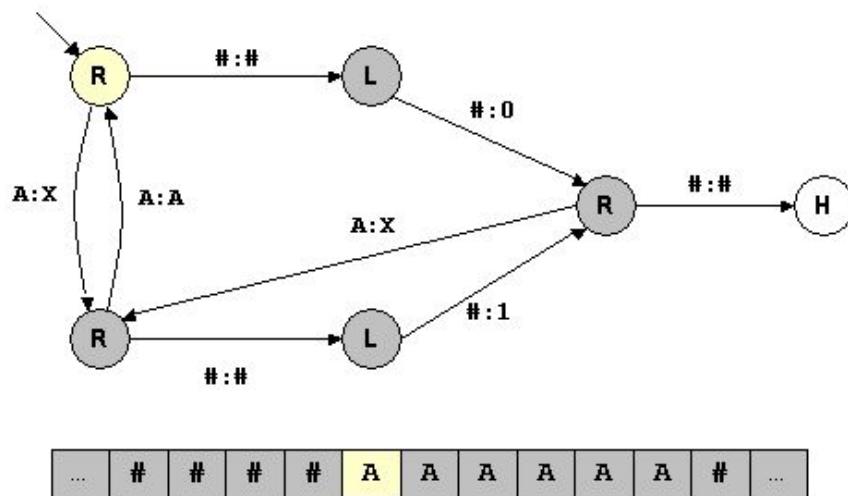
The “Corrupted Blood Incident”

Lack of State Diagram and Consequences:

In the design of the Corrupted Blood ability, it appears that the designers did not fully anticipate the state in which pets could carry the disease out of the dungeon and did not consider the transition between the dungeon state and the outside world state.



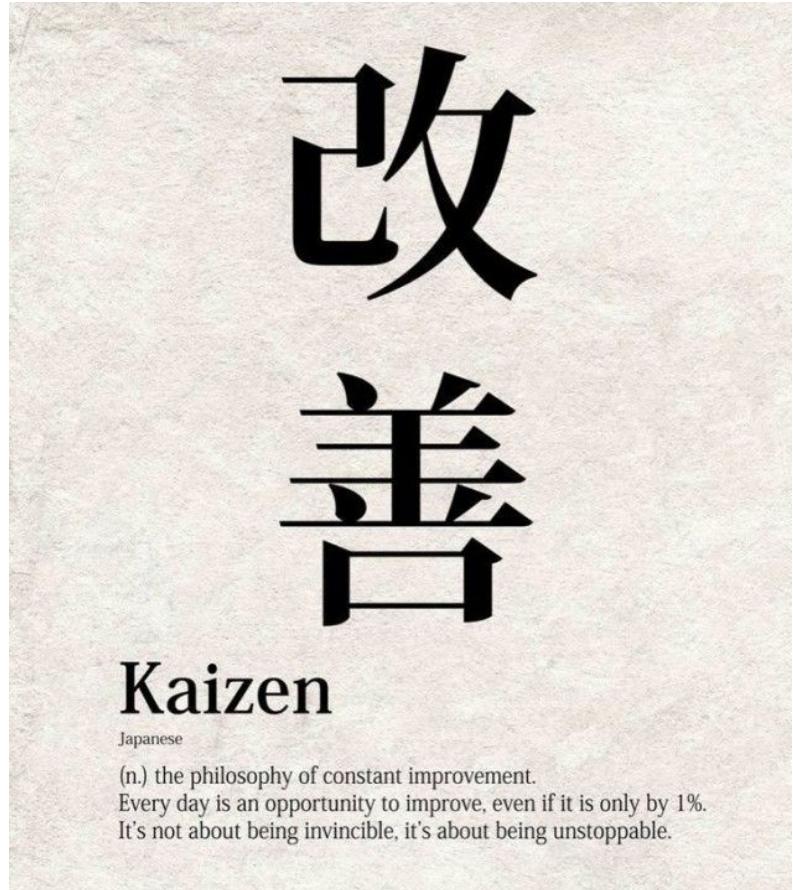
Turing machine (A Deterministic FSM)



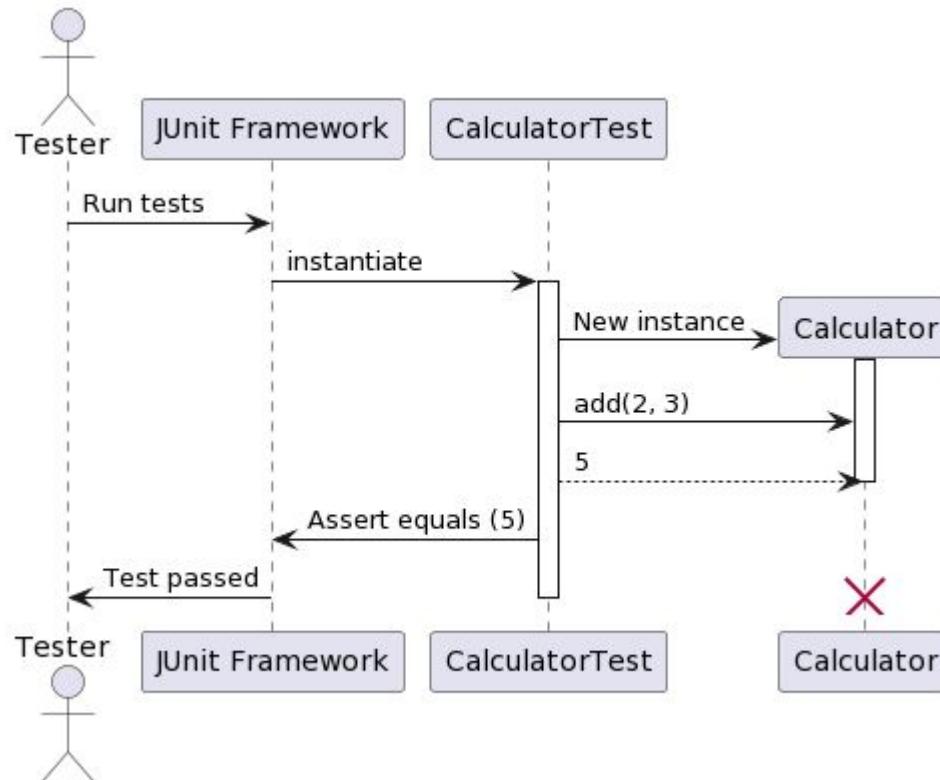
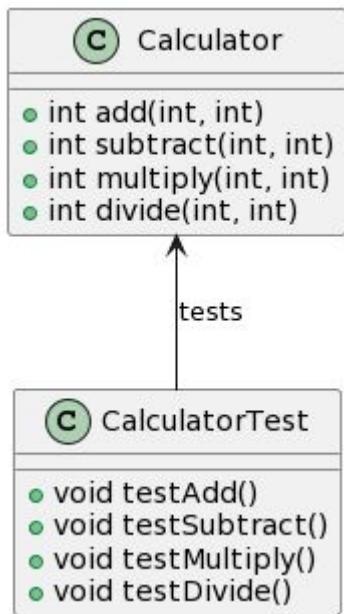
A Turing Machine

How to achieve quality

- Audit of potential and current suppliers
 - Supplier qualifications
- Development of specification
 - Measurable requirements
- Test
- Inspection
- Continuous Improvement
 - Kaizen
 - Continuous improvement of the value chain



Test (using JUnit)

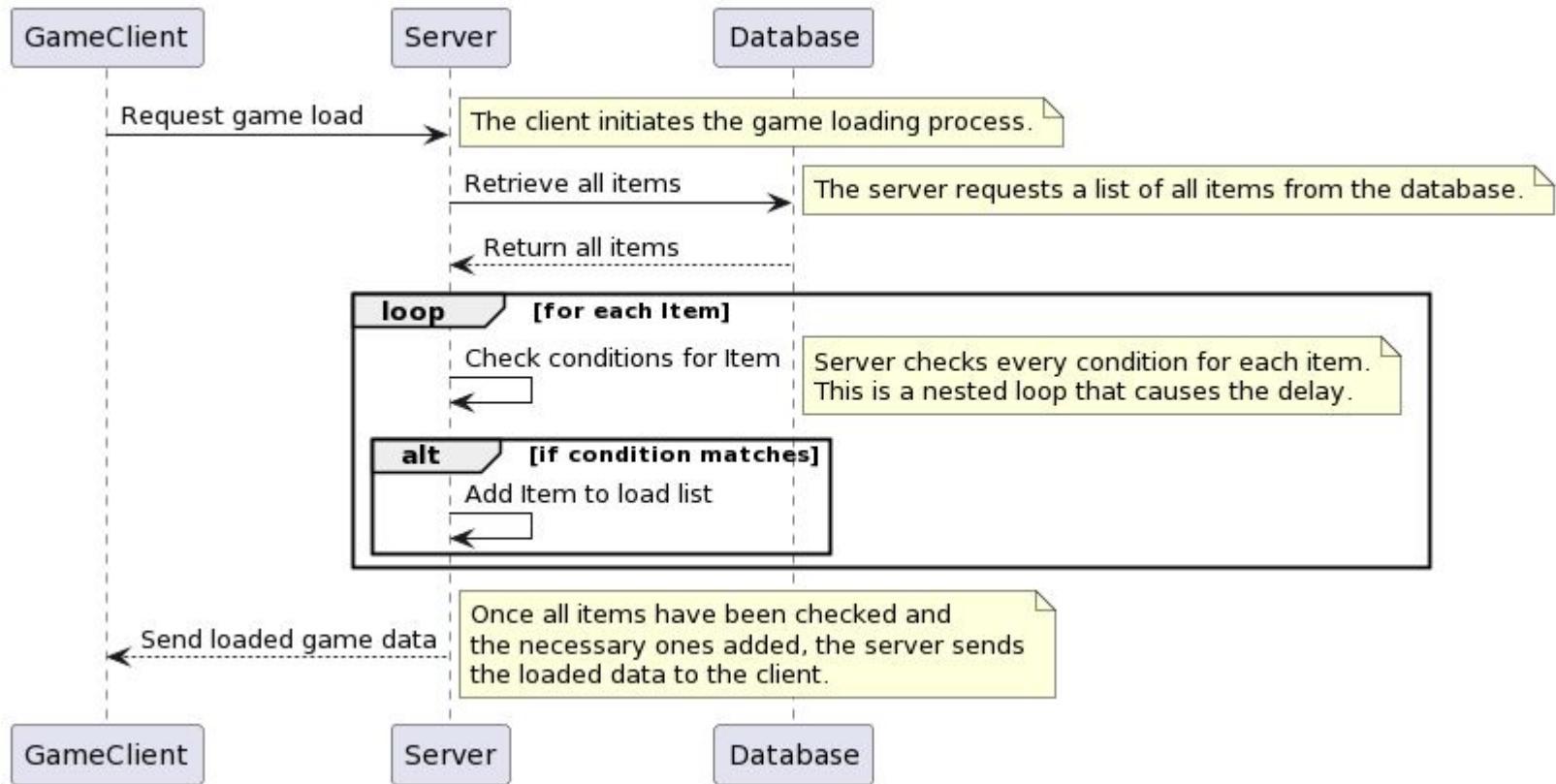


Test (using JUnit)

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    public int divide(int a, int b) {  
        if (b == 0) {  
            throw new IllegalArgumentException("Divide cannot be 0.");  
        }  
        return a / b;  
    }  
}
```

```
import static org.junit.Assert.*;  
import org.junit.Before;  
import org.junit.Test;  
  
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setUp() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdd() {  
        assertEquals("10 + 5 must be 15", 15, calculator.add(10, 5));  
    }  
  
    @Test  
    public void testSubtract() {  
        assertEquals("10 - 5 must be 5", 5, calculator.subtract(10, 5));  
    }  
  
    @Test  
    public void testMultiply() {  
        assertEquals("10 * 5 must be 50", 50, calculator.multiply(10, 5));  
    }  
  
    @Test  
    public void testDivide() {  
        assertEquals("10 / 5 must be 2", 2, calculator.divide(10, 5));  
    }  
  
    @Test(expected = IllegalArgumentException.class)  
    public void testDivideByZero() {  
        calculator.divide(10, 0);  
    }  
}
```

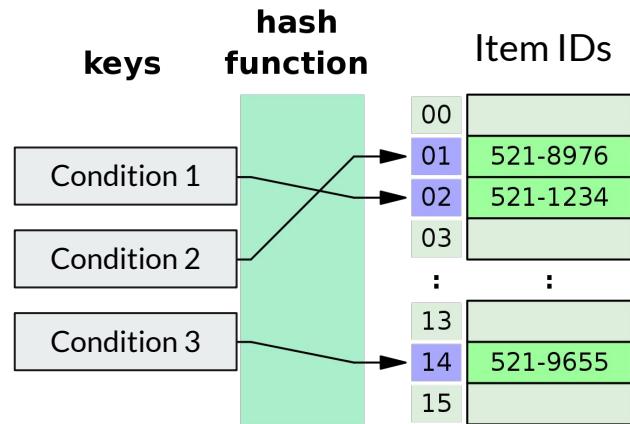
GTA V - loading screen bug



GTA V - loading screen bug

```
function loadGame() {
    items = getItemsFromDatabase()
    conditionsHashMap = createConditionsHashMap(conditions)
    itemsToLoad = []

    for each item in items {
        if (conditionsMap.containsKey(item.conditionId)) {
            itemsToLoad.add(item)
        }
    }
    continueLoadingGame(itemsToLoad)
}
```



Length of items and conditions is 63000

Executions: 63000

GTA V - unit test

```
public class GameLoaderTest {

    private GameLoader gameLoader;
    private List<Item> items;
    private List<Condition> conditions;

    @Before
    public void setUp() {
        gameLoader = new GameLoader();
        items = new ArrayList<>();
        conditions = new ArrayList<>();

        // Assuming we have 1000 items and 1000 conditions, with some overlap
        for (int i = 0; i < 1000; i++) {
            items.add(new Item(i));
            conditions.add(new Condition((int)(Math.random() * 1500)));
        }
    }

    @Test
    public void testLoadItemsWithHashMapIsFaster() {
        long startTime = System.nanoTime();
        List<Item> loadedItems = gameLoader.loadItems(items, conditions);
        long duration = System.nanoTime() - startTime;

        // Loading should be no more than 5 minutes (300 seconds)
        Assert.assertTrue(duration < 300);
    }
}
```

Today's program

- Review Techniques
- Project management

Review Techniques

Reviews and Inspections

- Informal
- Formal
 - Well-defined procedure
 - [1028-2008 - IEEE Standard for Software Reviews and Audits](#)

Standards for software review

IEEE Standard (1028-2008)

3.1 anomaly: Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences.

NOTE—Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.

3.2 audit: An independent examination of a software product, software process, or set of software processes performed by a third party to assess compliance with specifications, standards, contractual agreements, or other criteria.

3.3 inspection: A visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications.

NOTE—Inspections are peer examinations led by impartial facilitators who are trained in inspection techniques. Determination of remedial or investigative action for an anomaly is a mandatory element of a software inspection, although the solution should not be determined in the inspection meeting.

3.4 management review: A systematic evaluation of a software product or process performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches used to achieve fitness for purpose.

3.5 review: A process or meeting during which a software product, set of software products, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors or other interested parties for examination, comment or approval.

3.6 software product: (A) A complete set of computer programs, procedures, and associated documentation and data. (B) One or more of the individual items in (A).

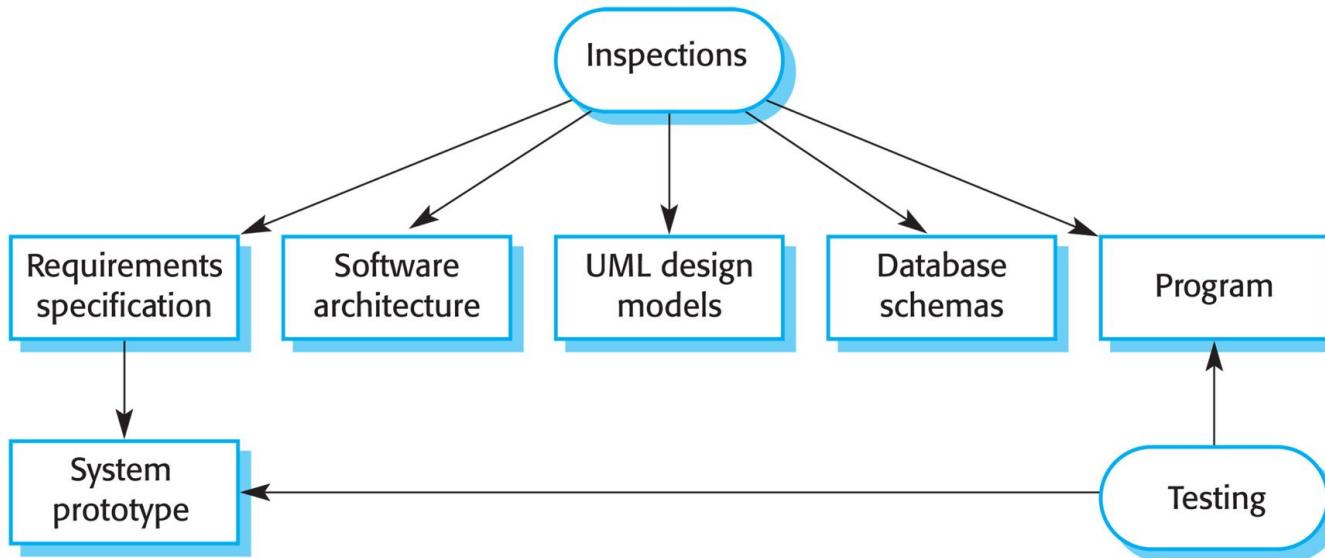
3.7 technical review: A systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.

NOTE—Technical reviews may also provide recommendations of alternatives and examination of various alternatives.

3.8 walk-through: A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems.

Reviews and Inspections

- Quality Assurance
 - Quality of Deliverables - Requirements, Software, documentation, etc.



Technical review

A systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.

- Systematic
- Qualified team
- Suitability for intended use
- Discrepancies

Example: Imagine a software development team is working on a new mobile banking application. The application is supposed to provide users with the ability to check their balance, transfer money, deposit checks using their phone camera, and pay bills.

Question: How do they perform technical review (with respect to the above four points)?



Walkthrough

A static analysis technique in which a **designer or programmer** leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible anomalies, violation of development standards, and other problems.

Table A.1—Comparison of review types

Characteristic	Management review	Technical review	Inspection	Walk-through	Audit
Objective	Monitor progress; set, confirm, or change objectives; change the allocation of resources	Evaluate conformance to specifications and plans; assess change integrity	Find anomalies; verify resolution; verify product quality	Find anomalies; examine alternatives; improve product; forum for learning	Independently evaluate conformance with objective standards and regulations
Decision-making	Management team charts course of action; decisions made at the meeting or as a result of recommendations	Review team requests management or technical leadership to act on recommendations	Review team chooses predefined product dispositions; anomalies should be removed	The team agrees on changes to be made by the author	Audited organization, initiator, acquirer, customer, or user
Change verification	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Leader verifies that action items are closed; change verification left to other project controls	Responsibility of the audited organization
Recommended group size	Two or more people	Three or more people	Three to six people	Two to seven people	One to five people
Group attendance	Management, technical leadership, and documented attendance	Technical leadership and peer mix; documented attendance	Peers meet with documented attendance	Technical leadership and peer mix; documented attendance	Auditors; the audited organization may be called upon to provide evidence
Group leadership	Usually the responsible manager	Usually the lead engineer	Trained facilitator	Facilitator or author	Lead auditor

Review type



Review type

Table A.1—Comparison of review types (continued)

Characteristic	Management review	Technical review	Inspection	Walk-through	Audit
Data collection	As required by applicable policies, standards, or plans	Not a formal project requirement. May be done locally.	Required	Recommended	Not a formal project requirement. May be done locally.
Output	Management review documentation; including the specification of action items, with responsibilities and dates for resolution	Technical review documentation, including the specification of action items, with responsibilities and dates for resolution	Anomaly list, anomaly summary, inspection documentation	Anomaly list, action items, decisions, follow-up proposals	Formal audit report; observations, findings, deficiencies

Technical review Procedure

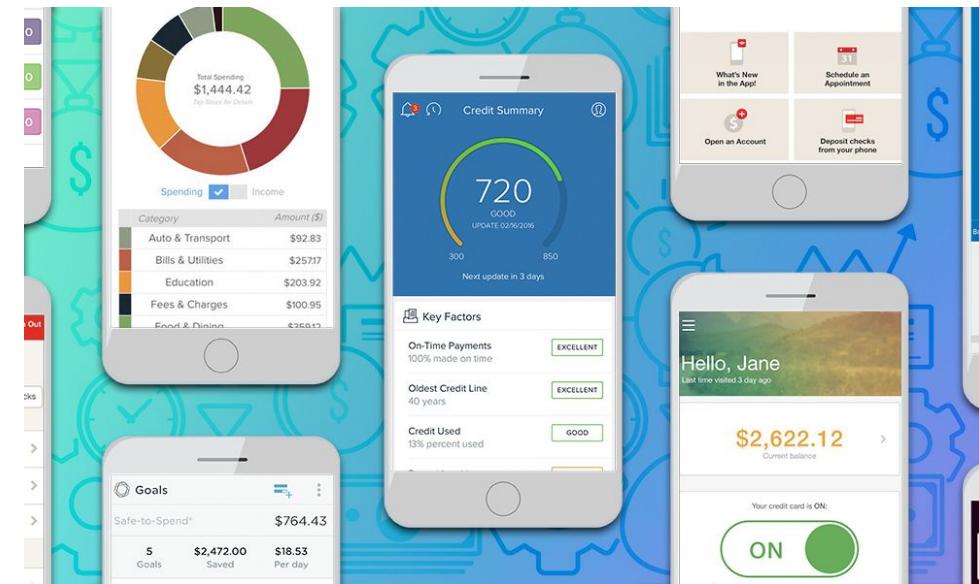
- Characteristics
 - Systematic
 - Qualified team
 - Suitability for intended use
 - Discrepancies/Deviations
- Elements subject to technical review, e.g.:
 - specifications
 - design description
 - test documentation
 - software development process descriptions
 - software architectural descriptions

(Examples of these elements are shown in the next few slides)

Technical review Procedure - Mobile Bank App

Specifications, including user stories and requirements for

- Secure login
- Account balance retrieval
- Funds transfer
- Mobile check deposit (camera support)
- Automatic bill pay
- Security requirement for encryption
- Multi-factor authentication



Technical review Procedure - Mobile Bank App

Design description, including wireframes and mockups for

- User interface
 - Flowcharts depicting the transaction process, etc.
- Class diagrams
- Database schemas
- Data pipelines
- Etc.

Technical review Procedure - Mobile Bank App

Test Documentation, including a suite of automated test scripts written to validate the functionalities of:

- New account registration
- Login
- Funds transfer
- Bill payment
- Performance testing
 - response time
 - robustness against peak load
 - security
 - up-time

Technical review Procedure - Mobile Bank App

Software Development Process Descriptions, a description of the iterative development process used for the app, including

- Sprint planning notes (Scrum)
 - priority of features (e.g. fingerprint login etc.)
 - sprint goals
 - daily meeting notes
 - sprint retrospective documentation
- Or similar, if other development methodologies are adopted.

Technical review Procedure - Mobile Bank App

Software Architectural Descriptions, an architectural overview document would detail the app's microservices architecture, including:

- Frontend and backend communications
 - How the user interface communicates with backend services
 - Account management
 - Transaction processing
 - Notification services
- Data pipeline (and, if exist, AI model integration)
- Security architecture
 - To show how customer data is protected, both at rest and in transit

Technical Review Procedure: Roles

- Decision maker
 - determines if objectives met
- Review leader
 - responsible for review
- Recorder
 - document anomalies, action items, decisions, and recommendations
- Technical reviewer
 - reviews and evaluates

Technical Review Procedure: Roles - Mobile Bank App

- Decision maker
 - Sarah, CTO, who has the final say on whether the app meets the strategic objectives of the company.
- Review leader
 - Michael, Project Manager, responsible for organizing the review.
 - He ensures that all necessary materials are prepared and that stakeholders from different departments are involved in the review process.
- Recorder
 - Priya, Business Analyst, takes detailed notes during the review meetings.
 - She documents any identified issues, and lists action items for improvement, decisions made by the group, and any recommendations for changes.
- Technical reviewer
 - Alex (Senior Software Engineer), Jeremy (Data Scientist), Maria (Machine Learning Engineer)
 - Work together to review and evaluate the technical aspects of the application
 - They assess the code quality, the performance of the AI models, data pipeline / database schemas

Technical Review Procedure: Preparation

- Identify the review team
- Assign specific responsibilities to the review team members
- Schedule and announce the meeting place
- Distribute review materials to participants, allowing adequate time for their preparation
- Set a timetable for distribution of review material and return of comments

Technical Review Procedure: Examination

- Determine if the software product is:
 - complete
 - conforms to the regulations, standards, guidelines, plans, specifications, specifications, and procedures applicable to the project
 - suitable for its intended use
 - ready for the next activity
- Identify anomalies and decide their criticality
- Generate a list of action items, emphasizing risks
- Document the meeting

Technical Review Procedure: Examination - Mobile Bank App

- **Complete:** All planned features for the current release, such as facial recognition login, account balance checks, and AI-powered fraud detection, have been implemented and are functioning as expected.
- **Conforms to regulations, standards, guidelines, plans, specifications, and procedures:** The app adheres to financial industry regulations like GDPR for data protection, follows the OWASP top 10 for security, and meets the accessibility standards outlined in WCAG. The AI components comply with ethical AI guidelines and data usage policies.
- **Suitable for its intended use:** User testing confirms that customers can easily navigate the app, use the AI chatbot for customer service efficiently, and the AI model for spending categorization helps users manage their finances effectively.
- **Ready for the next activity:** The app is stable enough to move to the next phase, which could be a broader beta test or a full production rollout, depending on the development stage.

Technical Review Procedure: Examination - Mobile Bank App

Identify anomalies and decide their criticality:

Example: An anomaly where the AI model fails to flag unusual transaction patterns that could indicate fraud is identified as a critical issue that requires immediate attention.

Generate a list of action items, emphasizing risks:

Action Item 1: Resolve instances of slow response times during peak usage hours to prevent potential customer dissatisfaction and attrition risk.

Action Item 2: Update encryption protocols for data transmission to address vulnerabilities and mitigate the risk of data breaches.

Action Item 3: Implement additional authentication steps for high-value transactions to reduce the risk of unauthorized access and fraud.

Action Item 4: Conduct thorough accessibility testing on new features to ensure compliance with legal standards and reduce the risk of regulatory penalties.

Action Item 5: Review and optimize the database schema to prevent potential performance bottlenecks as the user base grows.

Technical Review Procedure: Output

- Review objectives and whether they were met
- A list of software product anomalies
- A list of unresolved system or hardware anomalies or specification action items
- Recommendations on how to dispose of unresolved issues and anomalies
- Whether the software product meets the applicable regulations, standards, guidelines, plans, specifications, and procedures without deviations

Quality management and agile development

- Based on good practice rather than formal documentation
 - check before check in
 - never break the build
 - fix problems when you see them

Quality management and agile development - Mobile Bank App

- **Check before check-in**
 - Always run unit tests and perform a local build to ensure their changes
 - Errors are ensured to be eliminated before committing
- **Never break the build**
 - The continuous integration system is set up to build the app and run automated tests whenever new code is committed
 - Protocol of the team: The build after any commit must be always in a releasable state.
- **Fix problems when you see them**
 - Address flaws, errors immediately, rather than adding it to a backlog.
 - Remark: “TODO” is not error.

Where to put your Focus

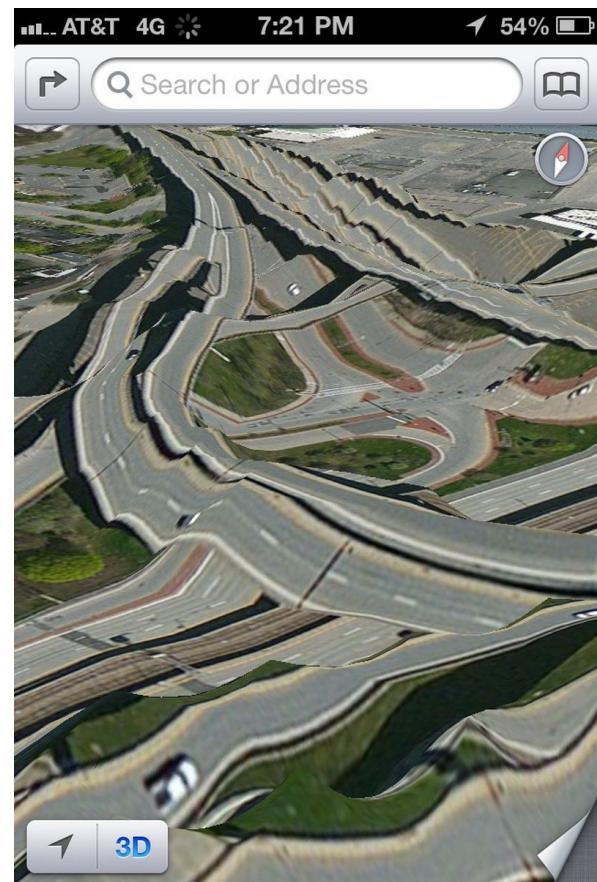
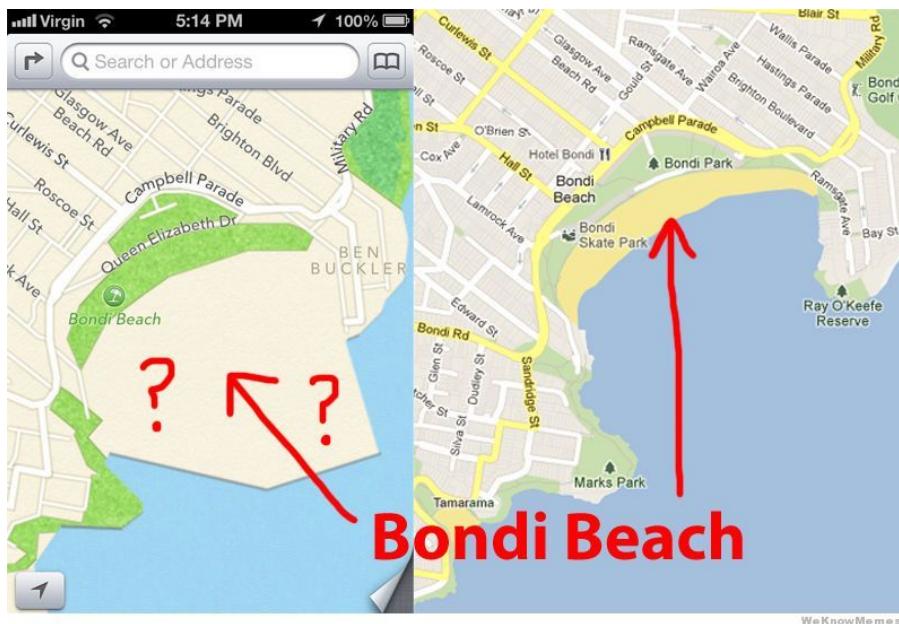




Where to put your Focus



“Apple maps bad” (Limited to its initial release in 2012)



“Apple maps bad” (Limited to its initial release in 2012)

Misaligned Priorities

- Apple's focus at the time was on replacing Google Maps as the default mapping system on iOS devices.
- Cutting off ties with Google vs. Better map quality than Google

Lack of Key Attention

- Map data quality?

Underestimation of Task Complexity

Inadequate Testing

If there had been more extensive real-world testing, many of the initial flaws might have been identified and fixed before the public release

Where to put focus? - Red Ring of Death (RROD)

RROD rate: **54.2 %**



Where to put focus? - Red Ring of Death (RROD)

Design Over Reliability: The Xbox 360 was designed with performance and aesthetics in mind, aiming to be powerful and visually appealing. However, the initial designs did not adequately address cooling, which led to overheating issues.

Time to Market Over Extensive Testing



Cost Microsoft more than **1 billion dollars**

BEST ANIMATIONS



Pareto Principle

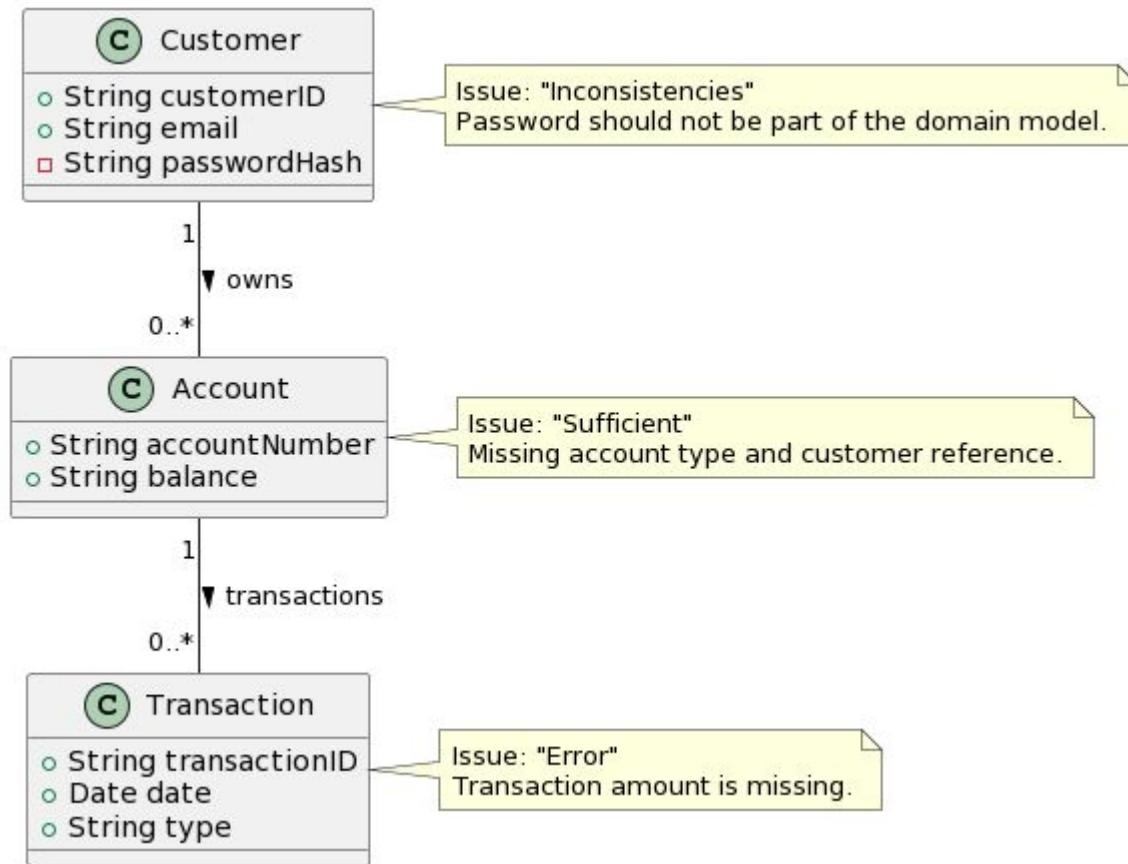
- For many events, roughly 80% of the effects come from 20% of the causes
- Named after Italian economist Vilfredo Pareto
 - noticed that 80% of Italy's land was owned by 20% of the population
 - also noticed about 20% of the peapods in his garden contained 80% of the peas
- Microsoft noted that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in a given system would be eliminated.



- Other examples: e.g. 20% of code contains 80% of bugs
- (Last 20% takes 80% of the time)

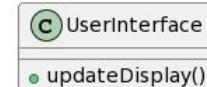
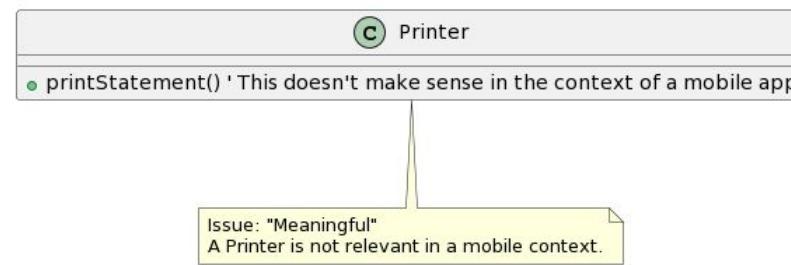
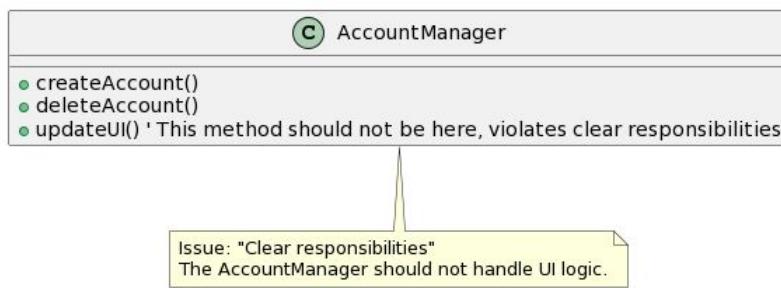
Review: Analysis

- Domain models
 - meaningful?
 - sufficient?
 - errors?
 - inconsistencies?
 - syntax?



Review: Design

- Meaningful?
- Clear responsibilities?
- Cohesion and coupling?
- Sufficient?
- Errors?
- Inconsistencies?
- Syntax?



Review: Requirements

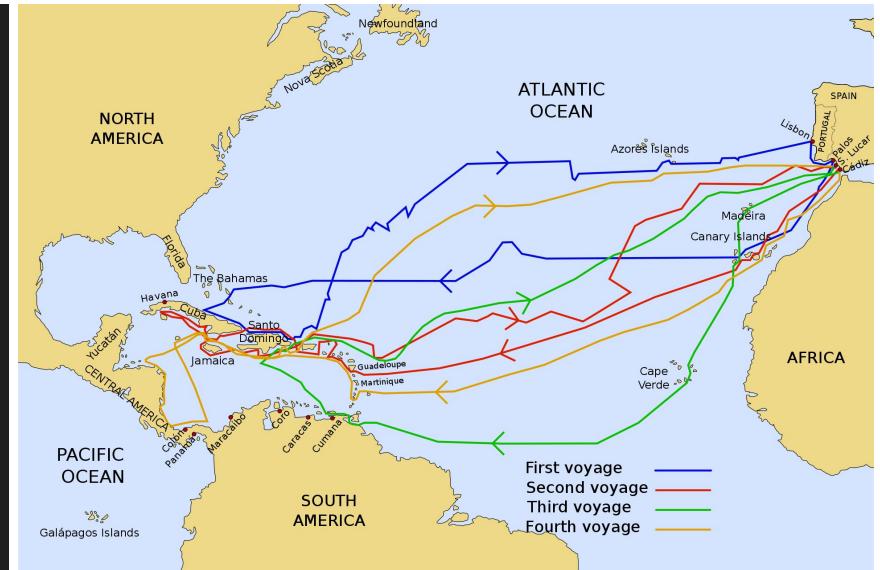
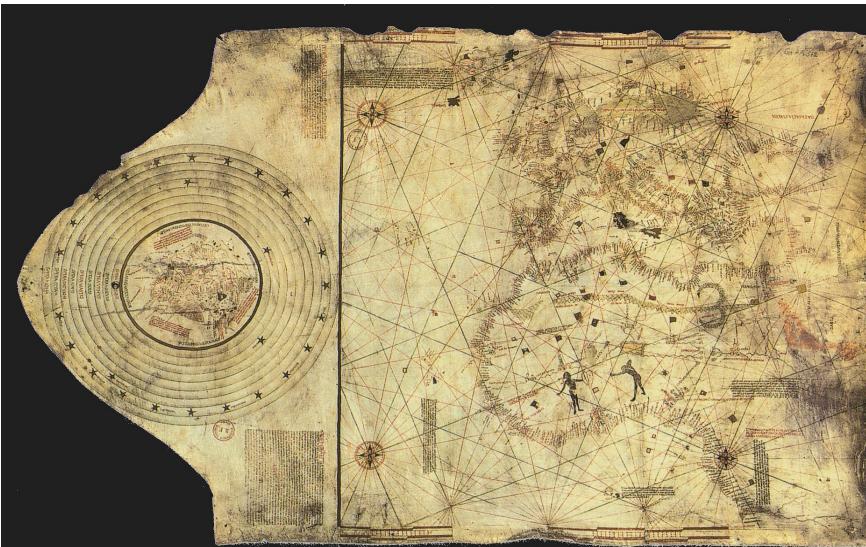
Cohesive	The requirement addresses one and only one thing.
Complete	The requirement is fully stated in one place with no missing information.
Consistent	The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.
Correct	The requirement meets all or part of a business need as authoritatively stated by stakeholders.
Current	The requirement has not been made obsolete by the passage of time.
Externally Observable	The requirement specifies a characteristic of the product that is externally observable or experienced by the user.
Feasible	The requirement can be implemented within the constraints of the project.
Unambiguous	The requirement is concisely stated without recourse to technical jargon, acronyms or other esoteric verbiage. It is subject to one and only one interpretation.
Mandatory	The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated.
Verifiable	The implementation of the requirement can be determined through one of four possible methods: inspection, analysis, demonstration, or test.

Cohesive	The requirement addresses one and only one thing.
Complete	The requirement is fully stated in one place with no missing information.
Consistent	The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.
Correct	The requirement meets all or part of a business need as authoritatively stated by stakeholders.
Current	The requirement has not been made obsolete by the passage of time.
Externally Observable	The requirement specifies a characteristic of the product that is externally observable or experienced by the user.
Feasible	The requirement can be implemented within the constraints of the project.
Unambiguous	The requirement is concisely stated without recourse to technical jargon, acronyms or other esoteric verbiage. It is subject to one and only one interpretation.
Mandatory	The requirement represents a stakeholder-defined characteristic the absence of which will result in a deficiency that cannot be ameliorated.
Verifiable	The implementation of the requirement can be determined through one of four possible methods: inspection, analysis, demonstration, or test.

Project Planning

Intro to Project Planning

- Project Scope Statement
 - Purpose, Desired result, Acceptance criteria, omitted
 - Direction
- Project plan
 - Subtasks, Estimates, Sequence, Deadlines
 - Route



Plan

- 
1. Risk management
 2. Project Schedule
 3. Budget
 4. Communication plan

1. Risk management

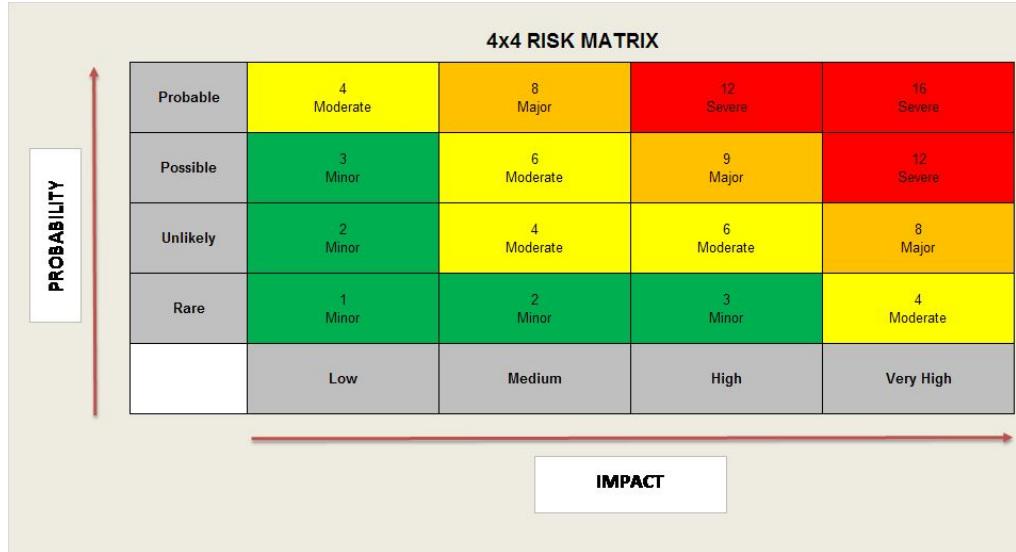
- A. Identify risks
- B. Assess risks
- C. Address risks

1.a Identify risks

- Brainstorm
 - Make a list of potential risks
- Ex - Mobile bank app
 - The project group members do not come to meetings...
 - Computer vision is much more difficult than expected
 - Security vulnerabilities
 - Scalability concerns
 - Data migration errors
 - Mobile platform updates

1.b Assess risks

- Risk matrix
 - Impact
 - Probability
- Score = Impact x Probability
- Decide 'cutoff'
 - What risk is acceptable?
 - Which level MUST be addressed?



1.c Address risks

- Unacceptable risks / critical risks
 - Transfer: Give the task to a 3rd party for which the risk is less (Specialist)
 - Acceptance: Accept risk
 - Mitigate: Reduce Impact and/or Probability
 - Eliminate: Remove risk
- Risk management plan

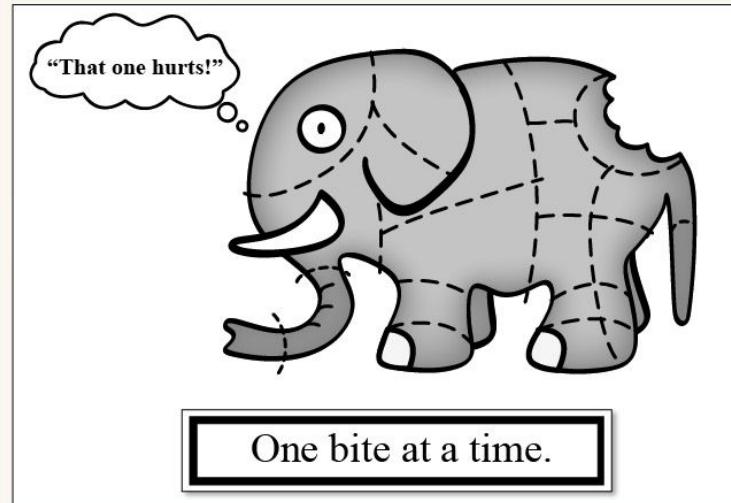
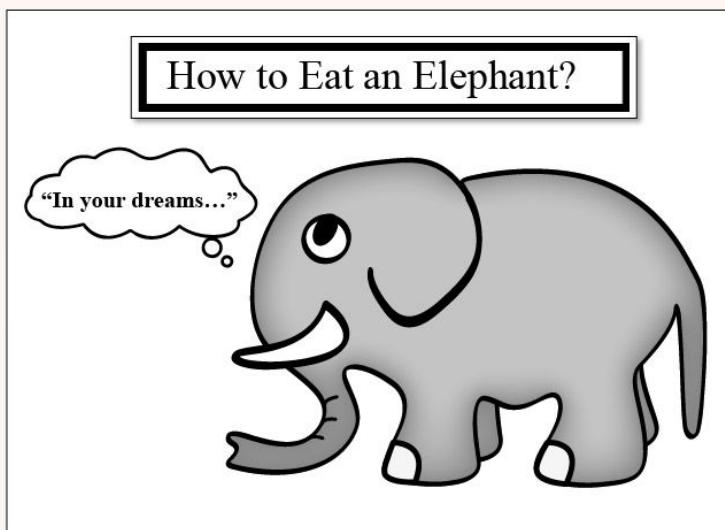
Risk	Score	Strategy	Responsibility

2. Project plan

- There are a lot of tools for project plans
 - Microsoft project (Part of 'Azure tools for education' - free for DTU students)
 - Jira
 - Project place
 - Trello + Teamgantt.com
 - [Asana](#) + [Instagantt](#)
 - Openproject
 - Monday.com

2.a Work Breakdown Structure

- Break the project down into manageable chunks
 - If the bite is difficult to estimate, it may be too big!
 - Very large elephants must be divided to avoid rotting

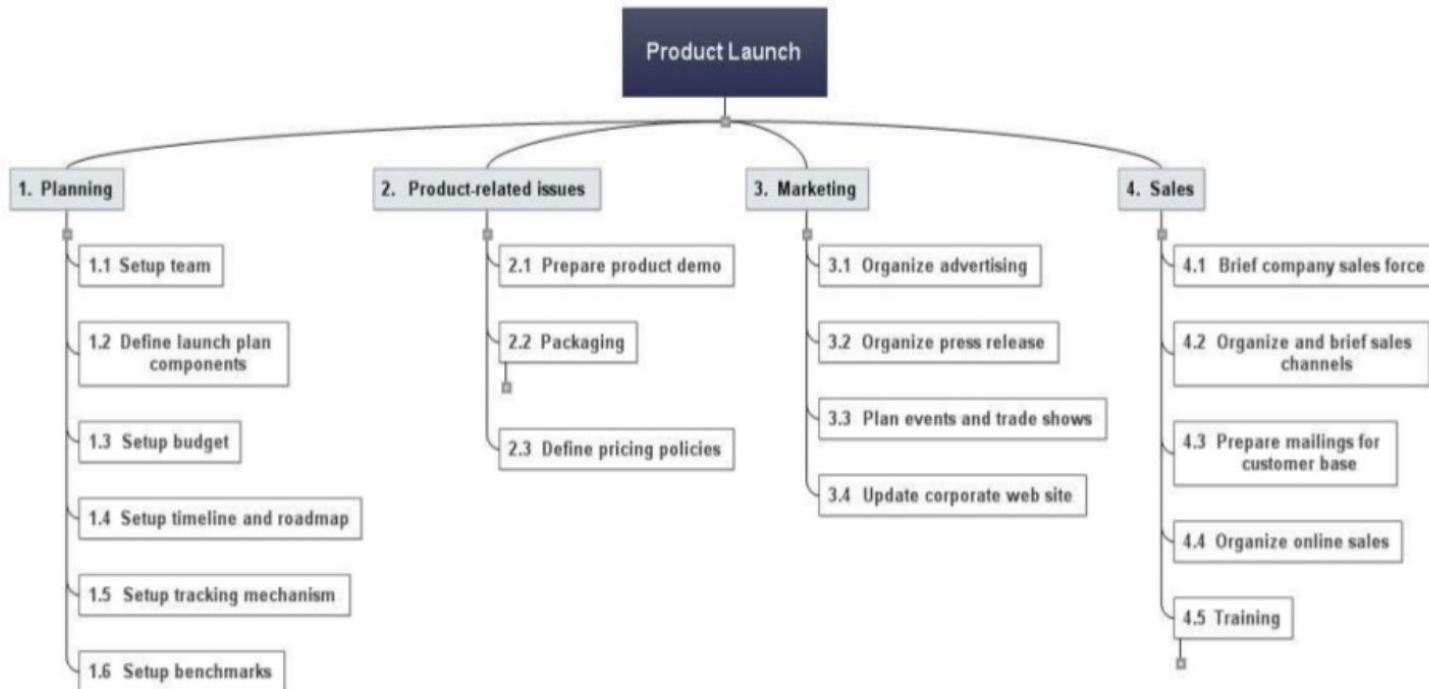


One bite at a time.

2.a Work Breakdown Structure

- The task is divided into smaller tasks
- Mindmap
- Post-its
 - Kanban
 - Scrum
- List

Mindmap



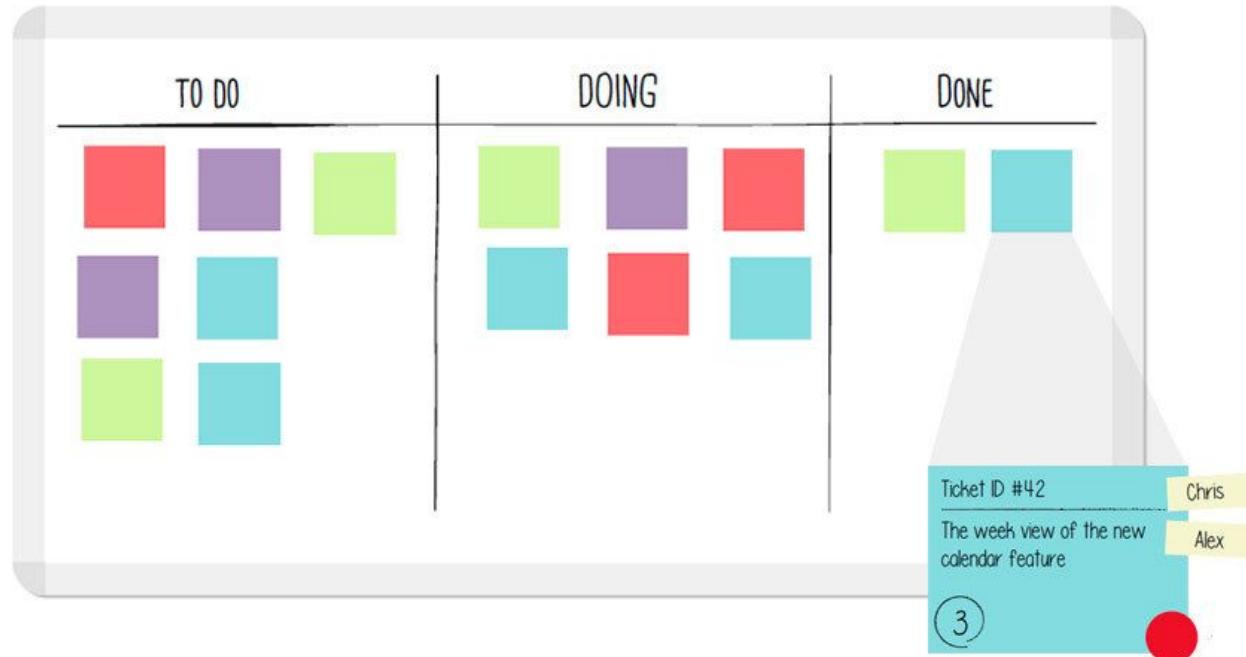
Post - Its

- Everyone writes assignments on post its
 - Placed under headings



Kanban and scrum (and scrumban)

- Suitable for sprints
- Cannot stand alone for large projects

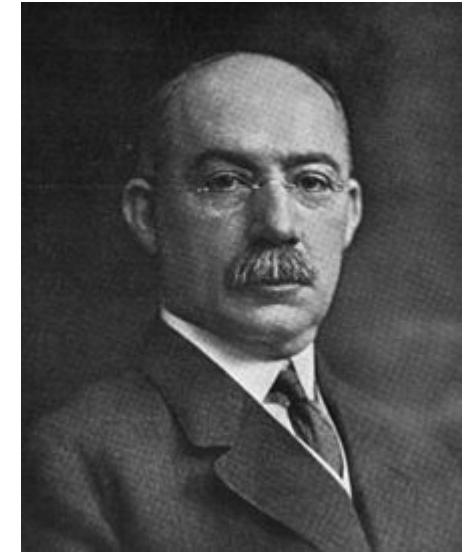


List

1. Prototyping
 - 1.1. Low model
 - 1.1.1. Low foundation
 - 1.1.2. Build model
2. Make working drawings
 - 2.1. Foundation drawings
 - 2.2. Essays
 - 2.3. ...
3.

2.b Put tasks in order

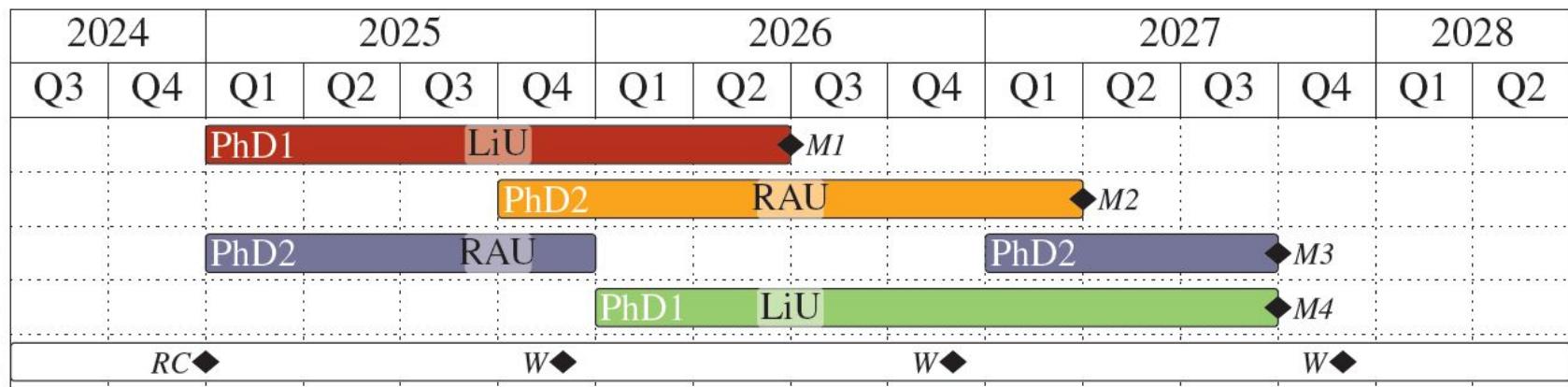
1. Mindmaps and Post-its are converted to lists.
2. Dependencies are identified
 - o Finish to start
 - o Start to start
 - o Finish to finish
3. Tasks that are bound are placed one after the other
 - o Bindings are important for critical path!



Henry Gantt



2.b Put tasks in order



2.c Identify the project team

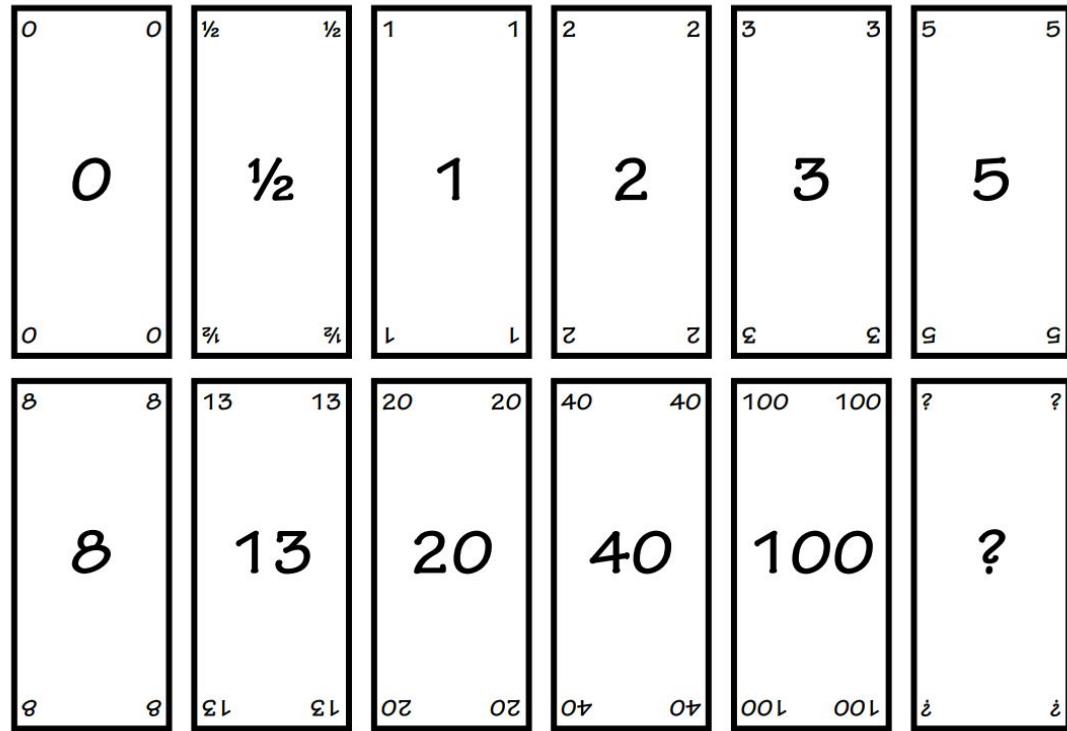
- Who has what skills?
- Who can solve which tasks?
- Who can you get hold of?
- Decide which tasks can run when
- Determine how many tasks can run in parallel
 - Some things are bad to parallelize!
 - Maybe even negative synergy!

2.d Estimate the duration of the various tasks

1. Use your experience
 2. Use the team's experience
 3. Use expert experience
 - a. (Beware! - they might just say what you want to hear!)
 4. Use estimation methods
 - a. PERT formula (requires experience): Expected = (Optimistic + 4 * Realistic + Pessimistic) / 6
 - b. Planning poker (surprisingly good for unfamiliar tasks) https://en.wikipedia.org/wiki/Planning_poker
- Tasks of approx. 10-50 hours...
 - (Time it with Pi)

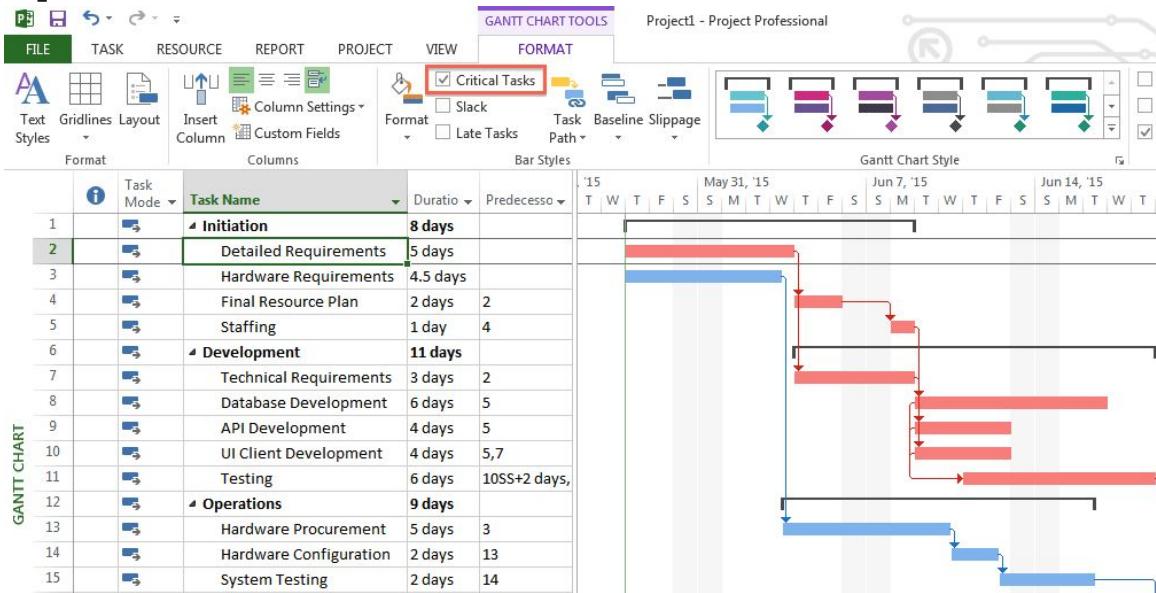
Planning poker (Scrum poker)

-
- Biggest and smallest explains
 - Large spread
 - Too big a bite
 - Need for research - Replace with?
 - Stick to consensus



2.e Identify critical path

- Construct Gantt chart
 - Use a tool
 - Parallelize!
 - Be aware of bindings
 - Find critical path!
- Critical path can change
 - Update
 - Monitor
- Set all sails on critical path!
 - Best resources



2.f Responsible for tasks

- Put responsibility on tasks!
 - No later than when the task is scheduled to start
 - Only one responsible!
 - If everyone has responsibility - NO ONE has responsibility
 - Perhaps the task should be divided into smaller ones?

Follow up

- Record time consumption every day (project manager's responsibility)
- Check whether the remaining time is sufficient
 - => Adjust estimate
 - => Lower quality
 - Push the deadline
 - Is critical path at risk?
- Update plan! (project manager)

2.g Milestones

- Divide the project into 'sprints'
- Typically 2-3 weeks of work per sprint
- Block for derailed project
- Lose a maximum of 3 weeks of work....

3. Budget

- Person hours
 - Overhead (10%)
- Other expenses

4. Communication plan

What	Who (responsible)	Audience	How	When
Team accountability session (Scrum meeting)	Project manager	Project team		
Status report	Project manager	Steering group	Meet	
Prototype update?				

62531 Development Methods for IT Systems

Glossary of terms

Deena Francis

October 7, 2023

Activity

It is a series of actions or operations. Thus, it is not atomic, that is, it can be decomposed into several simpler steps or actions. An activity generally takes some time to complete.

Example: Login user into the system: This activity can be considered as a series of actions such as get the user name, get the user password, check the user name and password, display error message (if authentication is wrong), approve login. In Figure 1 shown here, the activity called “Login user into system” has a number of operations or actions that will be carried out.

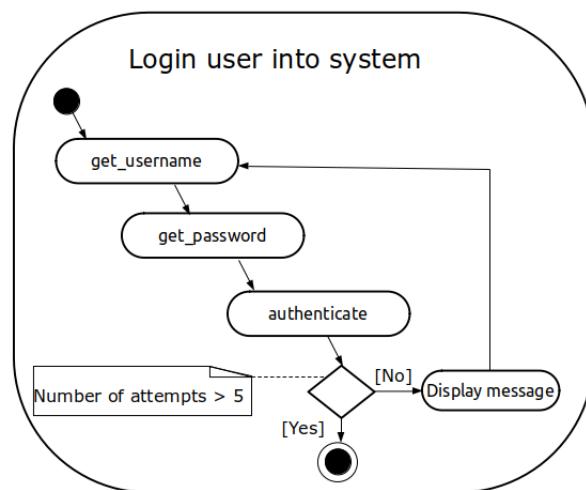


Figure 1: Example of an activity

Action

It is a simple step in an activity. Thus, it is atomic, that is, it cannot be decomposed into simpler steps. An action generally takes a short amount of time to complete.

Example: get_username: This action is a simple step that consists of getting the user name of a user. See Figure 1.

Activity diagrams

These are UML behavioral diagrams that show the flow or dynamics of control, execution of your system.

Why?:

These diagrams are created for the purpose of understanding the flow of your system at a high level. This can in turn, help model (or visualize) in detail complex activities.

Table 1 shows the various components of the activity diagram and their meaning.

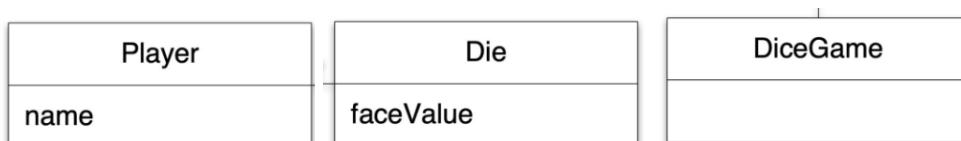
Symbol	Meaning
●	The start node or initial node. It indicates the start of your activity.
○	The stop node or activity final node. It indicates the end of your activity.
⊗	The flow final node. It indicates the end of a flow in your activity.
	The action node. It represents a step in the activity.
	The accept event action node. It represents a step in the activity that waits for an event to occur.
	The wait time action node. It represents a step in the activity that is a time event occurrence and has the time of occurrence attached to it.
	The branch or decision node. It indicates a check that is made. Notice a single arrow into this node and multiple arrows coming out of it. The arrows coming out indicate the paths to be followed when the condition takes either YES or NO.
	The merge node. It takes as input several flows and merges them into a single outgoing flow.
	The fork node. It has a single incoming flow, which it splits into multiple parallel flows.
	The join node. It has multiple incoming flows, which it joins into a single flow.

Table 1: Activity diagram components

Analysis class

It is a class that consists of both data and behavior. This is a high level view of your class because the implementation details are not specified. In the examples shown below, these analysis classes belong to a specific domain (area), namely the dice game.

Example:



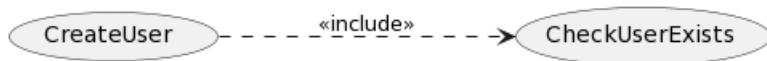
Association

It is a relationship. It can be between:

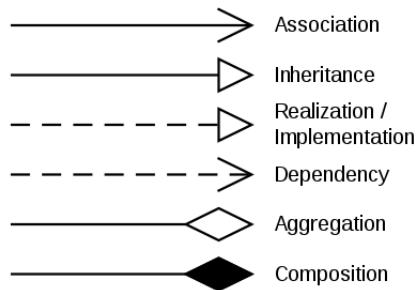
- Classes: Example: Class Dog inherits from class Animal, Class Classroom consists of a group of Students.
- Use cases: Example: The use case of SignIn includes the use case of CheckUserExists (Reference: Lecture 3 slides)
- Actors: Example: GroupLeader inherits from Researcher (Reference: Lecture 3 slides).
- Usecases and actors: Example: User is associated to a use case called Play game (Reference: Lecture 3 slides).

In UML it is represented by lines. There are different kinds of associations, and hence different kinds of lines (see below).

Example:



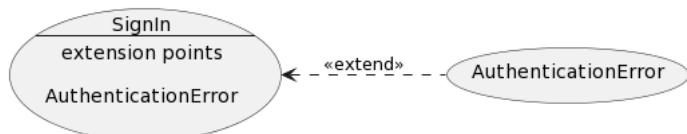
Types of associations:



Extend association

This association is used when one use case extends (or adds on to) the behavior of another use case. In UML, you will use a dashed arrow from the extending use case with the <<extend>> keyword on top of the arrow. The extending use case can be thought of as adding some extra functionality to the other use case. This extending use case can also be thought of as an optional use case, that is invoked under special conditions.

Example:

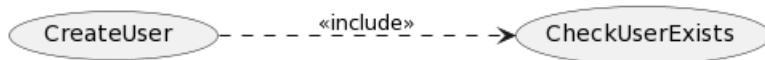


Include association

This association is used when a use case contains another ‘sub-use case’. This occurs in cases where we have complex use cases, which can be broken down further into other sub-use cases.

Example:

The included use case, that is the sub use case can be thought of as a mandatory part of the original use case. In the example here, the CheckUserExists use case is an essential part of CreateUser use case.



Object

They are real-world entities that have some properties (attributes) and behavior(methods). They are instances of classes.

Example:

Person p1;

Here p1 is the object of data type Person, which is a class. What can this object do? It will have all the data (attributes) and behavior (methods) of the class Person.

Class

It is a template for creating objects. They can be considered as high-level data types.

Example:

```

class Person{
    private String name;
    private String dateOfBirth;

    public getAge(){
        LocalDate dob = LocalDate.parse( this .dateOfBirth );
        LocalDate curDate = LocalDate.now();
        return Period.between(dob, curDate).getYears();
    }

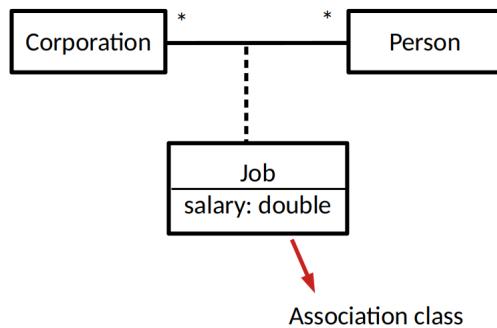
    public getName() {
        return this.name;
    }
}
  
```

In the example above, the class called Person has the properties (attributes) of name and dateOfBirth, which are variables that describe a Person. The methods (functions) getAge() and getName() calculate the age of the Person and gets the name respectively. In order to get the current date, we have used import java.time.LocalDate; and to find the number of years between two dates we have used import java.time.Period;

Association classes

They are classes that are part of the association between two other classes. Here, the association or relationship is defined by a class. The association class, just like any other class has attributes and methods.

Example:



In the example, a class called `Job` is defined, that describes the relationship between the classes `Corporation` and `Person`.

```
public class Corporation { ... }

public class Person { ... }

class Job {
    private Person owner;
    private Company company;
    private double salary;
    private float periodOfEmployment;

    public Job(Company c, Person p, double s) {
        this.company = c;
        this.person = p;
        this.salary = s;
    }
}
```

In the example above, the class Job is the association class. It describes the relationship between the two classes Corporation and Person. Notice how this class has attributes related to the Job such as the company, person, salary and the period of employment.

Class diagrams

It is a UML diagram showing the classes, their relationships (associations), attributes and methods. This diagram gives an overview of the software components that you will build.

Example:



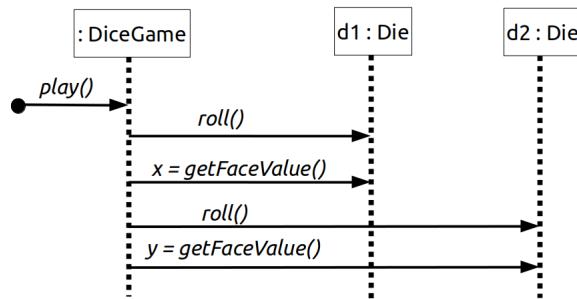
Unified Modelling Language (UML)

It is a visual language for specifying, constructing and documenting the artifacts of the system. This language can show both static and dynamic aspects of the system you are building.

Sequence diagrams or System sequence diagrams

This is a UML diagram that shows the interaction of processes in a time sequence. It captures the course of events within a use case.

A good reference for all the symbols and notation used in UML: Sequence diagrams in UML.



In the figure shown here, we have a sequence diagram showing the events that happen in the use case of Play a dice game. Here time is depicted in the horizontal dashed lines, where time is increasing as we go from the top to the bottom.

Use cases

They are text stories, used to discover and record requirements. It will contain the steps required to achieve a certain goal of the user. The use cases are made by experts. (Reference: Lecture 2 slides)

Example:

In the example here, the use case is in the brief form, where only the steps and the actors involved are

UC1: Send message

Actor:User of chat app

Use case text:

- ① The user opens the app.
- ② He/she presses start chat button.
- ③ He/she selects the person(s) to whom the message is to be sent.
- ④ He/she types the message in the text box.
- ⑤ He/she presses send message button.
- ⑥ If the message has been received, user gets confirmation, else gets status report.

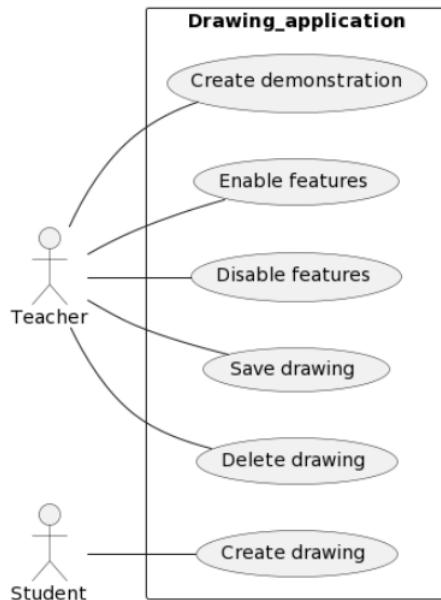
mentioned. There are more detailed kinds of use cases such as casual and fully dressed.

Use case diagrams

It is a UML diagram that shows an overview of actors, use cases and their relationships.

Example:

In this example, the system boundary is represented by the rectangle enclosing the ovals. The ovals represent



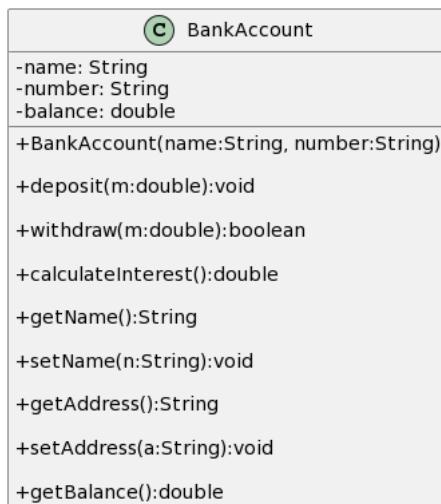
the use cases, the stick men called Teacher and Student are the actors, and the lines represent the interaction between the actors and the use cases.

Design class

They are classes that consists of data (attributes) and behavior (methods) such that the software implementation details are specified.

Example:

In this example we have a class called BankAccount that has data (attributes) such as name, number and



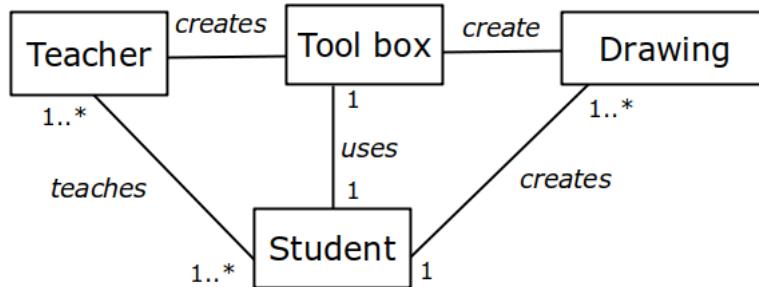
balance. It also has behavior (methods) such as deposit(), getName() etc. Notice the + and - symbols in front of the attributes and methods. These symbols denote the visibility. Here + means public and - means private. In java, public keyword attached to a variable or method means that all other classes can access this variable or method. In java, the private keyword attached to a variable or method means that this variable or method are accessible only to the members of this class.

Domain model

A diagram that shows concepts and their associations and attributes. The software implementation details are not provided in this diagram. This model is not strictly UML, but it can be converted to a UML class diagram.

Example:

In this example, we have a drawing application, where there are the concepts of Teacher, Toolbox, Drawing



and a Student. These concepts are related to each other by lines representing how they interact with each other. The multiplicity (the number of interacting elements) and a short keyword (a verb) describing the nature of the relationship are also contained in this diagram.

Unified Process (UP)

It is an iterative and incremental software development process. The term iterative means that it involves repetition, and incremental means that it happens in a series of additions or increases. It has several phases: Inception, Elaboration, Construction and transition.

Risk management

Risk is the probability of an adverse circumstance. Risk management is a project management task that involves handling risks associated with a project. (Reference: Lecture 4)

The risks can be:

- Project risks: It affects the project schedule or resources. Example: Loss of an experienced developer.
- Product risks: It affects the quality or performance of the software being developed. Example: Failure of a purchased component to perform as expected.
- Business risks: It affects the organization developing or procuring the software. Example: A competitor introducing a new product.

In risk management, one would typically do risk identification, risk analysis, risk planning and risk monitoring.

Algorithm

An algorithm is a sequence of steps used to solve particular types of problems or perform some computation. An algorithm has inputs and outputs, and the steps in the algorithm transform the inputs to the outputs.

Example:

Input: A sequence of n numbers a_1, a_2, \dots, a_n .

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Suppose there exists a function called Insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one

place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.

Algorithm: Begin at the left-most element of the array and invoke Insert to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

(Reference: Wikipedia)

Pseudocode

It is a description of the steps in an algorithm using a mix of conventions of different programming languages.

Example:

In this example, the array A is being sorted using the algorithm of insertion sort (see the section for

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Algorithms).

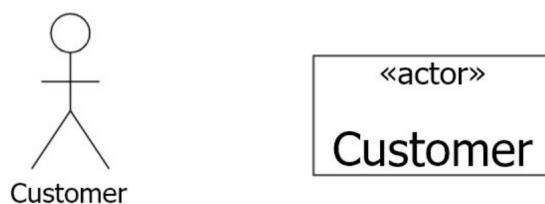
Code documentation

It is a way of explaining how the code works and how to use it. It can be done in the form of comments in the code, external documents with pictures, and textual explanations. The goal of documentation is to aid in the development and maintenance of a good software that is easy to understand, easy to test and maintain.

Actors

They are users of the system, and are involved in the use cases, that is they interact with the system in some way. In UML, actors are named after the role they play.

Example:



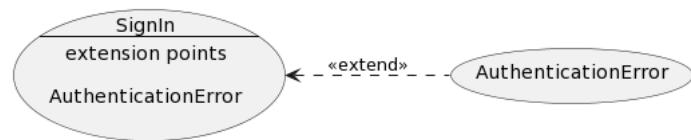
Stakeholders

They are people or systems that affects or is affected by the system. Examples include end users, supporters, customers, organizations etc. Some actors can also be stakeholders if they are affected by the system.

Extension points

They are points in the behavior of the use case where a behavior can be extended by some other (extending) use case.

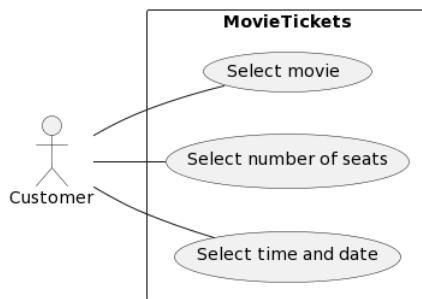
Example:



Primary actor

They are actors who start the use case. Goals of this actor are met by the execution of the use case. In UML, primary actors are placed on the left hand side of the use cases.

Example:

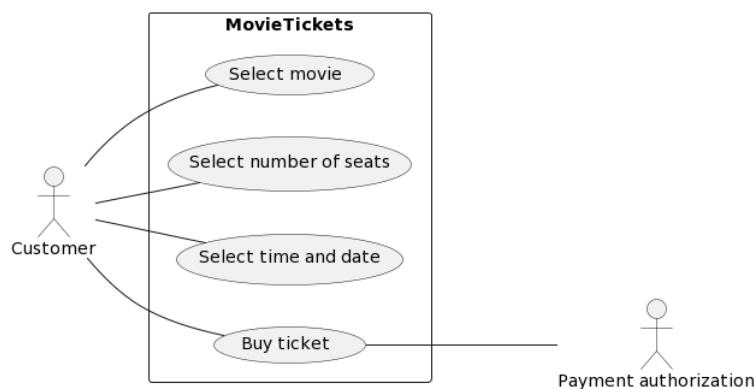


In this example, the primary actor is Customer. This actor starts the use case of Select movie and others.

Secondary actor

They are actors who never initiates the use case, but either supplies or receives information as part of the use case. They do not necessarily benefit from the use case. In UML, secondary actors are placed on the right hand side of the use cases.

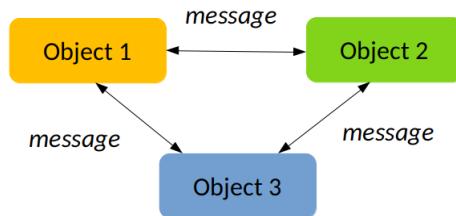
Example:



In this example, Payment authorization is a secondary actor, as this actor does not start the use case, it merely provides a service.

Object oriented programming

It is the paradigm of programming where computation or problem solving is done by the interaction of objects.



Attributes

These are the properties of a class. They are also called data members, fields or variables.

Example:

```

public class Person {
    private int age = 5;
    private String name = 'XYZ';
}
  
```

In this example, the class Person has two attributes age and name.

Methods

These are functions (a series of steps to do something specific) that define the behavior of a class.

Example:

```

public class Person {
    private int age = 5;
    private String name = 'XYZ';

    public void setAge( int x){
        this.age = x;
    }
}
  
```

In the example, the method setAge() sets the age of a Person.

Class, Responsibility, Collaborators (CRC) card

This is a tool to identify classes, their methods and associations with other classes. A CRC looks like the one shown in Figure 2.

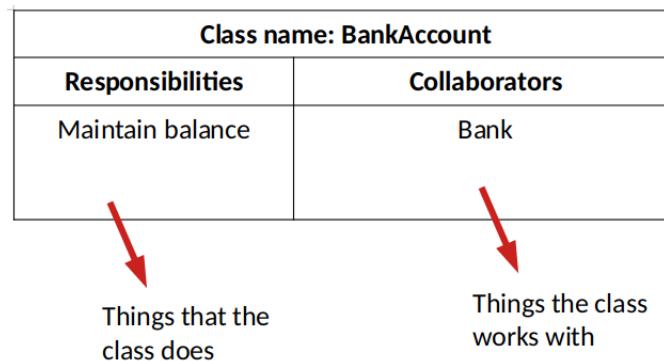


Figure 2: CRC card

The class name is written on top of this card. Its methods are listed in textual form in the column Responsibilities, and the other classes that it associates with are mentioned in the column Collaborators.

Multiplicity

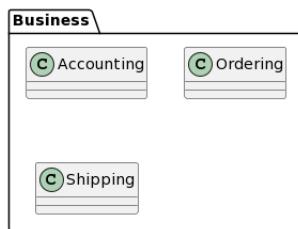
It is a number or a list of numbers that indicates how many entities are part of an association.



In the figure given above, the multiplicity is indicated by 1 and 2 on either side of the arrow. This association means that a DiceGame object is associated with 2 Die class objects.

Packages

It is a logical grouping of model elements such as classes. The package will have a name such as Business as in the example shown below. This package contains classes that are in the domain (area) of Business.



Glossary

It is a collection of all terms and their definitions. It can be stored in the form of a document. The purpose of this document is to help standardize terms used within the project so that all stakeholders and members of your team agree on the terms used.

Supplementary specification

It is a document that contains all textual description of other requirements. By other requirements, we mean the ones that are not documented in a use case or diagrams.

Example:

An example of a supplementary specification includes a document that contains the reports, documentation, packaging, supportability information etc.

Business rules

They are statements that provide the criteria and conditions for making a decision.

Example:

An example of a business rule is as follows. Net sale is defined as the total sales price of an order before discounts, allowances, shipping, and other charges.(Reference lecture 4).

Probability * Impact matrix in risk management

It is a matrix with severity (danger level) on the rows and the probability or likelihood on the columns. The severity and likelihood have a rating scale of 1,.., 5, where 1 indicates low and 5 indicates severe. In each cell of this matrix, we obtain the risk level by multiplying the corresponding row's rating and column's rating.

Example:

Database

It is an organized collection of structured information, or data, typically stored electronically in a computer system.

Vision

It is either a short statement or a longer description of the project/product that includes a description of:

- Need: Why do we need this product? Explain what this product provides.
- Approach: Explain the methodology or approach that the product uses to do what it does.
- Benefit: Explain the benefits of using this product.
- Competition: Explain the competing products out there, and why your product is better or comparable to the competition.

User stories

They are simple, single sentence description of a user's task that involves using the system. These are usually made by the users themselves.