

IT303

Software Engineering

Syllabus

- ***Week 1:*** Introduction, Software Process
- ***Week 2,3,4,5:*** Requirements Analysis, System Architecture and Design, OOAD, Design Patterns,
- ***Week 6,7,8,9,10 :*** Version Control, Testing, DevOps, Reliability, Performance of Computer Systems,
- ***Week 11,12,13,14:*** Research paper's implementation outcomes, Project submissions, Case Studies.

Evaluation Pattern

- **15% for Mid Sem**
- **25% for End Sem**
- **60% for Project, Assignments, Quizzes**

Software?

- *The product that software professionals **build** and then **support** over the long term.*
- *Software encompasses: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately store and manipulate information and (3) **documentation** that describes the operation and use of the programs.*

Types Software Products

- **Generic products**
 - Stand-alone systems that are marketed and sold to **any customer** who wishes to buy them.
 - Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- **Customized products**
 - Software that is commissioned by **a specific customer** to meet their own needs.
 - Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Software Costs

- Software costs often dominate computer system costs. The costs of software on a PC are often greater than the hardware cost.
- Software costs **more to maintain** than it does to develop. For systems with a long life, maintenance costs may be several times development costs.
- Software engineering is concerned with cost-effective software development.

Software Engineering

- The IEEE definition:
 - *Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*

Importance of Software Engineering

- More and more, individuals and society rely on advanced software systems. We need to be able to produce **reliable and trustworthy systems economically and quickly.**
- It is usually **cheaper, in the long run**, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the **costs of changing** the software after it has gone into use.

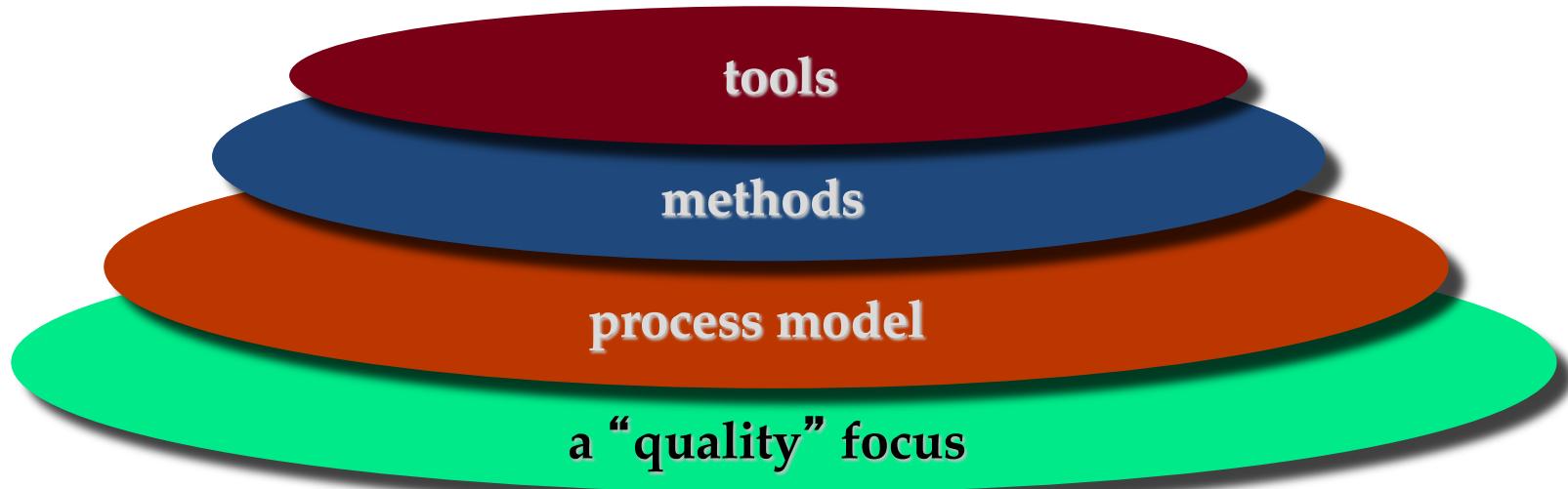
FAQ about software engineering

Question	Answer
What is software?	Computer programs, data structures and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

A Layered Technology



- Any engineering approach must rest on organizational commitment to **quality** which fosters a continuous process improvement culture.
- **Process** layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestone are established, quality is ensured and change is managed.
- **Method** provides technical how-to's for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.
- **Tools** provide automated or semi-automated support for the process and methods.

Software Process

- A process is a collection of activities, actions and tasks that are performed when some work product is to be created. It is **not a rigid prescription** for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work to pick and choose the **appropriate set of work actions** and tasks.
- Purpose of process is to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

- **Communication:** communicate with customer to understand objectives and gather requirements
 - **Planning:** creates a “map” defines the work by describing the tasks, risks and resources, work products and work schedule.
 - **Modeling:** Create a “sketch”, what it looks like architecturally, how the constituent parts fit together and other characteristics.
 - **Construction:** code generation and the testing.
 - **Deployment:** Delivered to the customer who evaluates the products and provides feedback based on the evaluation.
-
- These five framework activities can be used to all software development regardless of the application domain, size of the project, complexity of the efforts etc, though the details will be different in each case.
 - For many software projects, these framework activities are applied **iteratively** as a project progresses. Each iteration produces a software increment that provides a subset of overall software features and functionality.

Five Activities of a Generic Process framework

Umbrella Activities

Complement the five process framework activities and help team **manage and control** progress, quality, change, and risk.

- **Software project tracking and control:** assess progress against the plan and take actions to maintain the schedule.
- **Risk management:** assesses risks that may affect the outcome and quality.
- **Software quality assurance:** defines and conduct activities to ensure quality.
- **Technical reviews:** assesses work products to uncover and remove errors before going to the next activity.
- **Measurement:** define and collects process, project, and product measures to ensure stakeholder's needs are met.
- **Software configuration management:** manage the effects of change throughout the software process.
- **Reusability management:** defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- **Work product preparation and production:** create work products such as models, documents, logs, forms and lists.

Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry Out the Plan

- *Does the solutions conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

Software Myths

Erroneous beliefs about software and the process that is used to build it.

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,
but ...
- Invariably lead to bad decisions,
therefore ...
- Insist on reality as you navigate your way through software engineering

Software Myths Examples

- **Myth 1:** Once we write the program and get it to work, our job is done.
- Reality: the sooner you begin writing code, the longer it will take you to get done. 60% to 80% of all efforts are spent after software is delivered to the customer for the first time.
- **Myth 2:** Until I get the program running, I have no way of assessing its quality.
- Reality: technical review are a quality filter that can be used to find certain classes of software defects from the inception of a project.
- **Myth 3:** software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
- Reality: it is not about creating documents. It is about creating a quality product. Better quality leads to a reduced rework. Reduced work results in faster delivery times.
- Many people recognize the fallacy of the myths. Regrettably, **habitual attitudes and methods** foster poor management and technical practices, even when reality dictates a better approach.

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

Software Engineering 9/e
By Ian Sommerville

Verification and Validation of Safety Critical Systems

Presented By

Madhusmita Das

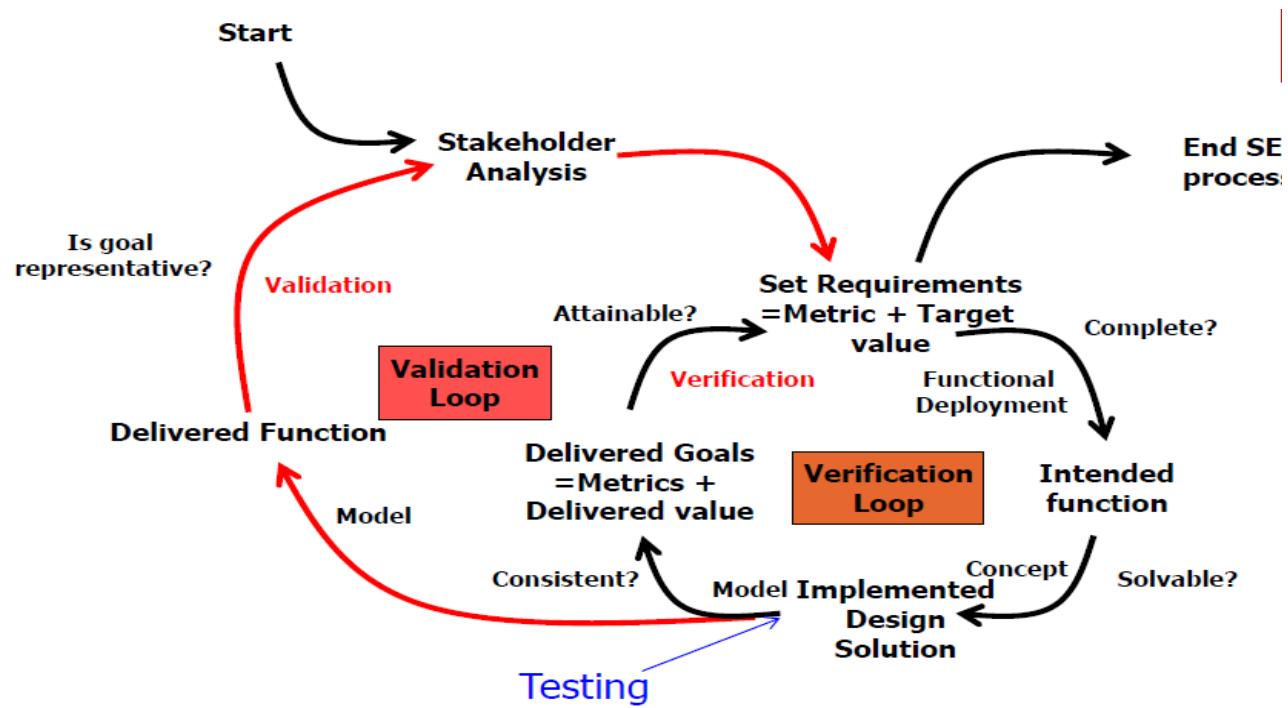
Guided By

Dr. Biju R. Mohan

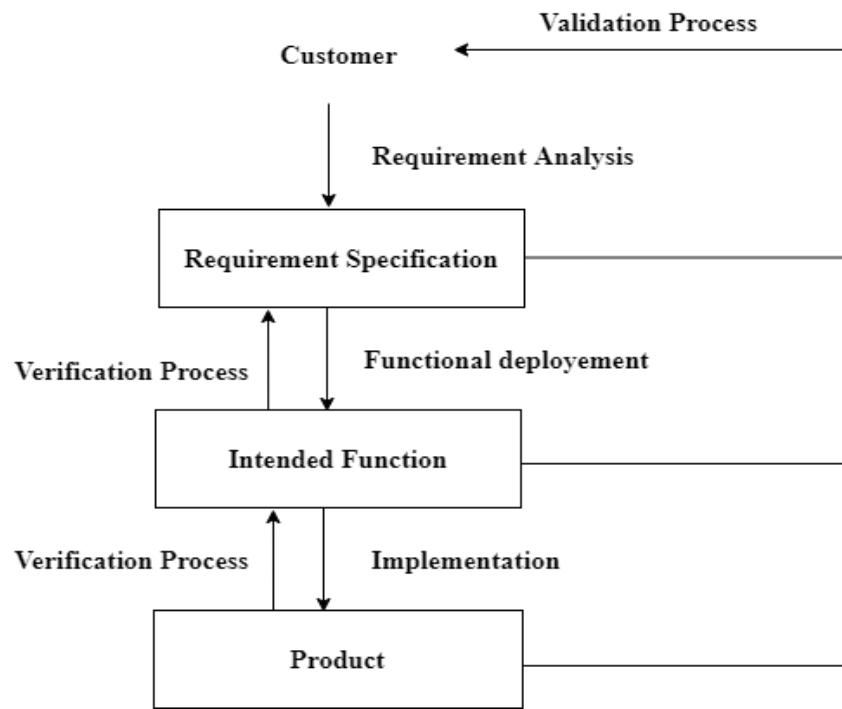
Verification and Validation process

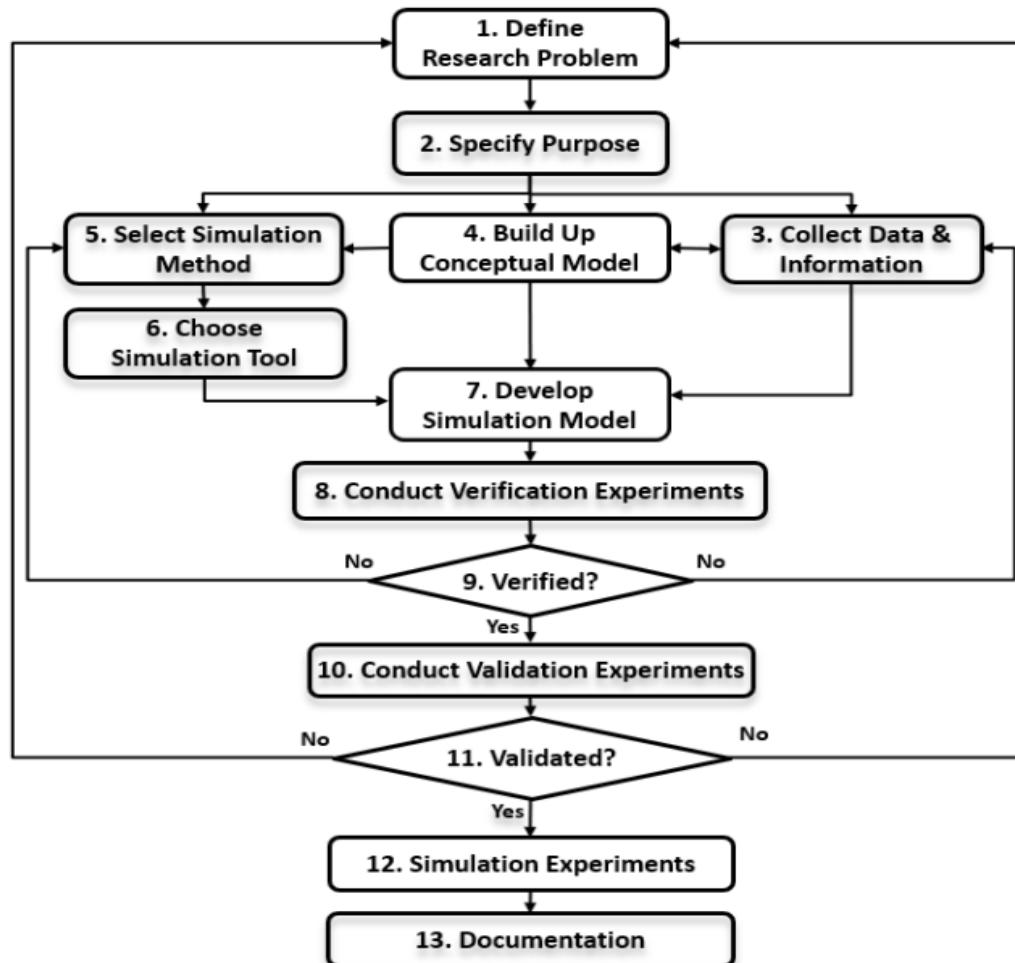
- In any process driven system or software development the verification process verifies if the system or software is being developed as per the requirement.
- The validation process confirms the system or software is developed as per the requirement.
- The verification process is carried out during the different developmental phases of the system or the software.
- The validation process is done at the end of the system or software development.
- 50-70% of the developmental resources are consumed by this process. This process consists of reviews, analysis, checklist generation and testing.

Verification and Validation



Verification and Validation Process





Modeling and simulation procedure

Safety-critical systems

- A system whose functionality affects the safety of the environment, human life, or the nation's interests is a safety critical system.
- More and more systems are safety-critical systems, such as avionics systems, grid cyber-physical systems (GCPSS), automotive systems and medical systems, etc. The failure or mishap of a safety-critical system may lead to the damage of systems or equipment, it may even result at the loss of life and personal injury. It also can give rise to large amounts of economic losses. Thus, safety analysis has attracted much more attention.

Guideline for safety analysis

- Many standards provide the guideline for safety analysis, such as, ARP 4761 , MIL-STD-882D, DO-178C, ISO 26262, IEC 61508, DO-333 , etc. In DO-333, formal methods, are recommended to ensure system safety during the system development. It is necessary to integrate formal methods with safety analysis approaches.

Categories of SCS

- Safety critical software
- Safety critical system
- Safety critical software system

Verification Techniques

- Dynamic Testing
 - Functional testing
 - Structural testing
 - Random testing
- Static testing
 - Consistency techniques
 - Measurement techniques

Validation Techniques

- Formal Method
- Fault Injection
 - Hardware fault injection
 - Software fault injection
- Dependability analysis
 - Hazard analysis
 - Risk analysis

Formal methods and critical systems

- The development of critical systems is one of the ‘success’ stories for formal methods
- Formal methods are mandated in Britain for the development of some types of safety-critical software for defence applications
- There is not currently general agreement on the value of formal methods in critical systems development

Formal methods and validation

- Specification validation
 - Developing a formal model of a system requirements specification forces a detailed analysis of that specification and this usually reveals errors and omissions
 - Mathematical analysis of the formal specification is possible and this also discovers specification problems
- Formal verification
 - Mathematical arguments (at varying degrees of rigour) are used to demonstrate that a program or a design is consistent with its formal specification

Motivation

- For every safety critical system development verification and validation plays an important role in assuring safety. The output of this process gives us the satisfactory fulfillment of the safety requirements.
- To make automation in verification and validation, formal method is introduced as it is cost effective.
- As testing cannot complement replaced by formal method so combining formal method and testing is a cost effective approach for system verification and validation.
- A lot of testing methods are available such as Ontology based testing, Model-based testing, Regression testing etc.

Safety proofs

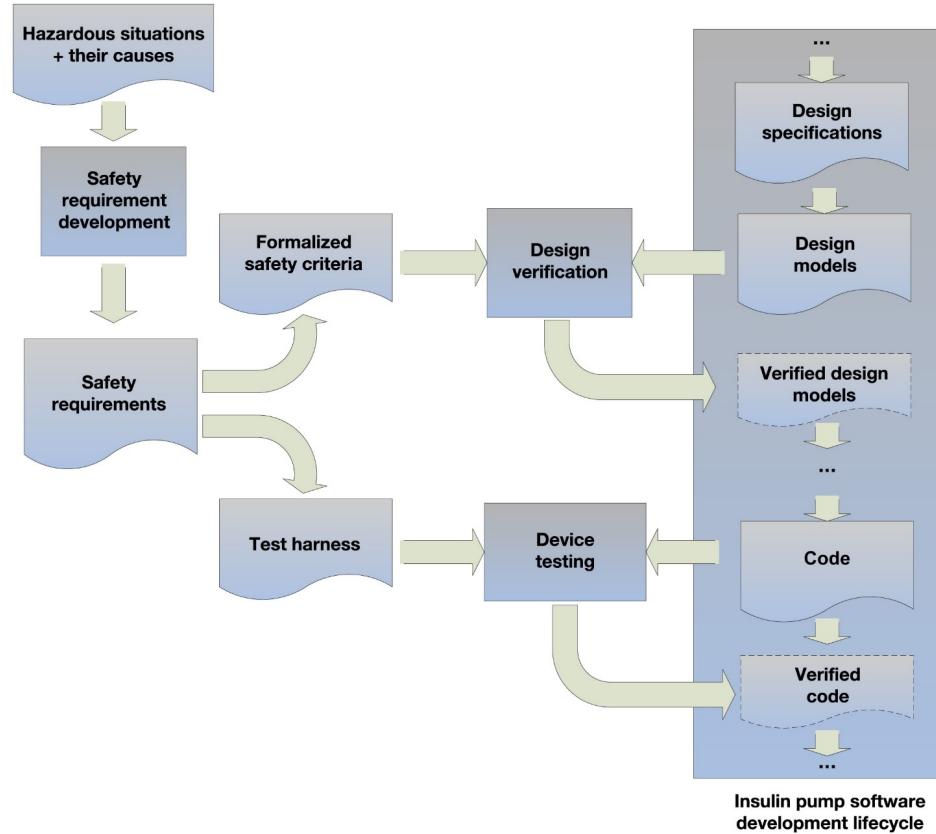
- Safety proofs are intended to show that the system cannot reach an unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code
- May be displayed graphically

Construction of a safety proof

- Establish the safe exit conditions for a component or a program
- Starting from the END of the code, work backwards until you have identified all paths that lead to the exit of the code
- Assume that the exit condition is false
- Show that, for each path leading to the exit that the assignments made in that path contradict the assumption of an unsafe exit from the component

Use Case:

Insulin pump systems

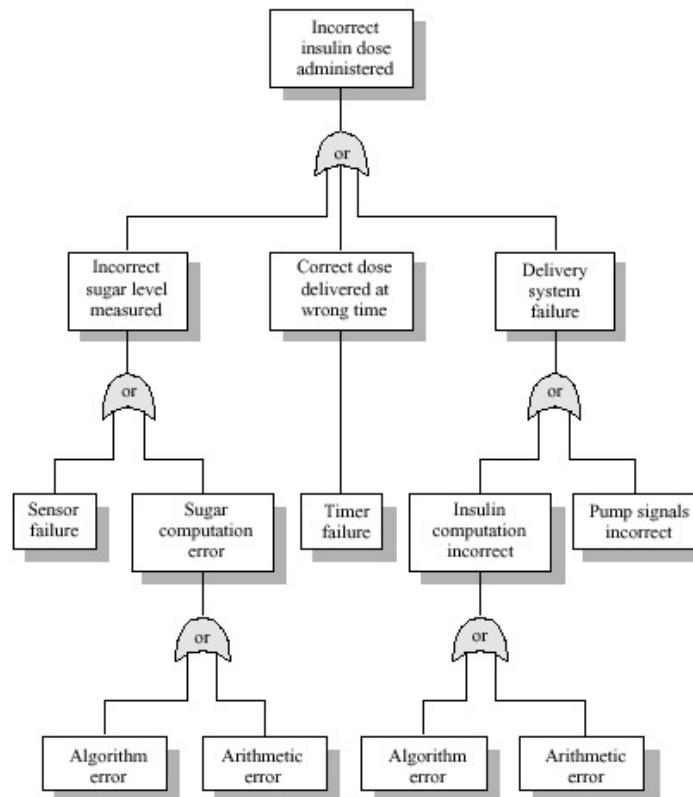


Integrating safety requirements into software development lifecycle.

Insulin system hazards

- insulin overdose or under dose (biological)
- power failure (electrical)
- machine interferes electrically with other medical equipment such as a heart pacemaker (electrical)
- parts of machine break off in patient's body(physical)
- infection caused by introduction of machine (biol.)
- allergic reaction to the materials or insulin used in the machine (biol).

Fault tree for software hazards



Safety proofs

- Safety proofs are intended to show that the system cannot reach an unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code

Insulin delivery system

- Safe state is a shutdown state where no insulin is delivered
 - If hazard arises, shutting down the system will prevent an accident
- Software may be included to detect and prevent hazards such as power failure
- Consider only hazards arising from software failure
 - Arithmetic error The insulin dose is computed incorrectly because of some failure of the computer arithmetic
 - Algorithmic error The dose computation algorithm is incorrect

Arithmetic errors

- Use language exception handling mechanisms to trap errors as they arise
- Use explicit error checks for all errors which are identified
- Avoid error-prone arithmetic operations (multiply and divide). Replace with add and subtract
- Never use floating-point numbers
- Shut down system if exception detected (safe state)

Algorithmic errors

- Harder to detect than arithmetic errors. System should always err on the side of safety.
- Use reasonableness checks for the dose delivered based on previous dose and rate of dose change.
- Set maximum delivery level in any specified time period.
- If computed dose is very high, medical intervention may be necessary anyway because the patient may be ill.

Procedure for Insulin delivery code

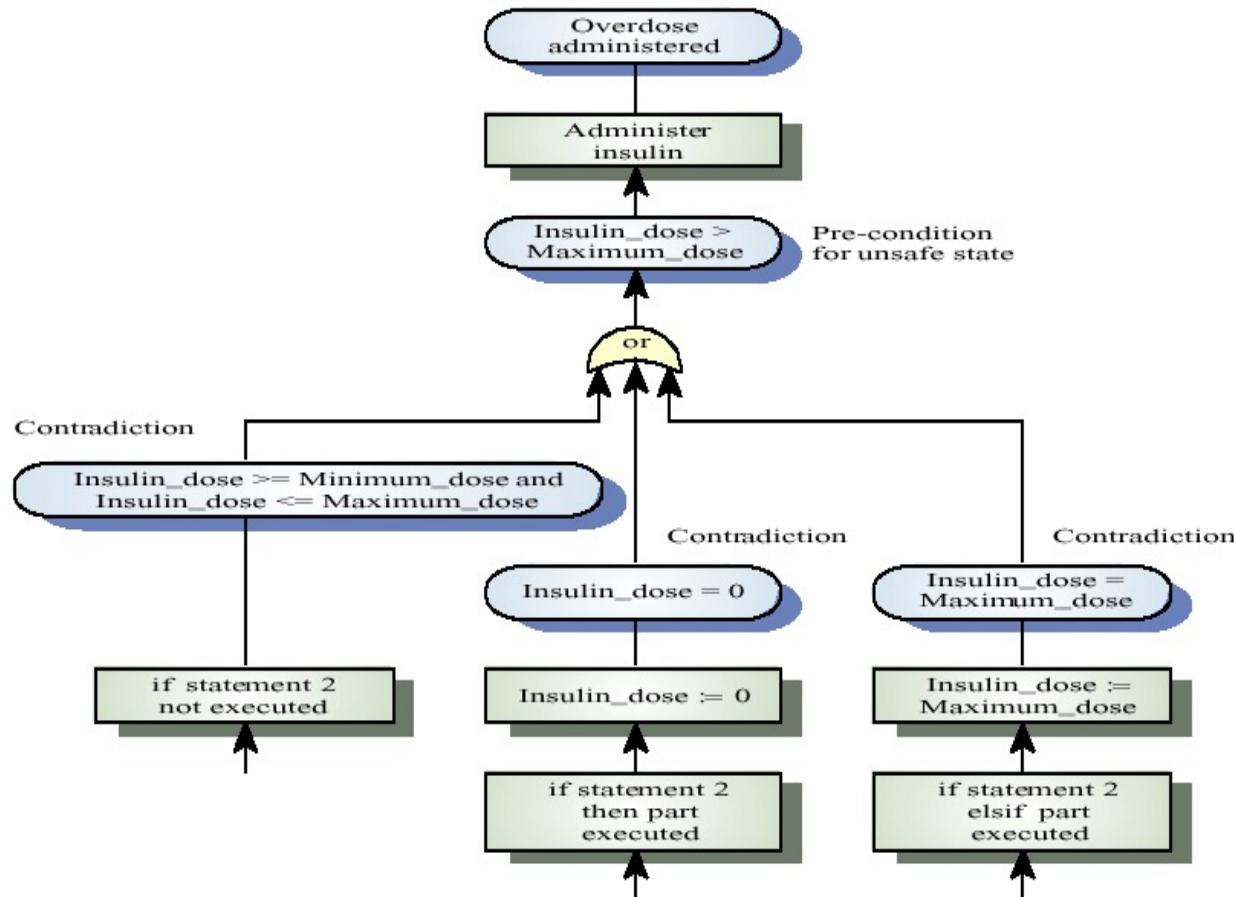
```
// The insulin dose to be delivered is a function of blood sugar level, the previous dose
// delivered and the time of delivery of the previous dose

    currentDose = computeInsulin () ;

    // Safety check - adjust currentDose if necessary

    if (previousDose == 0) {                                // if statement 1
        if (currentDose > 16)
            currentDose = 16 ;
    }
    else
        if (currentDose > (previousDose * 2) )
            currentDose = previousDose * 2 ;
    if ( currentDose < minimumDose )                      // if statement 2
        currentDose = 0 ;                                  // then branch
    else if ( currentDose > maxDose )                    // else branch
        currentDose = maxDose ;
    administerInsulin (currentDose) ;
```

Informal safety proof Fault tree



System testing

- System testing of the software has to rely on simulators for the sensor and the insulin delivery components.
- Test for normal operation using an operational profile. Can be constructed using data gathered from existing diabetics.
- Testing has to include situations where rate of change of glucose is very fast and very slow.
- Test for exceptions using the simulator.

Safety assertions

- Predicates included in the program indicating conditions which should hold at that point.
- May be based on pre-computed limits e.g. number of insulin pump increments in maximum dose.
- Used in formal program inspections or may be pre-processed into safety checks that are executed when the system is in operation.

Safety assertions

```
static void administerInsulin ( ) throws SafetyException {  
    int maxIncrements = InsulinPump.maxDose / 8 ;  
    int increments = InsulinPump.currentDose / 8 ;  
    // assert currentDose <= InsulinPump.maxDose  
    if (InsulinPump.currentDose > InsulinPump.maxDose)  
        throw new SafetyException (Pump.doseHigh);  
    else  
        for (int i=1; i<= increments; i++) {  
            generateSignal () ;  
            if (i > maxIncrements)  
                throw new SafetyException ( Pump.incorrectIncrements);  
        } // for loop  
} //administerInsulin
```

References

1. Singh N.K., Wang H., Lawford M., Maibaum T.S.E., Wassyng A. (2015) Stepwise Formal Modelling and Reasoning of Insulin Infusion Pump Requirements. In: Duffy V. (eds) Digital Human Modeling. Applications in Health, Safety, Ergonomics and Risk Management: Ergonomics and Health. DHM 2015. Lecture Notes in Computer Science, vol 9185. Springer, Cham. https://doi.org/10.1007/978-3-319-21070-4_39
2. Zhang, Yi & Jetley, Raoul & Jones, Paul & Ray, Arnab. (2011). Generic Safety Requirements for Developing Safe Insulin Pump Software. Journal of diabetes science and technology. 5. 1403-19. 10.1177/193229681100500612.

Tools and Languages

- Languages used are System Process Engineering Meta-model Specification Version 2.0, UML, TLA+, Z, Petri Nets (PNs) etc.
- To make automation many tools have already been developed known as UPPAAL, TIMES, PRISM, KLEE, Frama-C, Prema, ASSERT™ etc.

Thank You

Q & A

Model Verification and Validation Strategies and Methods

Presented By

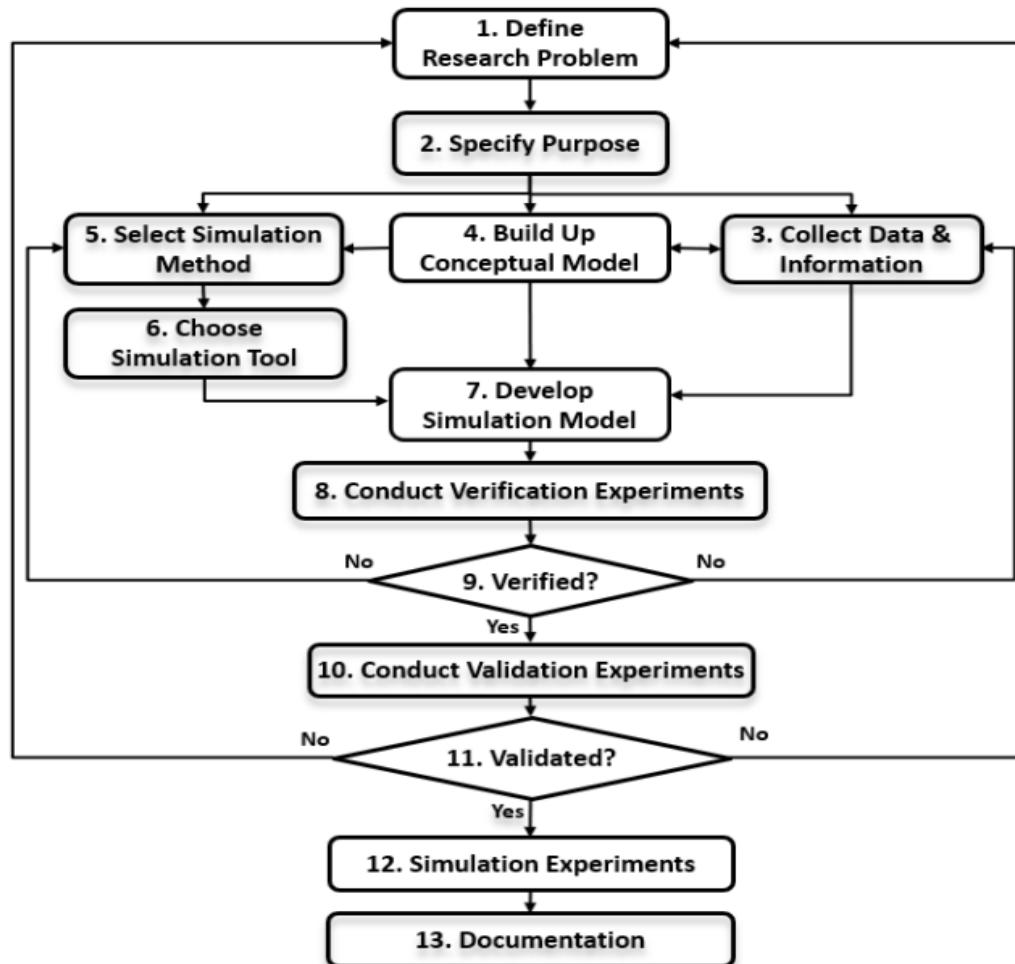
Madhusmita Das

Guided By

Dr. Biju R. Mohan

Introduction

- Model verification and validation involves a series of strategies, methods and activities that are an integral part of the simulation model design and development process.
- Model verification deals with the identification and removal of errors in the model by comparing simulation model outcomes to practical solutions from the real-world situation .
- Model validation determines how accurate the simulation model is as a representation of a real-world system for the simulation purpose.

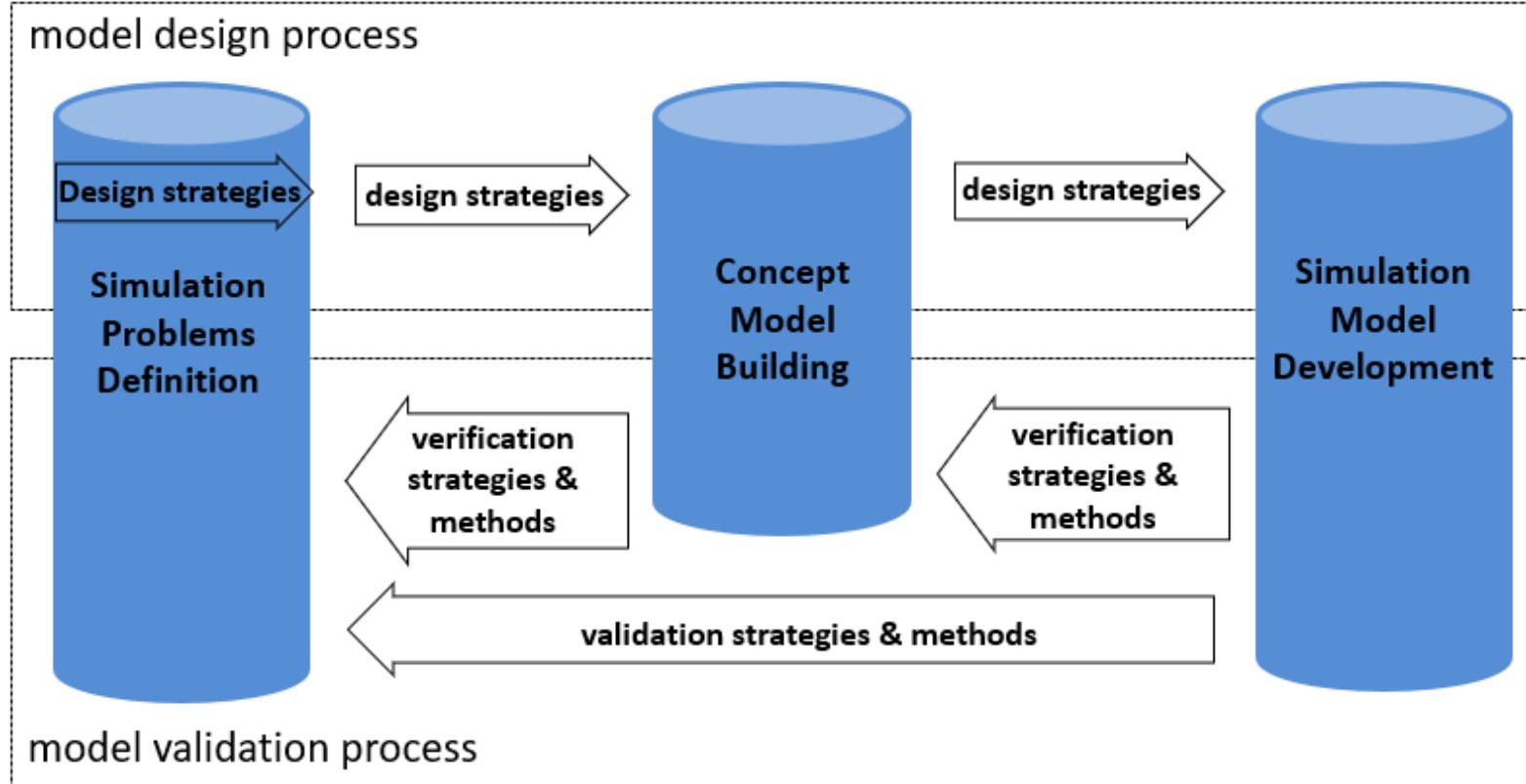


Modeling and simulation procedure

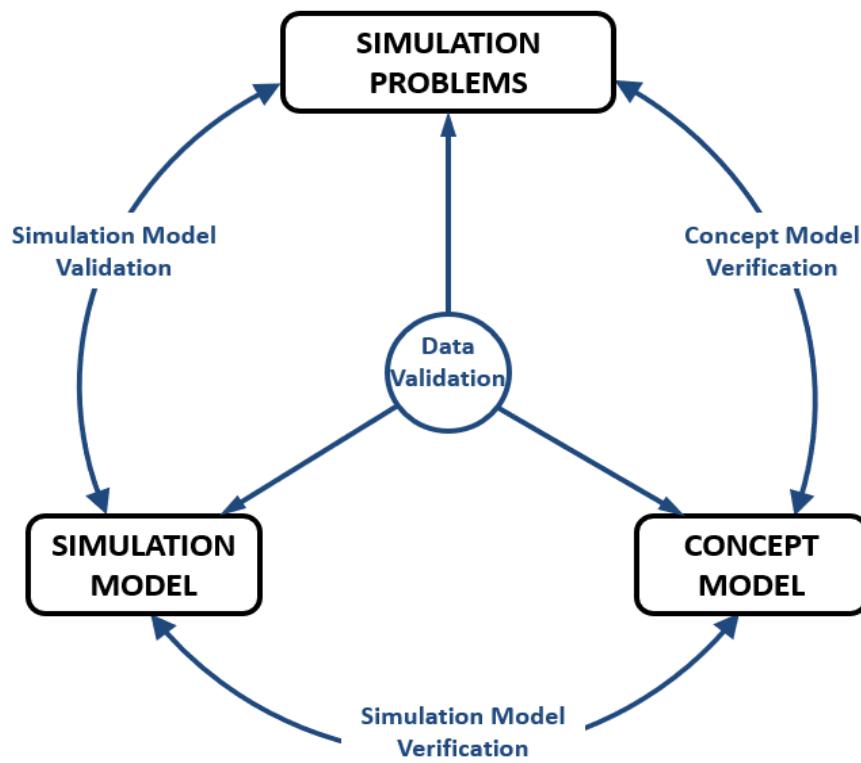
MODEL DESIGN AND VALIDATION PROCESSES

There are three research stages:

- Simulation problems definition
- Concept model building and
- Simulation model development



Model design and validation processes



Model verification and validation architecture

Validation types

- Animation Validation
- Model to Model Validation
- Event Validation
- Extreme Condition Validation
- Face Validation
- Historical Data Validation
- Operational Graphics Validation
- Sensitivity Analysis Validation
- Predictive Validation
- Traces Validation
- Turing Test Validation

Model Verification and Validation Methods Characteristics

Each validation method has distinct characteristics which make them suitable for different real-world simulation purposes and validation criteria :

Animation Validation: Simulation model operational behavior of each element is graphically displayed as the model runs over time.

Model to Model Validation: Outcomes of a simulation model being validated are compared with outcomes from another valid model, related to the same simulation problems.

Event Validation: Event occurrences in the simulation model are compared to those of the real-world system to determine how similar they are.

Extreme Condition Validation: The simulation model architecture and outputs are tested using extreme and unlikely combinations in the real-world.

Model Verification and Validation Methods Characteristics

Face Validation: Asking knowledgeable professionals about the system and whether the simulation model and its behaviors are reasonable .

Historical Data Validation: If historical data exists, some of the data is used to build the simulation model and the remaining data is used to determine whether the model performs as the system does .

Operational Graphics Validation: Observing entities' operational curves in simulation outputs, to determine whether the model performance is reasonable with respect to the real-world scenario .

Sensitivity Analysis Validation: This method consists of changing the values of the input data and parameters of a simulation model to determine the effect upon the performance of the model and its output. The same relationship should occur in the simulation model as in the real-world system.

Model Verification and Validation Methods Characteristics

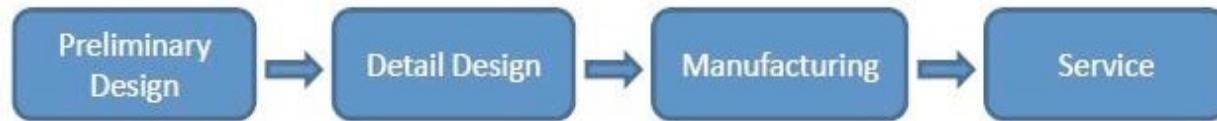
Predictive Validation: The simulation model is used to predict the system performance, and comparisons are made between the performance produced from the system and the forecast from the model, in order to determine if they are the same or similar enough.

Traces Validation: The behaviors of different types of specific entities in the simulation model are traced through the simulation model operation to determine whether the model's logic is correct and the necessary accuracy is obtained.

Turing Test Validation: Individuals who are knowledgeable about the real-world operational system are asked whether they can discriminate between outputs from the real-world system and the simulation model

MODEL VERIFICATION AND VALIDATION CASE STUDY

- A manufacturing operation system case study from a large UK-based manufacturing company was used in the research.
- The aim of the simulation case study was to develop a computer-based simulation model representing the manufacturing operation system, and examine the system performance under a range of different operating conditions .



Manufacturing operation system case study structure

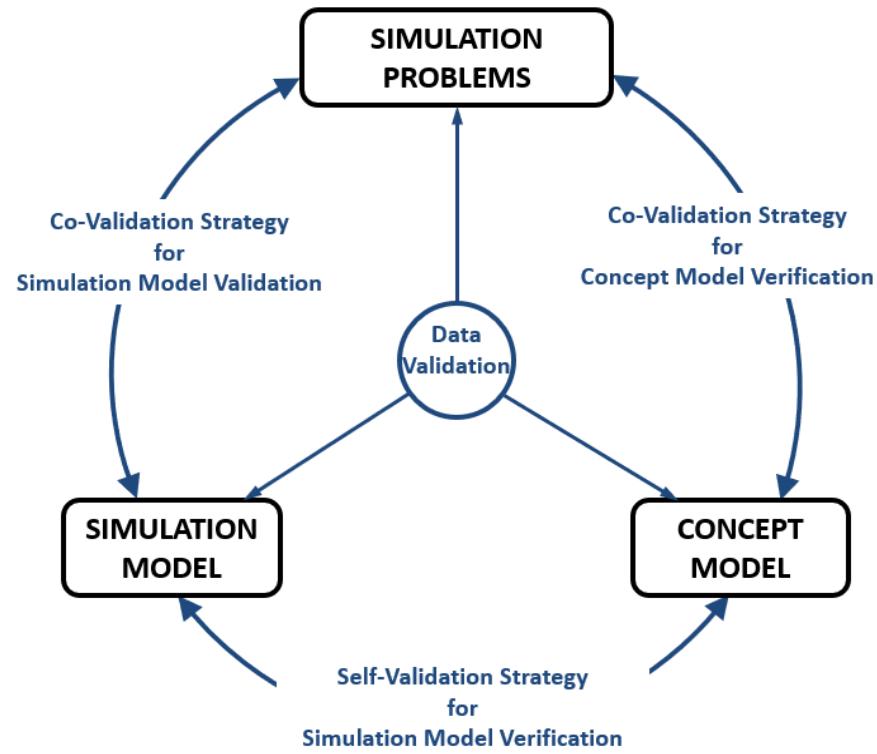


Simulation case study model verification and validation

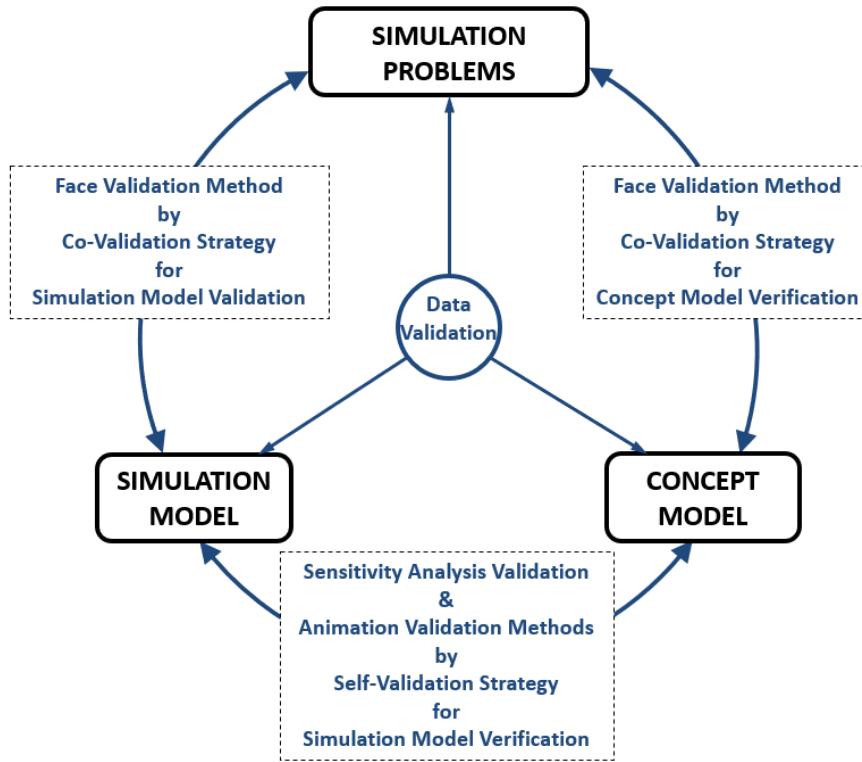
APPLICATION MODEL VERIFICATION AND VALIDATION STRATEGIES IN SIMULATION CASE STUDY

There are four primary verification and validation strategies used in model validation processes :

- Self-Validation
- Co-Validation
- Independent Validation
- Scoring Validation



Application model verification and validation strategies



Application model verification and validation methods

Summary

- A modeling and simulation procedure was demonstrated with focus on a real-world problem-solving case study. The procedure accommodates model verification & validation activities into model design & development process which, as a whole, forms a full cycle of modeling and simulation.
- The contribution of this paper is to bridge the gap between model design and development, and model verification and validation by providing such a modeling and simulation procedure.

References

- Yin, Chenggang & Mckay, Alison. (2018). Model verification & validation strategies and methods: an application case study.
- Yin, Chenggang & Mckay, Alison. (2018). Introduction to Modeling and Simulation Techniques.

Model-Based Simulation of Integrated Software Systems

- The main characteristic of an ISS is that it is made up of several large-scale and heterogeneous subsystems that have been developed by different suppliers and later put together to provide a set of advanced functions. To ensure that ISSs behave as intended and meet their functional, performance, and robustness requirements, these systems are subject to extensive Verification and Validation (V&V).
- A key aspect of V&V for an ISS is simulation-based testing, which is aimed at the detection of defects at design time, and before the ISS has been operationalized. So, we aim to develop an automated and effective technique for testing systems.
- To address this problem, we plan to precisely define what constitutes the state of a resource, and what are the dependencies between the test cases and the resources, that is, the "requires" and "taints" relationships. We further need a mechanism, for example, a rule language, to explicitly express the preconditions under which a test case can be run. Finally, we intend to develop a search-based optimization algorithm to compute an optimized ordering for running (a subset of) test procedures.

Thank You

Q & A

Class 2

Software Processes

An overview of conventional software process models

Objectives

- To introduce software process models
- To describe three generic process models and when they may be used
- To describe outline process models for requirements engineering, software development, testing and evolution

Topics covered

- Software process models
- Process iteration
- Process activities

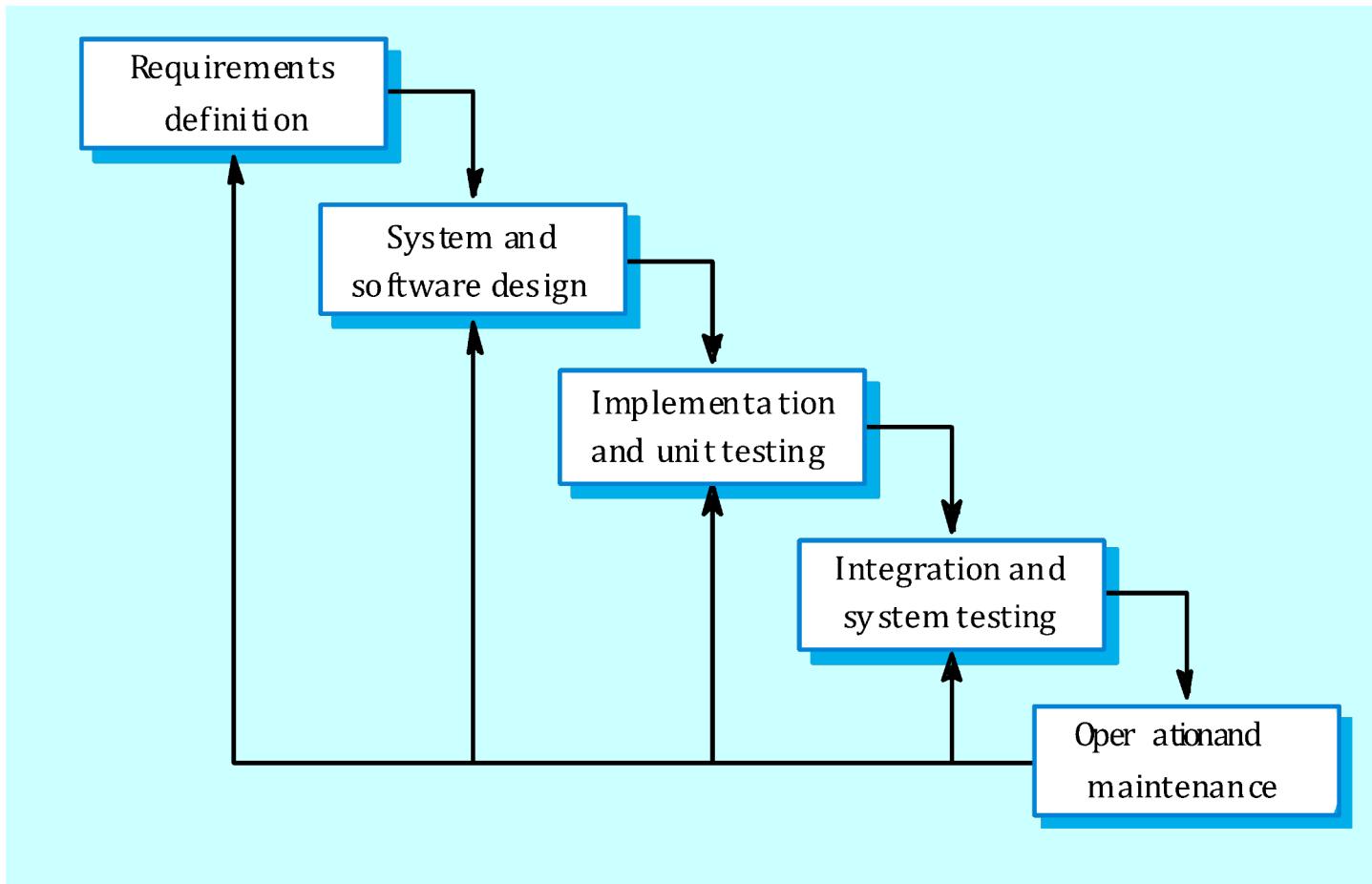
The software process

- A structured set of activities required to develop a software system
 - Specification;
 - Design;
 - Validation;
 - Evolution.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Generic software process models

- The waterfall model
 - Separate and distinct phases of specification and development.
- Evolutionary development
 - Specification, development and validation are interleaved.
- Component-based software engineering
 - The system is assembled from existing components.
- There are many variants of these models e.g. formal development where a waterfall-like process is used but the specification is a formal specification that is refined through several stages to an implementable design.

Waterfall model



Waterfall model phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.

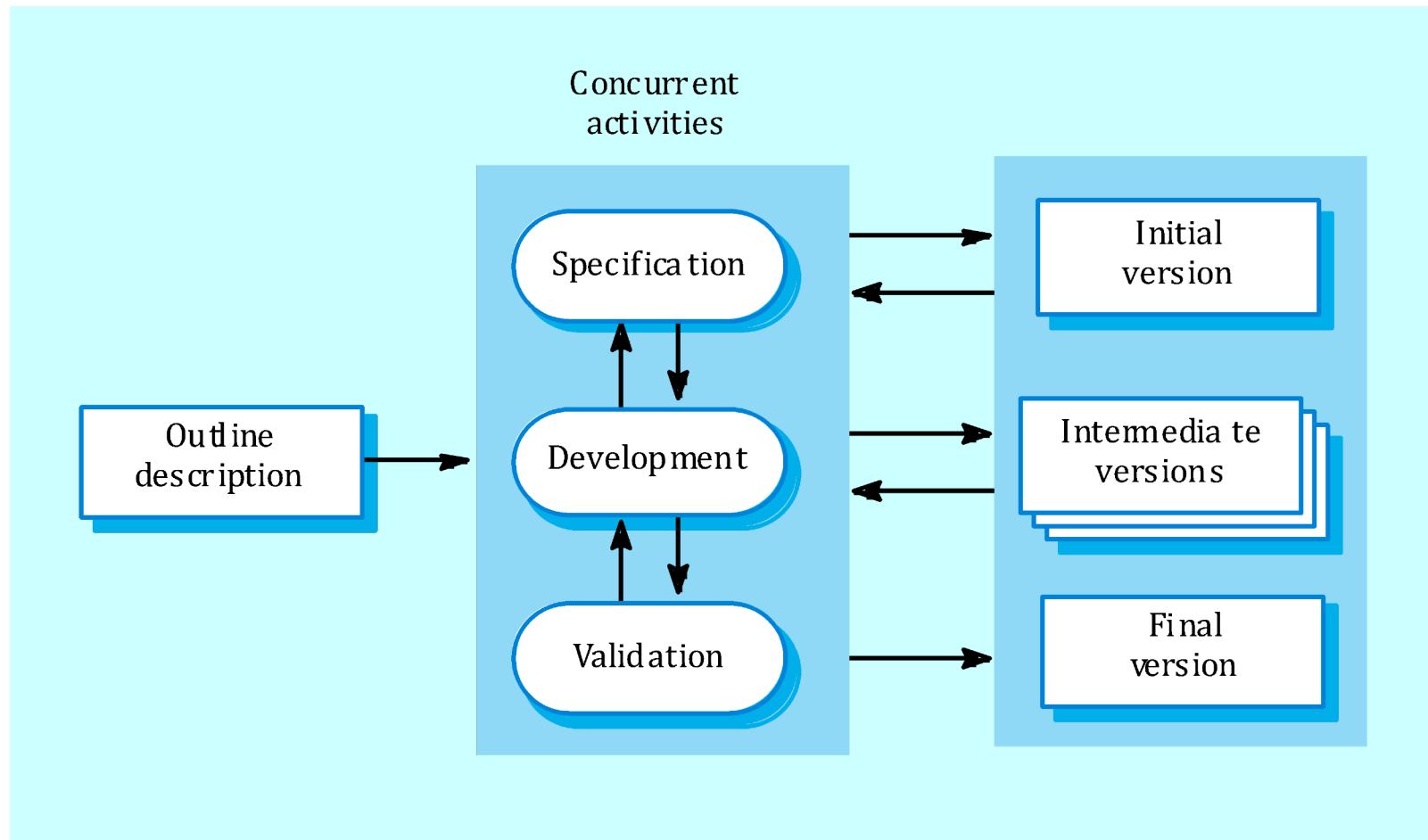
Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

Evolutionary development

- Exploratory development
 - Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements and add new features as proposed by the customer.
- Throw-away prototyping
 - Objective is to understand the system requirements. Should start with poorly understood requirements to clarify what is really needed.

Evolutionary development



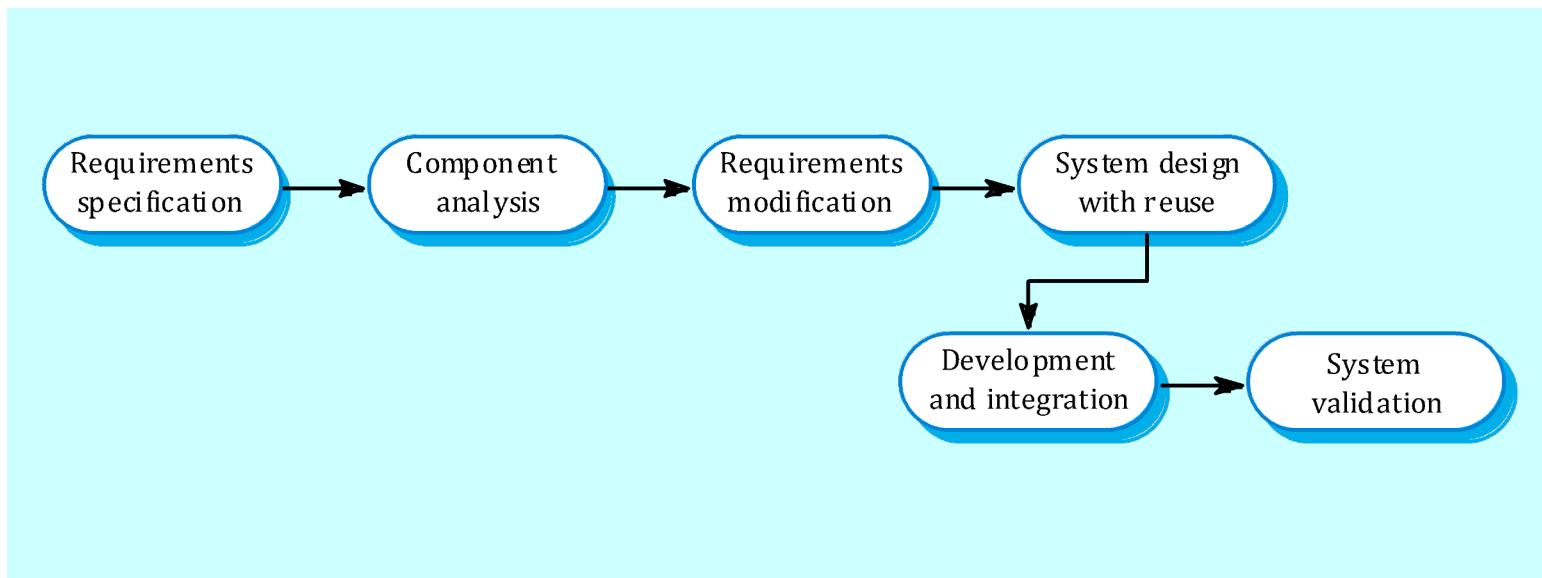
Evolutionary development

- Problems
 - Lack of process visibility;
 - Systems are often poorly structured;
 - Special skills (e.g. in languages for rapid prototyping) may be required.
- Applicability
 - For small or medium-size interactive systems;
 - For parts of large systems (e.g. the user interface);
 - For short-lifetime systems.

Component-based software engineering

- Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
 - Component analysis;
 - Requirements modification;
 - System design with reuse;
 - Development and integration.
- This approach is becoming increasingly used as component standards have emerged.

Reuse-oriented development



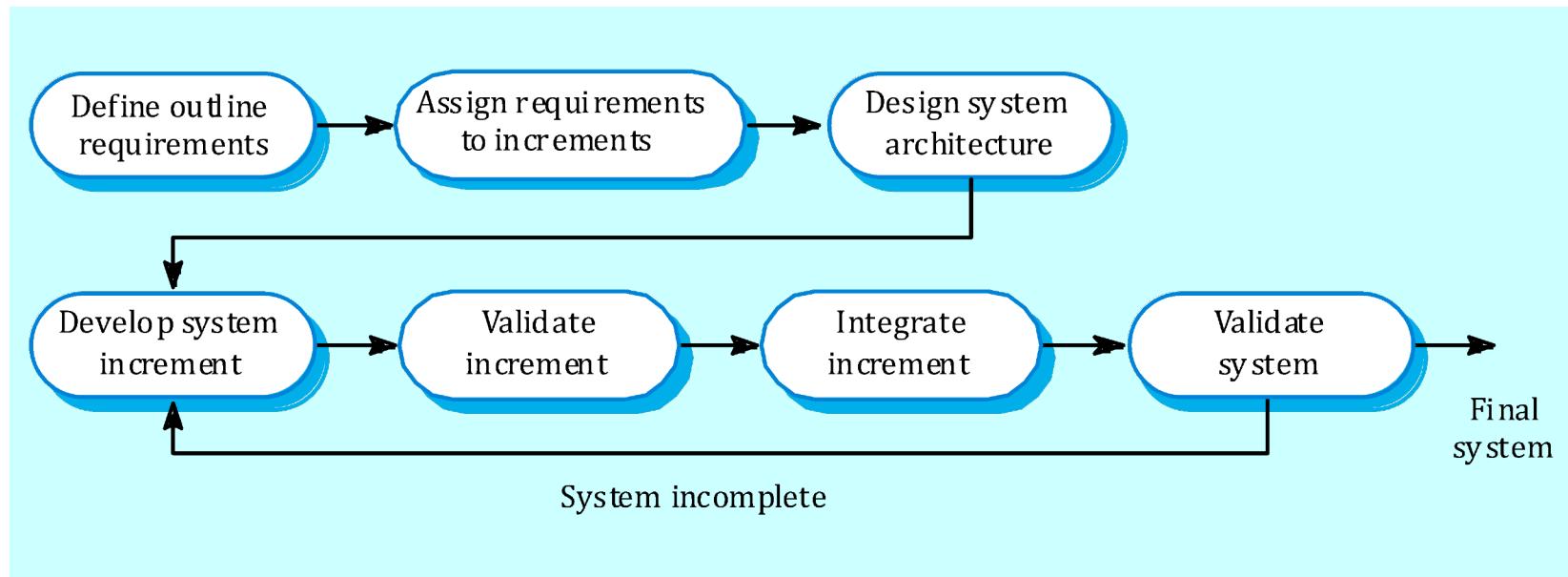
Process iteration

- System requirements **ALWAYS** evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems.
- Iteration can be applied to any of the generic process models.
- Two (related) approaches
 - Incremental delivery;
 - Spiral development.

Incremental delivery

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

Incremental development



Incremental development advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

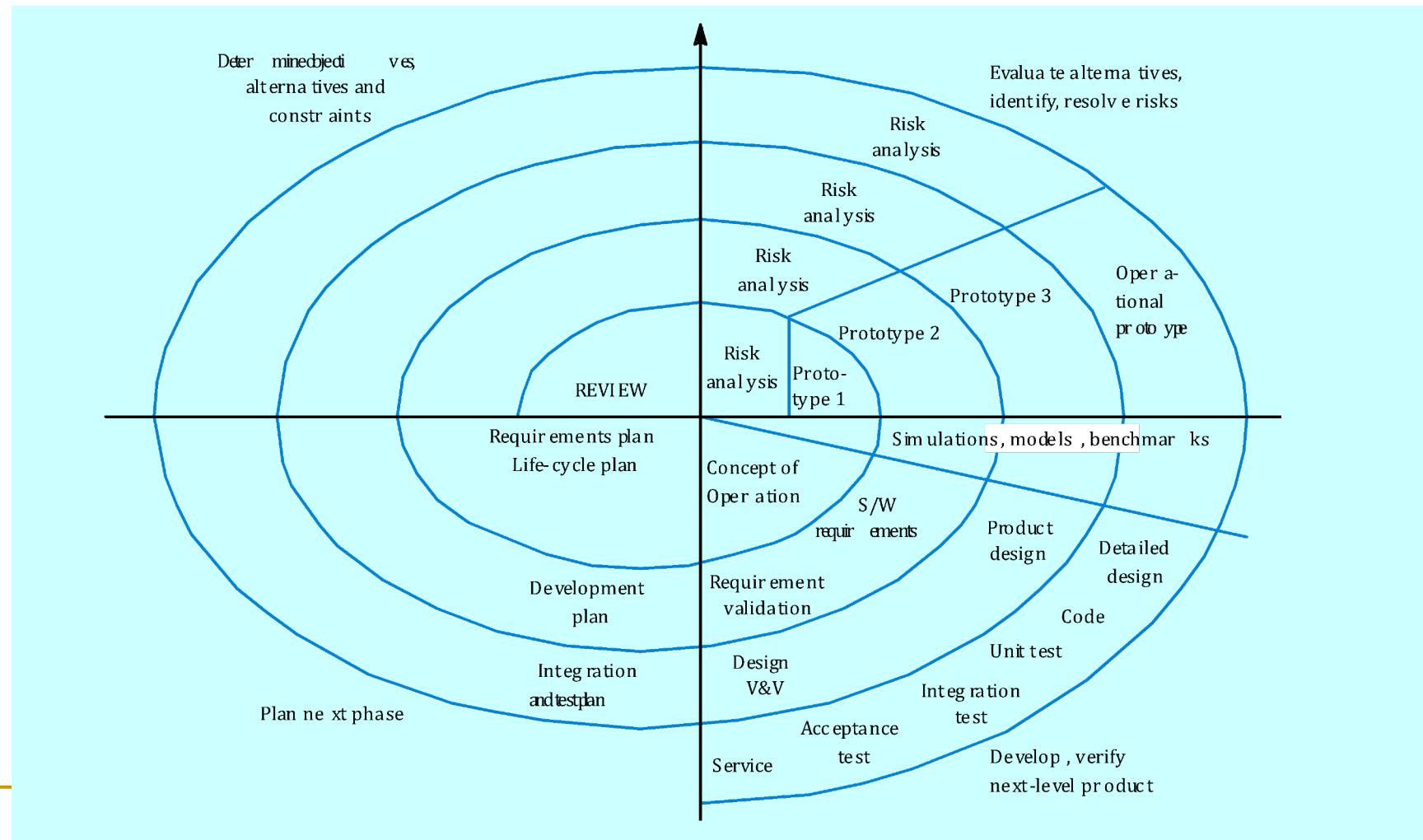
Extreme programming

- An approach to development based on the development and delivery of very small increments of functionality.
- Relies on constant code improvement, user involvement in the development team and pairwise programming.
- Covered in Chapter 17

Spiral development

- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

Spiral model of the software process



Spiral model sectors

- Objective setting
 - Specific objectives for the phase are identified.
- Risk assessment and reduction
 - Risks are assessed and activities put in place to reduce the key risks.
- Development and validation
 - A development model for the system is chosen which can be any of the generic models.
- Planning
 - The project is reviewed and the next phase of the spiral is planned.

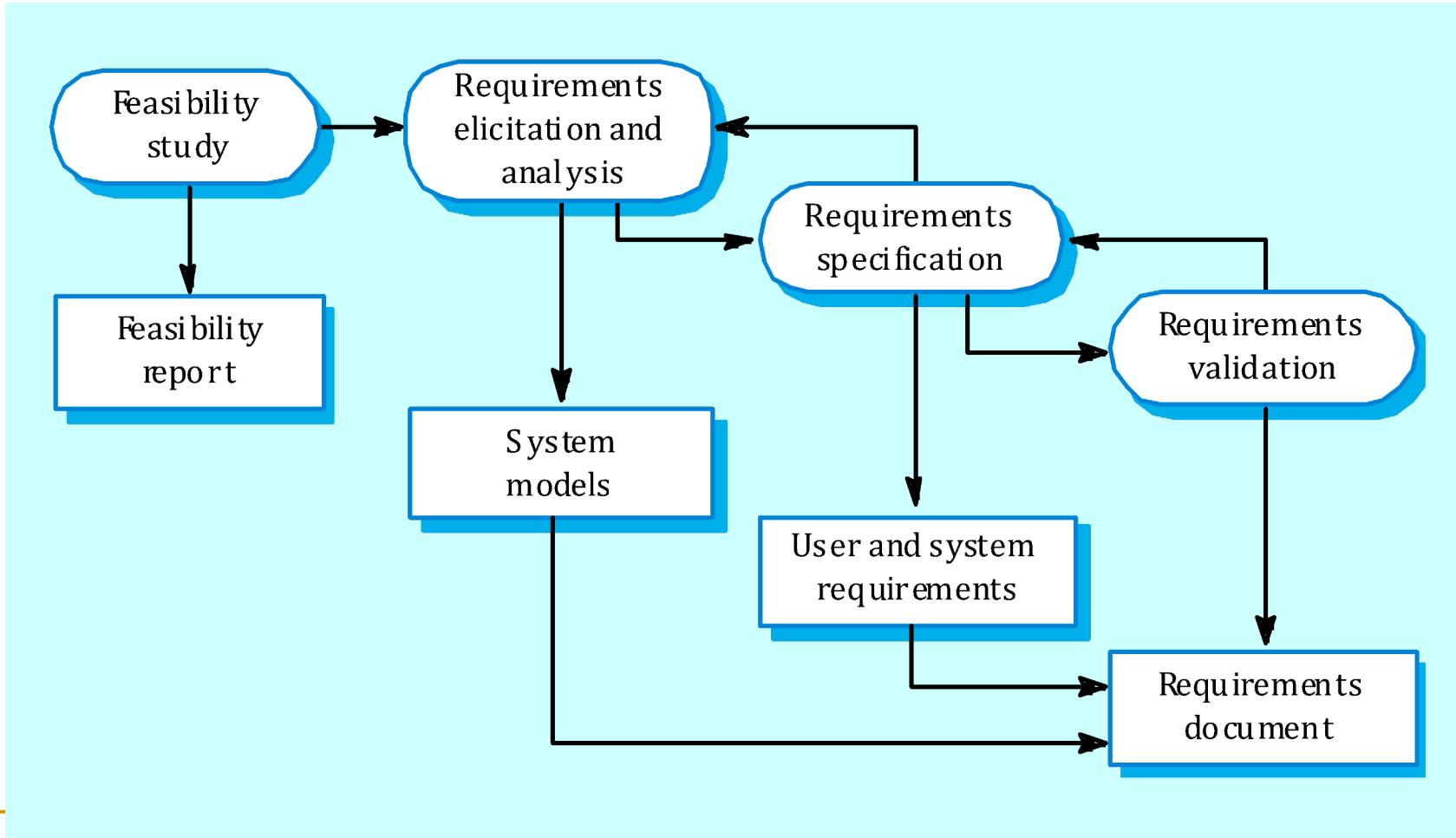
Process activities

- Software specification
- Software design and implementation
- Software validation
- Software evolution

Software specification

- The process of establishing what services are required and the constraints on the system's operation and development.
- Requirements engineering process
 - Feasibility study;
 - Requirements elicitation and analysis;
 - Requirements specification;
 - Requirements validation.

The requirements engineering process



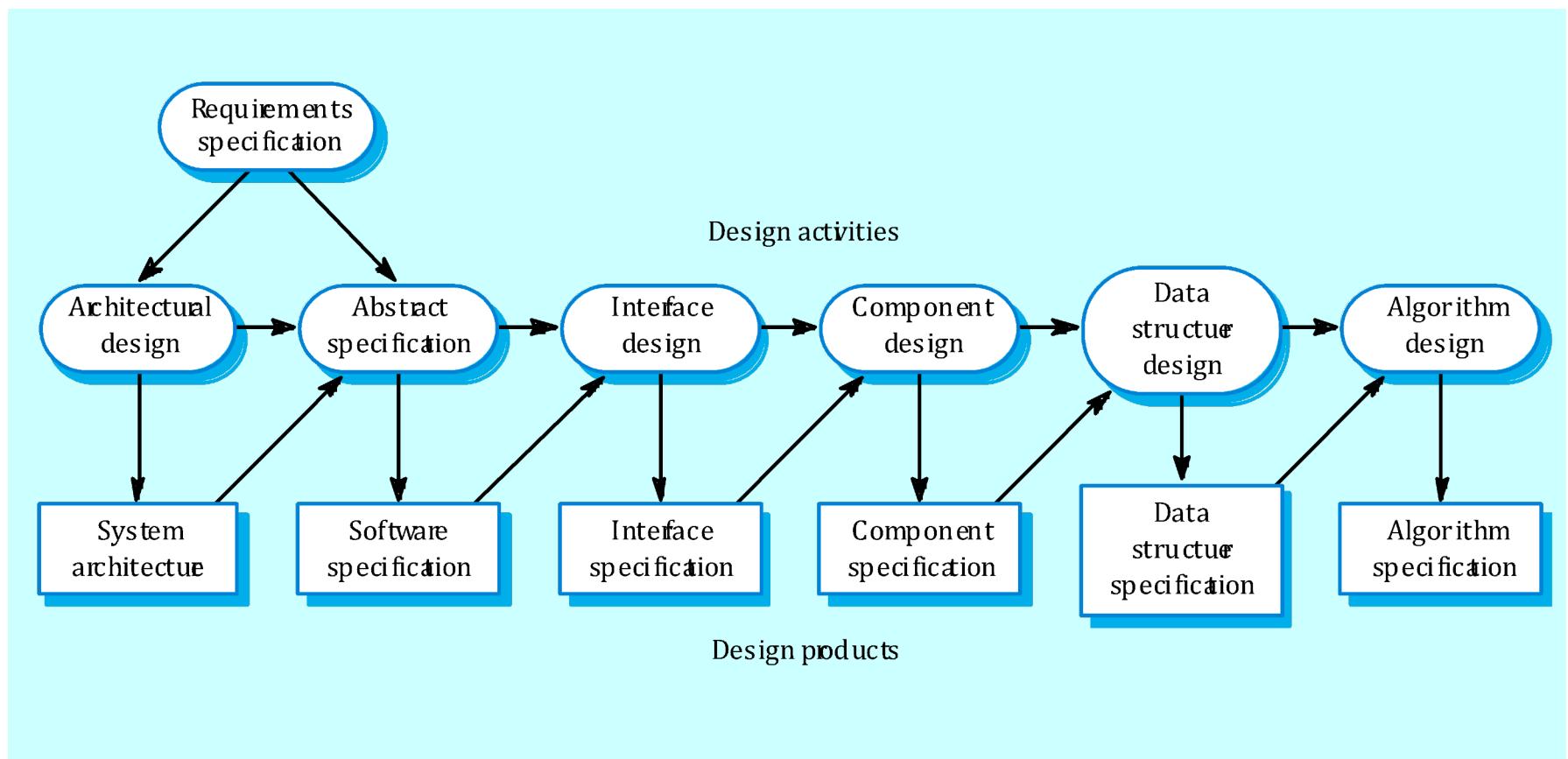
Software design and implementation

- The process of converting the system specification into an executable system.
- Software design
 - Design a software structure that realises the specification;
- Implementation
 - Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

Design process activities

- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design

The software design process



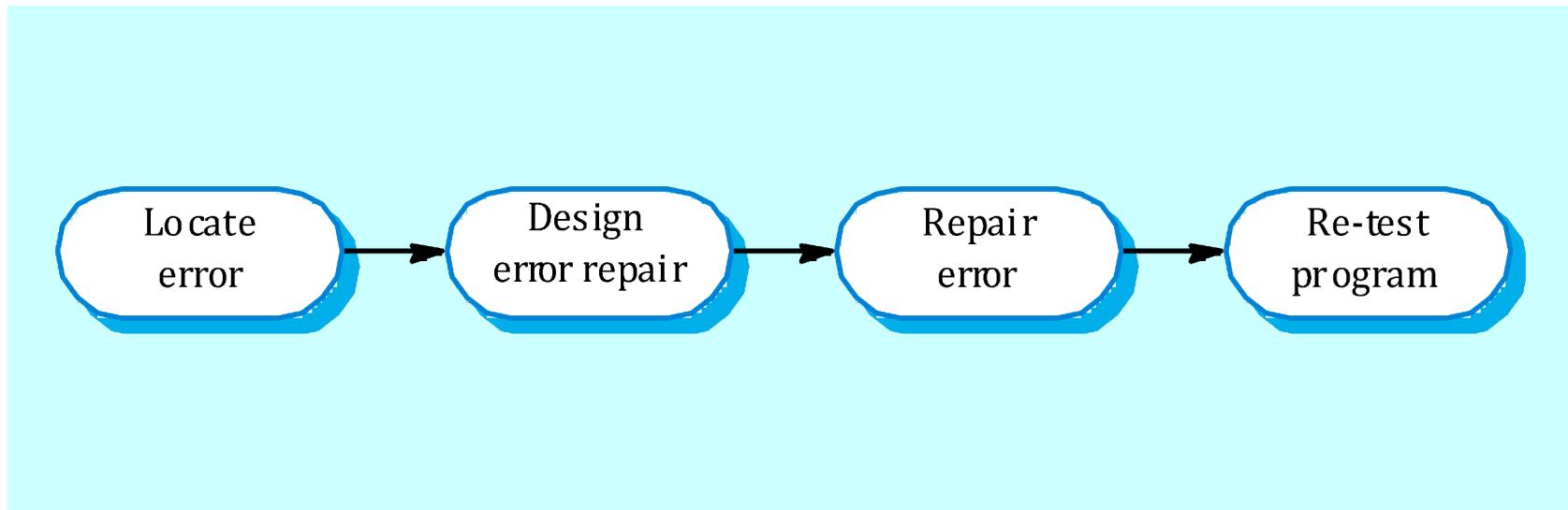
Structured methods

- Systematic approaches to developing a software design.
- The design is usually documented as a set of graphical models.
- Possible models
 - Object model;
 - Sequence model;
 - State transition model;
 - Structural model;
 - Data-flow model.

Programming and debugging

- Translating a design into a program and removing errors from that program.
- Programming is a personal activity - there is no generic programming process.
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process.

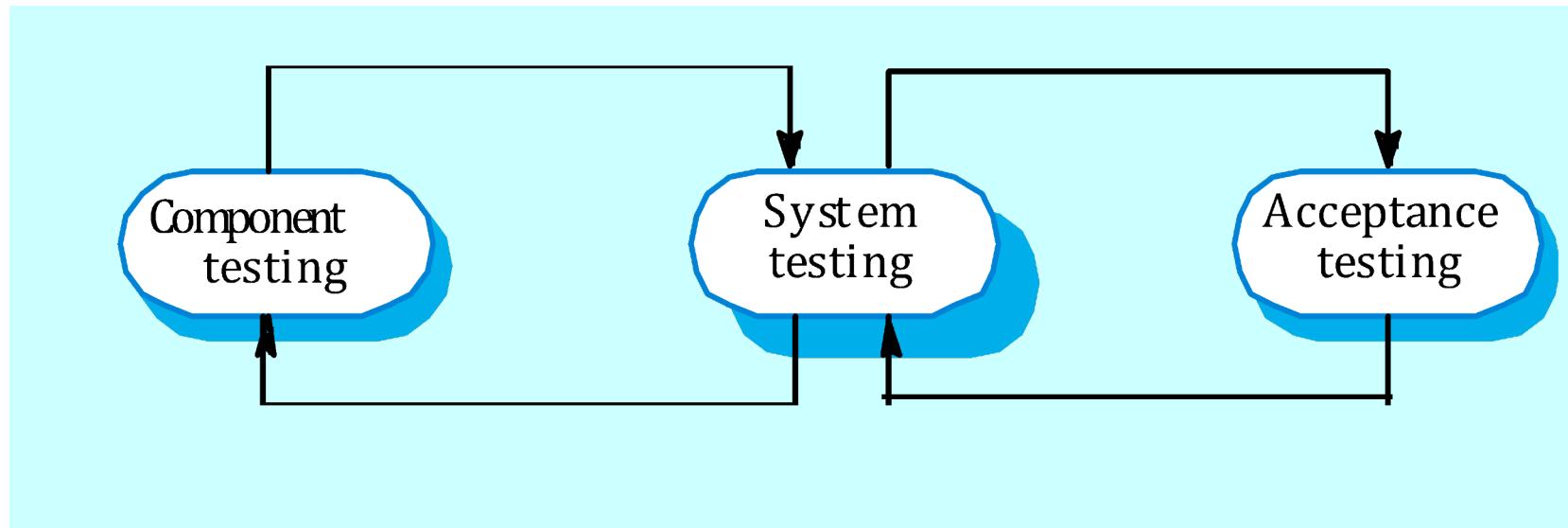
The debugging process



Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

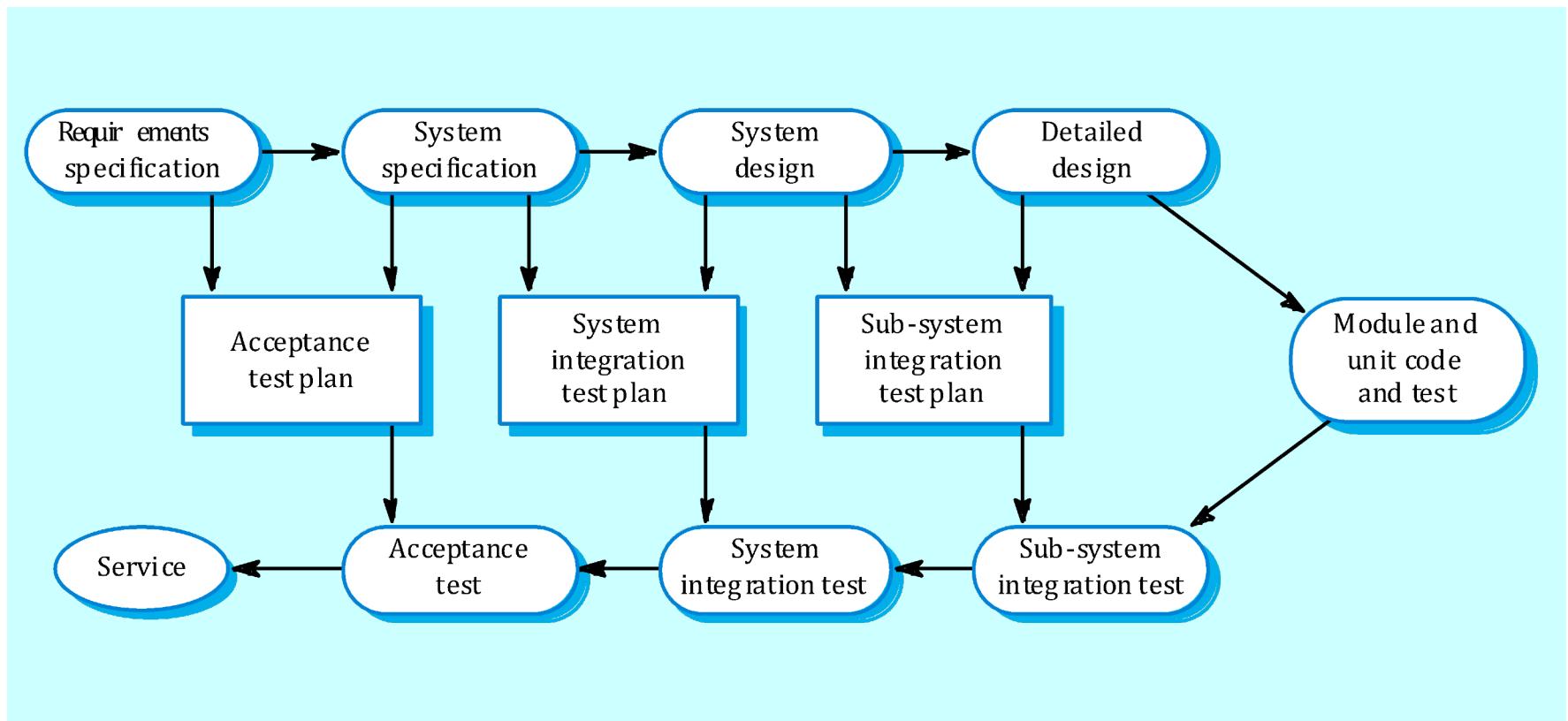
The testing process



Testing stages

- Component or unit testing
 - Individual components are tested independently;
 - Components may be functions or objects or coherent groupings of these entities.
- System testing
 - Testing of the system as a whole. Testing of emergent properties is particularly important.
- Acceptance testing
 - Testing with customer data to check that the system meets the customer's needs.

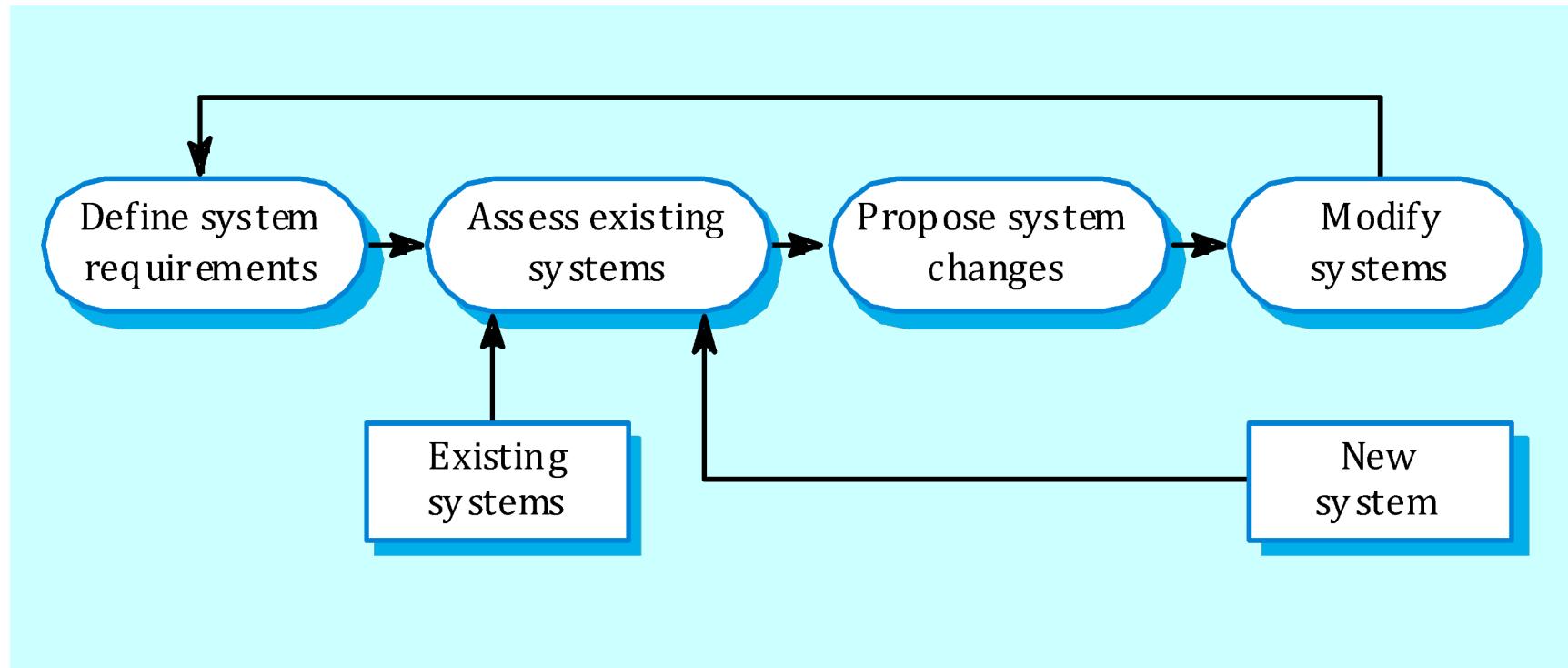
Testing phases



Software evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

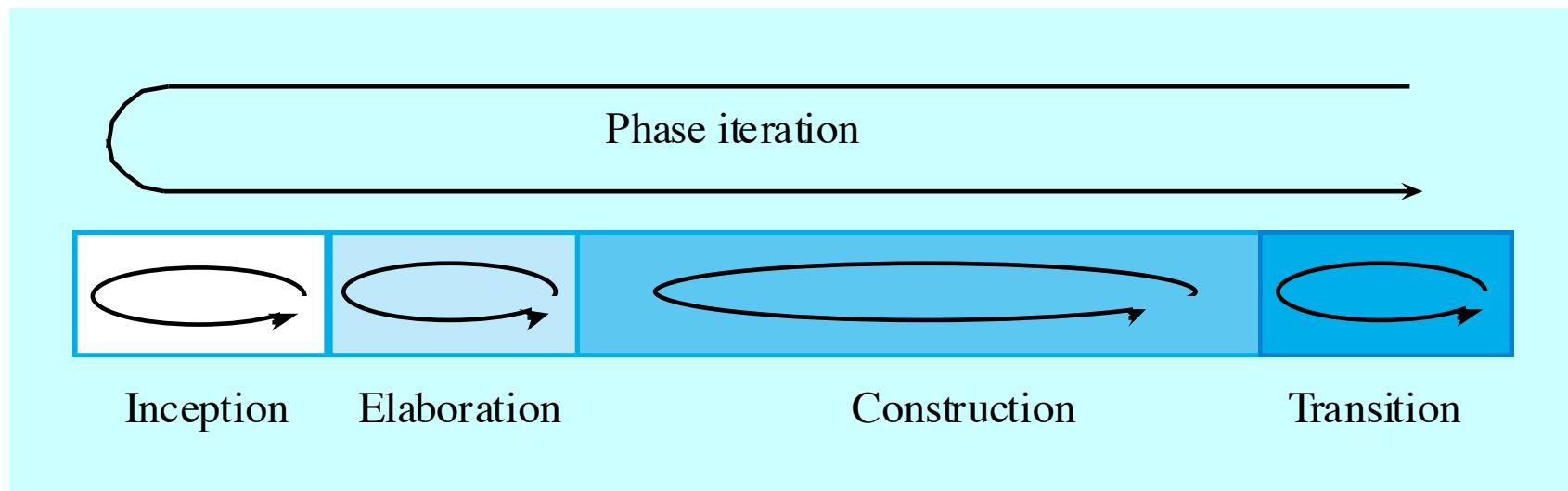
System evolution



The Rational Unified Process

- A modern process model derived from the work on the UML and associated process.
- Normally described from 3 perspectives
 - A dynamic perspective that shows phases over time;
 - A static perspective that shows process activities;
 - A practice perspective that suggests good practice.

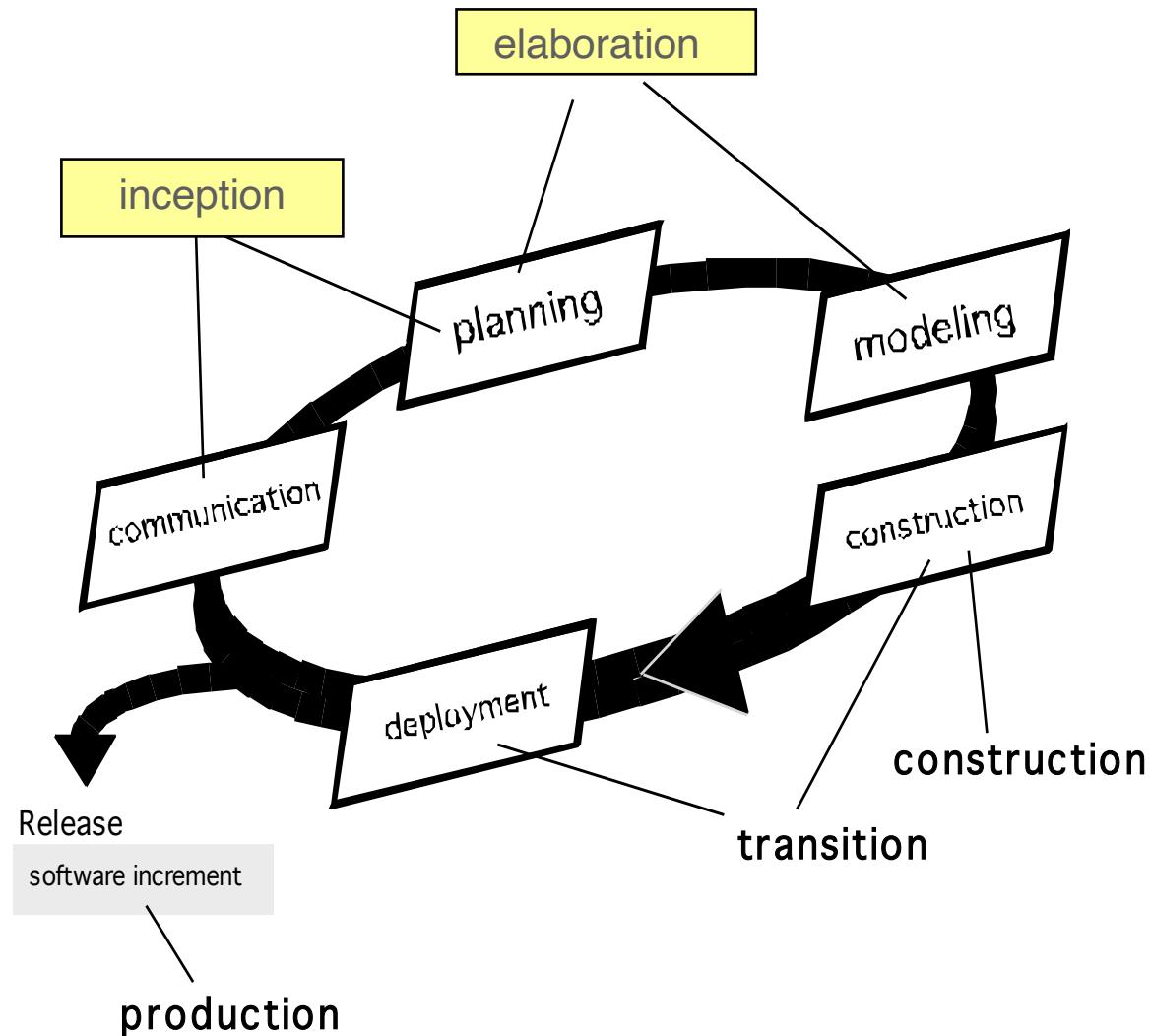
RUP phase model



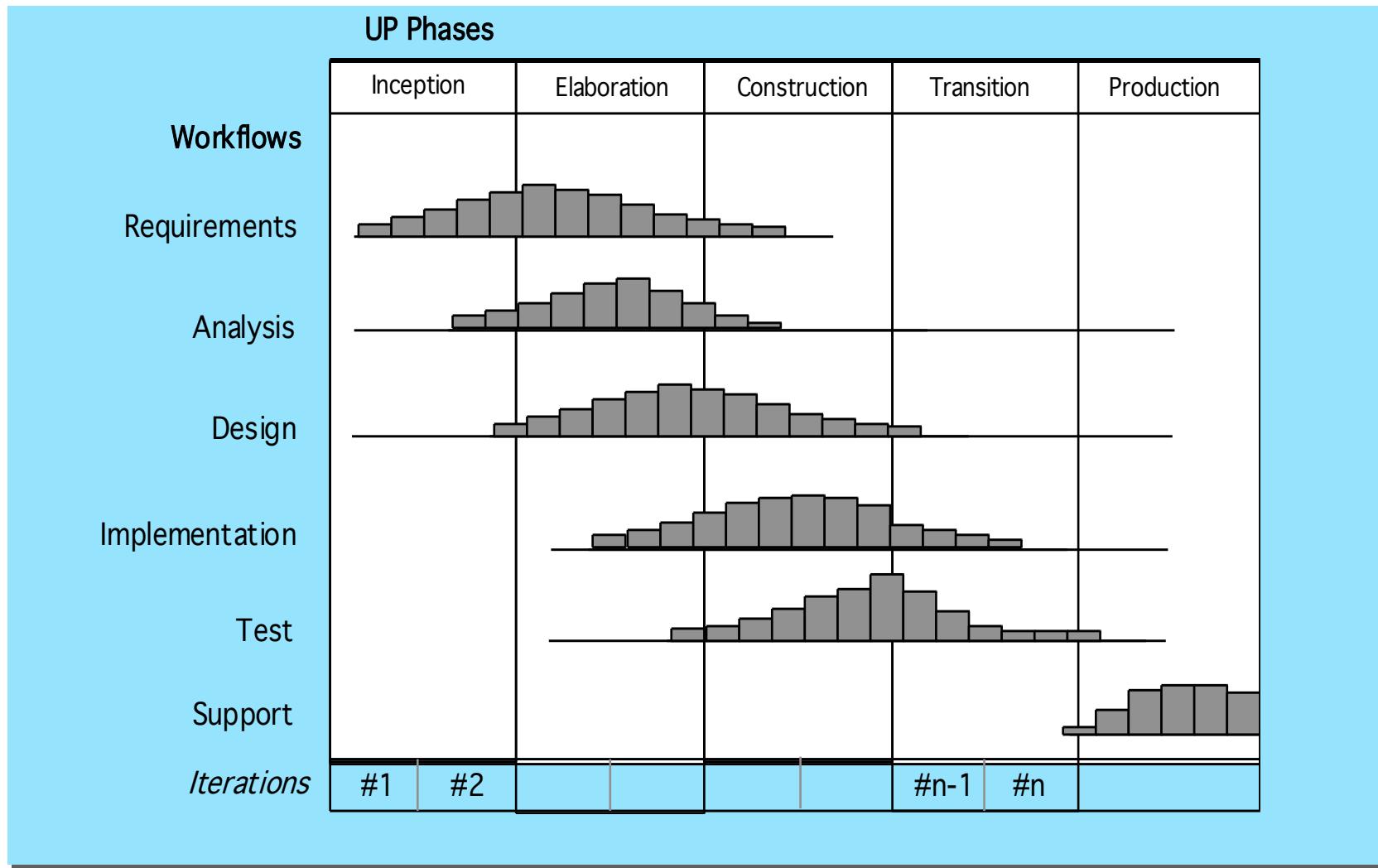
RUP phases

- Inception
 - Establish the business case for the system.
- Elaboration
 - Develop an understanding of the problem domain and the system architecture.
- Construction
 - System design, programming and testing.
- Transition
 - Deploy the system in its operating environment.

Rational Unified Process



RUP Phases



RUP Work Products

Inception phase

Vision document
Initial use-case model
Initial project glossary
Initial business case
Initial risk assessment.
Project plan,
phases and iterations.
Business model,
if necessary.
One or more prototypes

Elaboration phase

Use-case model
Supplementary requirements
including non-functional
Analysis model
Software architecture
Description.
Executable architectural
prototype.
Preliminary design model
Revised risk list
Project plan including
iteration plan
adapted workflows
milestones
technical work products
Preliminary user manual

Construction phase

Design model
Software components
Integrated software
increment
Test plan and procedure
Test cases
Support documentation
user manuals
installation manuals
description of current
increment

Transition phase

Delivered software increment
Beta test reports
General user feedback

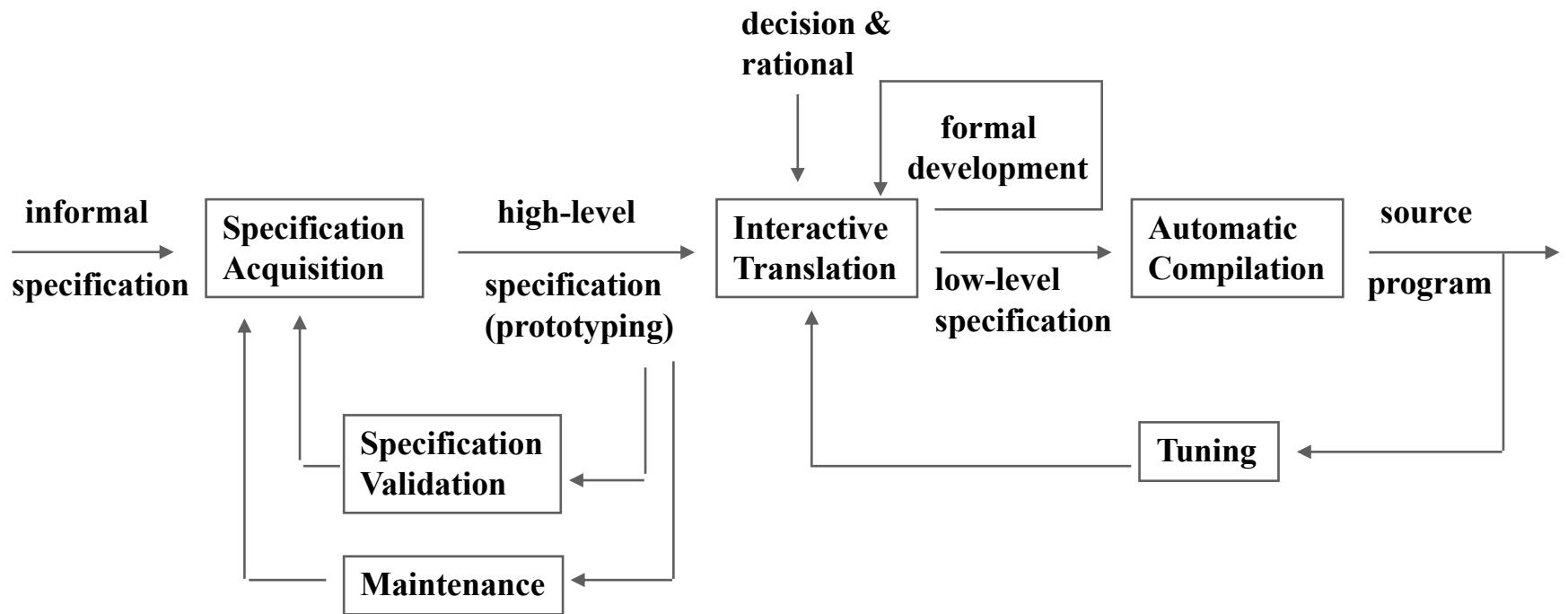
RUP good practice

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

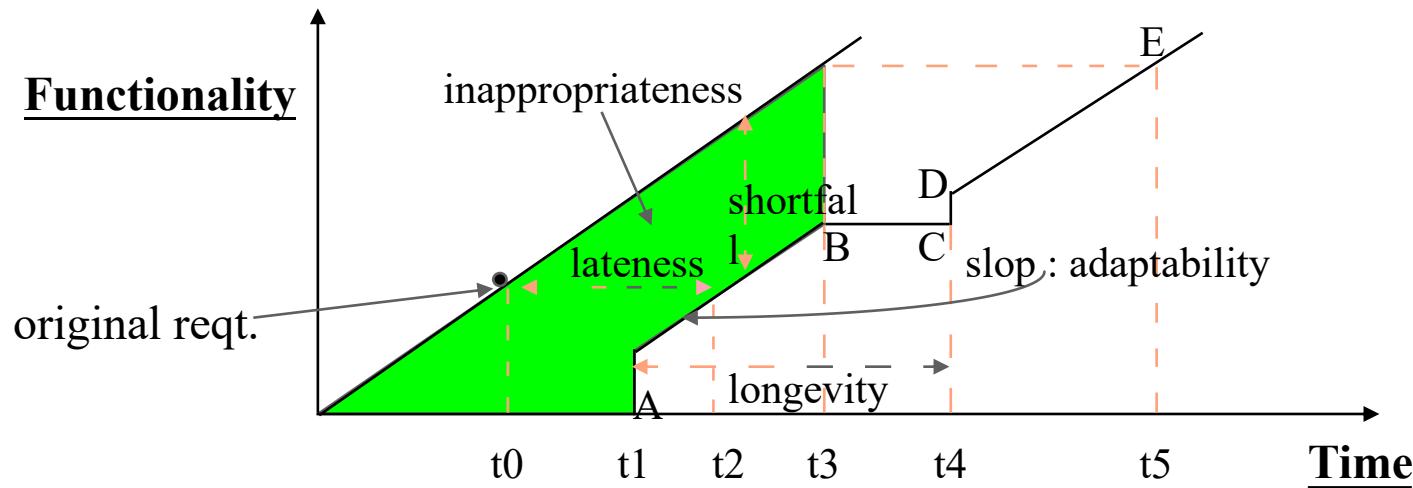
Static workflows

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Test	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Automated Synthesis Model



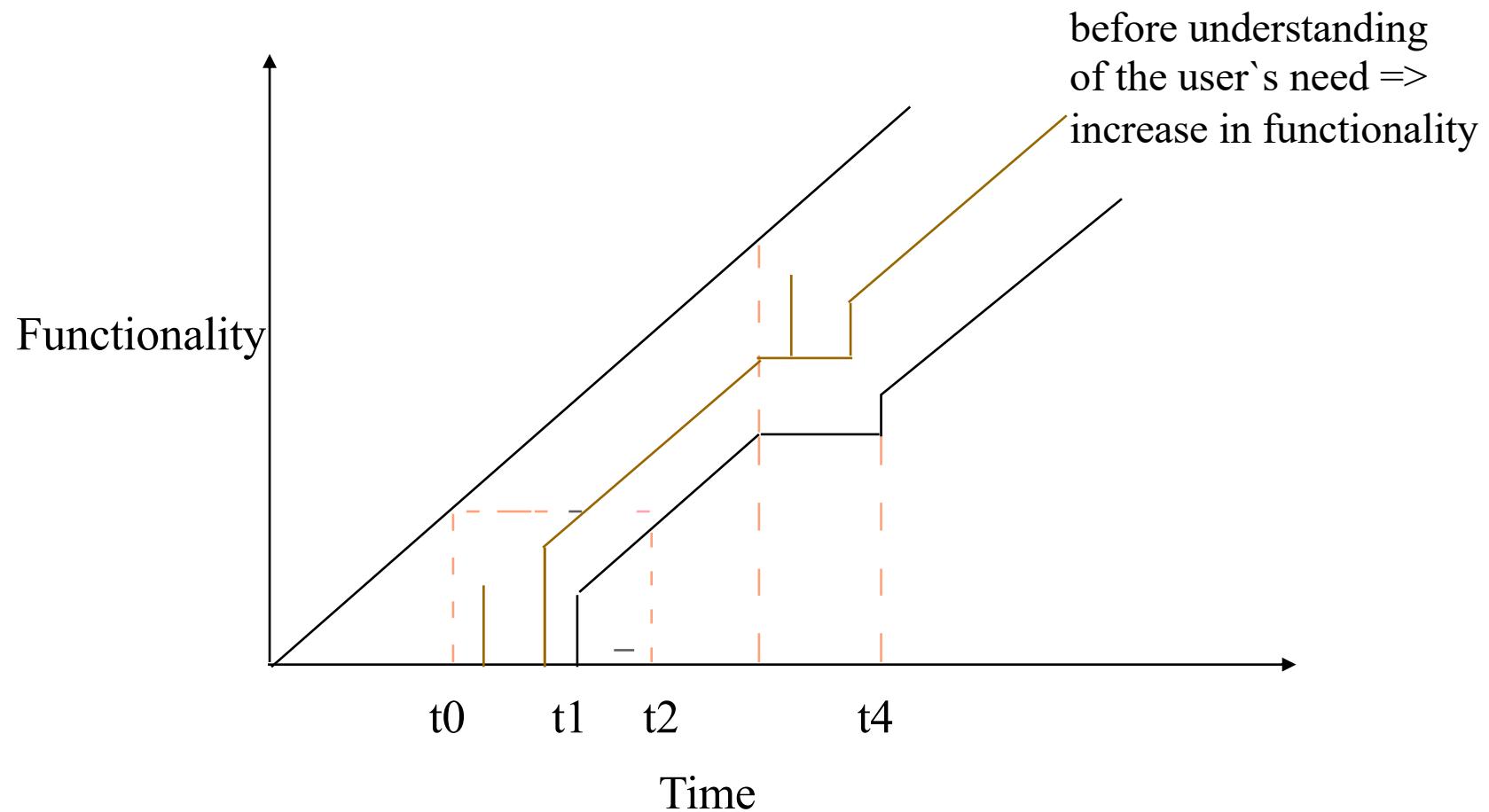
Comparing Various Process Models



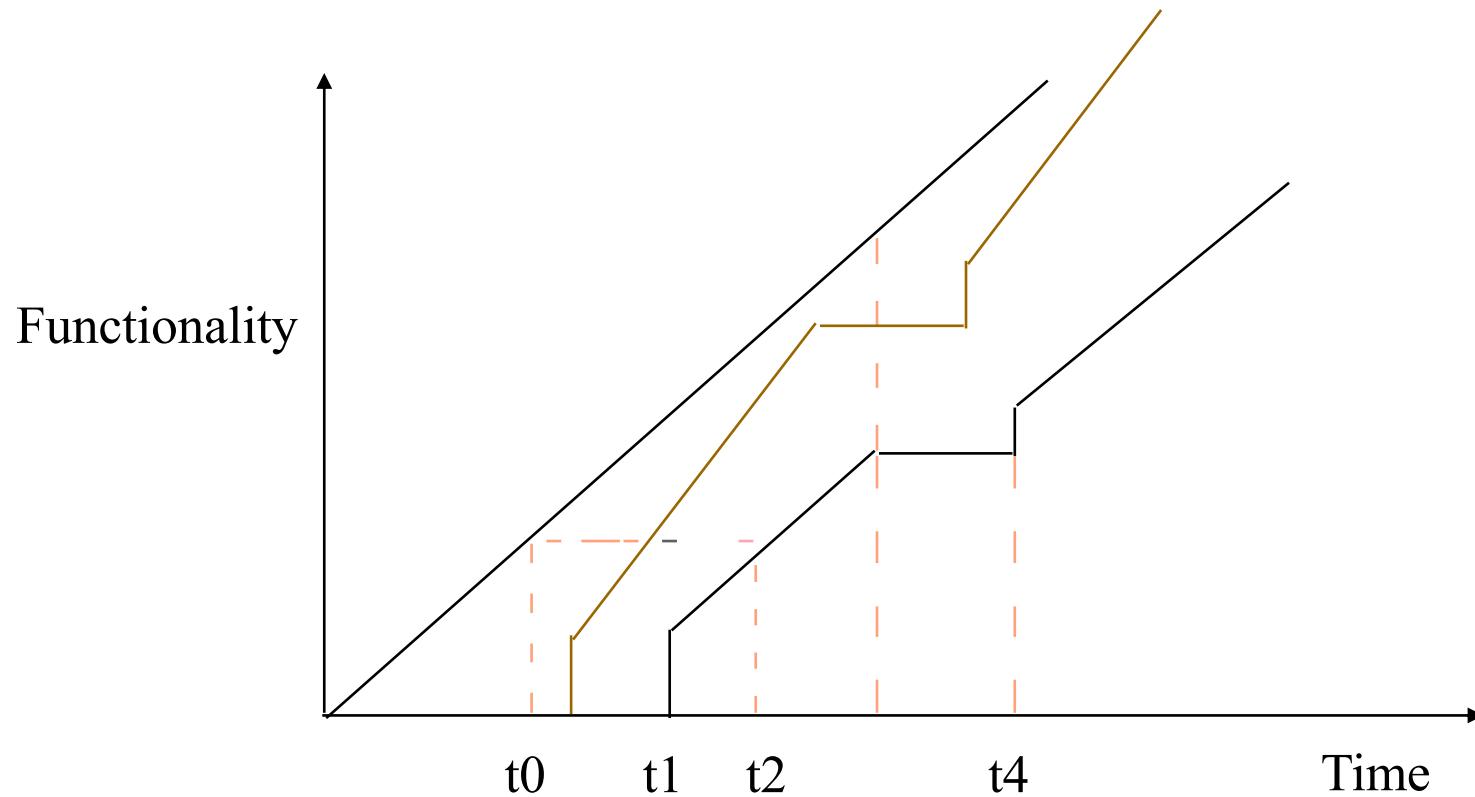
waterfall
model

- O : (t_0) original reqt.
- A : (at t_1) an operational product, not satisfying the old to needs because poor understanding of needs.
- A - B : undergo a series of enhancements.
- B - D : the cost of enhancements increase, to build a new system.
stop at t_4 .
- * cycle repeat itself

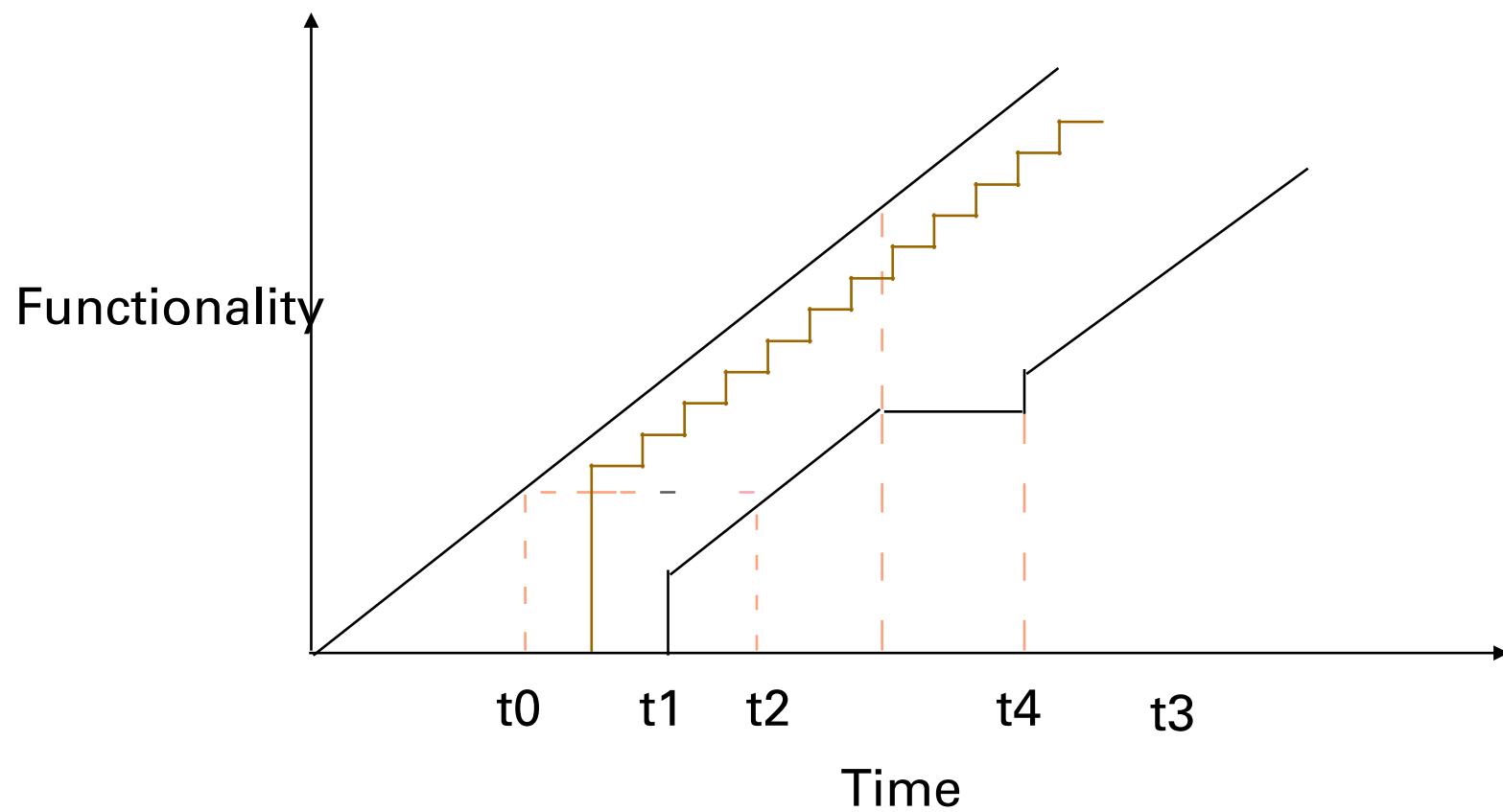
Throwaway Prototyping and Spiral Model



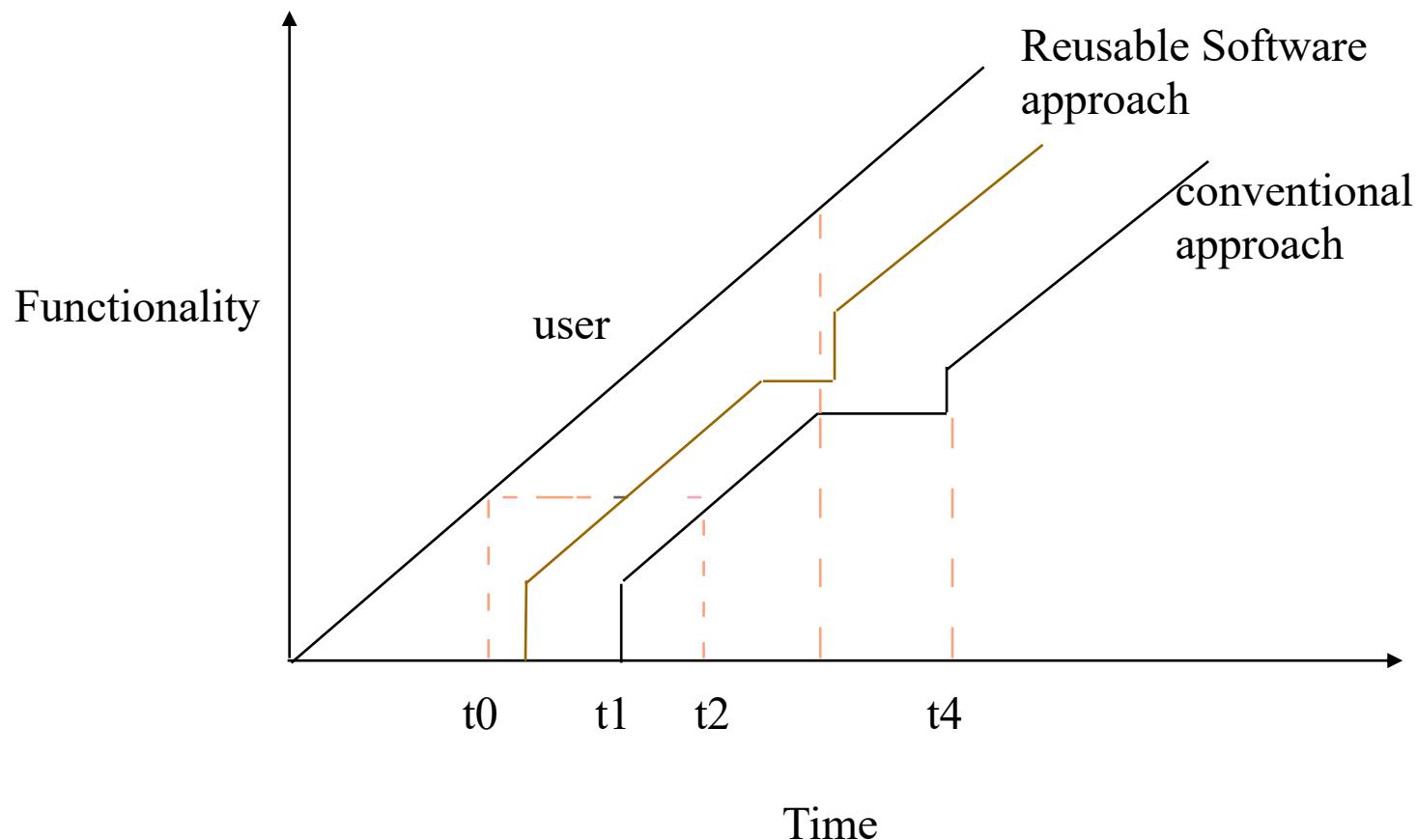
Evolutionary Prototyping



Automated Software Synthesis



Reusable Software versus Conventional



Computer-Aided Software Engineering

- Computer-Aided Software Engineering (CASE) is software to support software development and evolution processes.
- Activity automation
 - Graphical editors for system model development;
 - Data dictionary to manage design entities;
 - Graphical UI builder for user interface construction;
 - Debuggers to support program fault finding;
 - Automated translators to generate new versions of a program.

CASE technology

- CASE technology has led to significant improvements in the software process. However, these are not the order of magnitude improvements that were once predicted
 - Software engineering requires creative thought - this is not readily automated;
 - Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these.

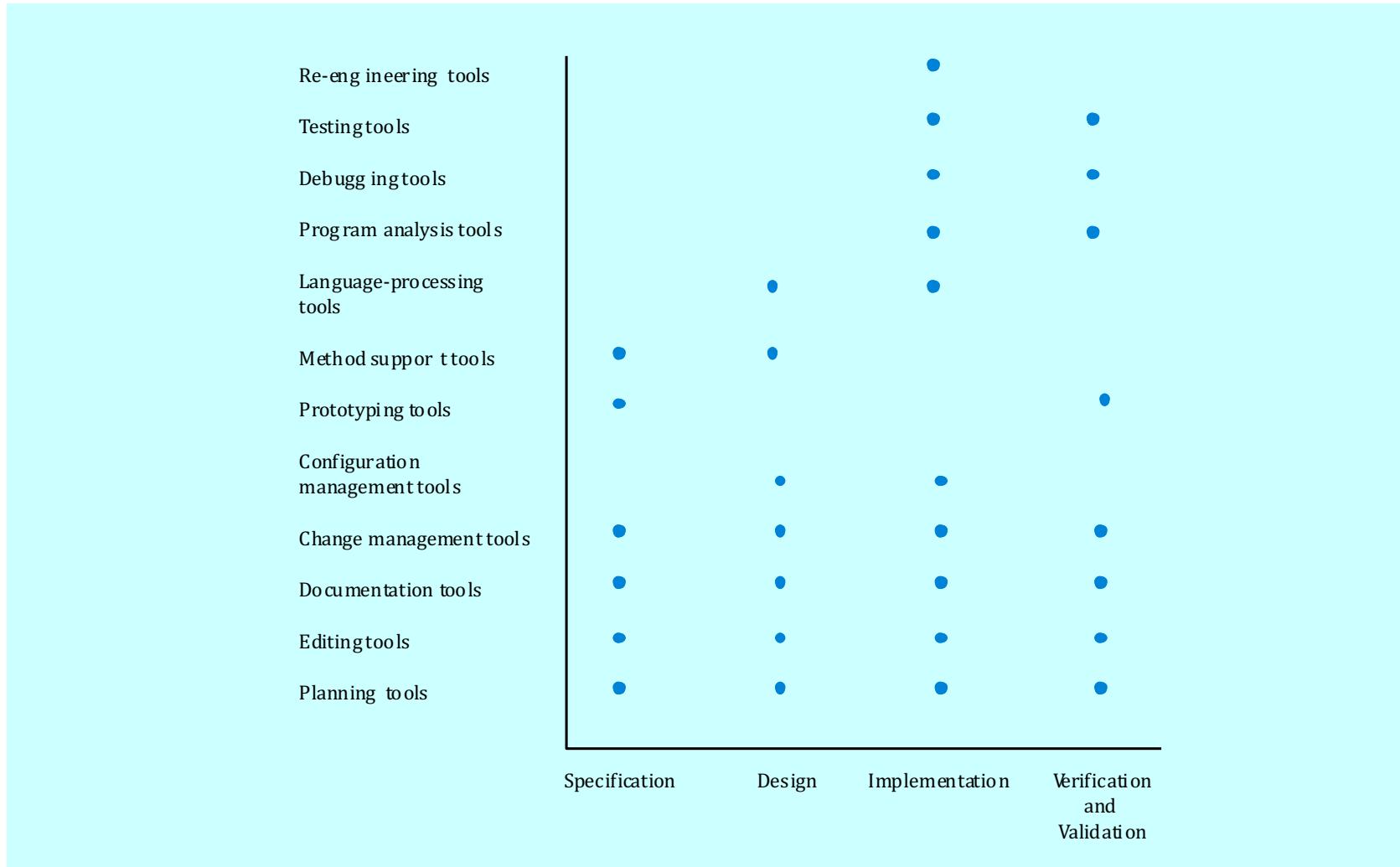
CASE classification

- Classification helps us understand the different types of CASE tools and their support for process activities.
- Functional perspective
 - Tools are classified according to their specific function.
- Process perspective
 - Tools are classified according to process activities that are supported.
- Integration perspective
 - Tools are classified according to their organisation into integrated units.

Functional tool classification

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

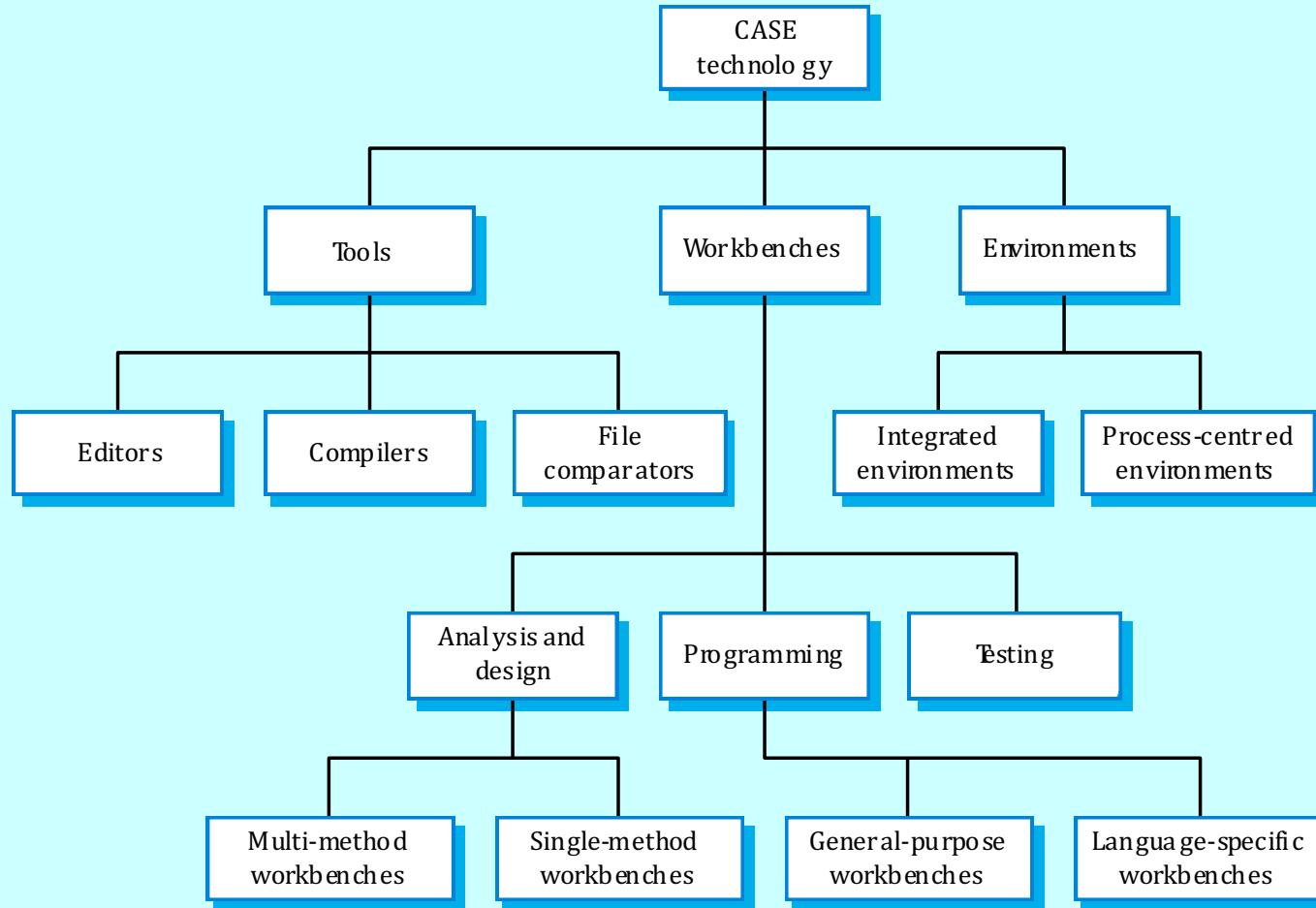
Activity-based tool classification



CASE integration

- Tools
 - Support individual process tasks such as design consistency checking, text editing, etc.
- Workbenches
 - Support a process phase such as specification or design, Normally include a number of integrated tools.
- Environments
 - Support all or a substantial part of an entire software process. Normally include several integrated workbenches.

Tools, workbenches, environments



Key points

- Software processes are the activities involved in producing and evolving a software system.
- Software process models are abstract representations of these processes.
- General activities are specification, design and implementation, validation and evolution.
- Generic process models describe the organisation of software processes. Examples include the waterfall model, evolutionary development and component-based software engineering.
- Iterative process models describe the software process as a cycle of activities.

Key points

- Requirements engineering is the process of developing a software specification.
- Design and implementation processes transform the specification to an executable program.
- Validation involves checking that the system meets to its specification and user needs.
- Evolution is concerned with modifying the system after it is in use.
- The Rational Unified Process is a generic process model that separates activities from phases.
- CASE technology supports software process activities.

Requirement Elicitation

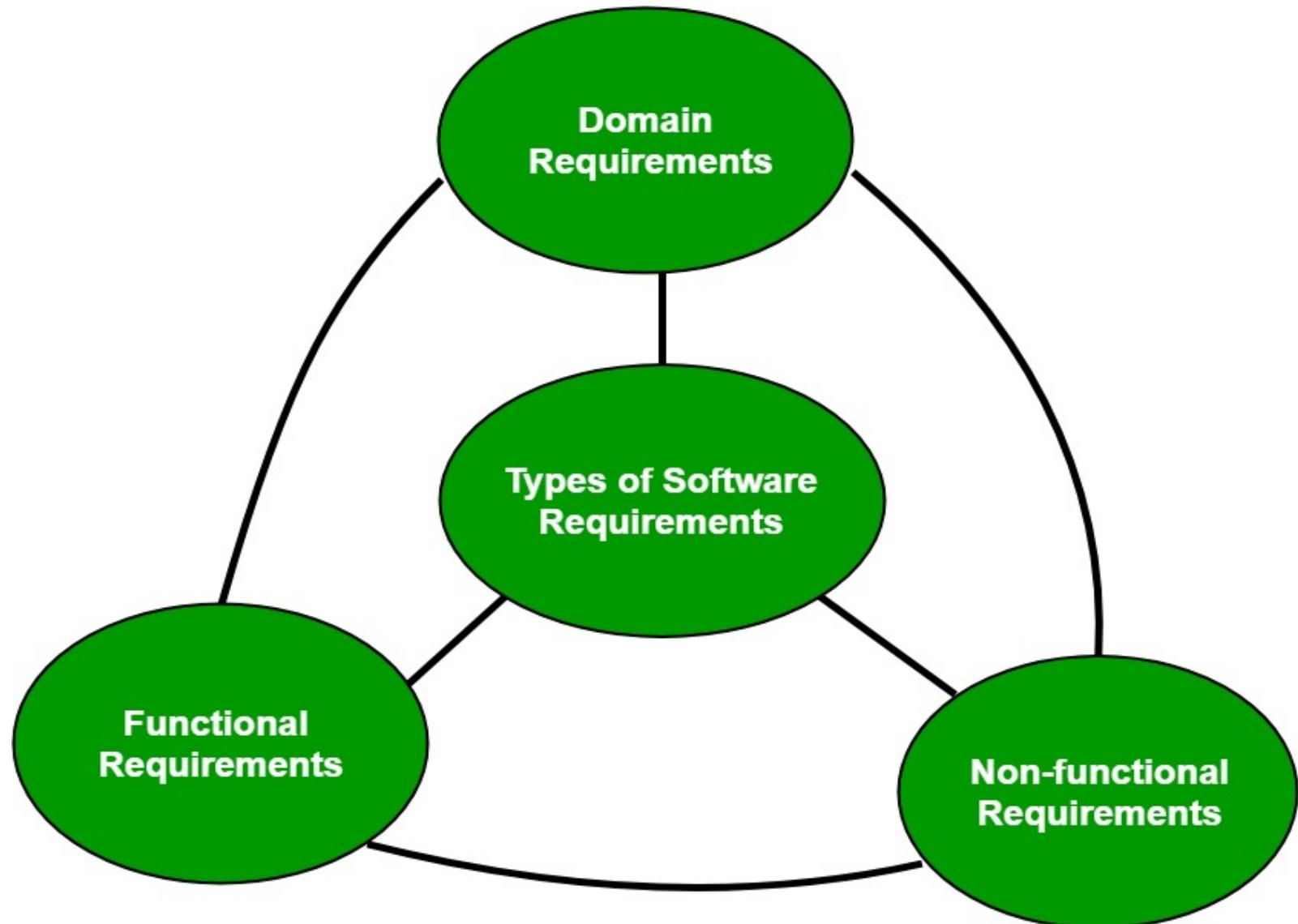
Lecture 3

Introduction

- Deciding precisely what to build is most important and most difficult
- Requirements are often buried under layers of assumptions, misconceptions, and politics
- Thorough understanding and constant communication with customers are essential

Classification

- A **software requirement can be of 3 types:**
 - Functional requirements
 - Non-functional requirements
 - Domain requirements



Functional Requirements:

- These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- All these functionalities need to be necessarily incorporated into the system as a part of the contract.
- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.
- They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

Non-functional requirements:

- These are basically the quality constraints that the system must satisfy according to the project contract.
- The priority or extent to which these factors are implemented varies from one project to other.
- They are also called non-behavioral requirements.

NFR

- They basically deal with issues like:
- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Domain requirements:

- Domain requirements are the requirements which are characteristic of a particular category or domain of projects.
- The basic functions that a system of a specific domain must necessarily exhibit come under this category.

Requirements Engineering

- Development, specification, and validation of requirements
- Elicitation and modeling
- Elicitation
 - Fact-finding, communication, and fact-validation
 - Output: requirements document ↗ Understood by customers unambiguously
- Modeling (based on requirements document)
- Requirements in a form understood by software engineers unambiguously

Requirements Elicitation

- The requirements elicitation process may appear simple: ask the customer, the users and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of business, and finally, how the system or product is to be used on a day-to-day basis. However, issues may arise that complicate the process. [1]

Challenges in Requirements Elicitation

- **Problems of scope'.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical details that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be “**obvious**,” specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous
- **Problems of volatility.** The requirements change over time. The rate of change is sometimes referred to as the level of requirement volatility

Requirements Quality

- **Visualization.** Using tools that promote better understanding of the desired end-product such as visualization and simulation.
- **Consistent language.** Using simple, consistent definitions for requirements described in natural language and use the business terminology that is prevalent in the enterprise.
- **Guidelines.** Following organizational guidelines that describe the collection techniques and the types of requirements to be collected. These guidelines are then used consistently across projects.
- **Consistent use of templates.** Producing a consistent set of models and templates to document the requirements.
- **Documenting dependencies.** Documenting dependencies and interrelationships among requirements.
- **Analysis of changes.** Performing root cause analysis of changes to requirements and making corrective actions.

Steps involved in Elicitation

- Identifying stakeholders
- Modeling goals
- Modeling context
- Discovering scenarios (or Use cases)
- Discovering "qualities and constraints" (Non-functional requirements)
- Modeling rationale and assumptions
- Writing definitions of terms
- Analyzing measurements (acceptance criteria)
- Analyzing priorities

Elicitation Techniques

- Interviews
- Brainstorming Sessions
- Facilitated Application Specification Technique (FAST)
- Quality Function Deployment (QFD)
- Use Case Approach

Interviews:

- Objective of conducting an interview is to understand the customer's expectations from the software.
- It is impossible to interview every stakeholder hence representatives from groups are selected based on their expertise and credibility.

Brainstorming Sessions:

- It is a group technique
- It is intended to generate lots of new ideas hence providing a platform to share views
- A highly trained facilitator is required to handle group bias and group conflicts.
- Every idea is documented so that everyone can see it.
- Finally a document is prepared which consists of the list of requirements and their priority if possible.

Facilitated Application Specification Technique:

- Each attendee is asked to make a list of objects that are-
 - Part of the environment that surrounds the system
 - Produced by the system
 - Used by the system
- Each participant prepares his/her list, different lists are then combined, redundant entries are eliminated, team is divided into smaller sub-teams to develop mini-specifications and finally a draft of specifications is written down using all the inputs from the meeting.

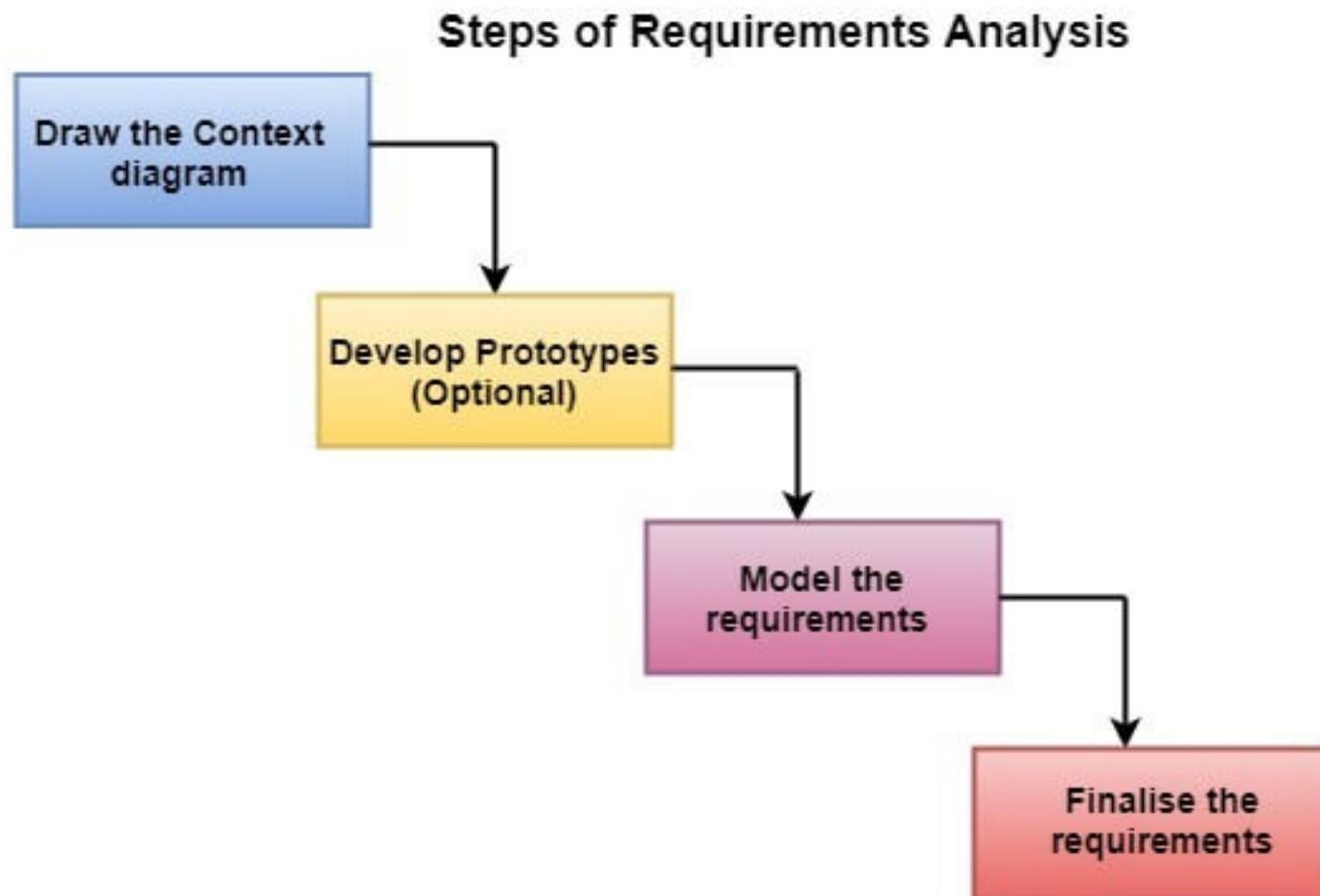
Quality Function Deployment:

- The major steps involved in this procedure are –
- Identify all the stakeholders, eg. Users, developers, customers etc
- List out all requirements from customer.
- A value indicating degree of importance is assigned to each requirement.
- In the end the final list of requirements is categorised as –
 - It is possible to achieve
 - It should be deferred and the reason for it
 - It is impossible to achieve and should be dropped off

Use Case Approach:

- **Actor** – It is the external agent that lies outside the system but interacts with it in some way. An actor maybe a person, machine etc. It is represented as a stick figure. Actors can be primary actors or secondary actors.
 - Primary actors – It requires assistance from the system to achieve a goal.
 - Secondary actor – It is an actor from which the system needs assistance.
- **Use cases** – They describe the sequence of interactions between actors and the system. They capture who(actors) do what(interaction) with the system. A complete set of use cases specifies all possible ways to use the system.
- **Use case diagram** – A use case diagram graphically represents what happens when an actor interacts with a system. It captures the functional aspect of the system.
 - A stick figure is used to represent an actor.
 - An oval is used to represent a use case.
 - A line is used to represent a relationship between an actor and a use case.

Requirements Analysis



SRS

- The software requirements specification document lists sufficient and necessary requirements for the project development.
- To derive the requirements, the developer needs to have clear and thorough understanding of the products under development.
- This is achieved through detailed and continuous communications with the project team and customer throughout the software development process

References

- [1]
https://en.wikipedia.org/wiki/Requirements_elicitation
- [2]
<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=278253>
- [3] <http://www.cse.msu.edu/~chengb/RE-491/Papers/SRSEExample-webapp.doc>

Requirements Modelling

Biju R Mohan

Requirement Modelling

- Paradigm Shift ?
- Structured Vs OOAD Vs Web Modelling

Basic Idea of Modeling

- The philosophy behind each of the phases.
- The workflows and the individual activities within each phase.
- The artifacts that should be produced (diagrams, textual descriptions and code).
- Dependencies between the artifacts.
- Notations for the different kinds of artifacts.
- The need to model static structure and dynamic behavior.

Structured Analysis and Structured Design

- Used to develop the Traditional Projects that uses procedural programming.
- **Analysis Phase:** It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
- **Design Phase:** It uses Structure Chart and Pseudo Code

OOAD

- Used to develop Object-oriented Projects that depends on Object-oriented programming.
- Uses common processes likes: analysis, design, implementation, and testing.
- Uses UML notations likes: use case, class diagram, communication diagram, development diagram and sequence diagram.

Web Modelling

- Web modeling (aka model-driven Web development) is a branch of Web engineering which addresses the specific issues related to design and development of large-scale Web applications.
- In particular, it focuses on the design notations and visual languages that can be used for the realization of robust, well-structured, usable and maintainable Web applications.
- Designing a data-intensive Web site amounts to specifying its characteristics in terms of various orthogonal abstractions.
- The main orthogonal models that are involved in complex Web application design are: data structure, content composition, navigation paths, and presentation model.

Lecture 5

Introduction OOAD

Agenda

- Compare and contrast analysis and design.
- Define object-oriented analysis and design (OOA/D).
- Illustrate a brief example.

Analysis

- Analysis emphasizes an *investigation of the problem and requirements, rather* than a solution.
- For example, if a new computerized library information system is desired, how will it be used?
- "Analysis" is a broad term, best qualified, as in *requirements analysis (an investigation of the requirements)* or *object analysis (an investigation of the domain objects)*.
- .

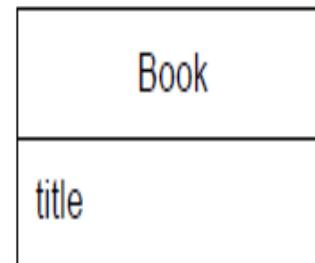
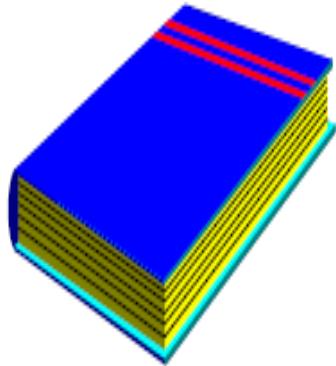
Design

- Design emphasizes a *conceptual solution that fulfills the requirements, rather than its implementation.*
- For example, a description of a database schema and software objects. Ultimately, designs can be implemented

What Is Object-Oriented Analysis and Design?

- **object-oriented analysis, there is an emphasis on finding and describing** the objects—or concepts—in the problem domain.
- For example, in the case of the library information system, some of the concepts include *Book*, *Library*, and *Patron*.
- **object-oriented design, there is an emphasis on defining software** objects and how they collaborate to fulfill the requirements.
- For example, in the library system, a *Book software object may have a title attribute and a getChapter method*

domain concept



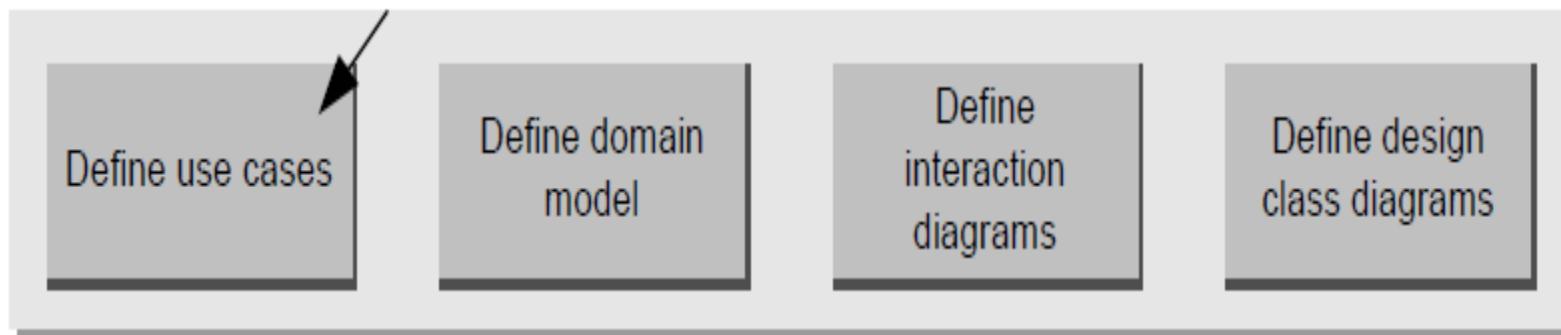
visualization of
domain concept

representation in an
object-oriented
programming language

public class Book
{
private String title;

public Chapter getChapter(int) {...}
}

Overall Picture of OOAD



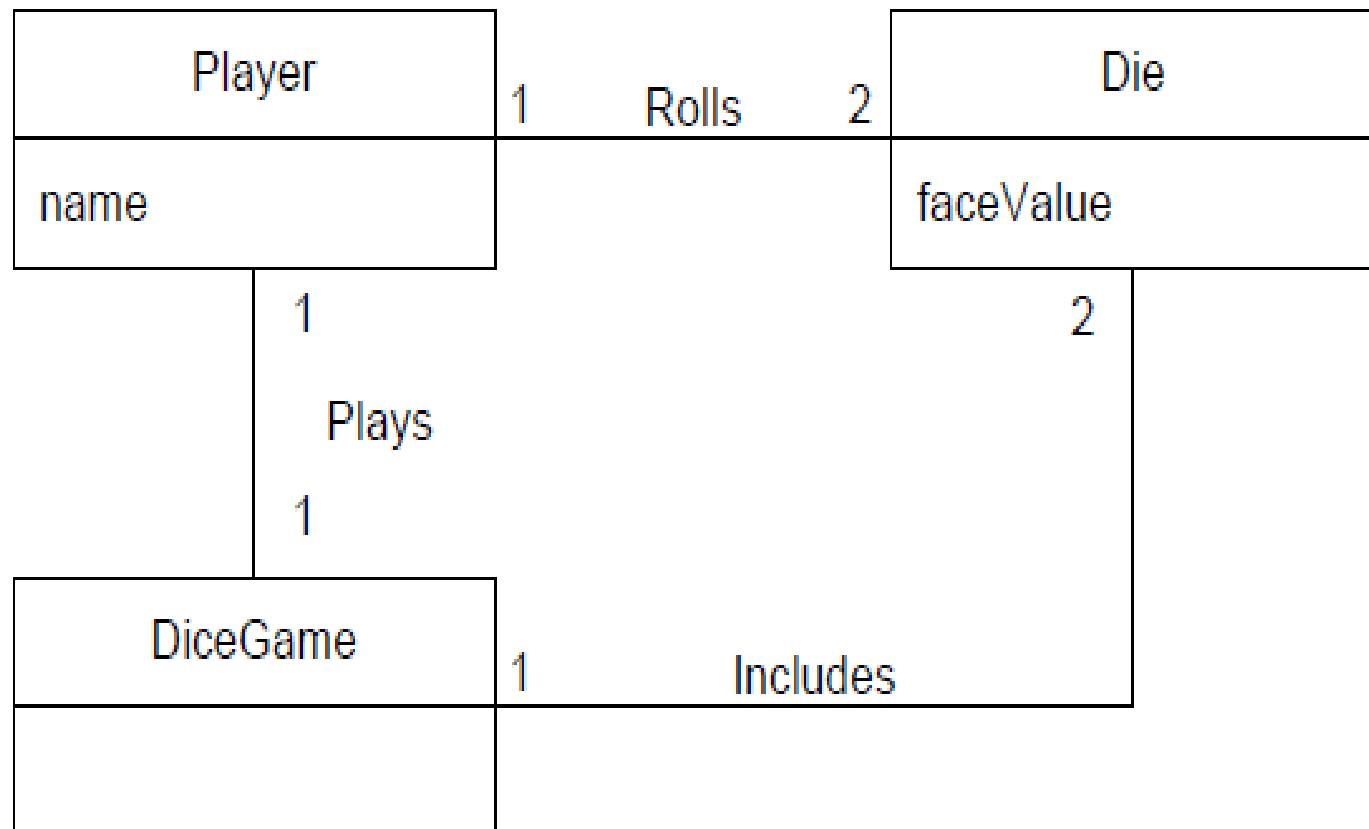
Define Use cases

- Use cases are not an object-oriented artifact—they are simply written stories.
- However, they are a popular tool in requirements analysis and are an important part of the Unified Process.
- For example, here is a brief version of the *Play a Dice Game use case*:
 - **Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.**

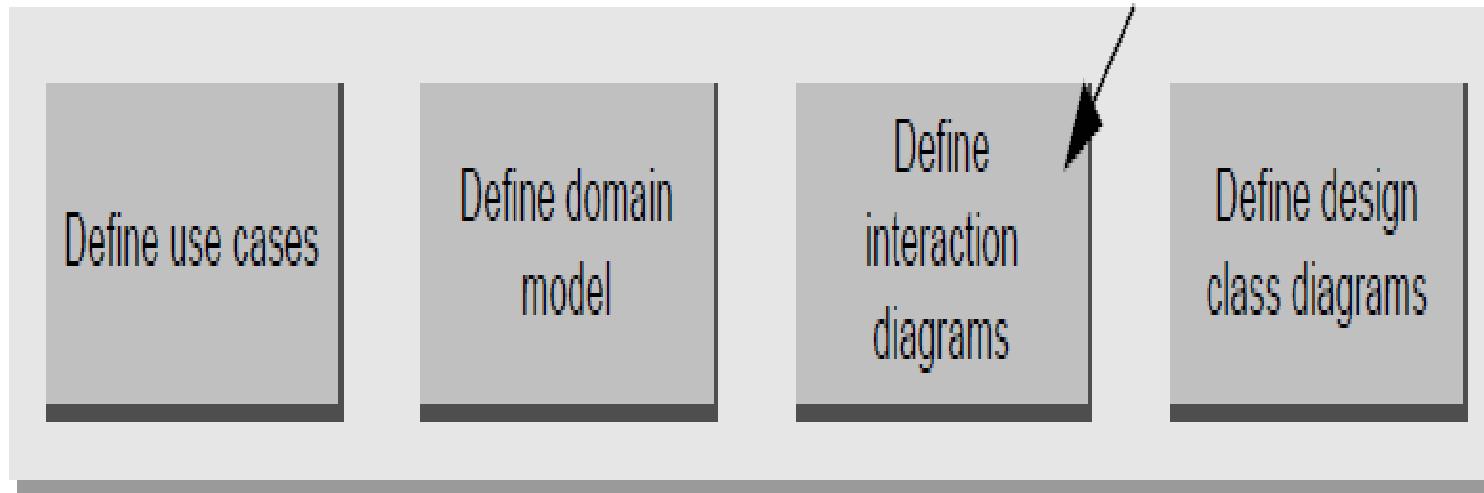
Define a Domain Model

- A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy.
- The result can be expressed in a **domain model, which** is illustrated in a set of diagrams that show domain concepts or objects.

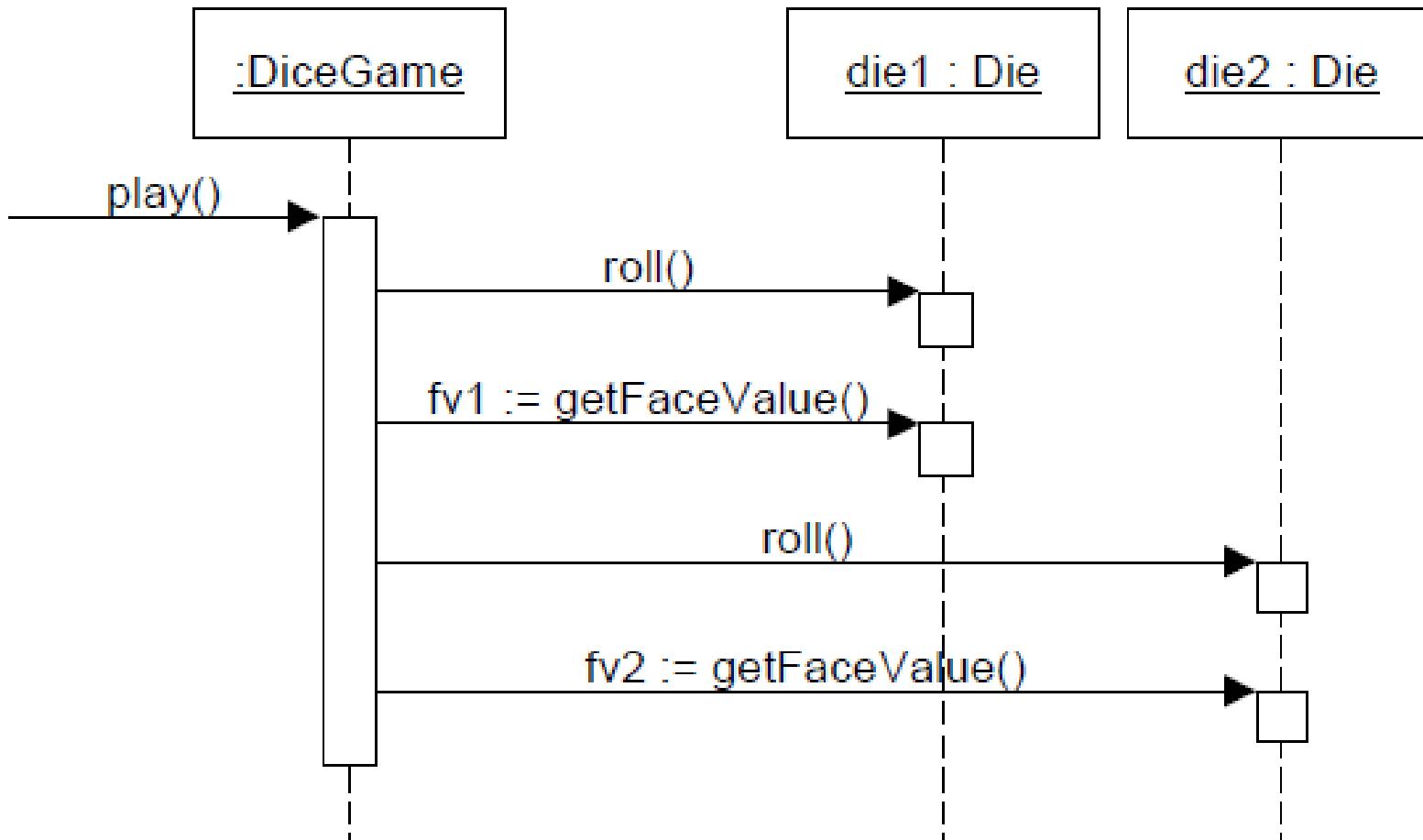
Partial domain model of the dice game.



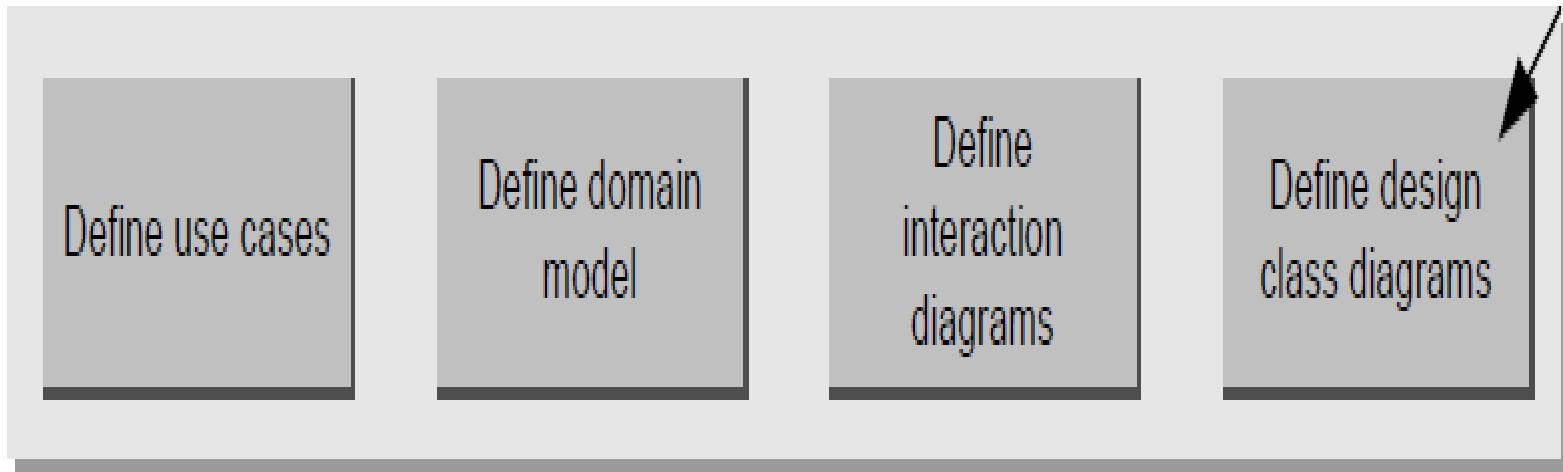
Define Interaction Diagrams



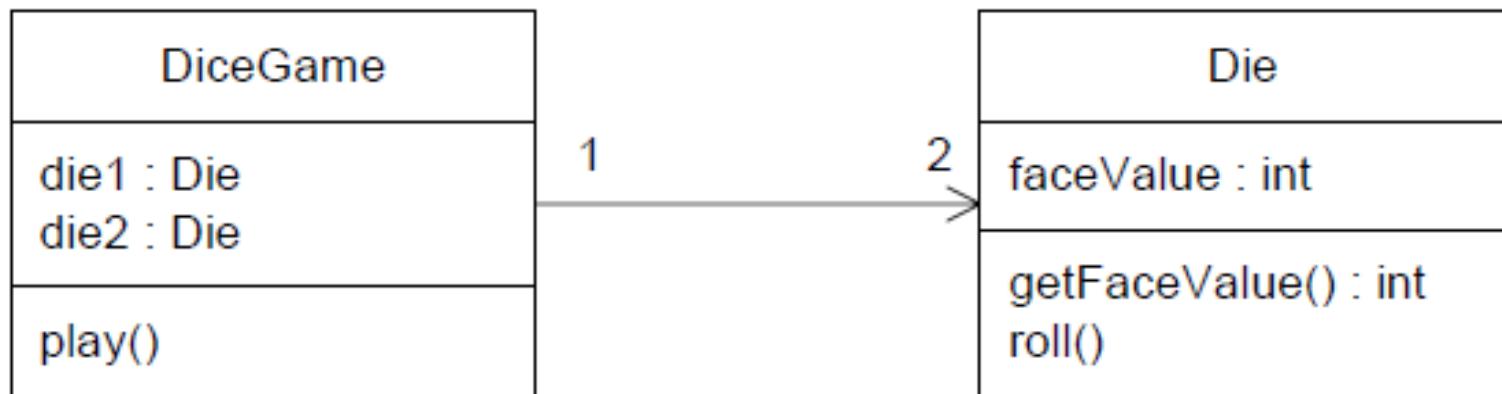
Interaction diagram illustrating messages between software objects.



Define Design Class Diagrams



Partial design class diagram



References

**Applying UML and Patterns: An Introduction to
Object-Oriented Analysis and Design and
Iterative Development**

Lecture 6

Unified Process

Agenda

- Define an iterative and adaptive process.
- Define fundamental concepts in the Unified Process.

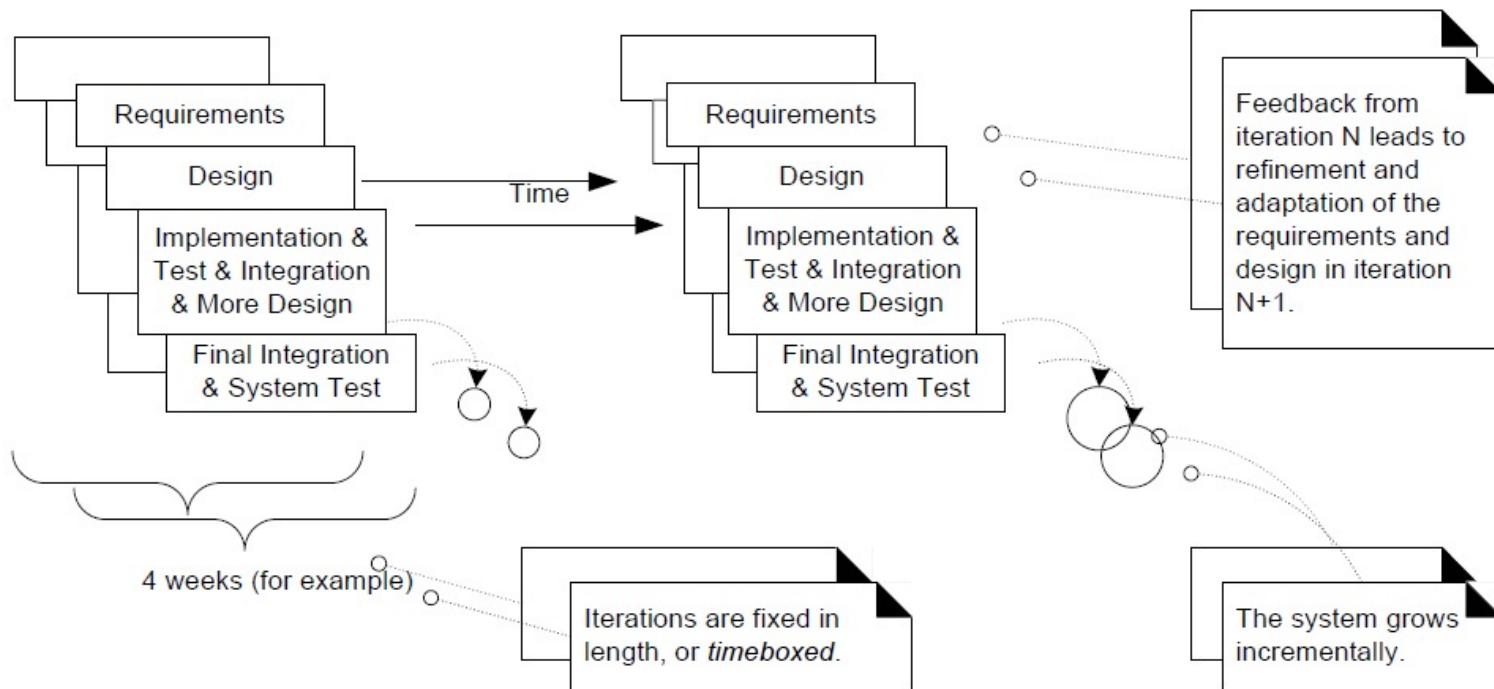
UP

- The **Unified Process** has emerged as a popular software development process for building object-oriented systems.
- The Unified Process (UP) combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented description.

Important Idea

- The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system.
- The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development**

Iterative and incremental development.



Highlights

- There is neither a rush to code, nor a long drawn-out design step that attempts to perfect all details of the design before programming.
- A "little" forethought regarding the design with visual modeling using rough and fast UML drawings is done; perhaps a half or full day by developers doing design work in pairs.
- The result of each iteration is an executable but incomplete system; it is not ready to deliver into production. The system may not be eligible for production deployment until after many iterations; for example, 10 or 15 iterations.

Contd..

- The output of an iteration is *not an experimental or throw-away prototype, and* iterative development is not prototyping.
- Rather, the output is a production-grade subset of the final system.
- each iteration tackles new requirements and incrementally extends the system, an iteration may occasionally revisit existing software and improve it;

Benefits of Iterative Development

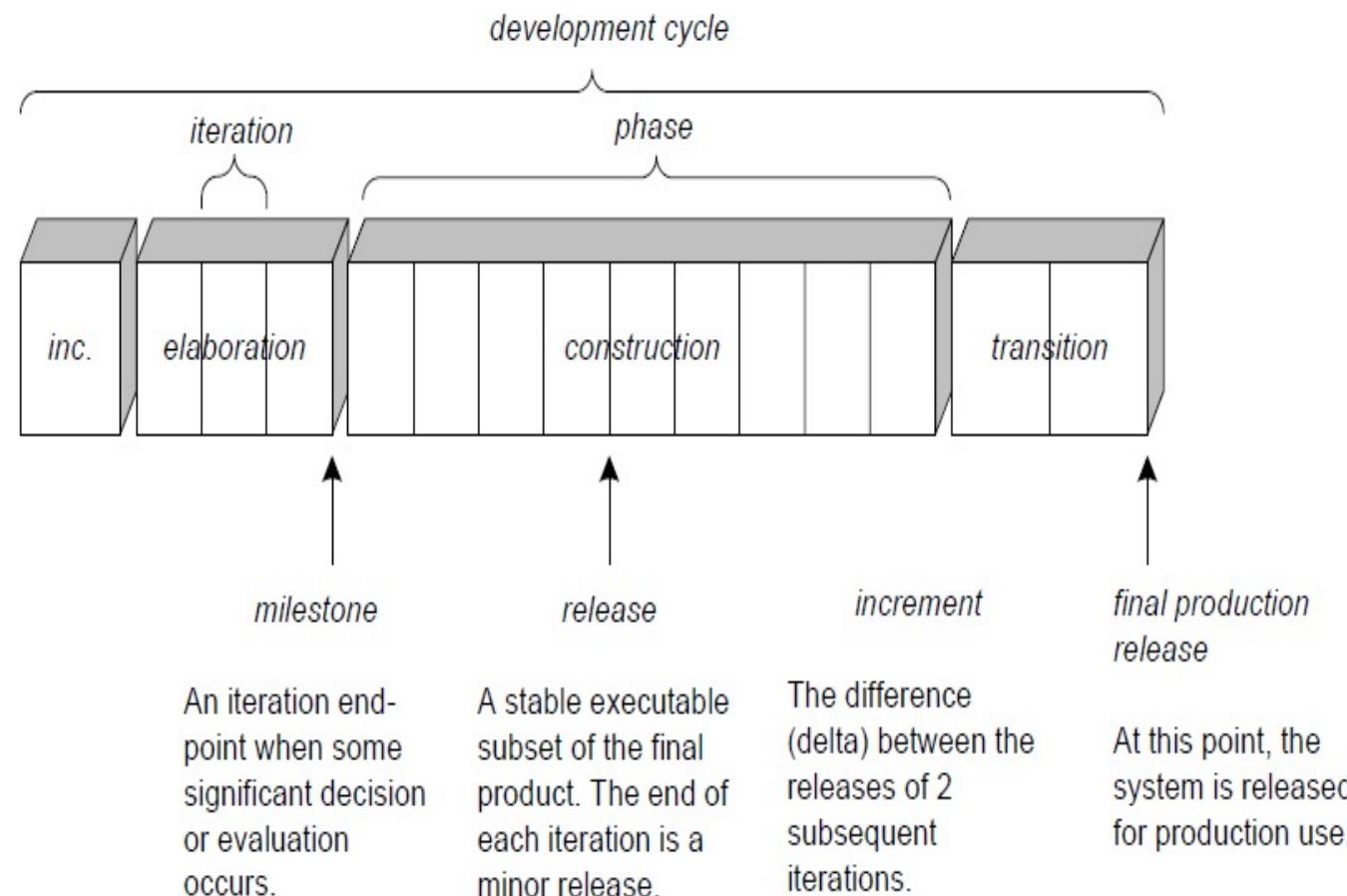
- early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
- early visible progress
- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
- the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

Phases

A UP project organizes the work and iterations across four major phases:

1. **Inception**— approximate vision, business case, scope, vague estimates.
2. **Elaboration**—refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction**—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition**—beta tests, deployment.

Schedule-oriented terms in the UP.



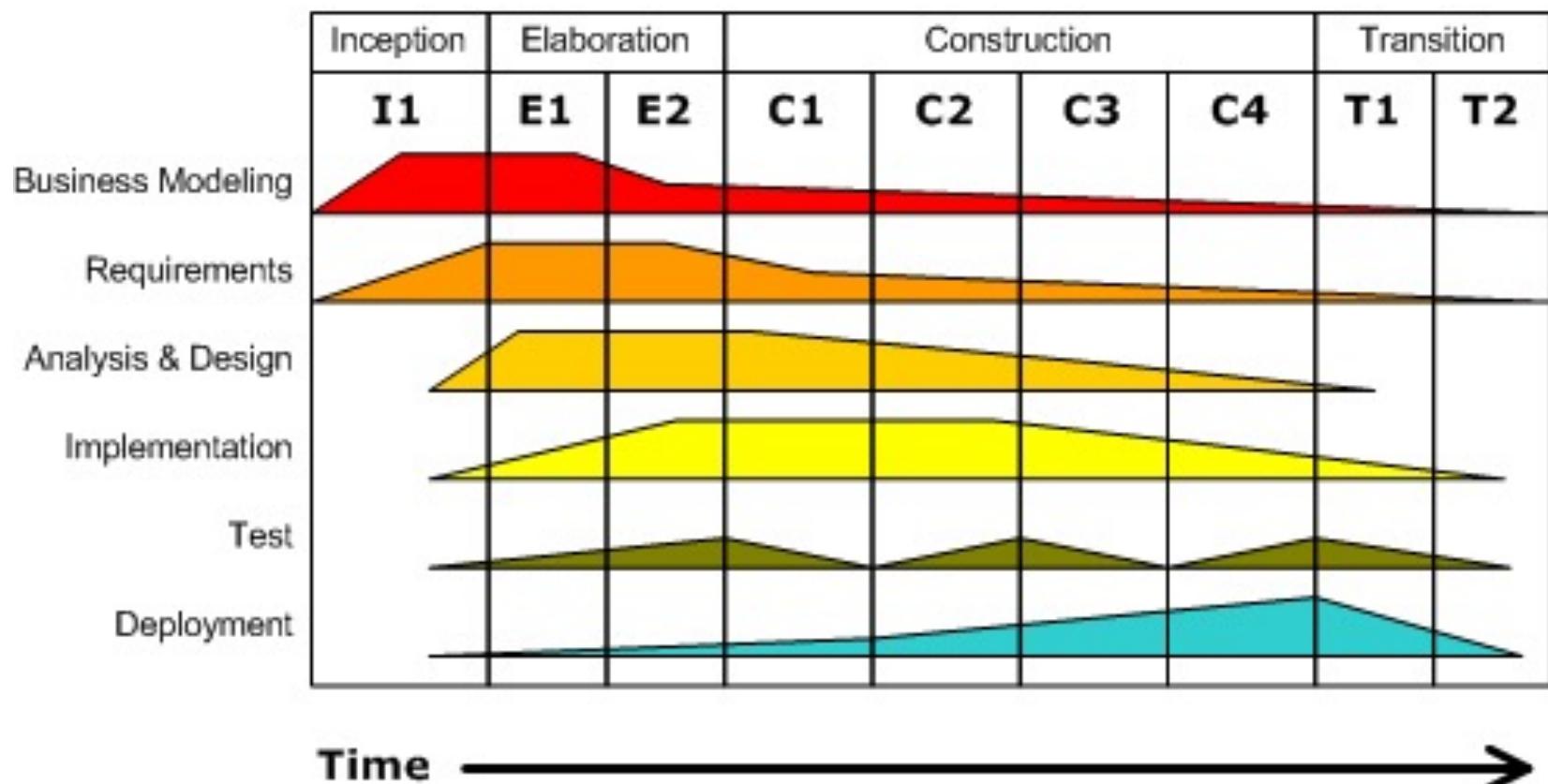
The UP Disciplines (was Workflows)

- The UP describes work activities, such as writing a use case, within **disciplines** (originally called **workflows**).
- In the UP, an **artifact** is the general term for any work product: **code**, Web graphics, database schema, text documents, diagrams, models, and so on.

Disciplines and phases

Iterative Development

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



Sample Development Case of UP artifacts, s - start; r - refine

Discipline	Artifact Iteration-*	Incep. 11	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

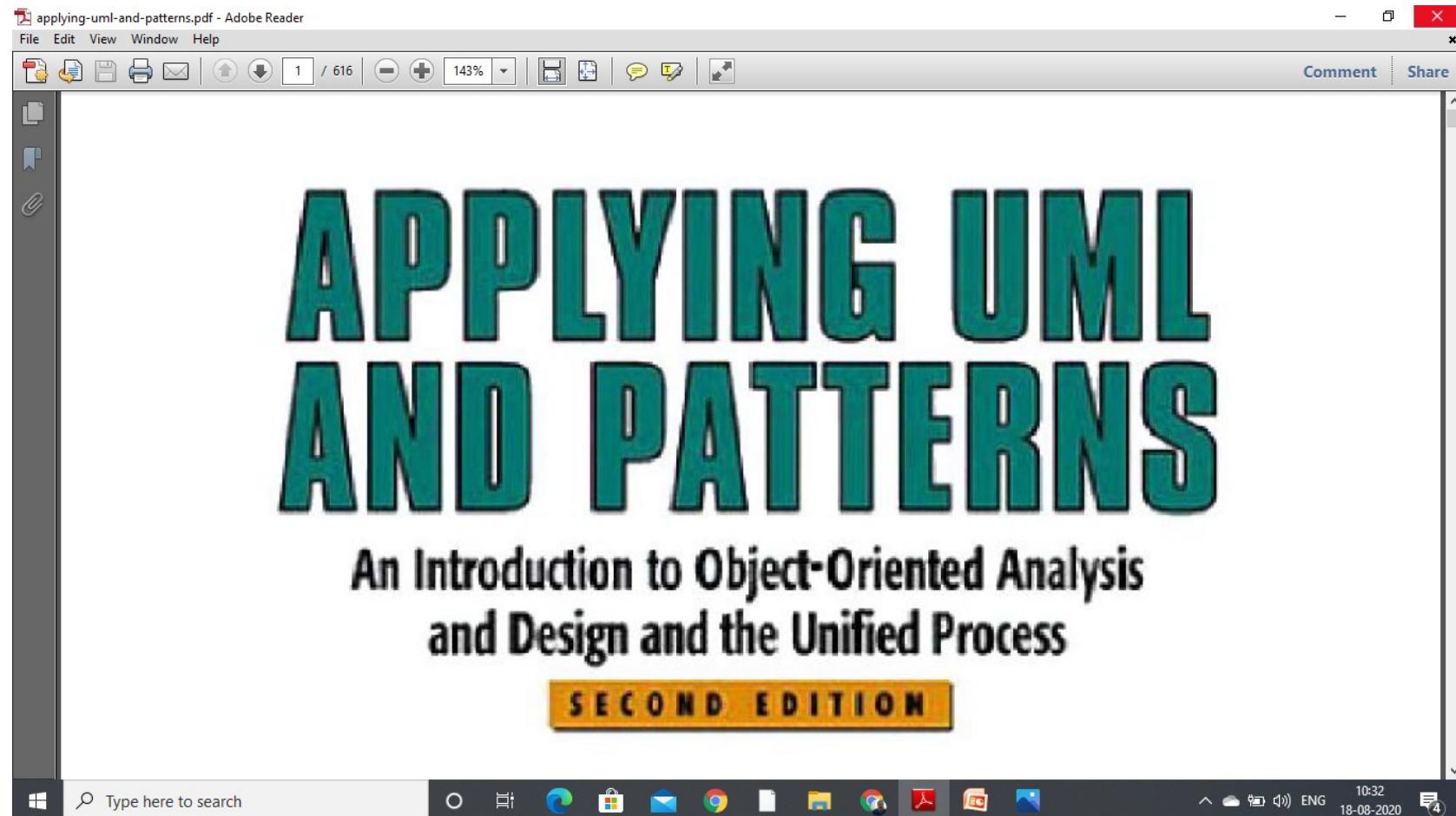
You Know You Didn't Understand the UP When...

- You think that inception = requirements, elaboration = design, and construction = implementation (that is, superimposing a waterfall lifecycle on to the UP).
- You think that the purpose of elaboration is to fully and carefully define models, which are translated into code during construction.
- You try to define most of the requirements before starting design or implementation.

Contd..

- You try to define most of the design before starting implementation; you try to fully define and commit to an architecture before iterative programming and testing.
- A "long time" is spent doing requirements or design work before programming starts.
- You think UML diagramming and design activities are a time to fully and accurately define designs and models in great detail, and of programming as a simple mechanical translation of these into code.

References



Lecture 7

INCEPTION

Questions?

- What is the vision and business case for this project?
- Feasible?
- Buy and/or build?
- Rough estimate of cost: Is it \$10K-100K or in the millions?
- Should we proceed or stop?

What Artifacts May Start in Inception?

Artifact ¹	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements, and related non-functional requirements.
Supplementary Specification	Describes other requirements.
Glossary	Key domain terminology.
Risk List & Risk Management Plan	Describes the business, technical, resource, schedule risks, and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

Use Case Modeling

Practical Way of Doing

Agenda

- Identify and write use cases.
- Relate use cases to user goals and elementary business processes.
- Use the brief, casual, and fully dressed formats, in an essential style.
- Relate use case work to iterative development.

Introduction

- The UP defines the **Use-Case Model within the Requirements discipline.**
- Essentially, this is the set of all use cases; it is a model of the system's functionality and environment

Use Case Diagrams

A formal way of representing how a business system interacts with its environment

Illustrates the activities that are performed by the users of the system

A scenario-based technique in the UML

A sequence of actions a system performs that yields a valuable result for a particular actor

Use Case Analysis

- What is an Actor?
 - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
- Use Cases describe scenarios that describe the interaction between users of the system (the actor) and the system itself.

Use Cases

- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*.
- Use case diagrams are closely connected to scenarios. A **scenario** is an example of what happens when someone interacts with the system.

Use Cases

- Here is a scenario for a medical clinic.
- *A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "*
- We want to write a use case for this scenario.
- Remember: A **use case** is a summary of scenarios for a single task or goal.

Use Cases

- Step 1 Identify the actors
- As we read the scenario, define those people or systems that are going to interact with the scenario.
- *A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "*

Use Cases

- Step 1 Identify the actors
- As we read the scenario, define those people or systems that are going to interact with the scenario.
- *A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "*

Questions for Identifying People Actors

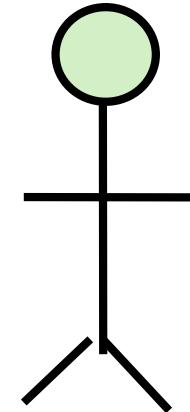
- Who is interested in the scenario/system?
- Where in the organization is the scenario/system be used?
- Who will benefit from the use of the scenario/system?
- Who will supply the scenario/system with this information, use this information, and remove this information?
- Does one person play several different roles?
- Do several people play the same role?

Questions for Identifying Other Actors

- What other entity is interested in the scenario/system?
- What other entity will supply the scenario/system with this information, use this information, and remove this information?
- Does the system use an external resource?
- Does the system interact with a legacy system?

Actors

- An Actor is outside or external the system.
- It can be a:
 - Human
 - Peripheral device (hardware)
 - External system or subsystem
 - Time or time-based event
- Represented by stick figure



Use Cases

- A **use case** is a summary of scenarios for a single task or goal.
- An **actor** is who or what initiates the events involved in the task of the use case. Actors are simply roles that people or objects play.
- So as we read our scenario, what or who is the actor????

Use Cases

- So as we read our scenario, what or who is the actor????
- *A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "*



- The actor is a **Patient**.

Use Cases

- The **use case** is a summary of scenarios for a single task or goal.
- So What is the Use Case????
- The Use Case is **Make Appointment**.
- It is a use case for the medical clinic.

Use Cases

- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication association** (or **communication** for short).



Actors are stick figures. Use cases are ovals. Communications are lines that link actors to use cases.

Use Case Components

- The use case has three components.
- The **use case** task referred to as the use case that represents a feature needed in a software system.
- The **actor(s)** who trigger the use case to activate.
- The **communication** line to show how the actors communicate with the use case.

Use Case Diagram - Use Case

- A major process performed by the system that benefits an actor(s) in some way
- Models a dialogue between an actor and the system
- Represents the functionality provided by the system

Use Case

- Each use case in a use case diagram describes one and only one *function* in which users interact with the system
 - May contain several “paths” that a user can take while interacting with the system
 - Each path is referred to as a scenario

Use Case

- Labelled using a descriptive verb-noun phrase
- Represented by an oval

Use Case - Actor

- Labelled using a descriptive noun or phrase
- Represented by a stick character



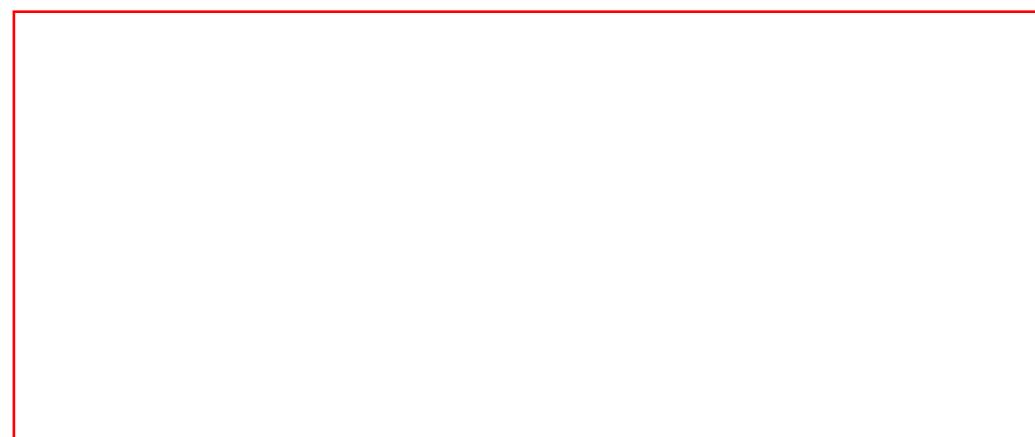
Use Case - Relationships

- Relationships
 - Represent communication between actor and use case
 - Depicted by line or double-headed arrow line
 - Also called association relationship

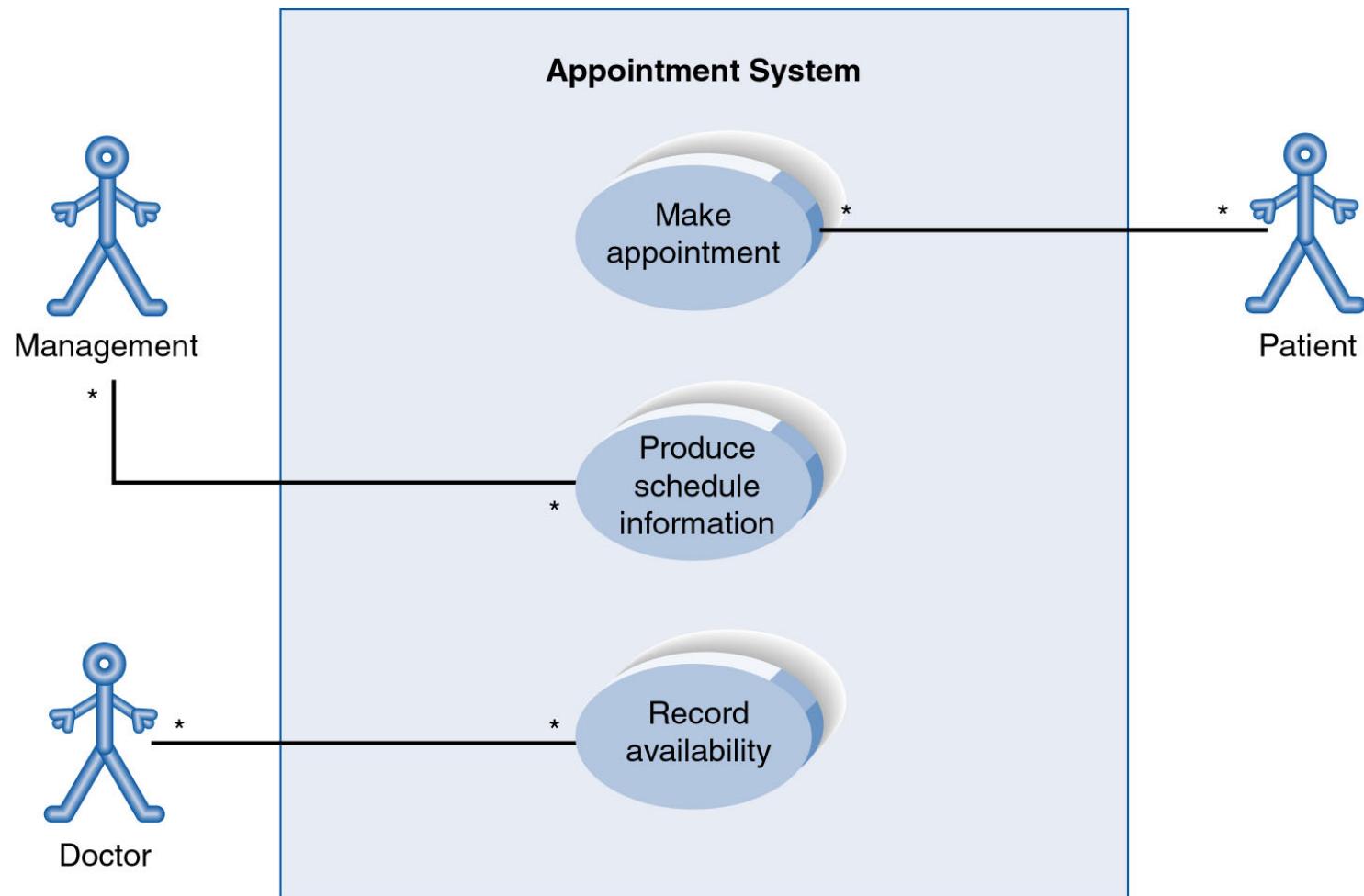


Use Case - Relationships

- Boundary
 - A boundary rectangle is placed around the perimeter of the system to show how the actors communicate with the system.



Use-Case Diagram



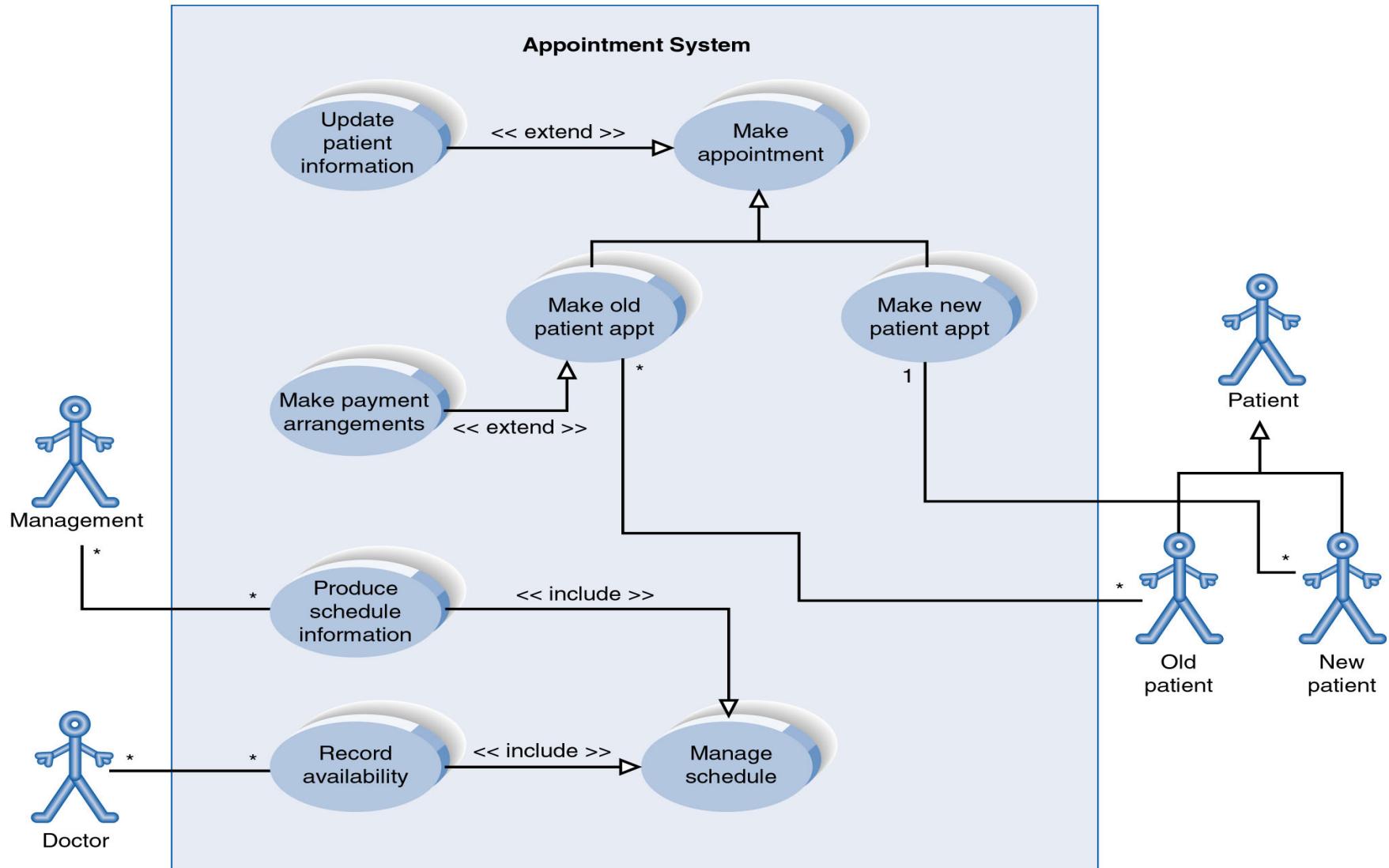
Use Case Diagram

- Other Types of Relationships for Use Cases
 - Generalization
 - Include
 - Extend

Components of Use Case Diagram

- Generalization Relationship
 - Represented by a line and a hollow arrow
 - From child to parent

Example of Relationships



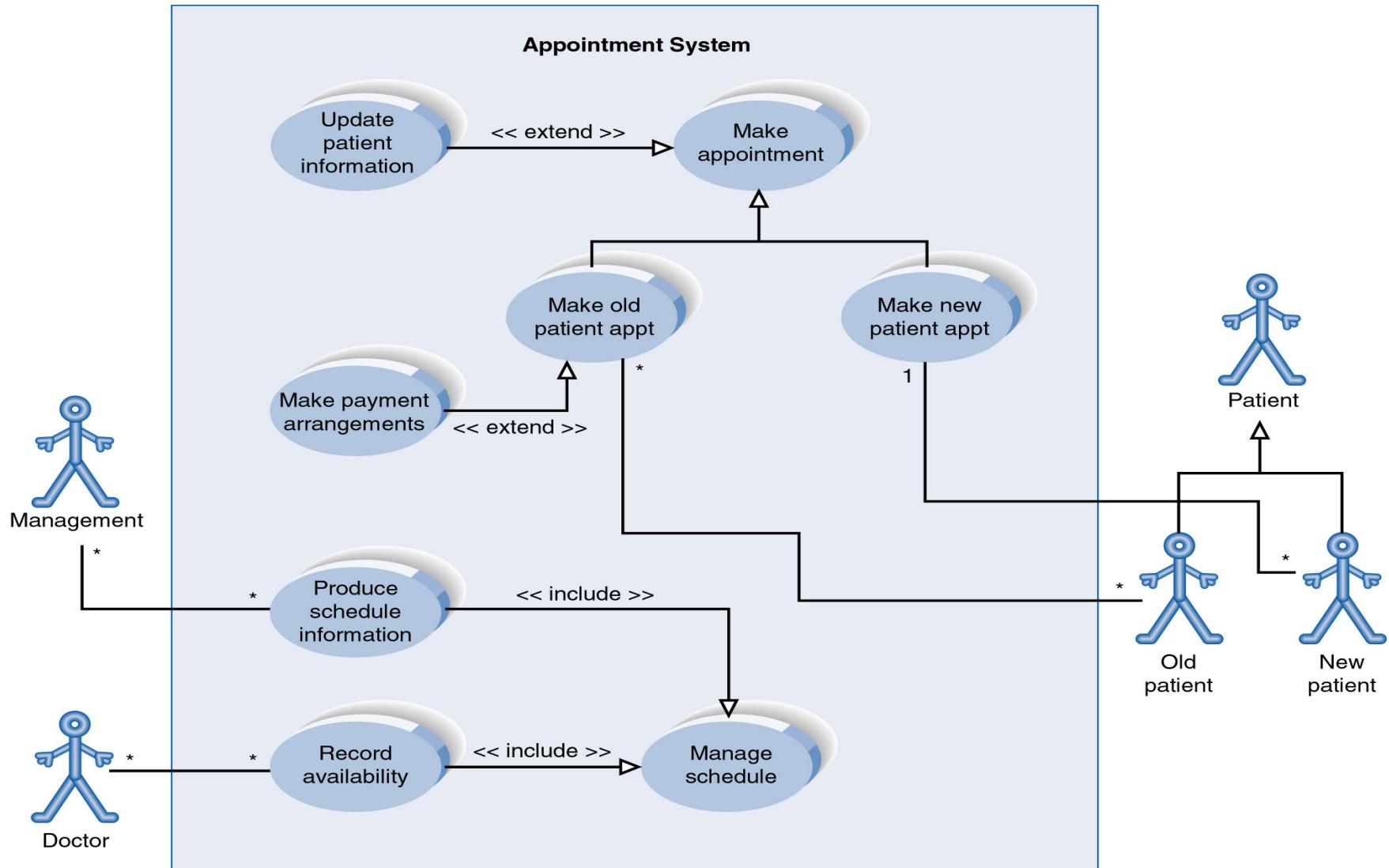
Use Case Diagram

- Include Relationship
 - Represents the inclusion of the functionality of one use case within another
 - Arrow is drawn from the base use case to the used use case
 - Write << include >> above arrowhead line

Use Case Diagram

- Extend relationship
 - Represents the extension of the use case to include optional functionality
 - Arrow is drawn from the extension use case to the base use case
 - Write << extend >> above arrowhead line

Example of Relationships



Use Case Relationships

- Pro:
 - Reduces redundancy in use cases
 - Reduces complexity within a use case
- Con
 - May introduce complexity to use case diagram

Benefits of Use Cases

- Use cases are the primary vehicle for requirements capture in UP
- Use cases are described using the language of the customer (language of the domain which is defined in the glossary)
- Use cases provide a contractual delivery process (UP is Use Case Driven)
- Use cases provide an easily-understood communication mechanism
- When requirements are traced, they make it difficult for requirements to fall through the cracks
- Use cases provide a concise summary of what the system should do at an abstract (low modification cost) level.

Difficulties with Use Cases

- As functional decompositions, it is often difficult to make the transition from functional description to object description to class design
- Reuse at the class level can be hindered by each developer “taking a Use Case and running with it”. Since UCs do not talk about classes, developers often wind up in a vacuum during object analysis, and can often wind up doing things their own way, making reuse difficult
- Use Cases make stating non-functional requirements difficult (where do you say that X must execute at Y/sec?)
- Testing functionality is straightforward, but unit testing the particular implementations and non-functional requirements is not obvious

Use Case Model Survey

- The Use Case Model Survey is to illustrate, in graphical form, the universe of Use Cases that the system is contracted to deliver.
- Each Use Case in the system appears in the Survey with a short description of its main function.
 - Participants:
 - Domain Expert
 - Architect
 - Analyst/Designer (Use Case author)
 - Testing Engineer

A Case Study

Lecture 8

Agenda

- The NextGen POS System
- Use Case

Introduction

- The case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are very interesting requirement and design problems to solve. In addition, it is a realistic problem; organizations really do write POS systems using object technologies.

Definition

- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

Architectural Layers

- A typical object-oriented information system is designed in terms of several architectural layers or subsystems
- User Interface—graphical interface; windows.
- Application Logic and Domain Objects—software objects representing domain concepts (for example, a software class named Sale) that fulfill application requirements.
- Technical Services—general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

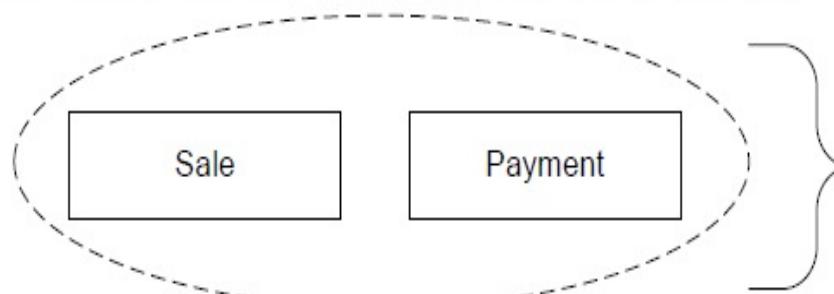
Interface



minor focus

explore how to connect to other layers

application logic and domain object layer



primary focus of case study

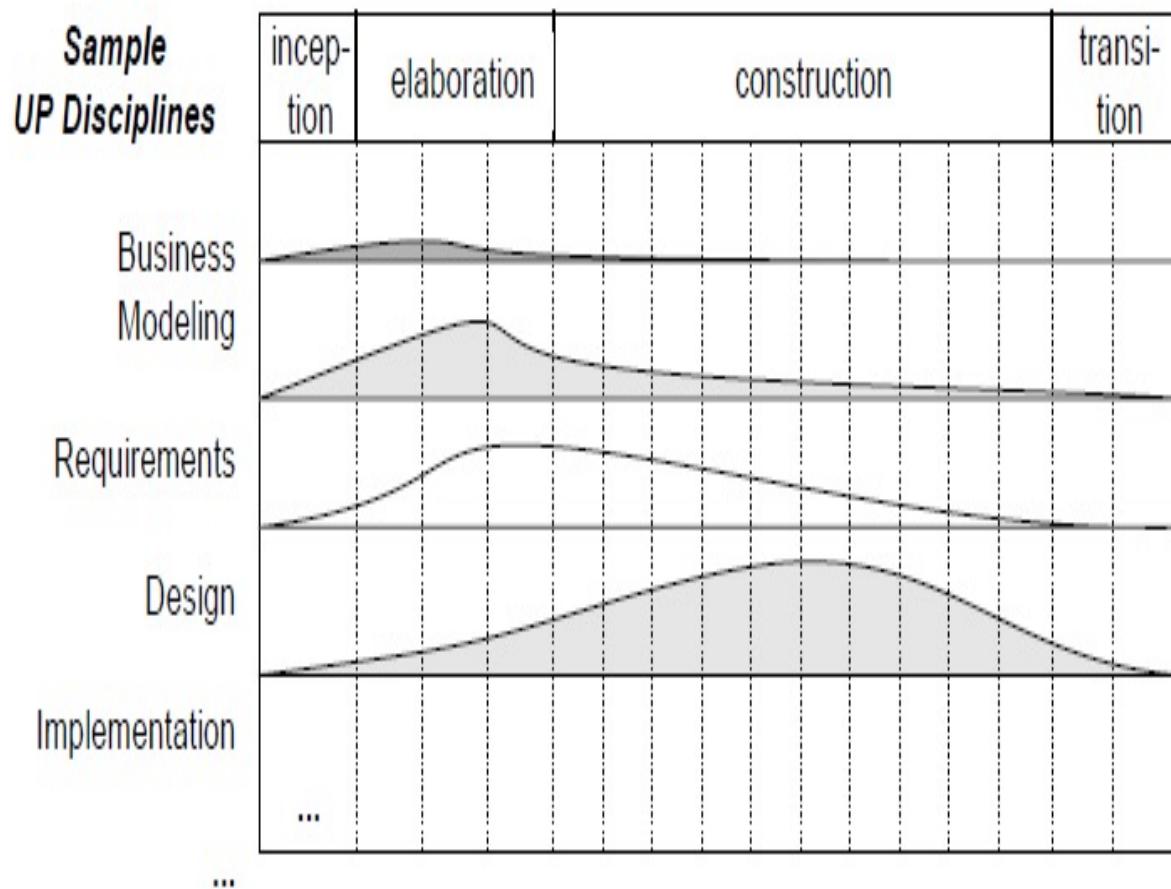
explore how to design objects

technical services layer



secondary focus

explore how to design objects



The relative effort in disciplines shifts across the phases.

This example is suggestive, not literal.

Different Formats

- **brief**—terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.
- **casual**—informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.
- **fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

Only What not How?

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Brief Format

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Casual Format

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

If the credit authorization is reject, inform the customer and ask for an alternate payment method.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external tax calculator system, ...

Fully Dressed

POS

Primary Actor: Cashier

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer short ages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions and Postconditions

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario

Main Success Scenario (or Basic Flow):

1. Customer arrives at 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.
Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

Main Success Scenario

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternate Flows)

Extensions (or Alternative Flows):

*a. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.

2a. System detects anomalies preventing recovery:

1. System signals error to the Cashier, records the error, and enters a clean state.
2. Cashier starts a new sale.

Extensions (or Alternate Flows)

3a. Invalid identifier:

1. System signals error and rejects entry.
- 3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters item identifier for removal from sale.
2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS terminal.

Extensions (or Alternate Flows)

- 5a. System detects failure to communicate with external tax calculation system service:
 1. System restarts the service on the POS node, and continues.
 - 1a. System detects that the service does not restart.
 1. System signals error.
 2. Cashier may manually calculate and enter the tax, or cancel the sale.
- 5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
 1. Cashier signals discount request.
 2. Cashier enters Customer identification.
 3. System presents discount total, based on discount rules.
- 5c. Customer says they have credit in their account, to apply to the sale:
 1. Cashier signals credit request.
 2. Cashier enters Customer identification.
 3. System applies credit up to price=0 and reduces remaining credit

Non Functional Requirements

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.

Technology and Data Variations List:

- 3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

Lecture 9

Use Case Modelling
Inception Phase Requirements
Discipline

Agenda

- Goals and Scope of a Use Case
- Finding Primary Actors, Goals, and Use Cases

How should use cases be discovered?

- Which of these is a valid use case?
- Negotiate a Supplier Contract
- Handle Returns
- Log In

EBP

- For requirements analysis for a computer application, focus on use cases at the level **of elementary business processes (EBPs)**.
 - A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state. e.g., Approve Credit or Price Order [original source lost].

Use Cases and Goals

- And it leads to a recommended procedure:
- 1. Find the user goals.
- 2. Define a use case for each.

Finding Primary Actors, Goals, and Use Cases

- Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
- Identify the primary actors.
- For each, identify their user goals. Raise them to the highest user goal level that satisfies the EBP guideline.
- Define use cases that satisfy user goals; name them according to their goal.

Step 1: Choosing the System Boundary

- For this case study, the POS system itself is the system under design;
- everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on.

Steps 2 and 3: Finding Primary Actors and Goals

In addition to obvious primary actors and user goals, the following questions help identify others that may be missed:

Who starts and stops the system?

Who does user and security management?

Is there a monitoring process that restarts the system if it fails?

How are software updates handled?
Push or pull update?

Who does system administration?

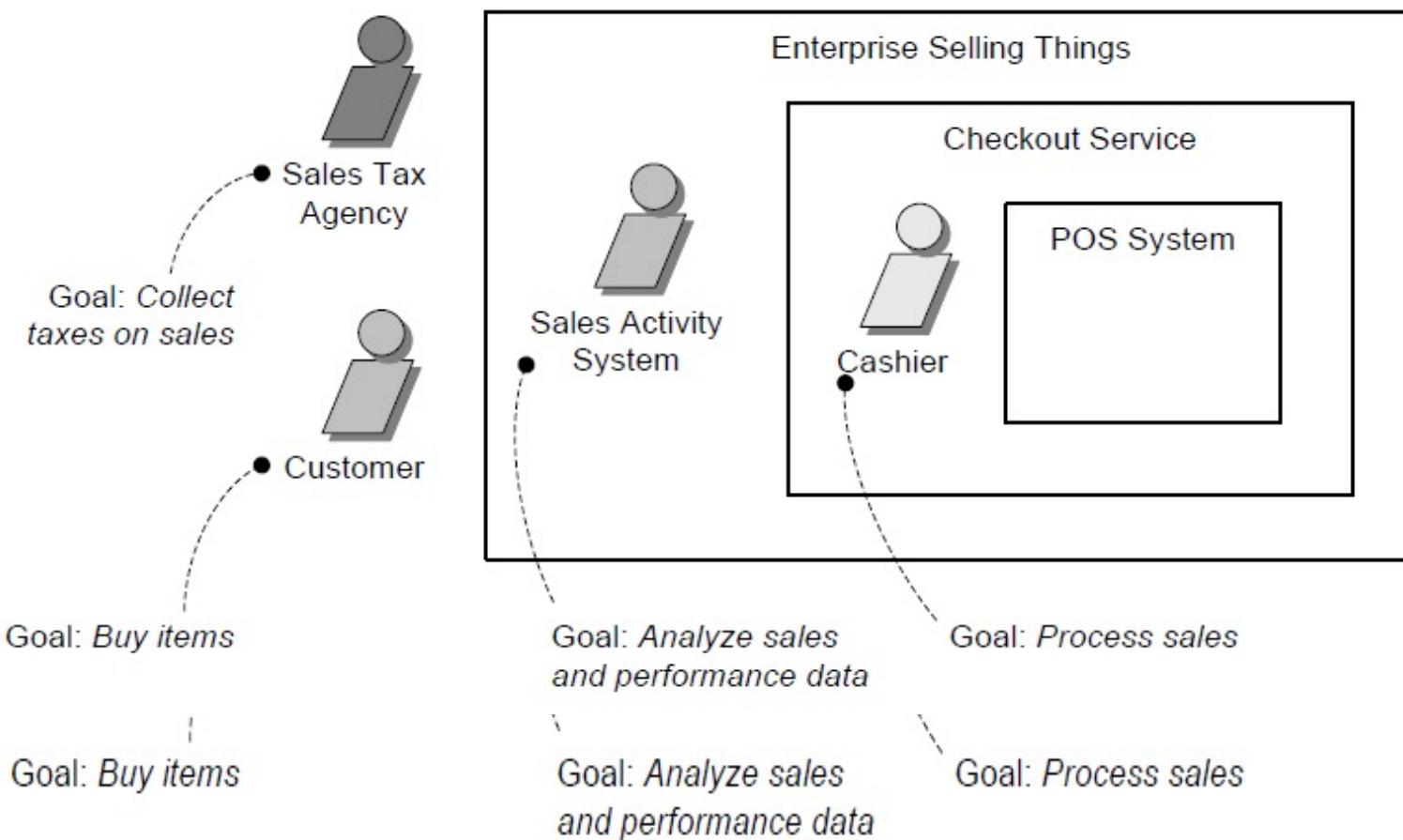
Is "time" an actor because the system does something in response to a time event?

Who evaluates system activity or performance?

Who evaluates logs? Are they remotely retrieved?

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out ...	System Administrator	add users modify users delete users manage security manage system tables ...
Manager	start up shut down ...	Sales Activity System	analyze sales and performance data
...

Primary actors and goals at different system boundaries

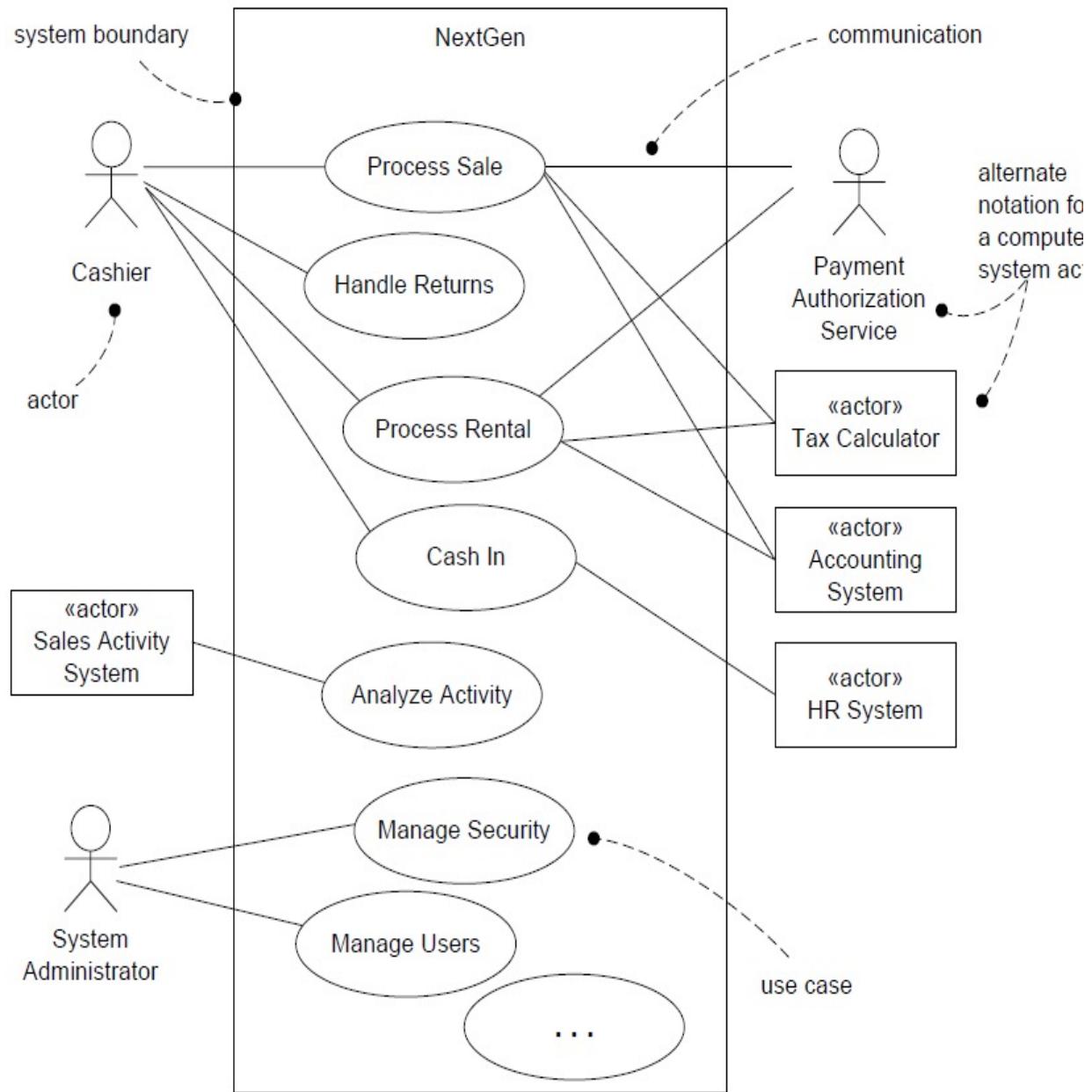


Actors and Goals via Event Analysis

External Event	From Actor	Goal
enter sale line item	Cashier	process a sale
enter payment	Cashier or Customer	process a sale
...		

Actors

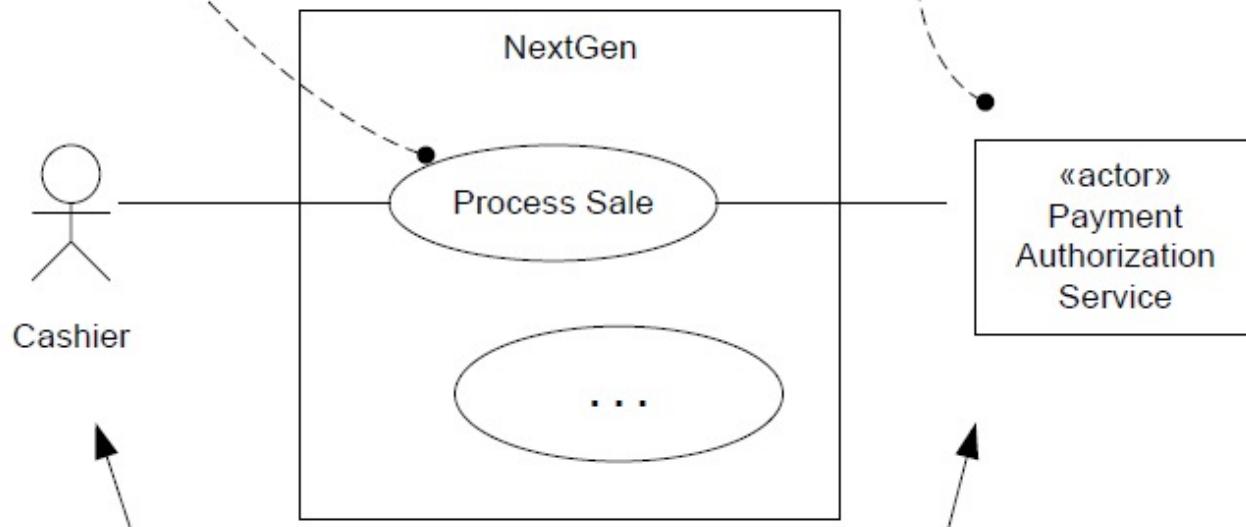
- **Primary actor**—has user goals fulfilled through using services of the SuD. For example, the cashier.
 -) Why identify? To find user goals, which drive the use cases.
- **Supporting actor**—provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
 -) Why identify? To clarify external interfaces and protocols.
- **Offstage actor**—has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
 -) Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.



Diagramming Suggestions

For a use case context diagram, limit the use cases to user-goal level use cases.

Show computer system actors with an alternate notation to human actors.



primary actors on
the left

supporting actors
on the right

High-Level System Feature Lists Are Acceptable

Summary of System Features

- sales capture
- payment authorization (credit, debit, check)
- system administration for users, security, code and constants tables, and so on
- automatic offline sales processing when external components fail
- real-time transactions, based on industry standards, with third-party systems, including inventory, accounting, human resources, tax calculators, and payment authorization services
- definition and execution of customized "pluggable" business rules at fixed, common points in the processing scenarios
- ...

Use Cases Are Not Object-Oriented

- use cases are a broadly applicable requirements analysis tool that can be applied to non-object-oriented projects, which increases their usefulness as a requirements method.
- However, as will be explored, use cases are a pivotal input into classic OOA/D activities

Discipline	Artifact Iteration->	Incep. II	Elab. El..En	Const. CL..Cn	Trans. T1..T2
Business Modeling	Domain Model		S		
Requirements	<i>Use-Case Model</i>	S	I		
	Vision	S	I		
	Supplementary Specification	S	I		
	Glossary	S	I		
Design	Design Model		S	I	
	SW Architecture Document		S		
	Data Model		S	I	
Implementation	Implementation Model		S	I	I
Project Management	SW Development Plan	S	I	I	I
Testing	Test Model		S	I	
Environment	Development Case	S	I		

References

- Chapter 6 Applying UML and Patterns by Craig Larman

Lecture 10

INCEPTION TO
ELABORATION

Agenda

- Define the elaboration step.

Elaboration

- The majority of requirements are discovered and stabilized
- The major risks are mitigated or retired
- The core architectural elements are implemented and proven

What is Inception Phase

- may last only one week
- The artifacts created should be brief and incomplete
- It is not the requirements phase of the project, but a short step to determine basic feasibility, risk, and scope, and decide if the project is worth more serious investigation, which occurs in elaboration

Activities and Artifacts in inception

- a short requirements workshop
- most actors, goals, and use cases named
- most use cases written in brief format; 10-20% of the use cases are written in fully dressed detail to improve understanding of the scope and complexity
- most influential and risky quality requirements identified
- version one of the Vision and Supplementary Specification written
- Risk list

Contd..

- technical proof-of-concept prototypes and other investigations to explore the technical feasibility of special requirements
- user interface-oriented prototypes to clarify the vision of functional requirements
- recommendations on what components to buy/build/reuse, to be refined in elaboration
- Candidate Tool List

Elaboration

- the core architecture, clarifies most requirements, and tackles the high-risk issues
- Elaboration often consists of between two and four iterations; each iteration is recommended to be between two and six weeks, unless the team size is massive
- During this phase, one is not creating throw-away prototypes; rather, the code and design are production-quality portions of the final system.

Fuzzy Grouping of requirements

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging ...	Scores high on all ranking criteria. Pervasive. Hard to add late. ...
Medium	Maintain Users ...	Affects security subdomain. ...
Low

Plans

- **Iteration Plan**
- **Change Request**
- **Software Development Plan.**

What Artifacts May Start in Elaboration?

- Domain Model
 - This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
- Design Model
 - This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.

Contd..

- Software Architecture Document
 - A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
- Data Model
 - This includes the database schemas, and the mapping strategies between object and non-object representations.

- Test Model
 - A description of what will be tested, and how.
- Implementation Model
 - This is the actual implementation — the source code, executables, database, and so on.
- Use-Case Storyboards, UI Prototypes
 - A description of the user interface, paths of navigation, usability models, and so forth.

References

- Chapter 8 **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**

Lecture 11

USE-CASE MODEL: DRAWING SYSTEM
SEQUENCE DIAGRAMS

Agenda

- Identify system events.
- Create system sequence diagrams for use cases.

System Sequence Diagram

– SYSTEM SEQUENCE DIAGRAM:-

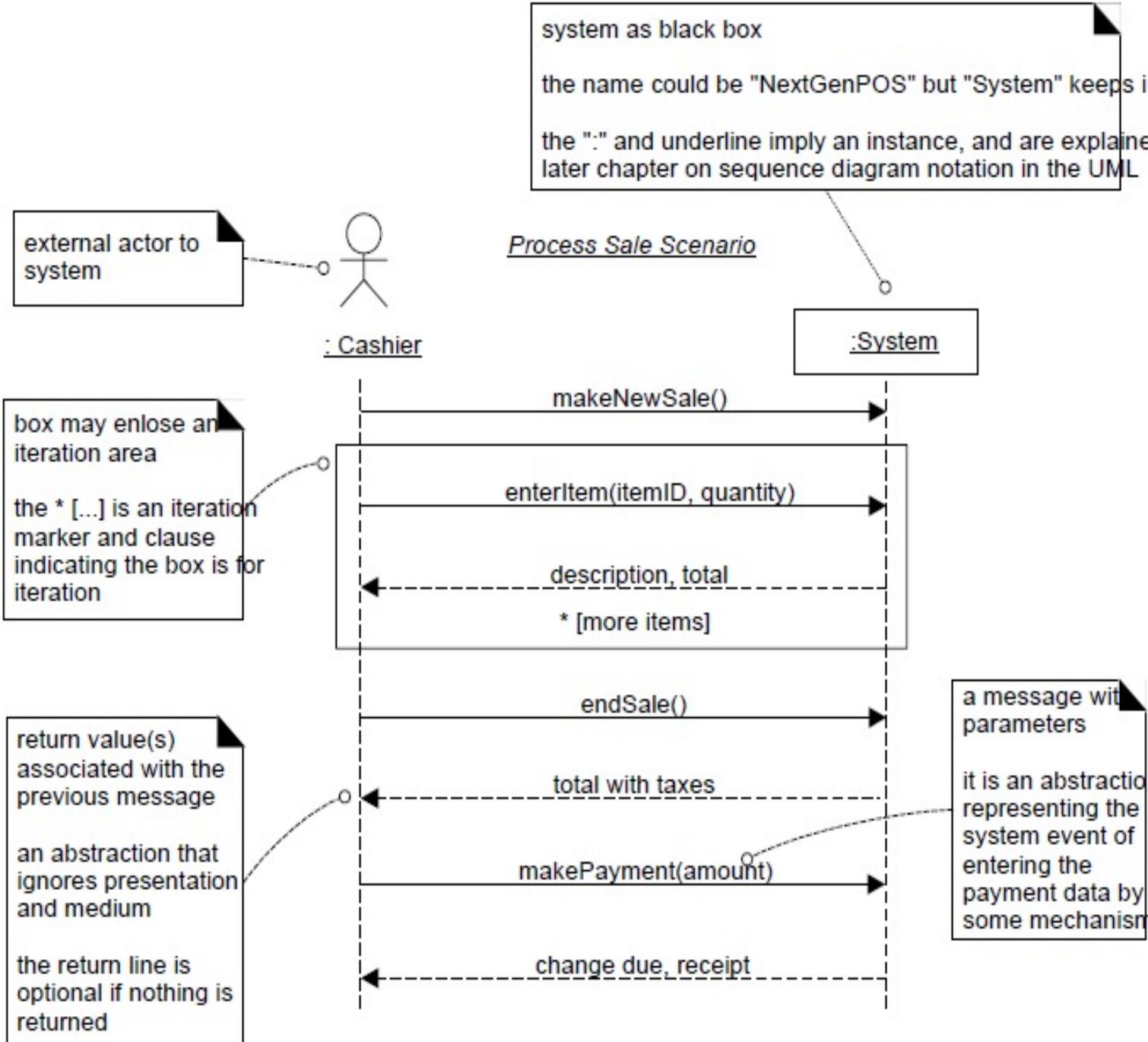
- Illustrates input and output events related to the system.
- Here the system behaves as “Black Box”.

Use cases describe

- How actors interact with system.
- Typical course of events that external actors generate.
- The order of the events.

Sequence Diagram

- A System sequence diagram visualizes a use case, while a sequence diagram visualizes a method of a class.
- The elements participating (exchanging messages) in a **system sequence diagram** are Actors and Systems. The messages exchanged by these elements could be any type depending on the systems (from web service calls to data input from a human).
- The elements participating in a **sequence diagram** are objects (instances of various classes). The messages exchanged by these elements are method invocations.



Phases

- **Inception**—SSDs are not usually motivated in inception.
- **Elaboration**—Most SSDs are created during elaboration

Discipline	Artifact Iteration→	Incep. 11	Elab. El. En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	<i>Use-Case Model (SSDs)</i>	s	r		
	Vision	s	r		
Design	Supplementary Specification	s	r		
	Glossary	s	r		
	Design Model		s	r	
	SW Architecture Document		s		
Implementation	Data Model		s	r	
	Implementation Model		s	r	R
Project Management	SW Development Plan	s	r	r	R
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 9.1 Sample UP artifacts and timing, s - start; r - refine

References

- Chapter 9 Applying UML Patterns (**Applying UML Patterns: An Introduction To Object-Oriented Analysis And Design**) Craig L

DOMAIN MODEL VISUALIZING CONCEPTS

Biju R Mohan

Lecture 12

Agenda

- Identify conceptual classes related to the current iteration requirements.
- Create an initial domain model.
- Distinguish between correct and incorrect attributes.
- Add *specification conceptual classes, when appropriate.*
- Compare and contrast conceptual and implementation views.

What is domain model ?

- A domain model illustrates meaningful conceptual classes in a problem domain; it is the most important artifact to create during object-oriented analysis.
- Identifying a rich set of objects or conceptual classes is at the heart of object-oriented analysis, and well worth the effort in terms of payoff during the design and implementation work.
- The identification of conceptual classes is part of an investigation of the problem domain. The UML contains notation in the form of class diagrams to illustrate domain models.

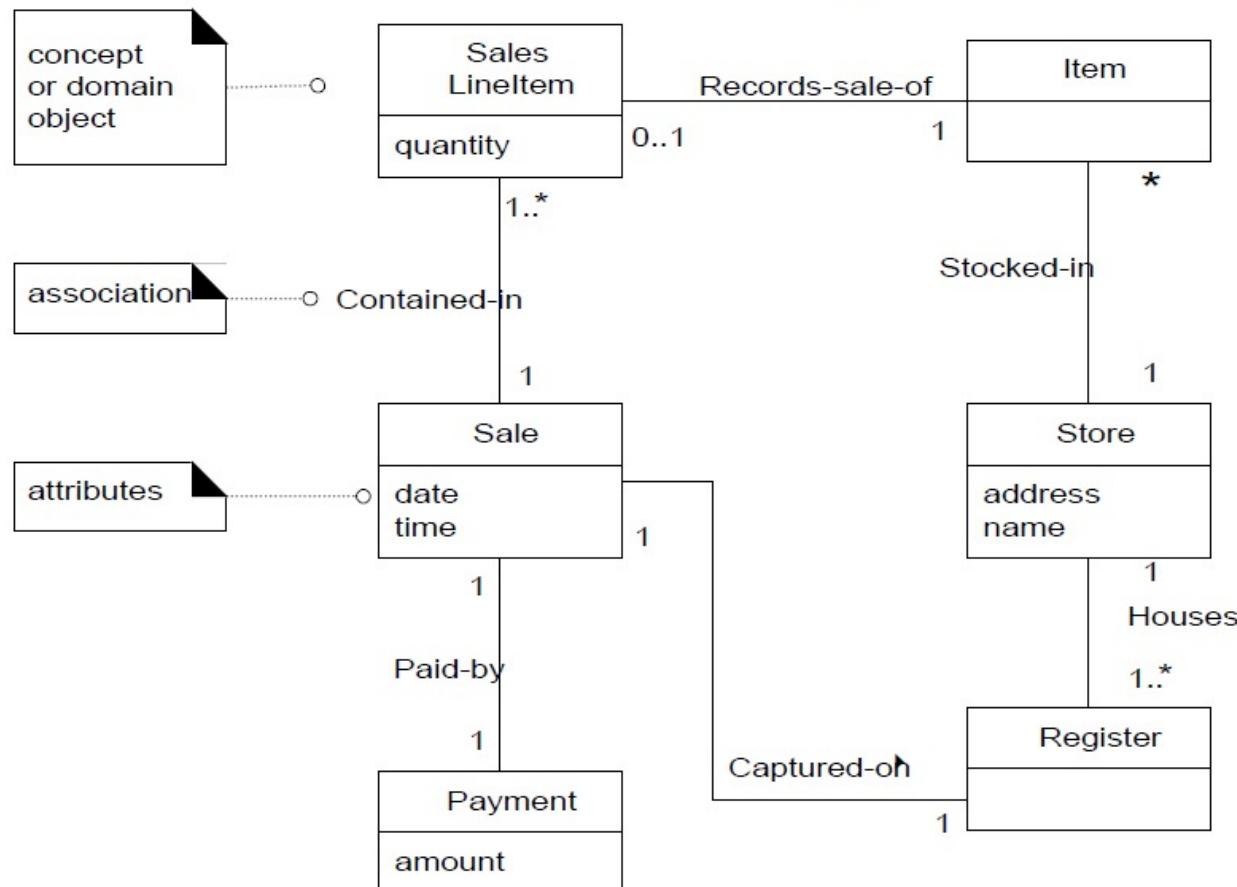
Key Point

- A domain model is a representation of real-world conceptual classes, not of software components. It is *not a set of diagrams describing software classes, or software objects with responsibilities.*

Domain Model

- The UP defines a Domain Model as one of the artifacts that may be created in the Business Modeling discipline.
- Using UML notation, a domain model is illustrated with a set of **class diagrams**
 - domain objects or conceptual classes
 - associations between conceptual classes
 - attributes of conceptual classes

Partial Domain Model of POS problem



Domain Models Are not Models of Software Components

- A domain model is a visualization of things in the realworld domain of interest, *not of software components such as a Java or C++*
- Therefore, the following elements are not suitable in a domain model:
 - Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
 - Responsibilities or methods.

Example



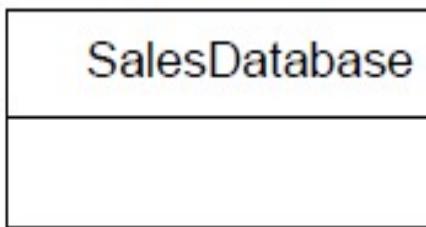
0

visualization of a real
world concept in the
domain of interest

it is ~~an~~ a picture of a
software class

Avoid

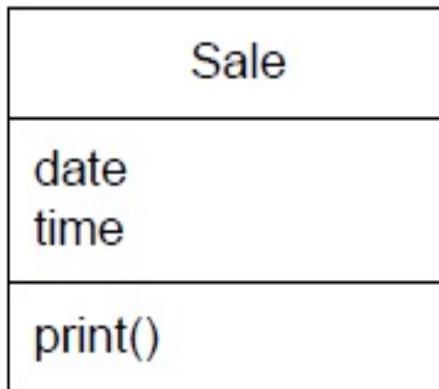
avoid



0.....

software artifact; not part
of domain model

avoid



0.....

software class; not part
of domain model

Conceptual Classes

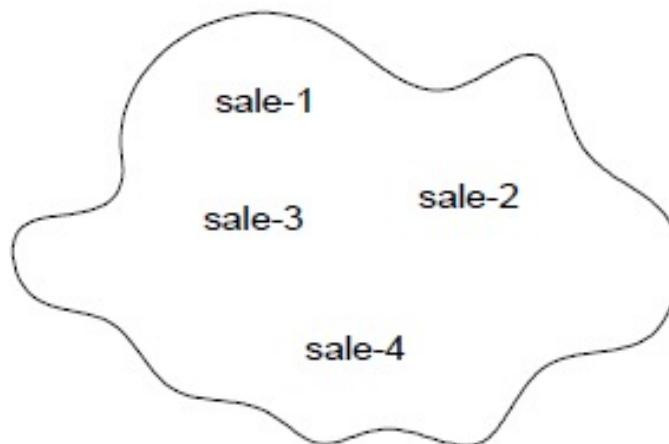
- Informally, a conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension
 - **Symbol**—words or images representing a conceptual class.
 - **Intension**—the definition of a conceptual class.
 - **Extension**—the set of examples to which the conceptual class applies



concept's symbol

"A sale represents the event of a purchase transaction. It has a date and time."

concept's intension



o

concept's extension

OOAD Vs Structured Analysis

- A central distinction between object-oriented and structured analysis is: division by conceptual classes (objects) rather than division by functions

Conceptual Class Identification

- Two techniques are presented in the following sections:
 - 1. Use a conceptual class category list.
 - 2. Identify noun phrases.

Conceptual Class Category	Examples
physical or tangible objects	<i>Register</i> <i>Airplane</i>
specifications, designs, or descriptions of things	<i>ProductSpecification</i> <i>FlightDescription</i>
places	<i>Store</i> <i>Airport</i>
transactions	<i>Sale</i> , <i>Payment</i> <i>Reservation</i>
transaction line items	<i>SalesLineItem</i>
roles of people	<i>Cashier</i> <i>Pilot</i>
containers of other things	<i>Store</i> , <i>Bin</i> <i>Airplane</i>
things in a container	<i>Item</i> <i>Passenger</i>

Finding Conceptual Classes with Noun Phrase Identification

Main Success Scenario (or Basic Flow):

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**. Price calculated from a set of price rules.
Cashier repeats steps 2-3 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.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions**) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

Candidate Conceptual Classes for the Sales Domain

Register

ProductSpecification

Item

SalesLineItem

Store

Cashier

Sale

Customer

Payment

Manager

ProductCatalog

How to Make a Domain Model

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory (discussed in a subsequent chapter).
4. Add the attributes necessary to fulfill the information requirements (discussed in a subsequent chapter).

On Naming and Modeling Things

Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory.
- Exclude irrelevant features.
- Do not add things that are not there.

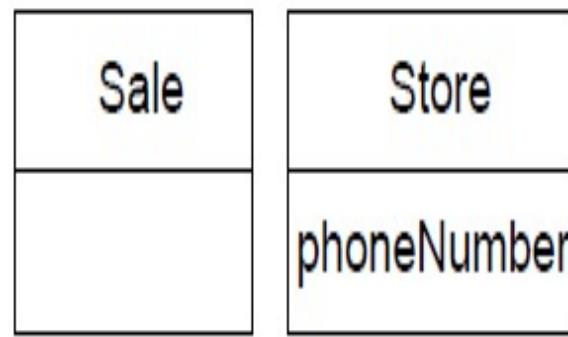
A Common Mistake in Identifying Conceptual Classes

- If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

Example



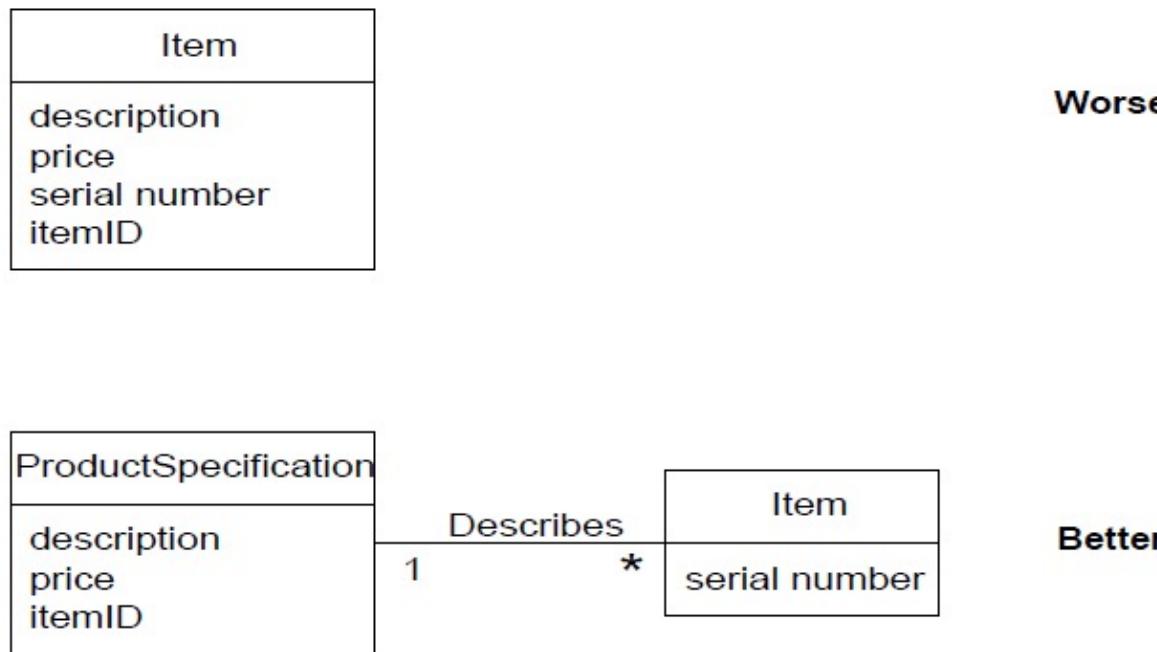
or... ?



Modeling the *Unreal World*

- Some software systems are for domains that find very little analogy in natural or business domains; software for telecommunications is an example.
- For example, here are some candidate conceptual classes related to a telecommunication
- switch: *Message, Connection, Port, Dialog, Route, Protocol.*

The Need for Specification or Description Conceptual Classes



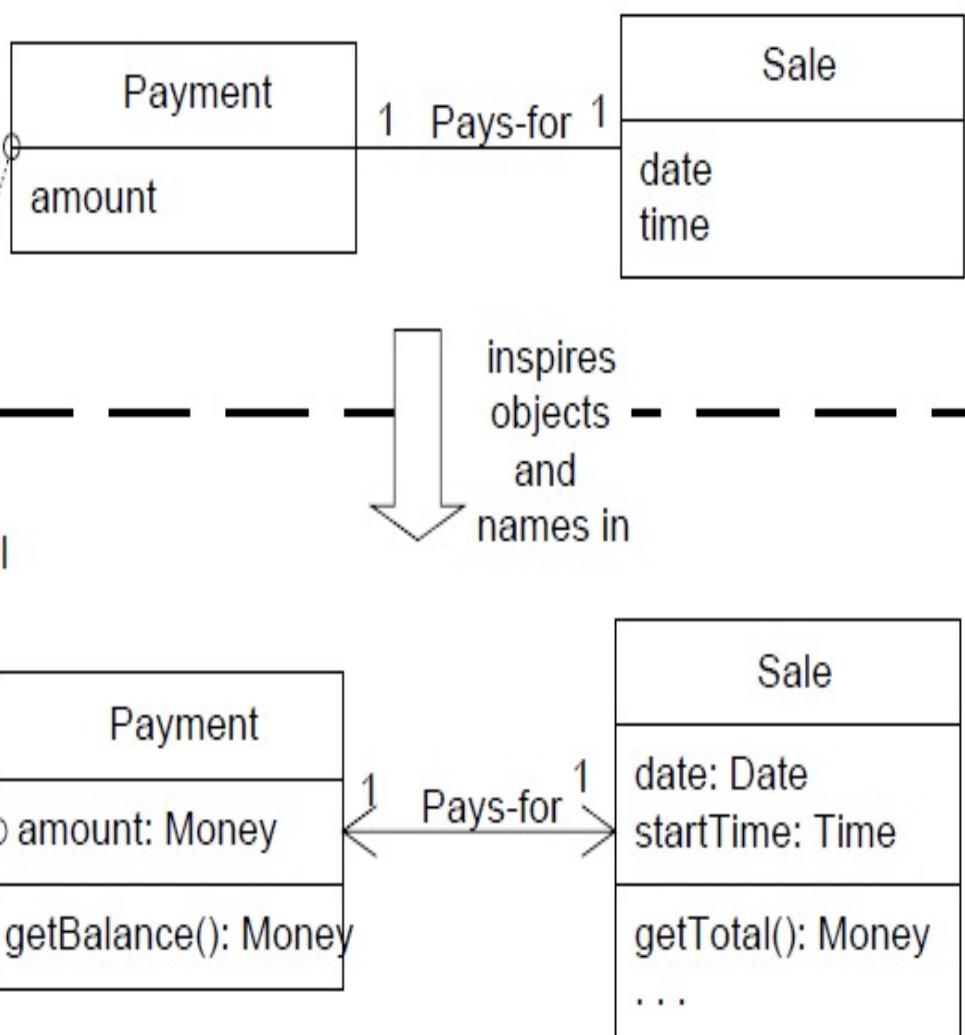
UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former ~~inspires~~ the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



UP Design Model

The object-oriented developer has taken inspiration from the real world in creating software classes.

References

- Chapter 10 Applying UML Patterns (**Applying UML Patterns: An Introduction To Object-Oriented Analysis And Design**) Craig L

DOMAIN MODEL: ADDING ASSOCIATIONS

Biju R Mohan

Lecture 13

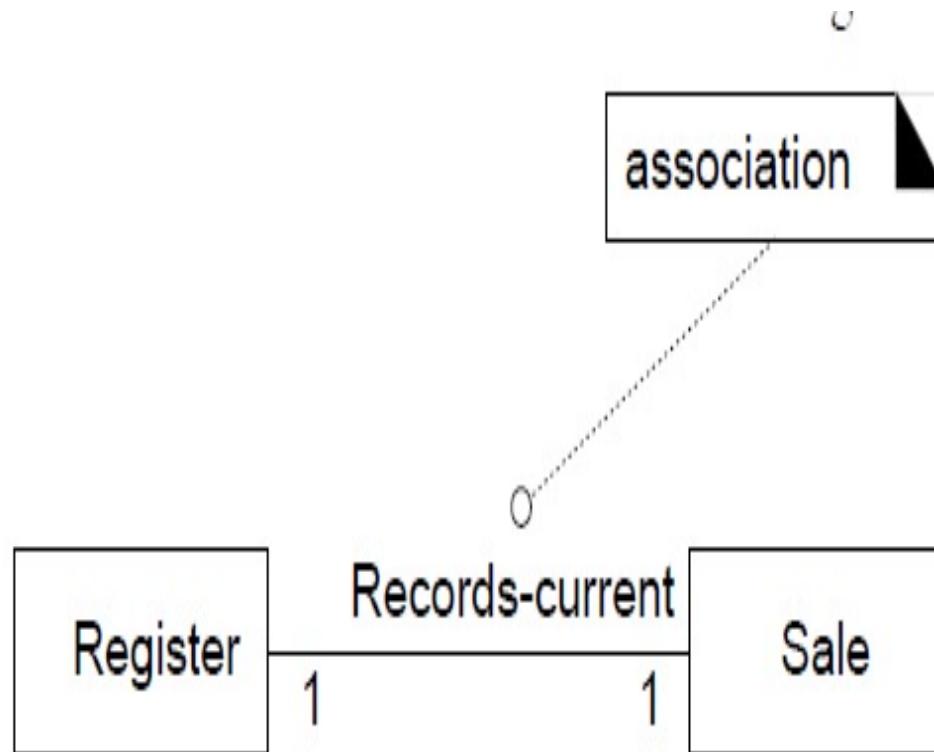
Agenda

- Identify associations for a domain model.
- Distinguish between need-to-know and comprehension-only associations.

Associations

- An **association is a relationship between types (or more specifically, instances of those types)** that indicates some meaningful and interesting connection

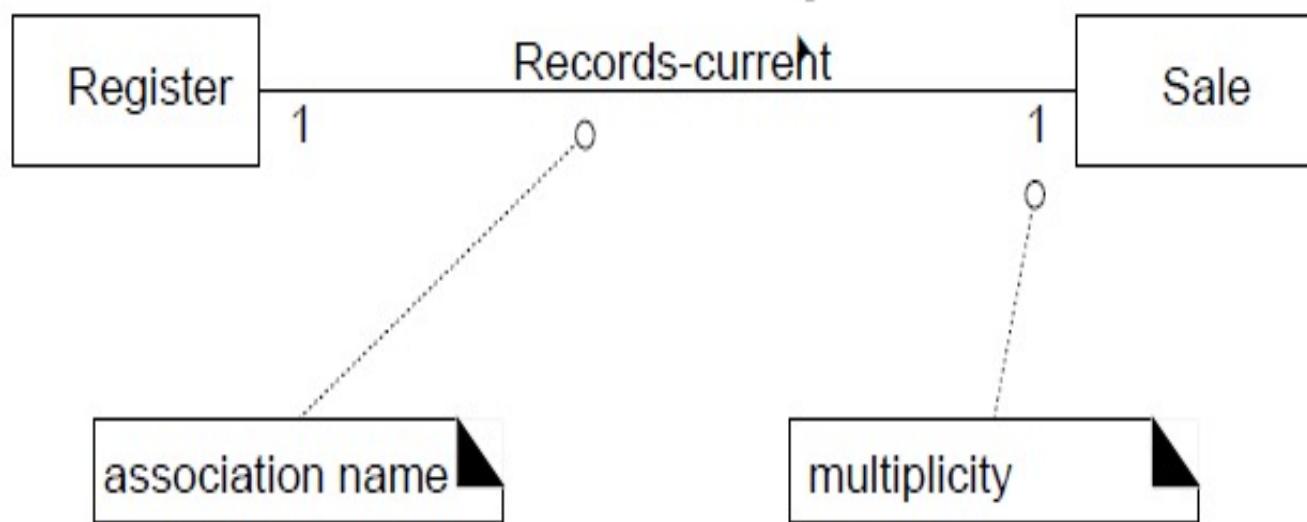
UML Notation of Association



Types

- Consider including the following associations in a domain model:
 - Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
 - Associations derived from the Common Associations List.

-"reading direction arrow"
-it has no meaning except to indicate direction
reading the association label
-often excluded



Category	Examples
A is a physical part of B	<i>Drawer</i> — <i>Register</i> (or more specifically, a <i>POST</i>) <i>Wing</i> — <i>Airplane</i>
A is a logical part of B	<i>SalesLineItem</i> — <i>Sale</i> <i>FlightLeg</i> — <i>FlightRoute</i>
A is physically contained in/on B	<i>Register</i> — <i>Store</i> , <i>Item</i> — <i>Shelf</i> <i>Passenger</i> — <i>Airplane</i>
A is logically contained in B	<i>ItemDescription</i> — <i>Catalog</i> <i>Flight</i> — <i>FlightSchedule</i>
A is a description for B	<i>ItemDescription</i> — <i>Item</i> <i>FlightDescription</i> — <i>Flight</i>
A is a line item of a transaction or report B	<i>SalesLineItem</i> — <i>Sale</i> <i>Maintenance Job</i> — <i>Maintenance-Log</i>
A is known/logged/recorded/reported/captured in B	<i>Sale</i> — <i>Register</i> <i>Reservation</i> — <i>FlightManifest</i>
A is a member of B	<i>Cashier</i> — <i>Store</i> <i>Pilot</i> — <i>Airline</i>

A is a member of B	<i>Cashier</i> — <i>Store</i> <i>Pilot</i> — <i>Airline</i>
A is an organizational subunit of B	<i>Department</i> — <i>Store</i> <i>Maintenance</i> — <i>Airline</i>
A uses or manages B	<i>Cashier</i> — <i>Register</i> <i>Pilot</i> — <i>Airplane</i>
A communicates with B	<i>Customer</i> — <i>Cashier</i> <i>Reservation Agent</i> — <i>Passenger</i>
A is related to a transaction B	<i>Customer</i> — <i>Payment</i> <i>Passenger</i> — <i>Ticket</i>
A is a transaction related to another transaction B	<i>Payment</i> — <i>Sale</i> <i>Reservation</i> — <i>Cancellation</i>
A is next to B	<i>SalesLineItem</i> — <i>SalesLineItem</i> <i>City</i> — <i>City</i>

Category	Examples
A is owned by B	<i>Register</i> — <i>Store</i> <i>Plane</i> — <i>Airline</i>
A is an event related to B	<i>Sale</i> — <i>Customer</i> , <i>Sale</i> — <i>Store</i> <i>Departure</i> — <i>Flight</i>

High-Priority Associations

Here are some high-priority association categories that are invariably useful to include in a domain model:

- A is a *physical or logical part* of B.
- A is *physically or logically contained* in/on B.
- A is *recorded in* B.

Association Guidelines

- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- It is more important to identify *conceptual classes* than to identify associations.
- Too many associations tend to confuse a domain model rather than illuminate it. Their discovery can be time-consuming, with marginal benefit.
- Avoid showing redundant or derivable associations.

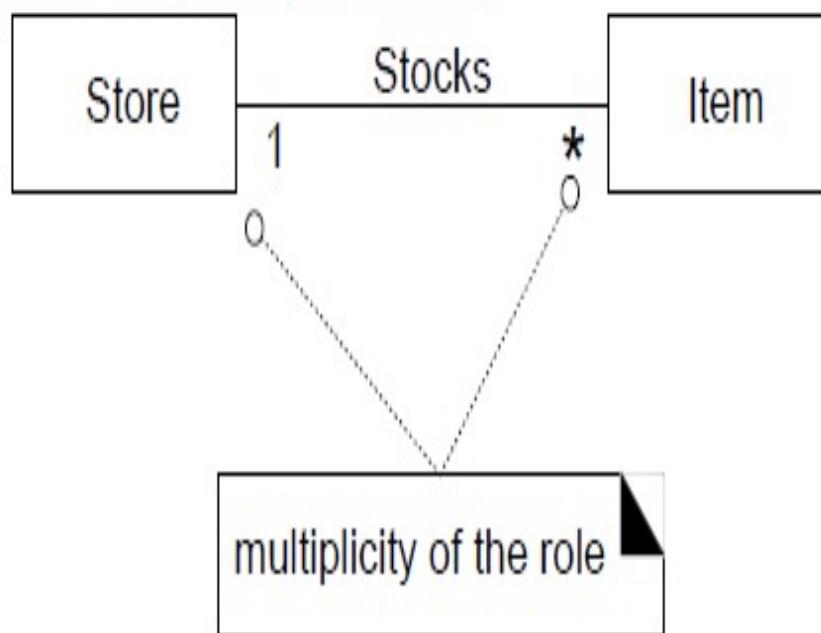
Roles

Each end of an association is called a **role**. Roles may optionally have:

- name
- multiplicity expression
- navigability

Multiplicity

Multiplicity defines how many instances of a class *A* can be associated with one instance of a class *B* (see Figure 11.3).





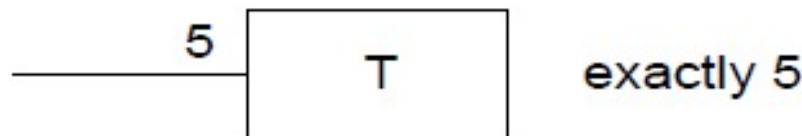
zero or more;
"many"



one or more



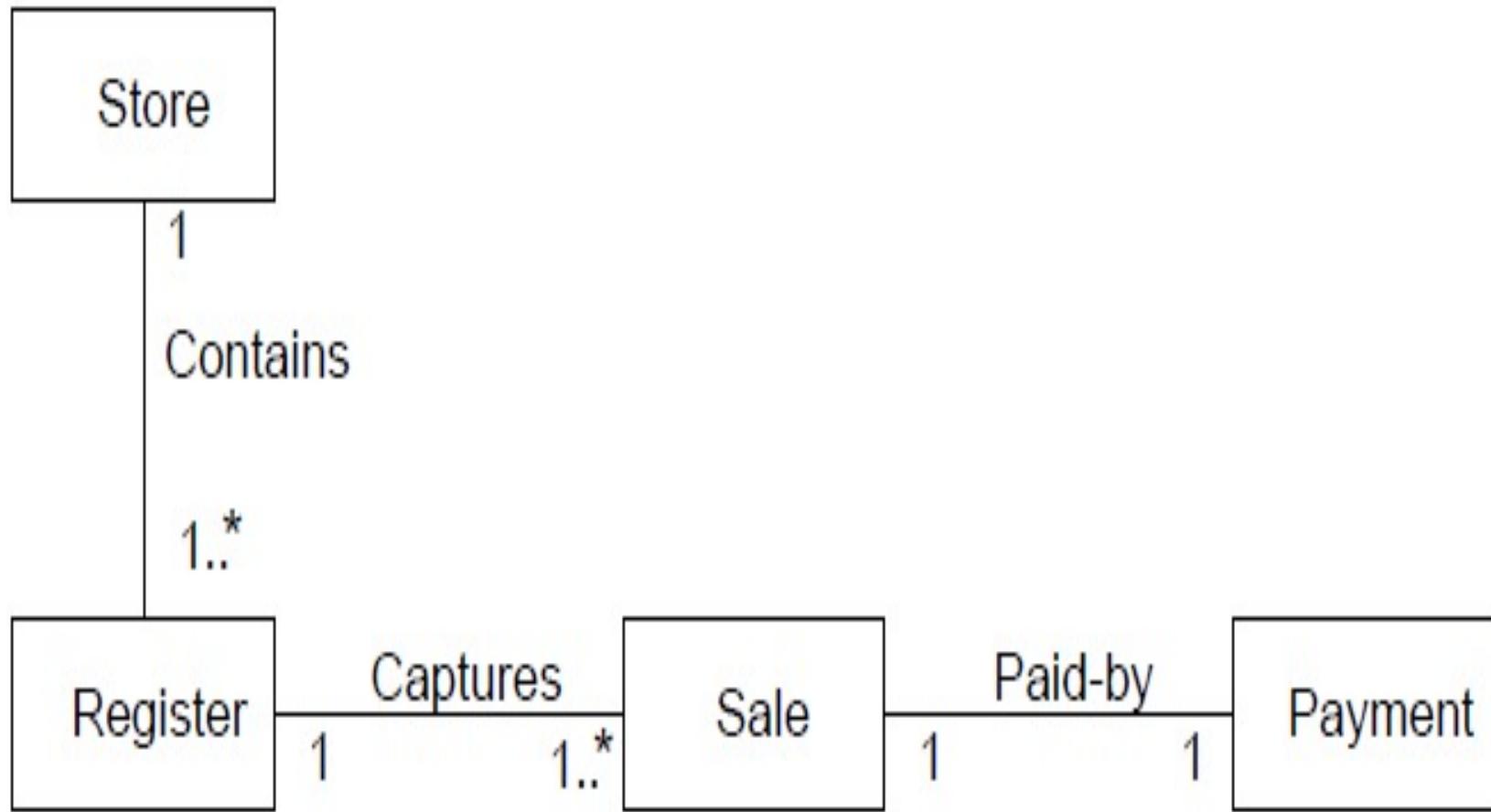
one to 40

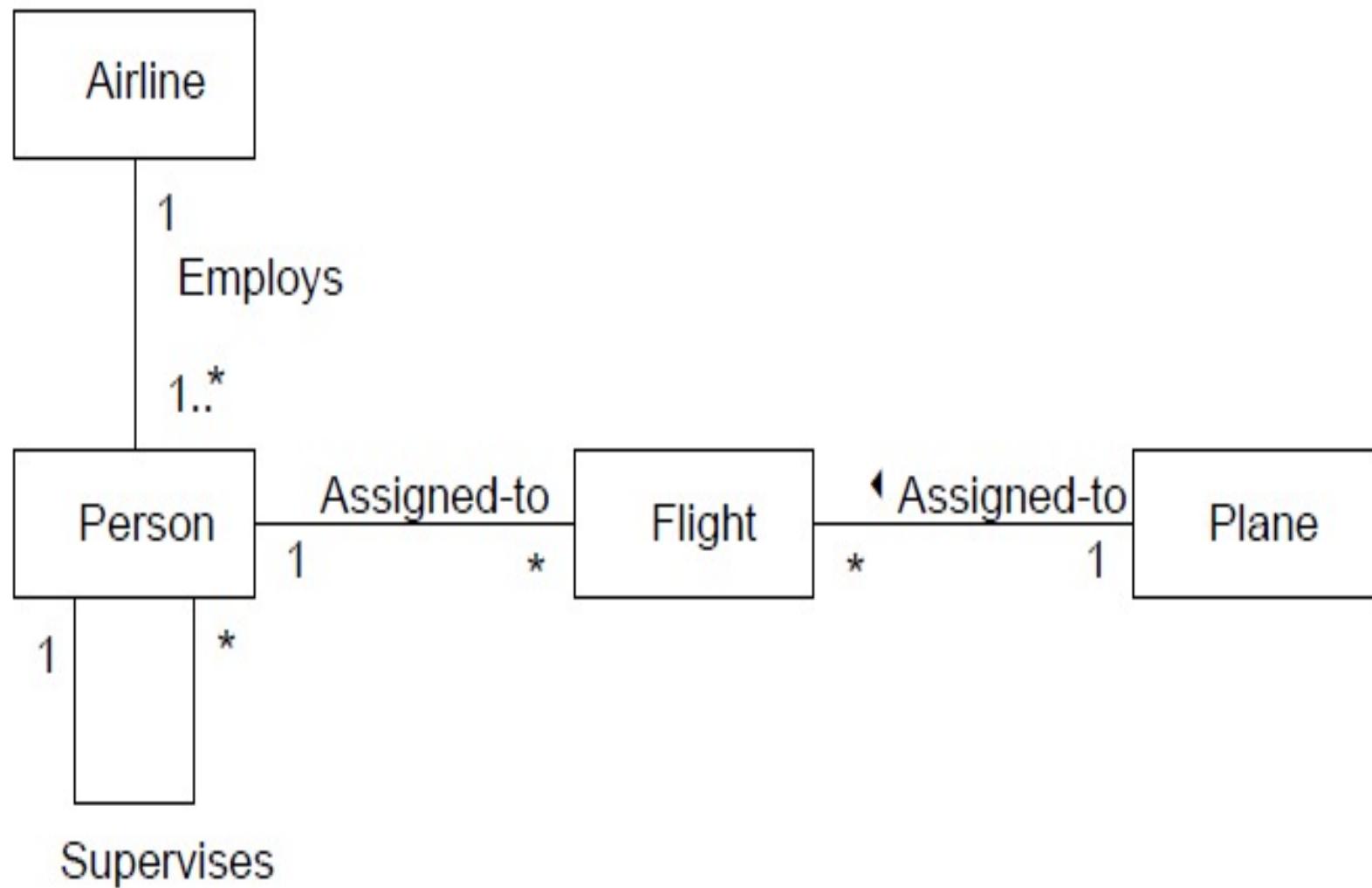


exactly 5

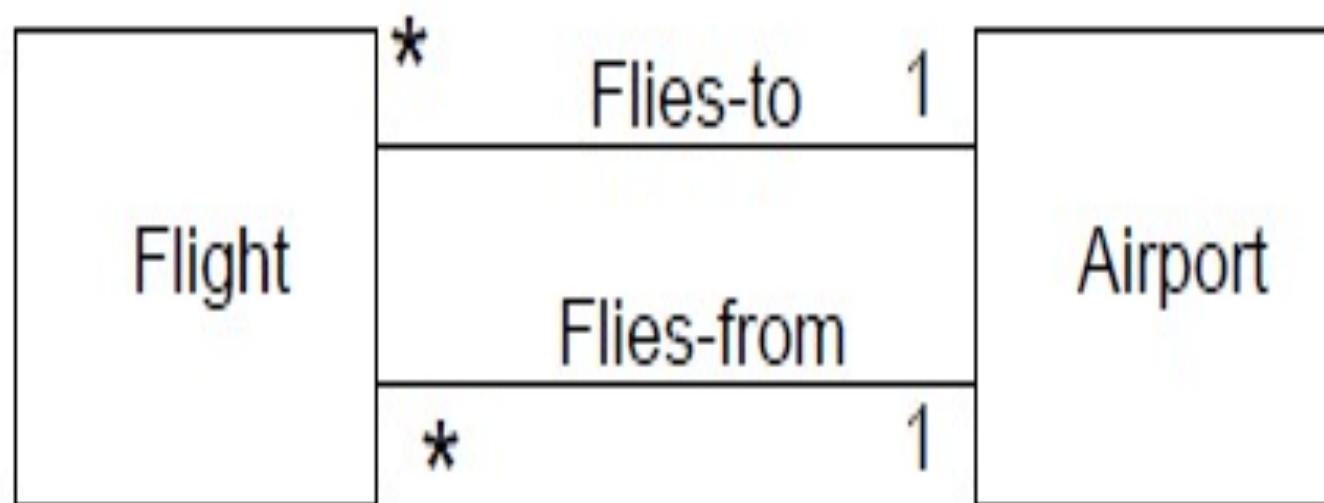


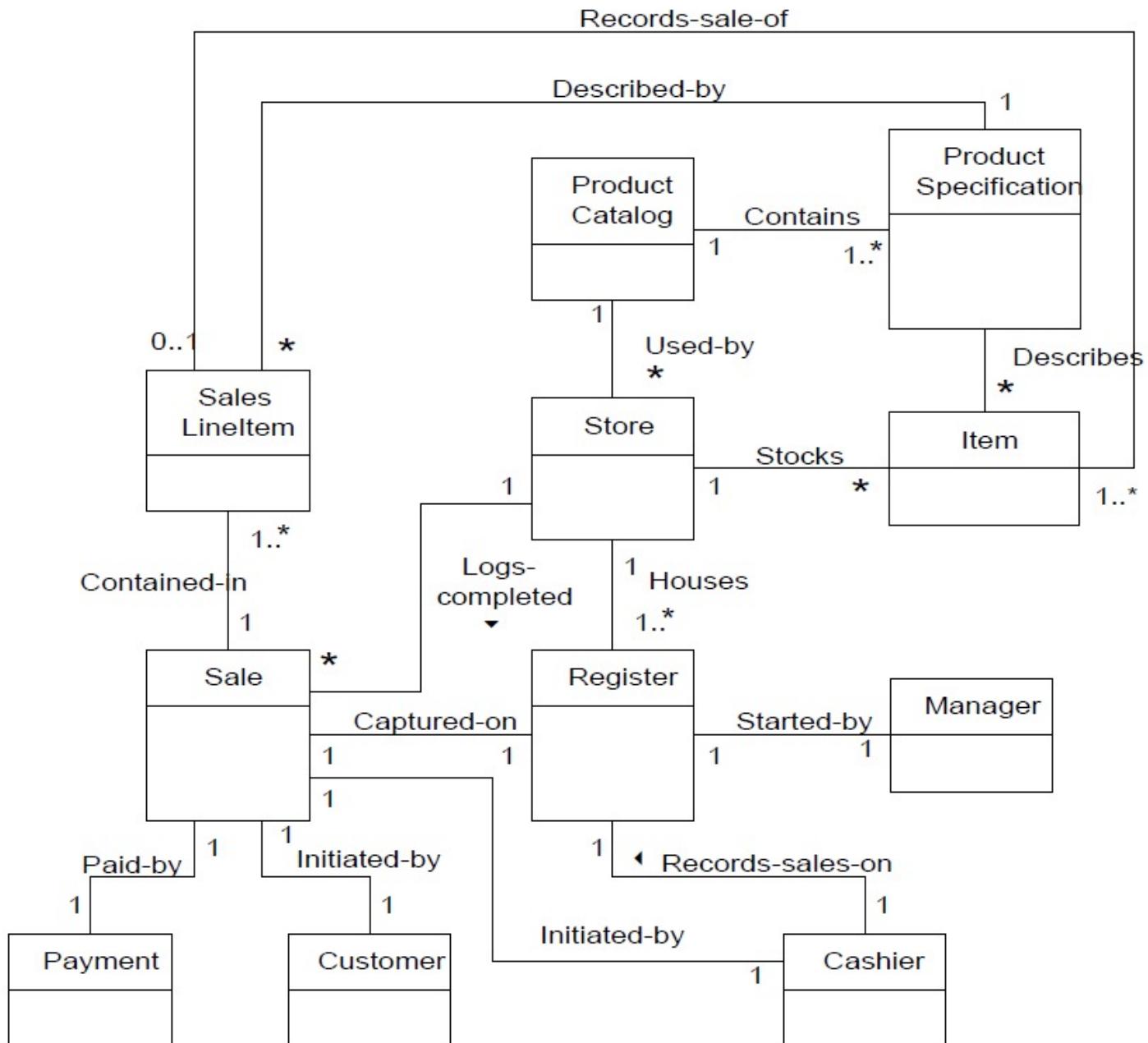
exactly 3, 5, or





Multiple Associations Between Two Types





Reference

- Chapter 11 Applying UML Patterns (**Applying UML Patterns: An Introduction To Object-Oriented Analysis And Design**) Craig L

Lecture 14

Design Patterns

What is a Design Pattern?

- A (Problem, Solution) pair.
- A technique to repeat designer success.
- Borrowed from Civil and Electrical Engineering domains.

How Patterns are used?

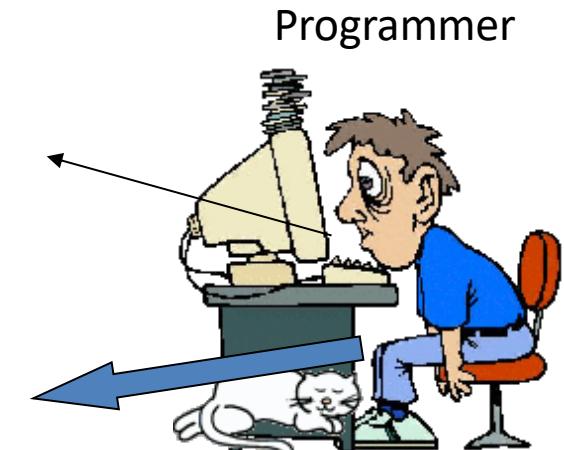
Designer

- Design Problem.
- Solution.

Implementation details.



Reduce gap



Implementation

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. 1995.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-oriented software architecture: a system of patterns*. 2002.

Design patterns you have already seen

- Encapsulation (Data Hiding)
- Subclassing (Inheritance)
- Iteration
- Exceptions

Encapsulation pattern

- **Problem:** Exposed fields are directly manipulated from outside, leading to undesirable dependences that prevent changing the implementation.
- **Solution:** Hide some components, permitting only stylized access to the object.

Subclassing pattern

- **Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.
- **Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

Iteration pattern

- **Problem:** Clients that wish to access all members of a collection must perform a specialized traversal for each data structure.
- **Solution:** Implementations perform traversals. The results are communicated to clients via a standard interface.

Exception pattern

- **Problem:** Code is cluttered with error-handling code.
- **Solution:** Errors occurring in one part of the code should often be handled elsewhere. Use language structures for throwing and catching exceptions.

Derived Conclusion

- ~~Patterns are Programming language features.~~
- Programming languages are moving towards Design.
- Many patterns are being implemented in programming languages.

Pattern Categories

- **Creational Patterns** concern the process of object creation.
- **Structural Patterns** concern with integration and composition of classes and objects.
- **Behavioral Patterns** concern with class or object communication.

What is the addressing Quality Attribute?

- Modifiability, Exchangeability, Reusability, Extensibility, Maintainability.

What properties these patterns provide?

- More general code for better Reusability.
- Redundant code elimination for better Maintainability.

What is the Singleton Pattern?

- The Singleton pattern ensures that a class is only instantiated once and provides a global access point for this instance

Creational Pattern

- Abstracts the instantiation process
- Object Creational Pattern
 - Delegates the instantiation to another object

Why use the Singleton Pattern?

- It is important for some classes to only have one instance
- It is also important that this single instance is easily accessible
- Why not use a global object?
 - Global variables make objects accessible, but they don't prevent the instantiation of multiple objects.

Solution

- Make the class responsible for keeping track of its only instance
 - The class can ensure that no other instances are created
 - This provides a useful way to access the instance
 - This is the Singleton pattern

Examples of Singleton Patterns

- Print Spooler
- File Systems
- Window Managers
- Digital Filters
- Accounting Systems

Sample Class

```
class Singleton
{
    // static variable single_instance of
    private static Singleton single_instar

    // variable of type String
    public String s;
```

Sample Implementation

```
// private constructor restricted to th:  
private Singleton()  
{  
    s = "Hello I am a string part of Sing:  
}  
// static method to create instance of :  
}  
e;  
};
```

```
public static Singleton  
getInstance()  
{  
    if (single_instance == null)  
        single_instance = new  
Singleton();  
  
    return single_instance;  
}
```

Things to Notice

- `getInstance()` function ensures that only one instance is created and that it is initialized before use
- Singleton constructor is declared as **private**
 - Direct instantiation causes errors at compile time

Benefits of the Singleton Pattern

- Controlled access to sole instance
- Reduced name space
- Allows refinement of operations and representation
- Allows a variable number of instances
- More flexible than class operations

Use the Singleton Pattern when...

- There must be exactly one instance of a class
 - This instance must be accessible from a well-known access point
- The instance should be extensible by subclassing
 - Clients should be able to use a derived class without modifying their code

Facade Pattern

Lecture 15

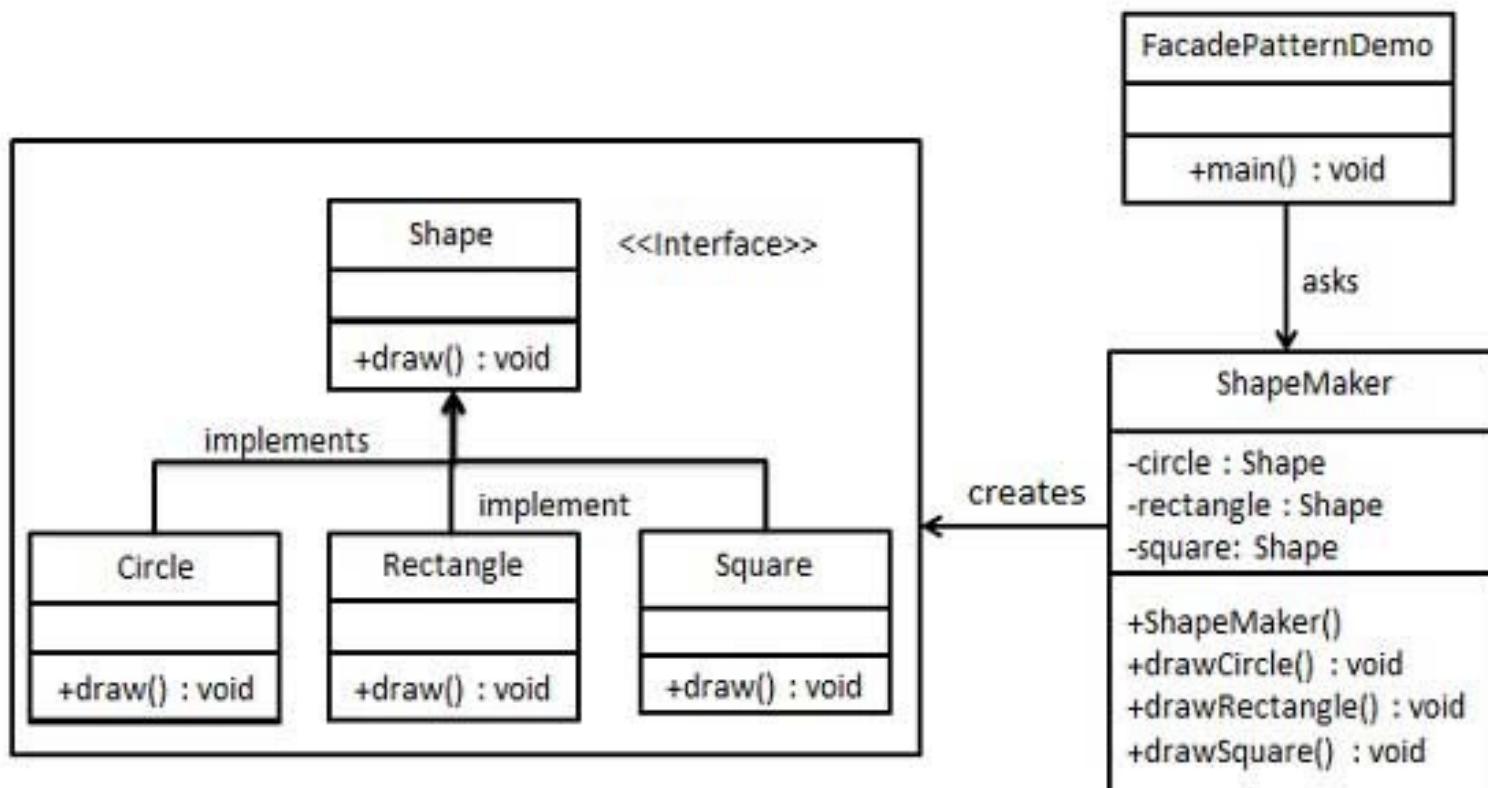
Facade

- “Provide a unified interface to a set of interfaces in a collection of subsystems.
- Facade defines a higher-level interface that makes the subsystems easier to use.”

Motivation

- “Common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.”
- With façade clients don't have to be concerned with exactly which object in a subsystem they're dealing with. They just call methods on the facade in blissful ignorance.

Implementation



Step 1

Create an interface.

Shape.java

```
public interface Shape {  
    void draw();  
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}
```

Square.java

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}
```

Circle.java

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

```
public class ShapeMaker {
    private Shape circle;
    private Shape rectangle;
    private Shape square;

    public ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
    }

    public void drawCircle(){
        circle.draw();
    }
    public void drawRectangle(){
        rectangle.draw();
    }
    public void drawSquare(){
        square.draw();
    }
}
```

Use the facade to draw various types of shapes.

FacadePatternDemo.java

```
public class FacadePatternDemo {  
    public static void main(String[] args) {  
        ShapeMaker shapeMaker = new ShapeMaker();  
  
        shapeMaker.drawCircle();  
        shapeMaker.drawRectangle();  
        shapeMaker.drawSquare();  
    }  
}
```

Step 5

Verify the output.

```
Circle::draw()  
Rectangle::draw()  
Square::draw()
```

References

- https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

Builder Pattern

Lecture 16

Builder Pattern

- The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming.
- The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

Problem

- The Builder design pattern solves problems like:
 - How can a class (the same construction process) create different representations of a complex object?
 - How can a class that includes creating a complex object be simplified?

Solution

- The Builder design pattern describes how to solve such problems:
- Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
- A class delegates object creation to a Builder object instead of creating the objects directly.

Intent

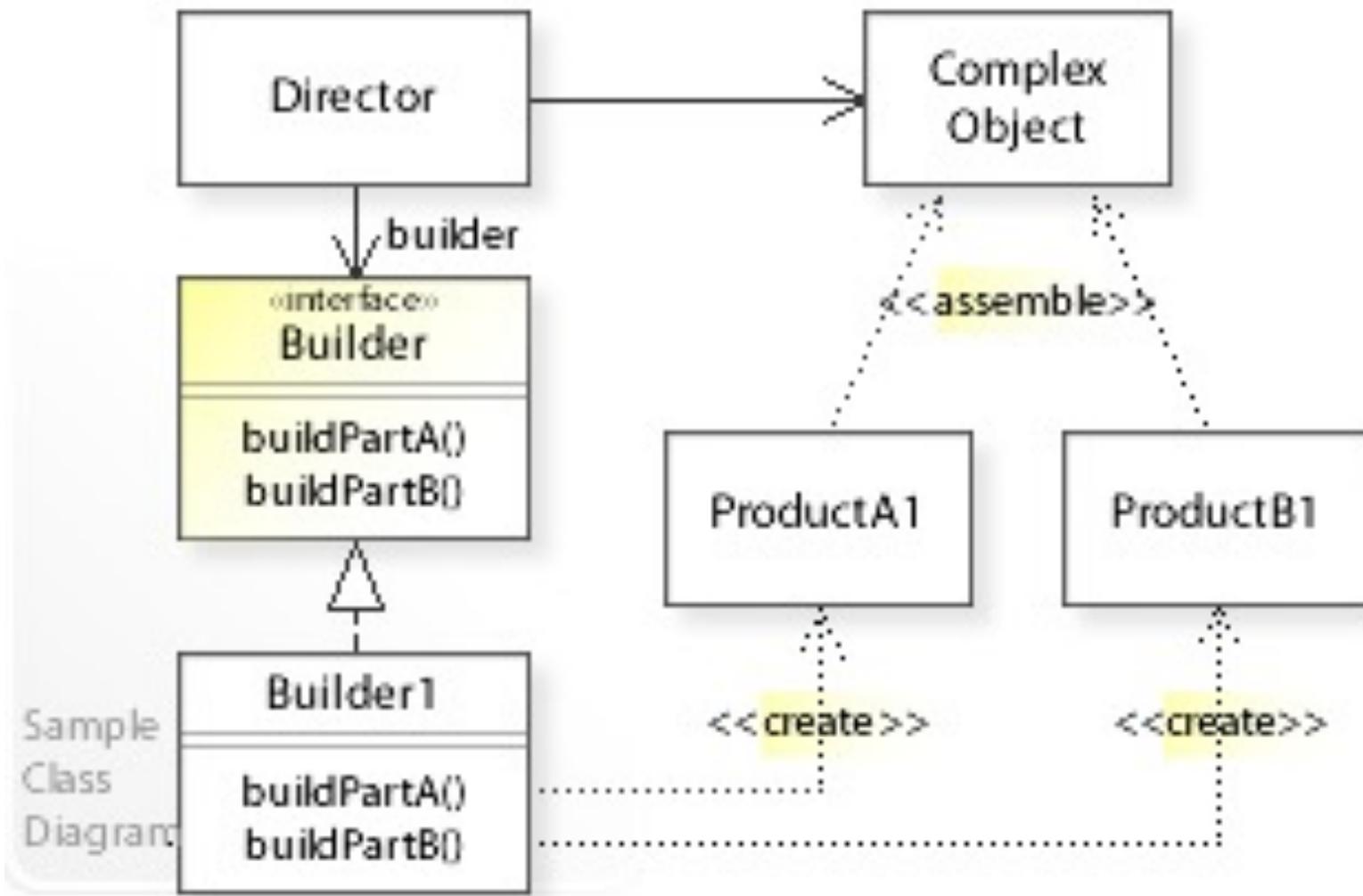
- The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so the same construction process can create different representations.

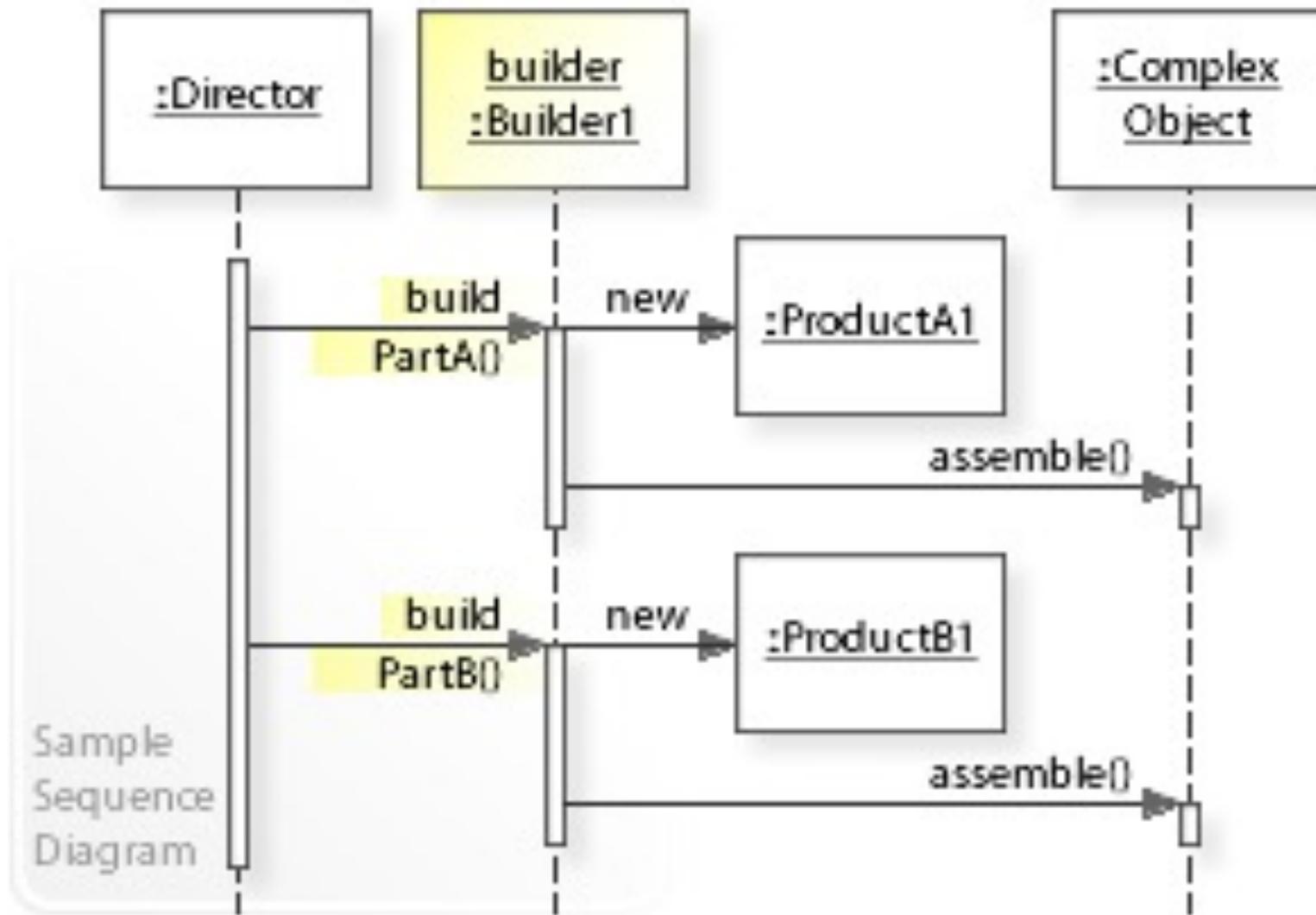
Advantages

- Advantages of the Builder pattern include:
 - Allows you to vary a product's internal representation.
 - Encapsulates code for construction and representation.
 - Provides control over steps of construction process.

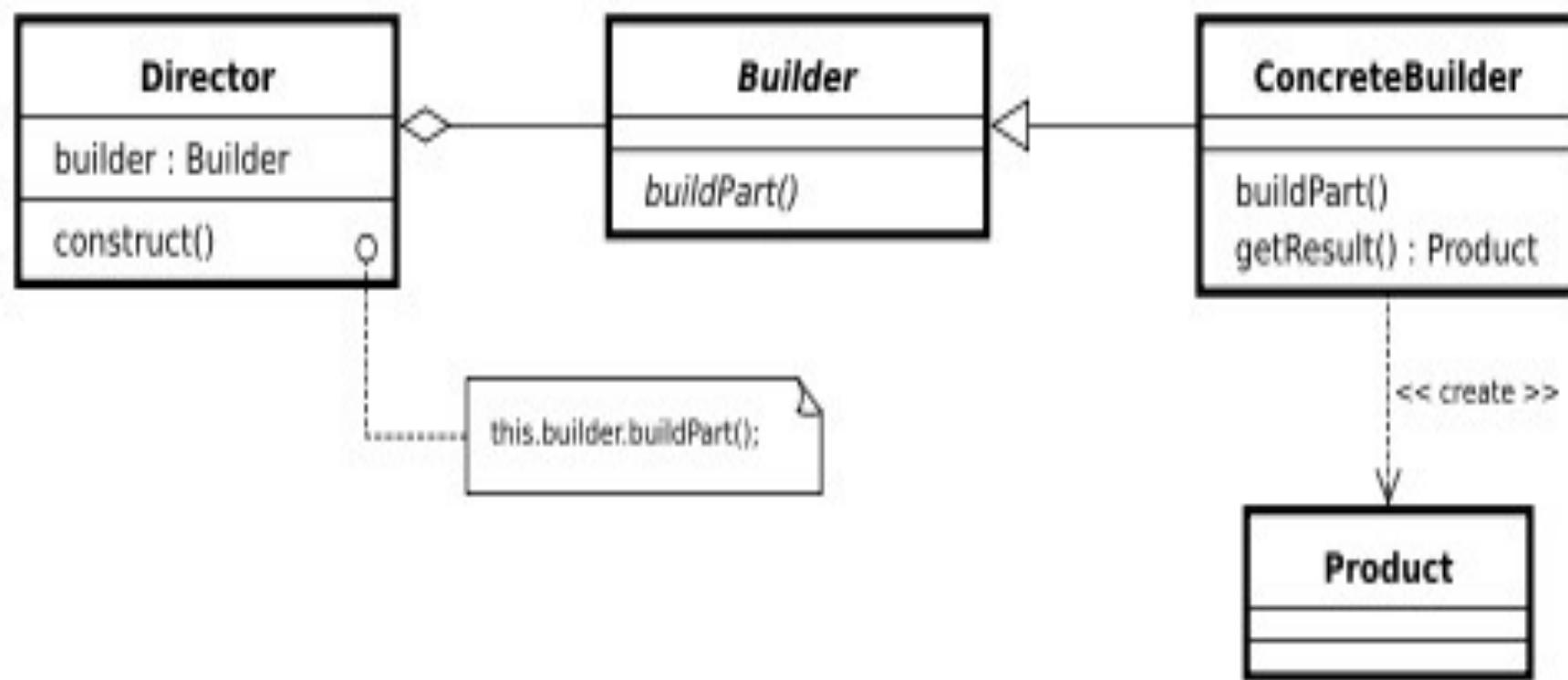
Disadvantages

- Disadvantages of the Builder pattern include:
 - Requires creating a separate `ConcreteBuilder` for each different type of product.
 - Requires the builder classes to be mutable.
 - Dependency injection may be less supported.





Class diagram



Represents a product created by the builder

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int NumDoors { get; set; }
    public string Colour { get; set; }

    public Car(string make, string model, string colour, int
numDoors)
    {
        Make = make;      Model = model;
        Colour = colour;  NumDoors = numDoors;
    }
}
```

```
/// The builder abstraction
public interface ICarBuilder
{
    string Colour { get; set; }
    int NumDoors { get; set; }

    Car GetResult();
}
```

```
/// Concrete builder implementation
public class FerrariBuilder : ICarBuilder
{
    public string Colour { get; set; }
    public int NumDoors { get; set; }

    public Car GetResult()
    {
        return NumDoors == 2 ? new Car("Ferrari",
"488 Spider", Colour, NumDoors) : null;
    }
}
```

```
// The director
public class SportsCarBuildDirector
{
    private ICarBuilder _builder;
    public SportsCarBuildDirector(ICarBuilder builder)
    {
        _builder = builder;
    }

    public void Construct()
    {
        _builder.Colour = "Red";
        _builder.NumDoors = 2;
    }
}
```

```
public class Client
{
    public void DoSomethingWithCars()
    {
        var builder = new FerrariBuilder();
        var director = new
SportsCarBuildDirector(builder);

        director.Construct();
        Car myRaceCar = builder.GetResult();
    }
}
```

References

- https://en.wikipedia.org/wiki/Builder_pattern

Factory Method

Lecture 17

Factory Method

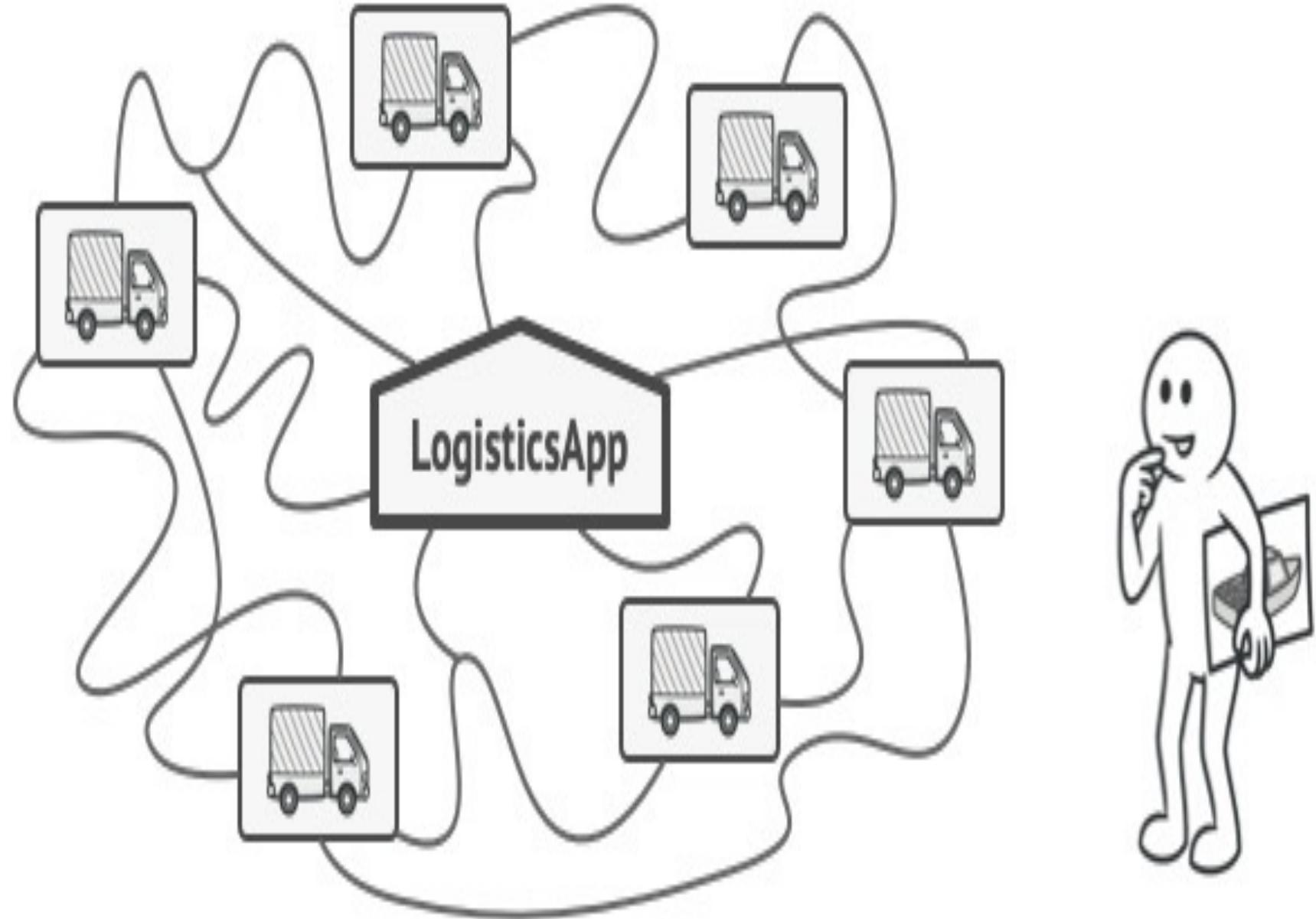
- Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Intent

- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Problem

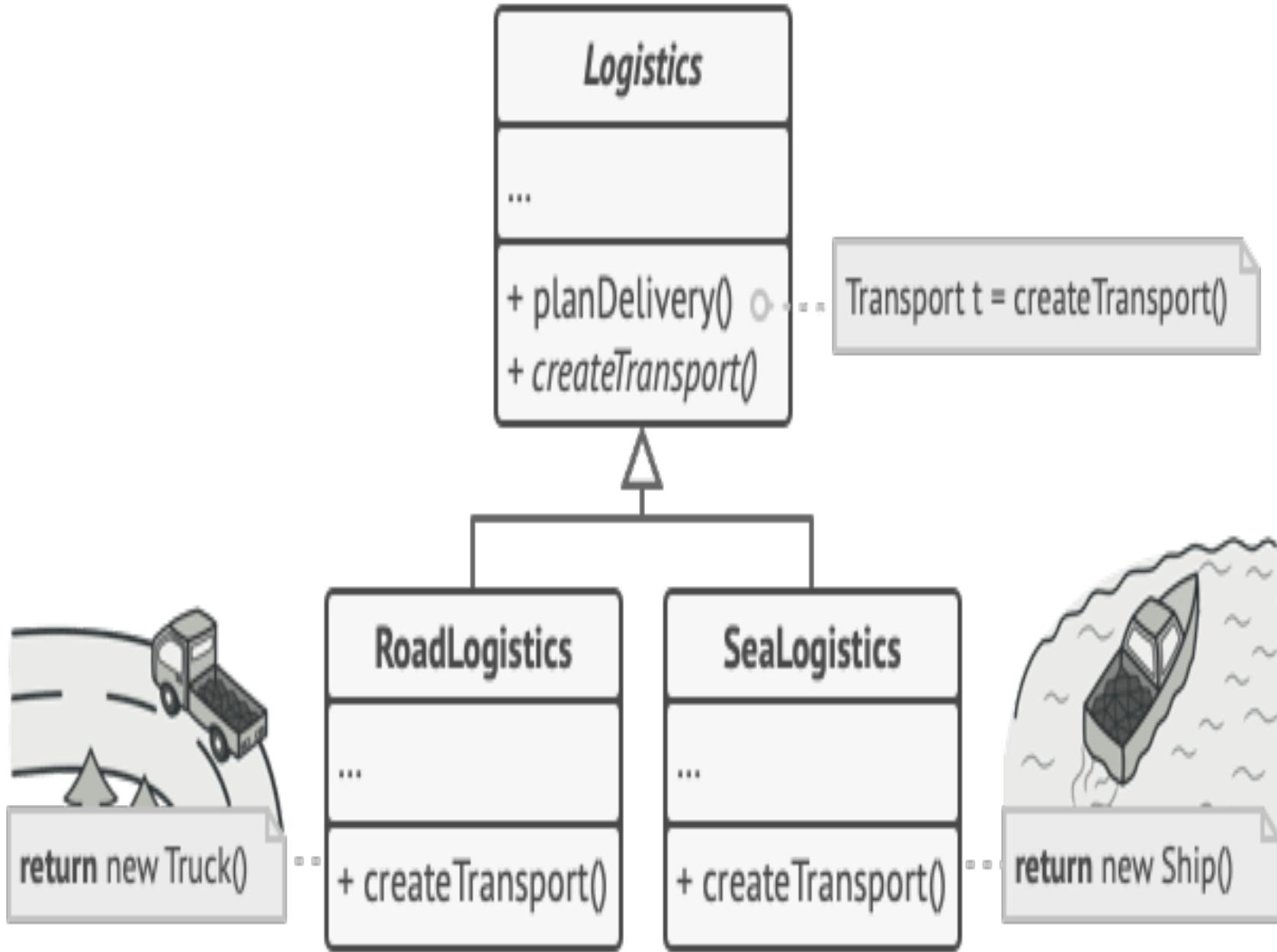
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the `Truck` class.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.
-



- At present, most of your code is coupled to the Truck class.
- Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special *factory* method.
- The objects are still created via the new operator, but it's being called from within the factory method.
- Objects returned by a factory method are often referred to as *products*.

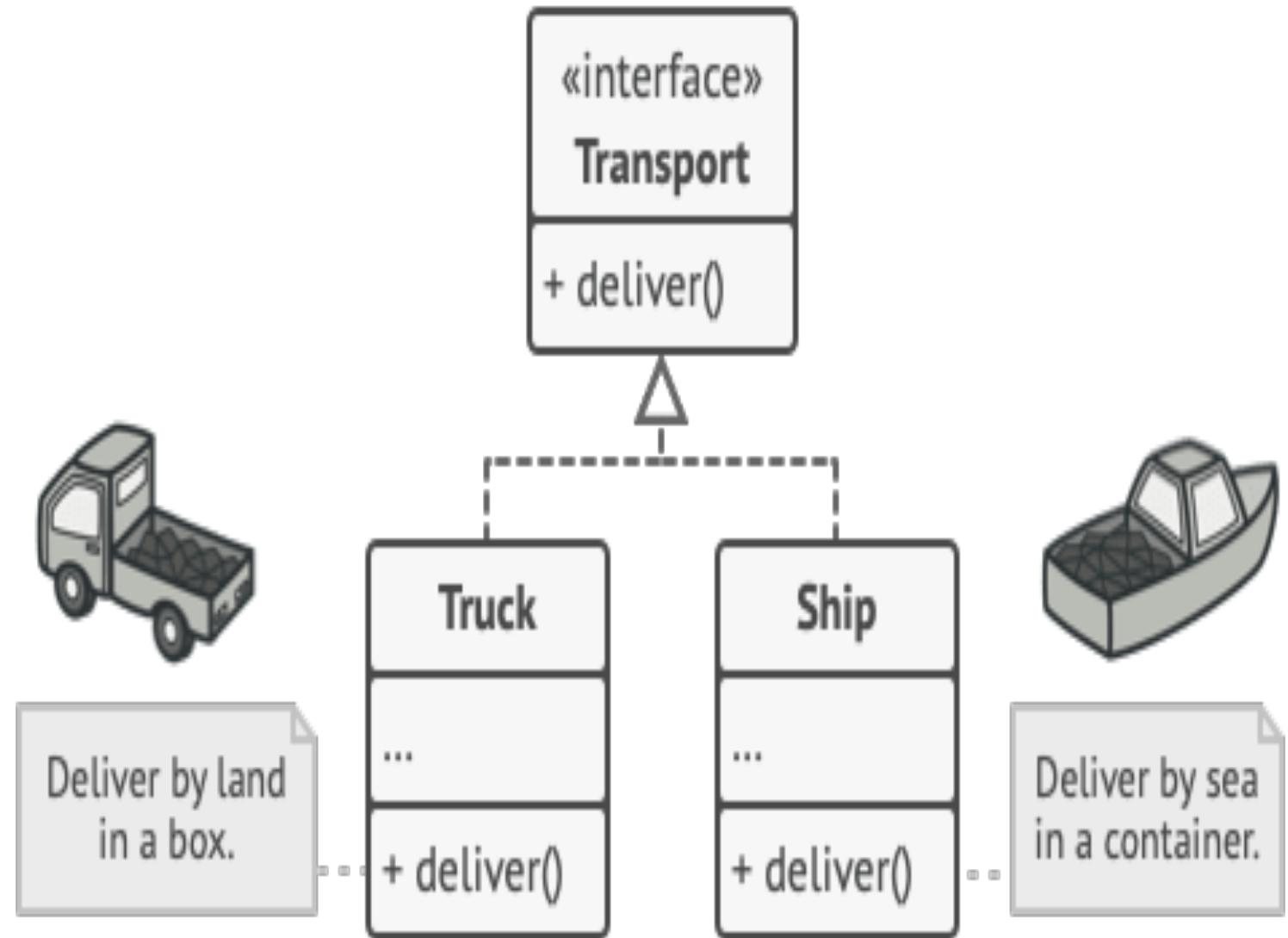


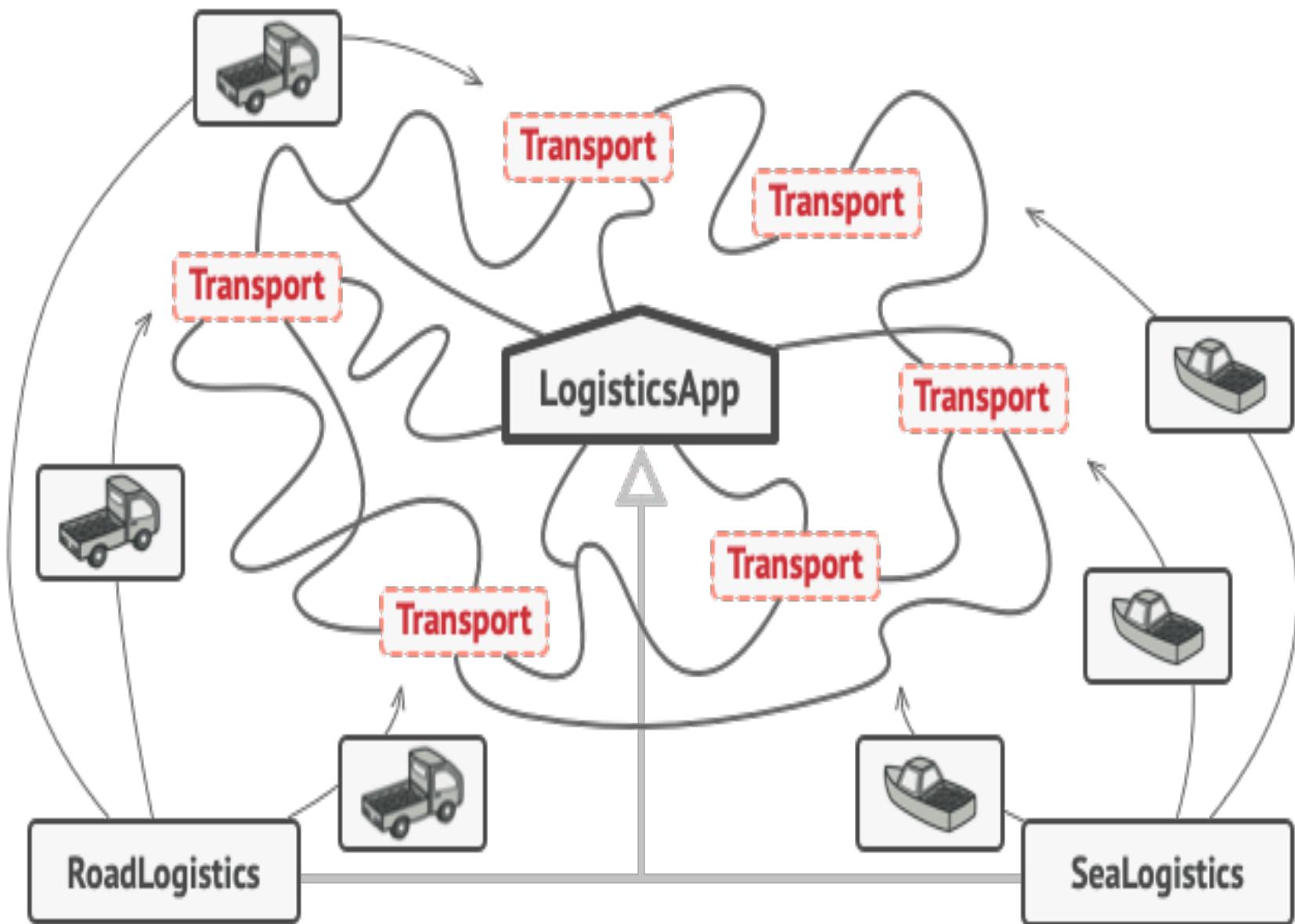
Advantages

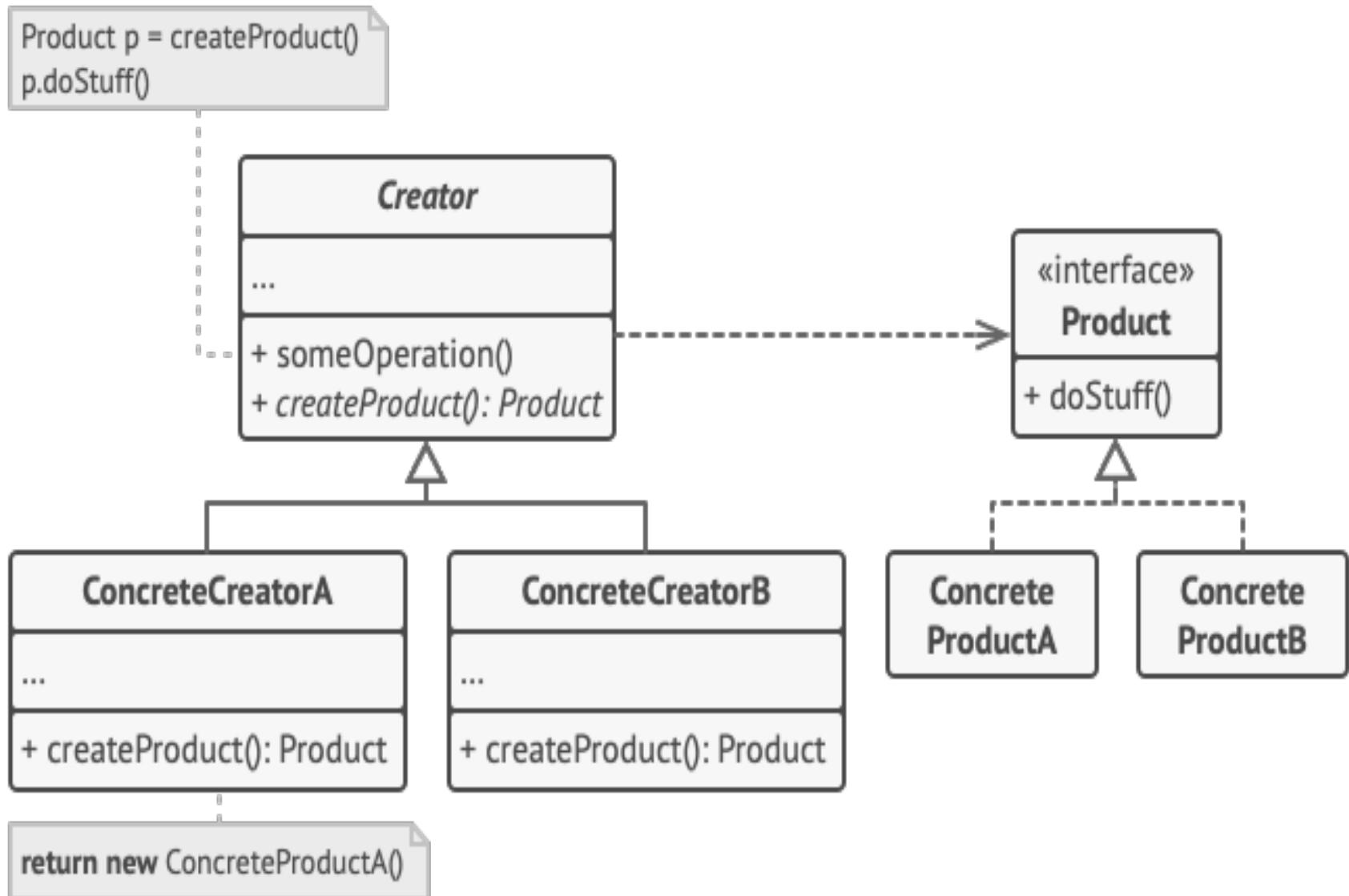
- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

Limitation

- There's a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

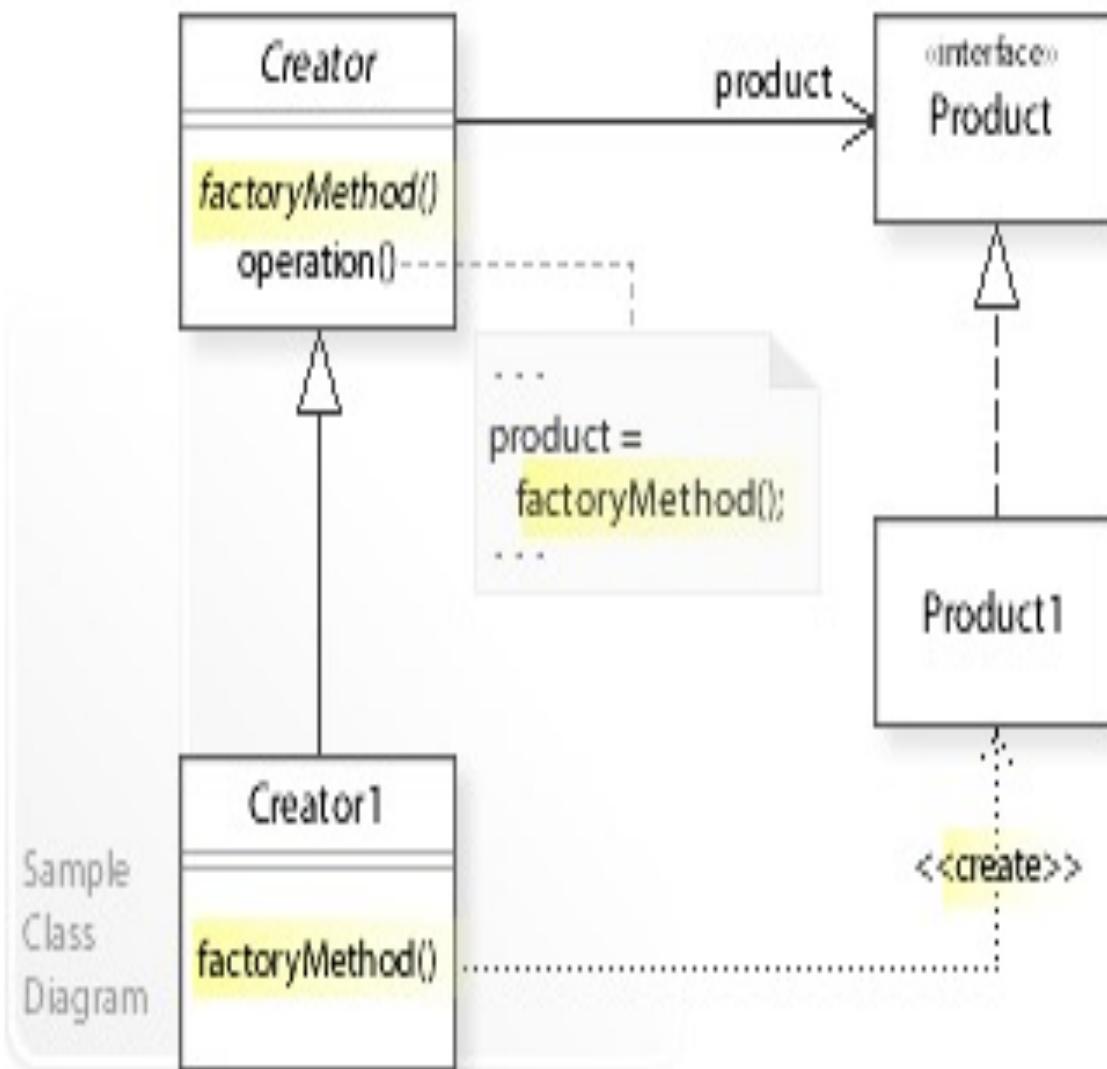


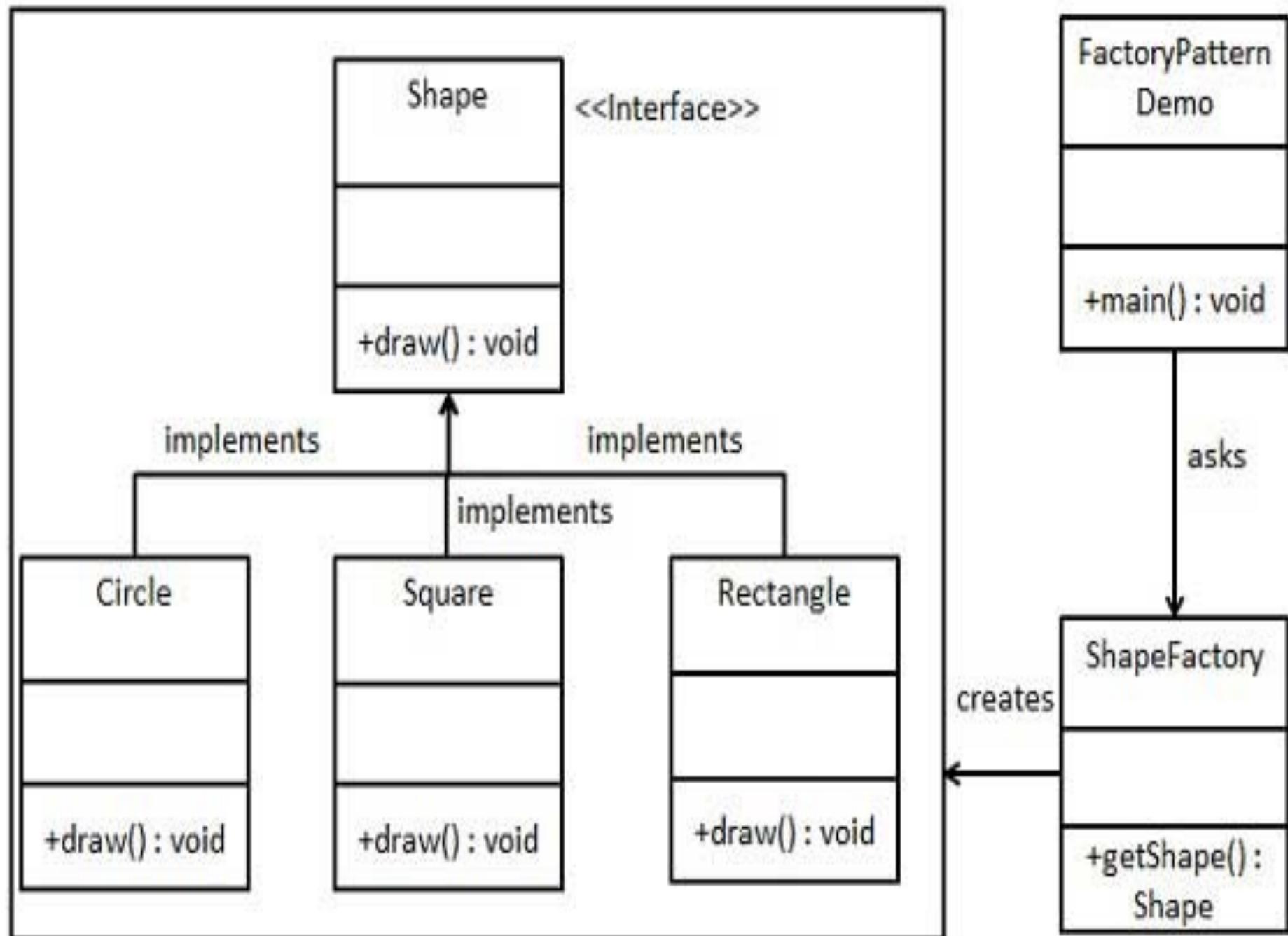




Example

- *FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE* / *RECTANGLE* / *SQUARE*) to *ShapeFactory* to get the type of object it needs.





Step 1

Create an interface.

Shape.java

```
public interface Shape {  
    void draw();  
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

Square.java

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

Circle.java

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

ShapeFactory.java

```
public class ShapeFactory {  
  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
  
        return null;  
    }  
}
```

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Circle and call its draw method.  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Circle  
        shape1.draw();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Rectangle  
        shape2.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
  
        //call draw method of square  
        shape3.draw();  
    }  
}
```

References

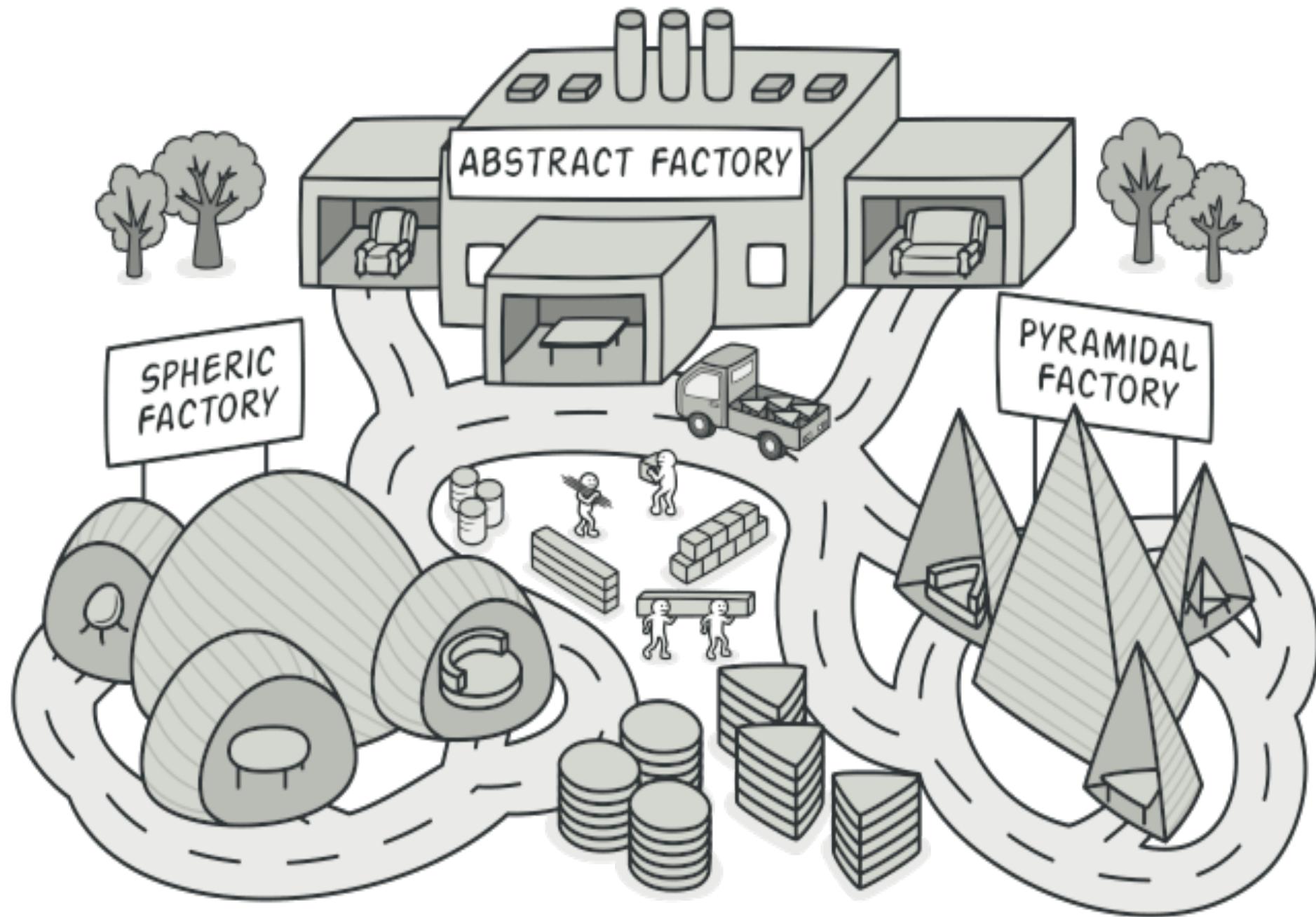
- <https://refactoring.guru/design-patterns/factory-method>
- [https://en.wikipedia.org/wiki/Factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern)
- [https://www.tutorialspoint.com/design pattern/factory pattern.htm](https://www.tutorialspoint.com/design_pattern/factory_pattern.htm)

Abstract Factory

Lecture 18

Intent

- **Abstract Factory** is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.



Problem

- Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
- A family of related products, say: Chair + Sofa + CoffeeTable.
- Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants:
 - Modern, Victorian, ArtDeco.

**Coffee
Table**

Sofa



Art Deco



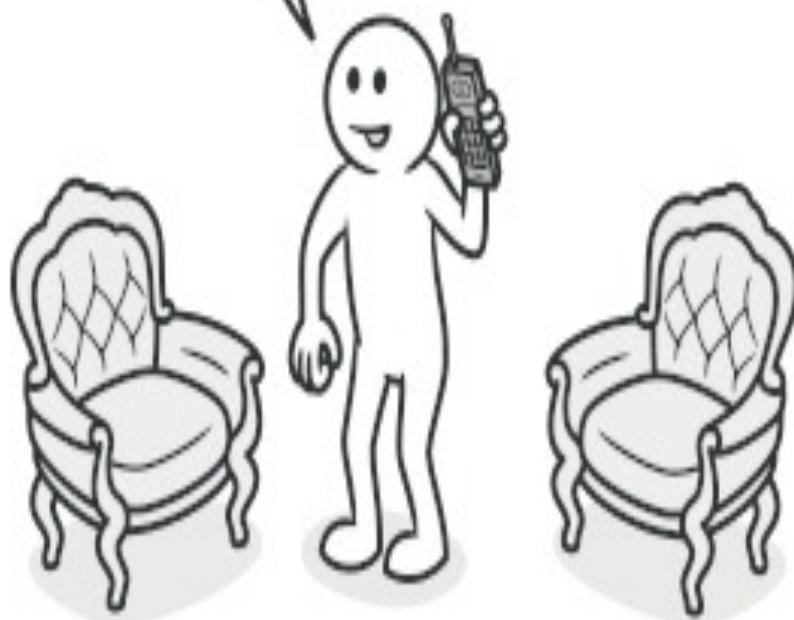
Victorian



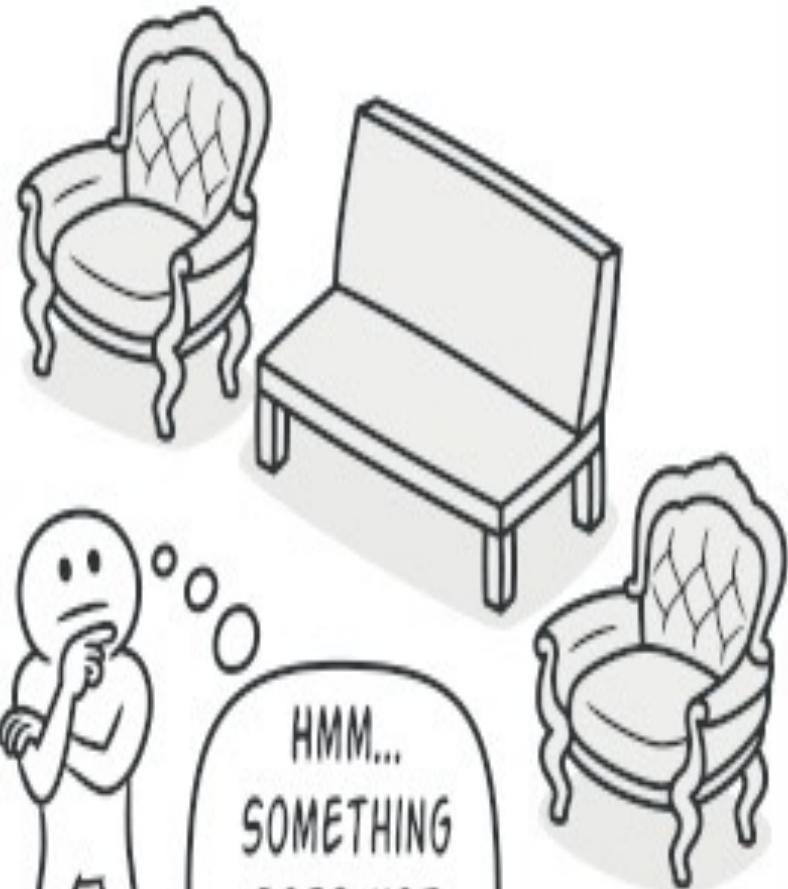
Modern



LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...



HMM... SOMETHING DOES NOT LOOK RIGHT.

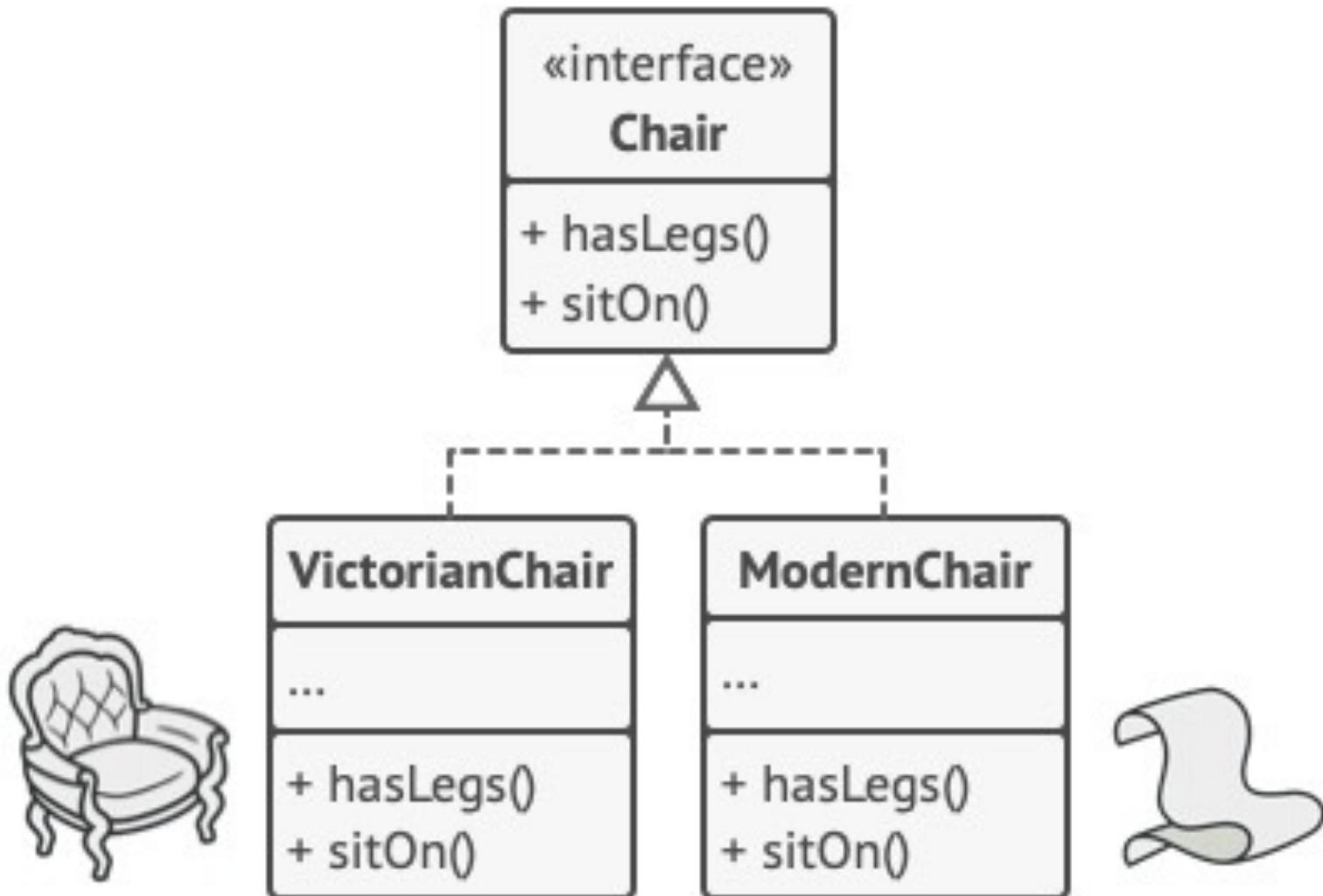


Problem

- You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.
- Also, you don't want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn't want to change the core code each time it happens.

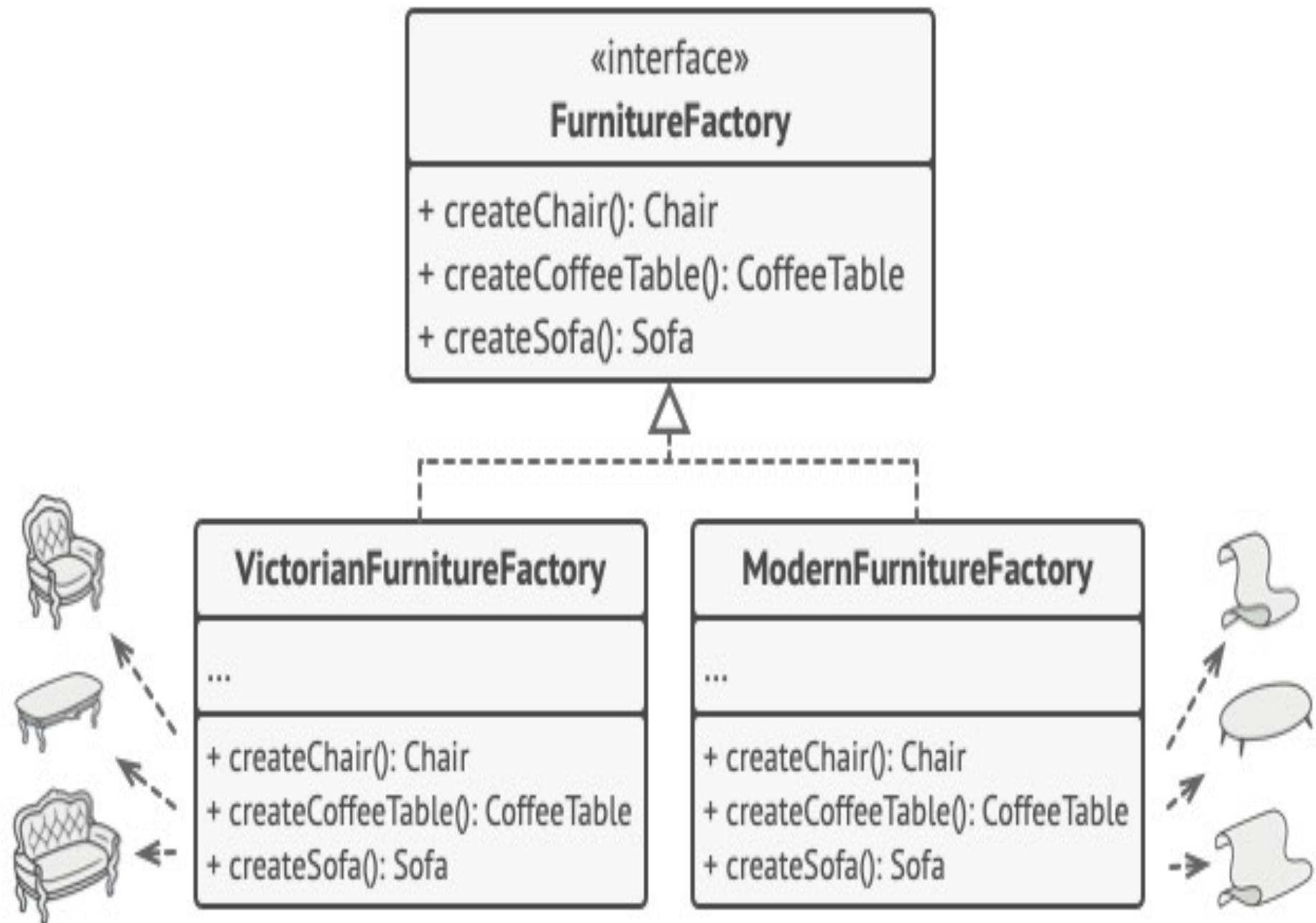
Solution

- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table).
- Then you can make all variants of products follow those interfaces.
- For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.



Solution

- The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, `createChair`, `createSofa` and `createCoffeeTable`).
- These methods must return abstract product types represented by the interfaces we extracted previously: `Chair`, `Sofa`, `CoffeeTable` and so on.

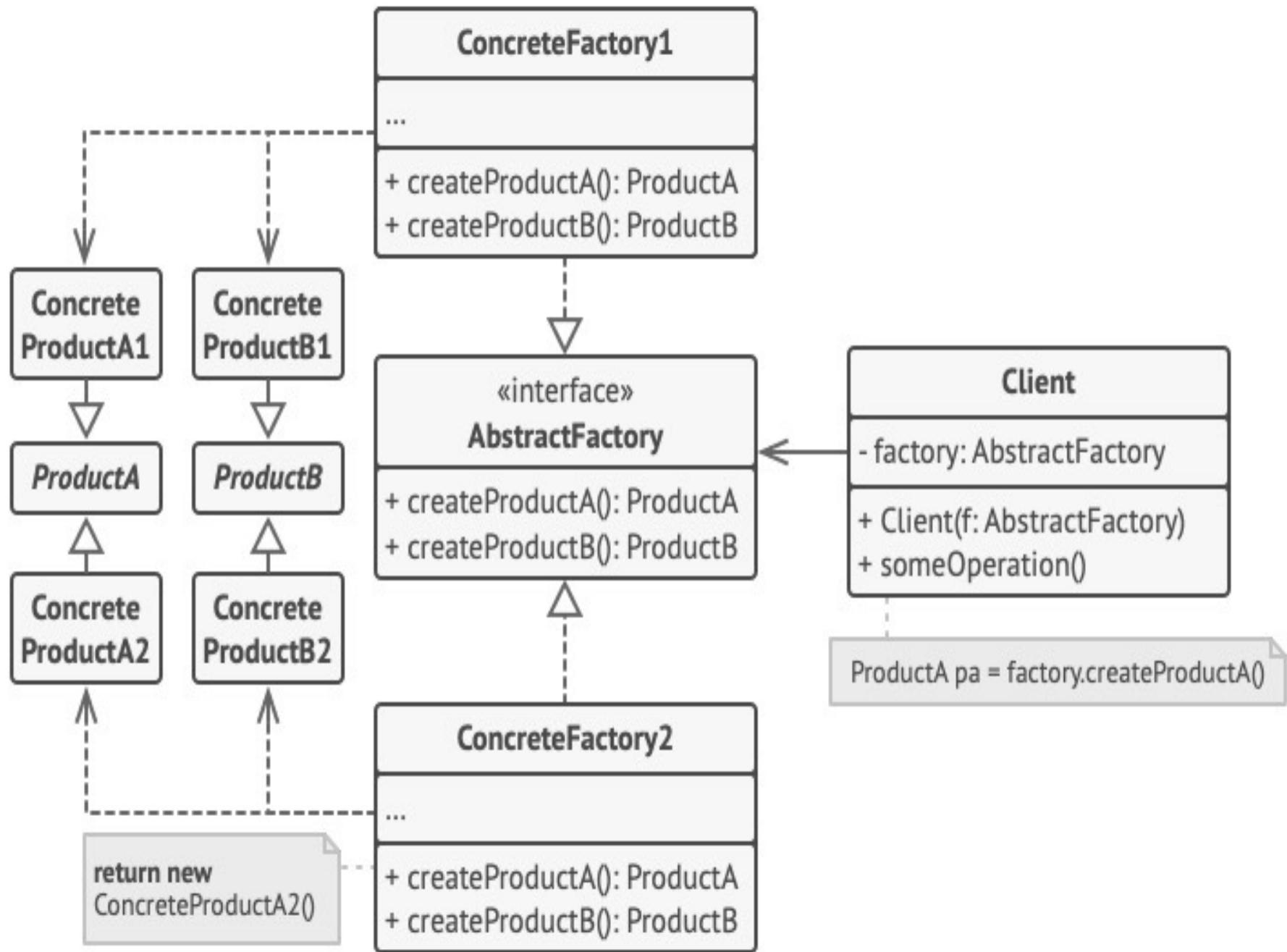


Explanation

- Now, how about the product variants? For each variant of a product family, we create a separate factory class based on the `AbstractFactory` interface.
- A factory is a class that returns products of a particular kind.
- For example, the `ModernFurnitureFactory` can only create `ModernChair`, `ModernSofa` and `ModernCoffeeTable` objects.

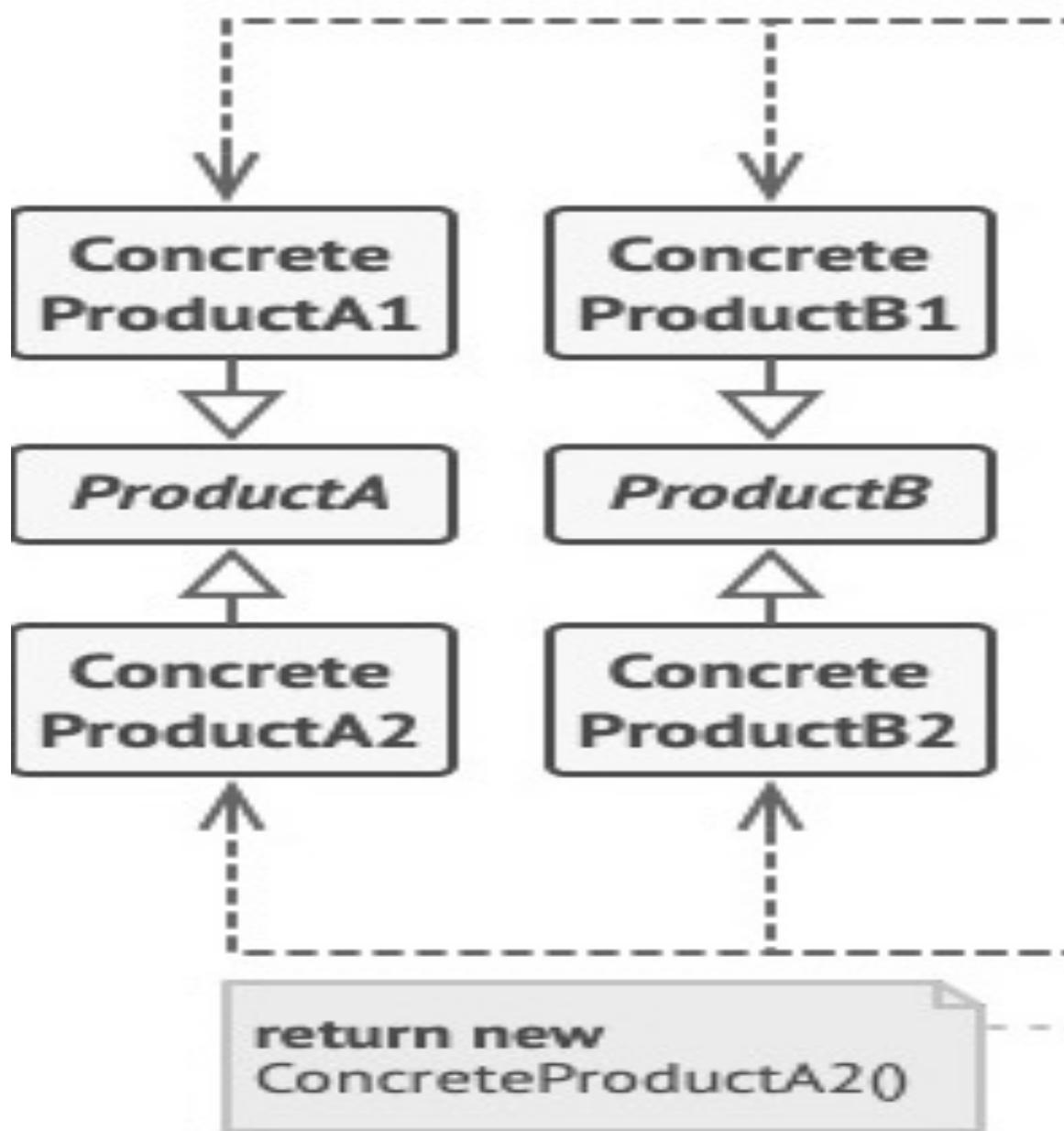
Explanation

- The client code has to work with both factories and products via their respective abstract interfaces.
- This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.



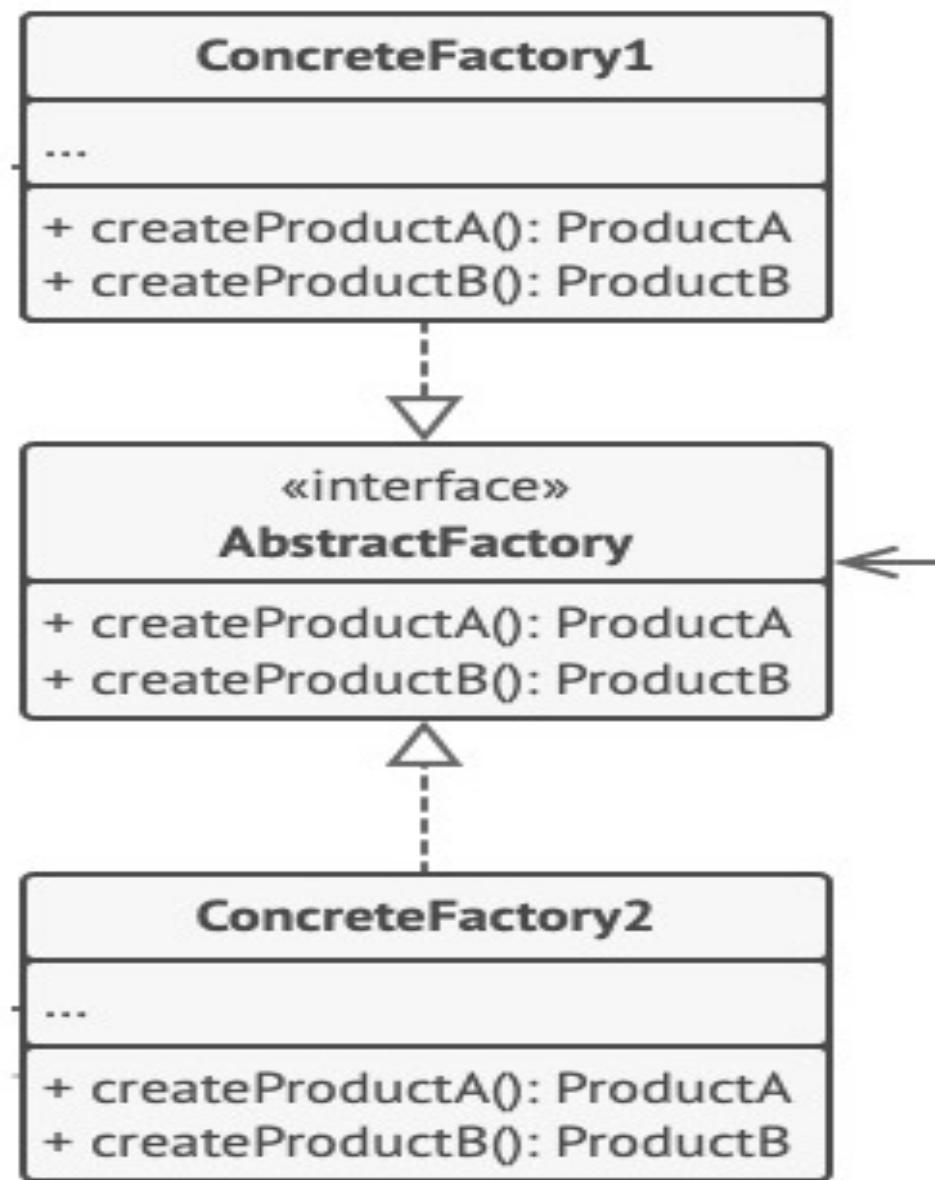
Structure

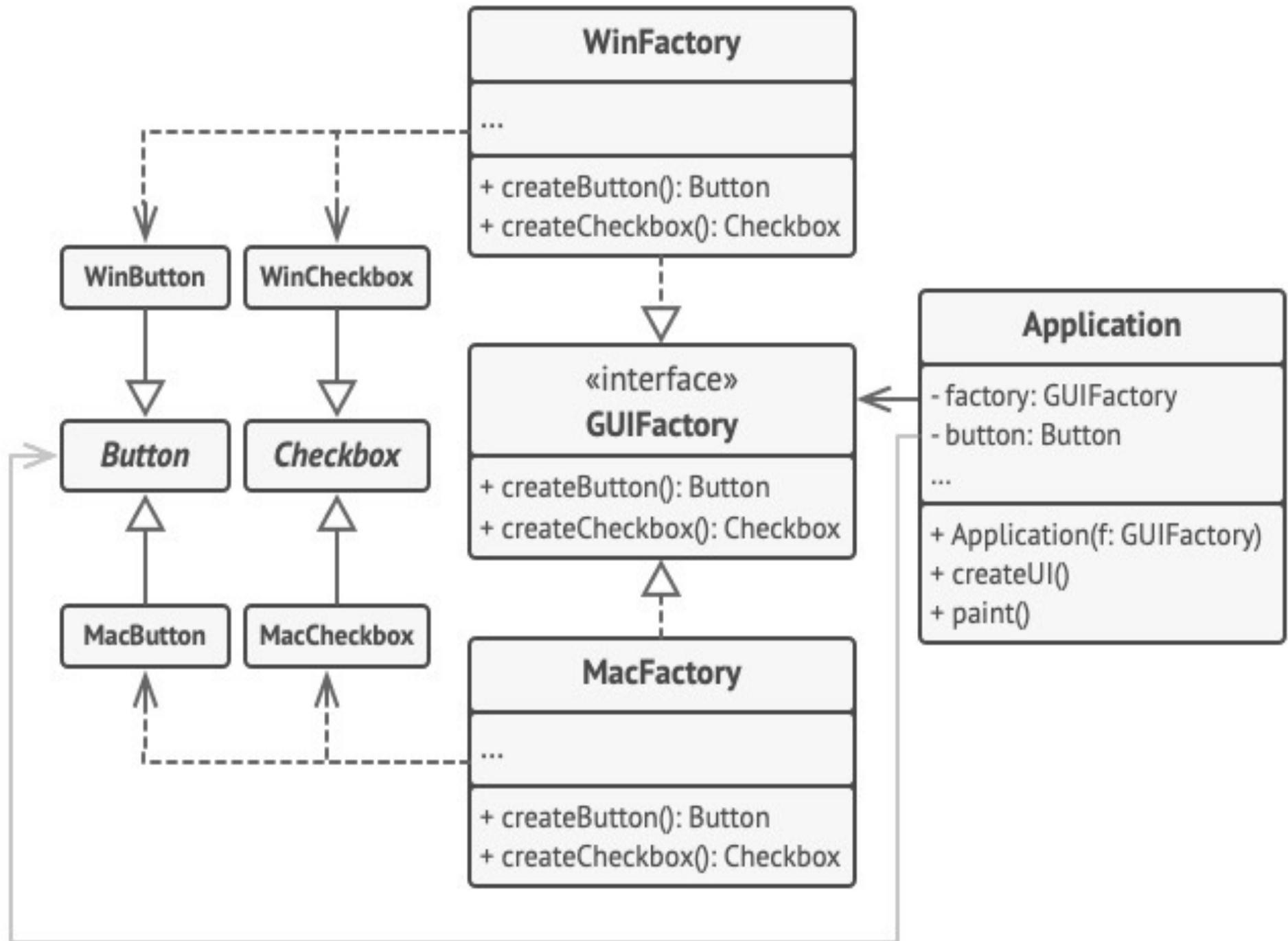
- Step 1
 - **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
- Step 2
 - **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).



Structure

- Step 3
 - The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
- Step 4
 - **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.





Pseudocode

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

```
// interface.  
interface Button is  
    method paint()  
  
    // Concrete products are created by corresponding concrete  
    // factories.  
class WinButton implements Button is  
    method paint() is  
        // Render a button in Windows style.  
  
class MacButton implements Button is  
    method paint() is  
        // Render a button in macOS style.
```

```
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in macOS style.
```

```
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()
```

```
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

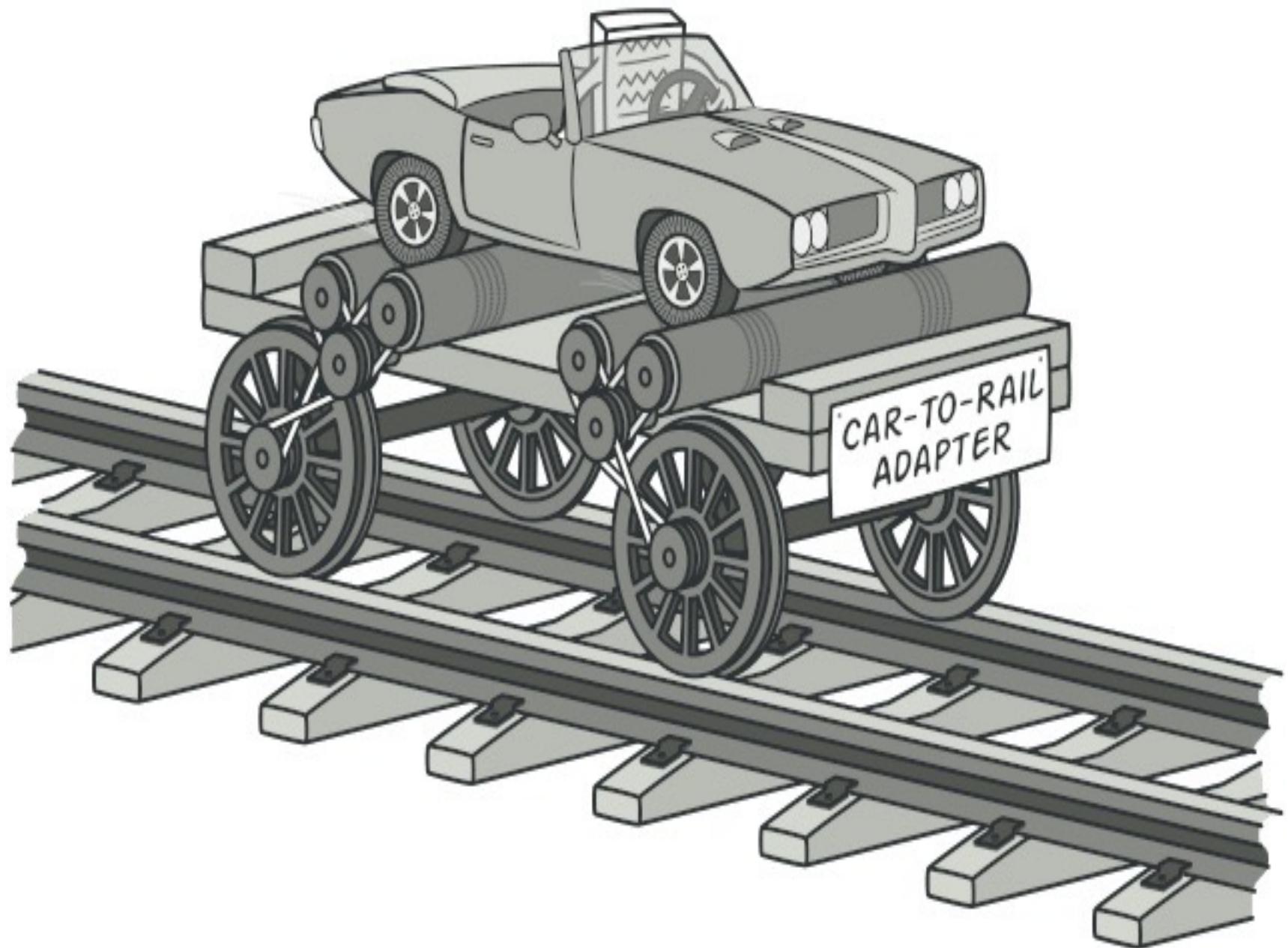
References

- <https://refactoring.guru/design-patterns/abstract-factory>

Adaptor

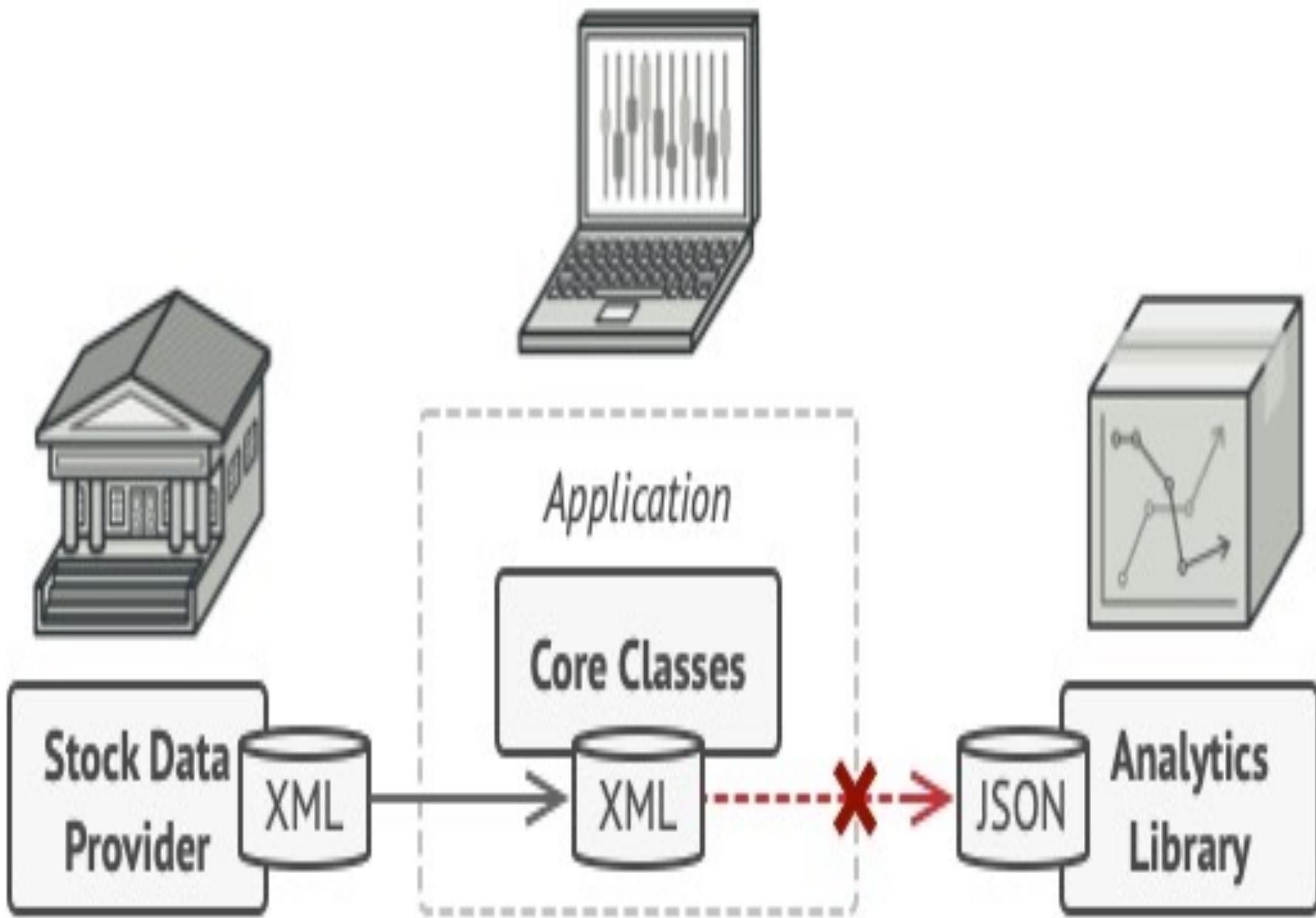
Intent

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



Problem

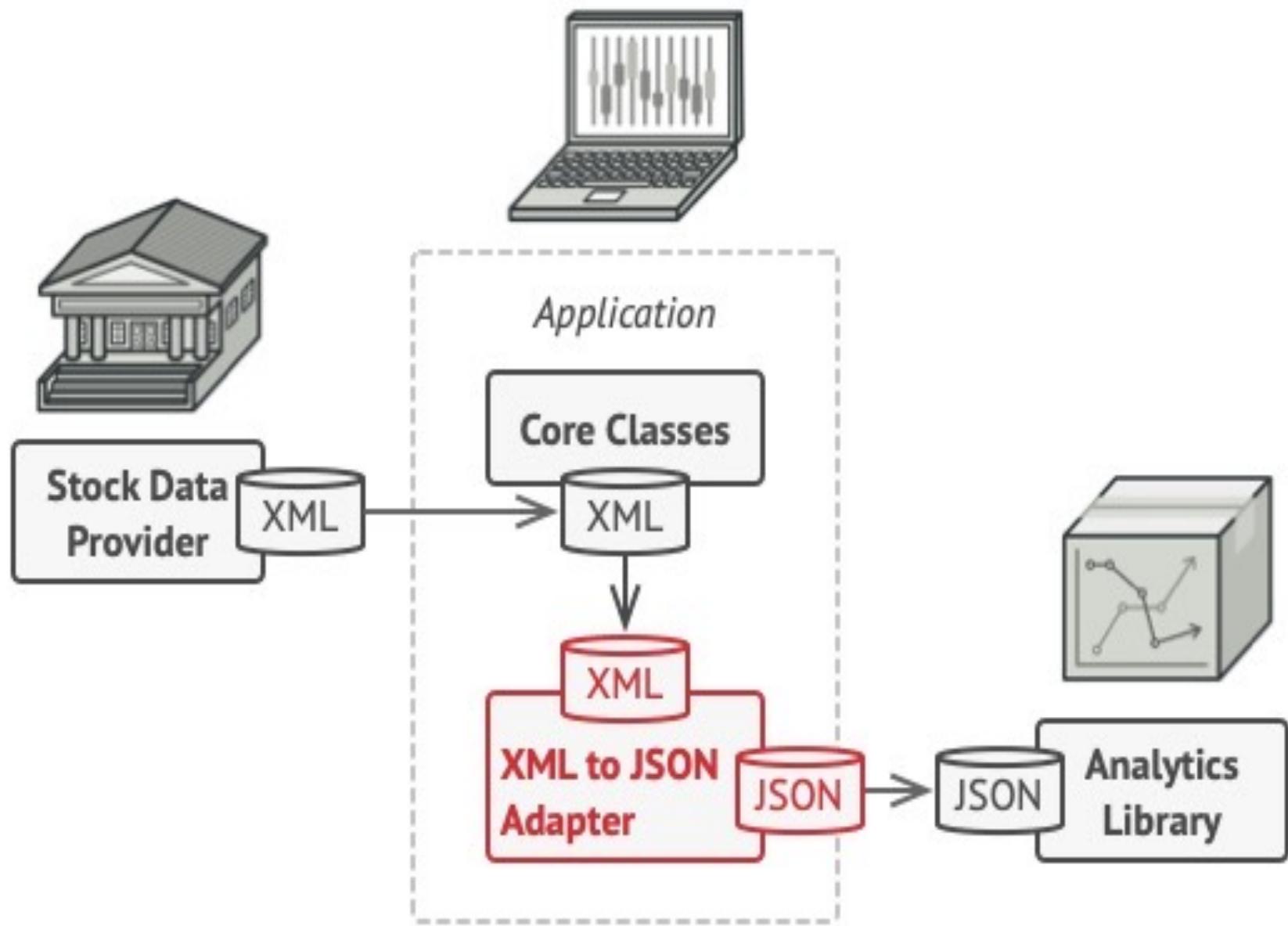
- You could change the library to work with XML.
- However, this might break some existing code that relies on the library.
- And worse, you might not have access to the library's source code in the first place, making this approach impossible.

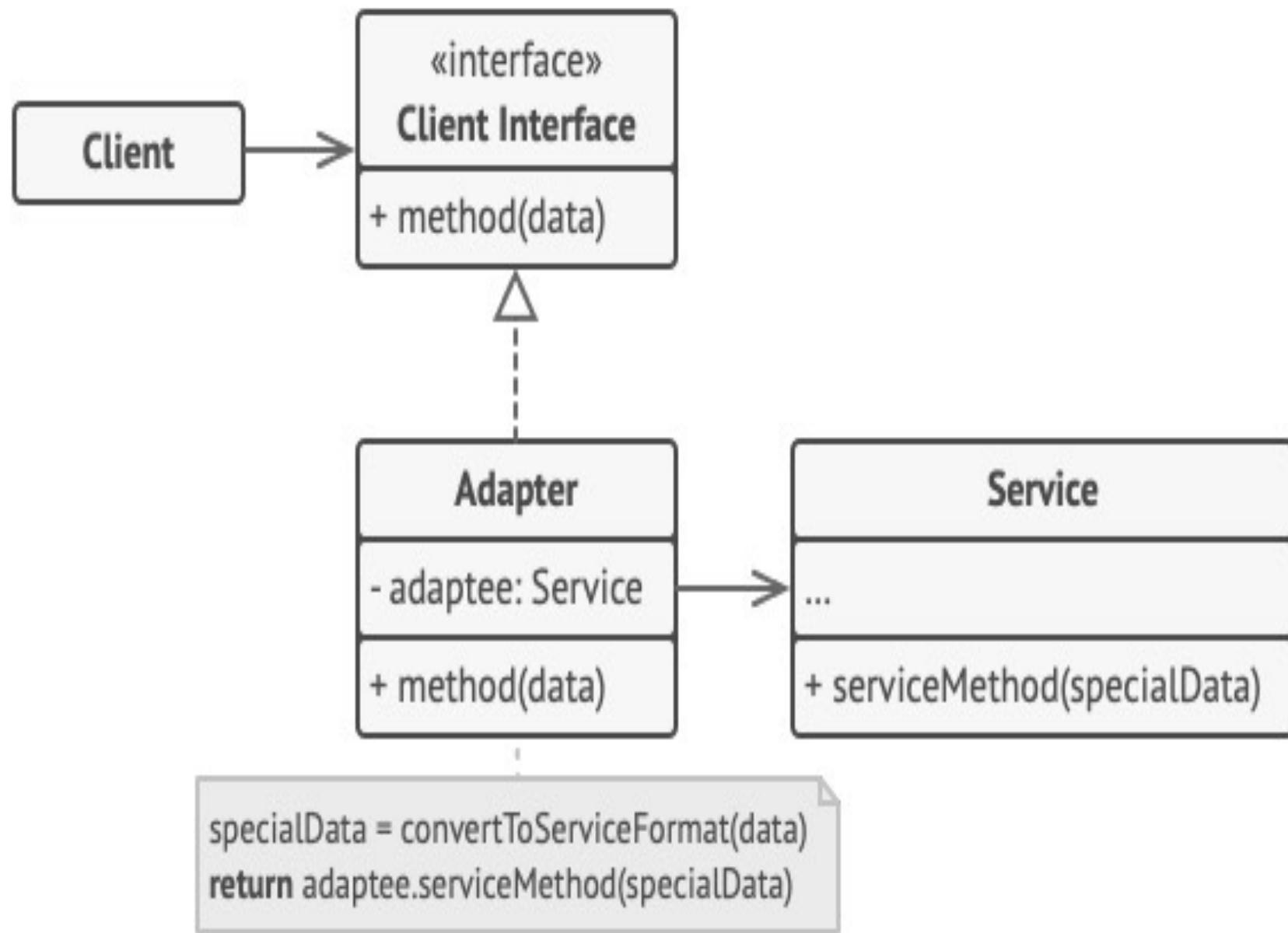
Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
 - The adapter gets an interface, compatible with one of the existing objects.
 - Using this interface, the existing object can safely call the adapter's methods.
 - Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.





Structure

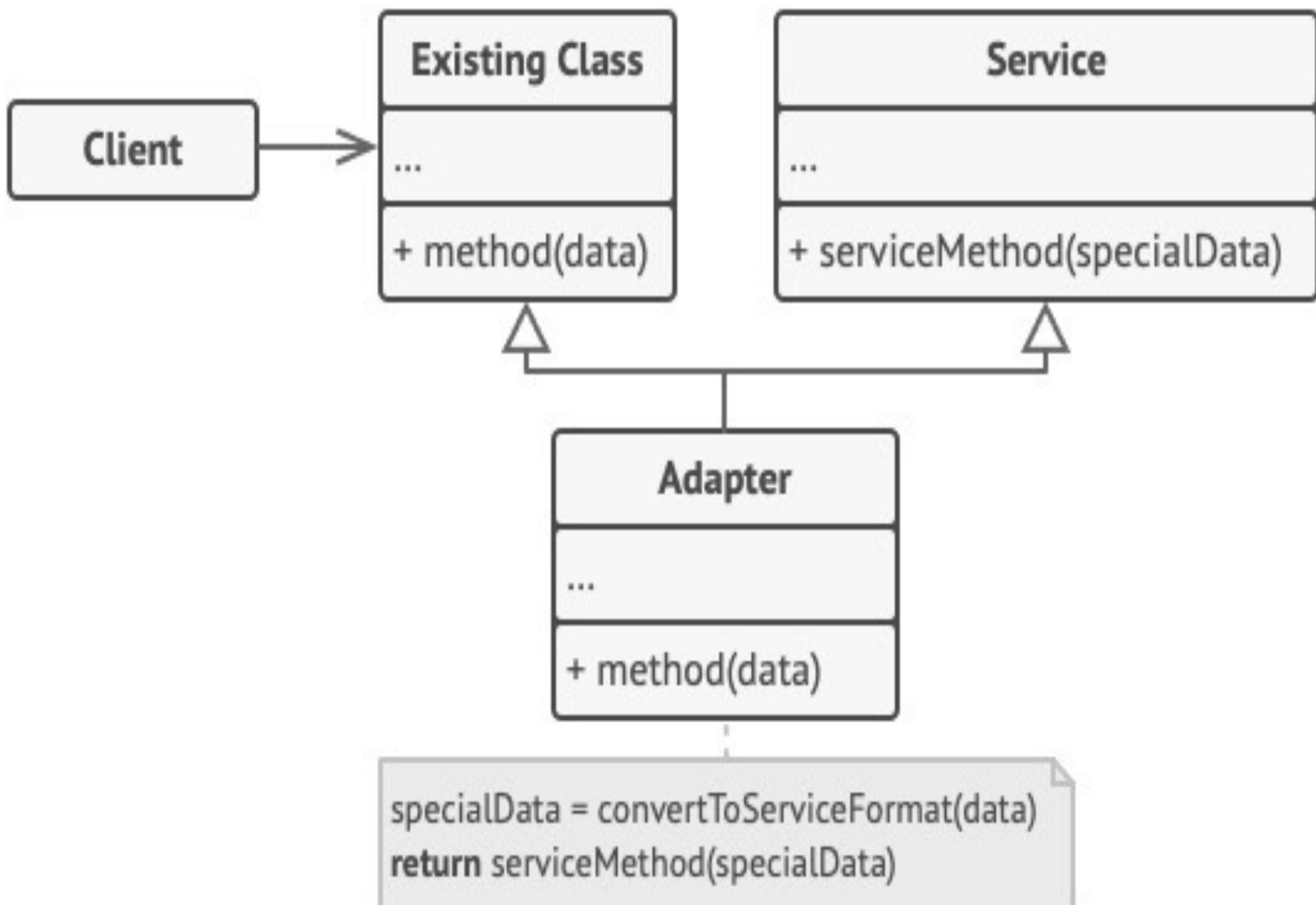
- The **Client** is a class that contains the existing business logic of the program.
- The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.
- The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

Explanation

- The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

Explanation

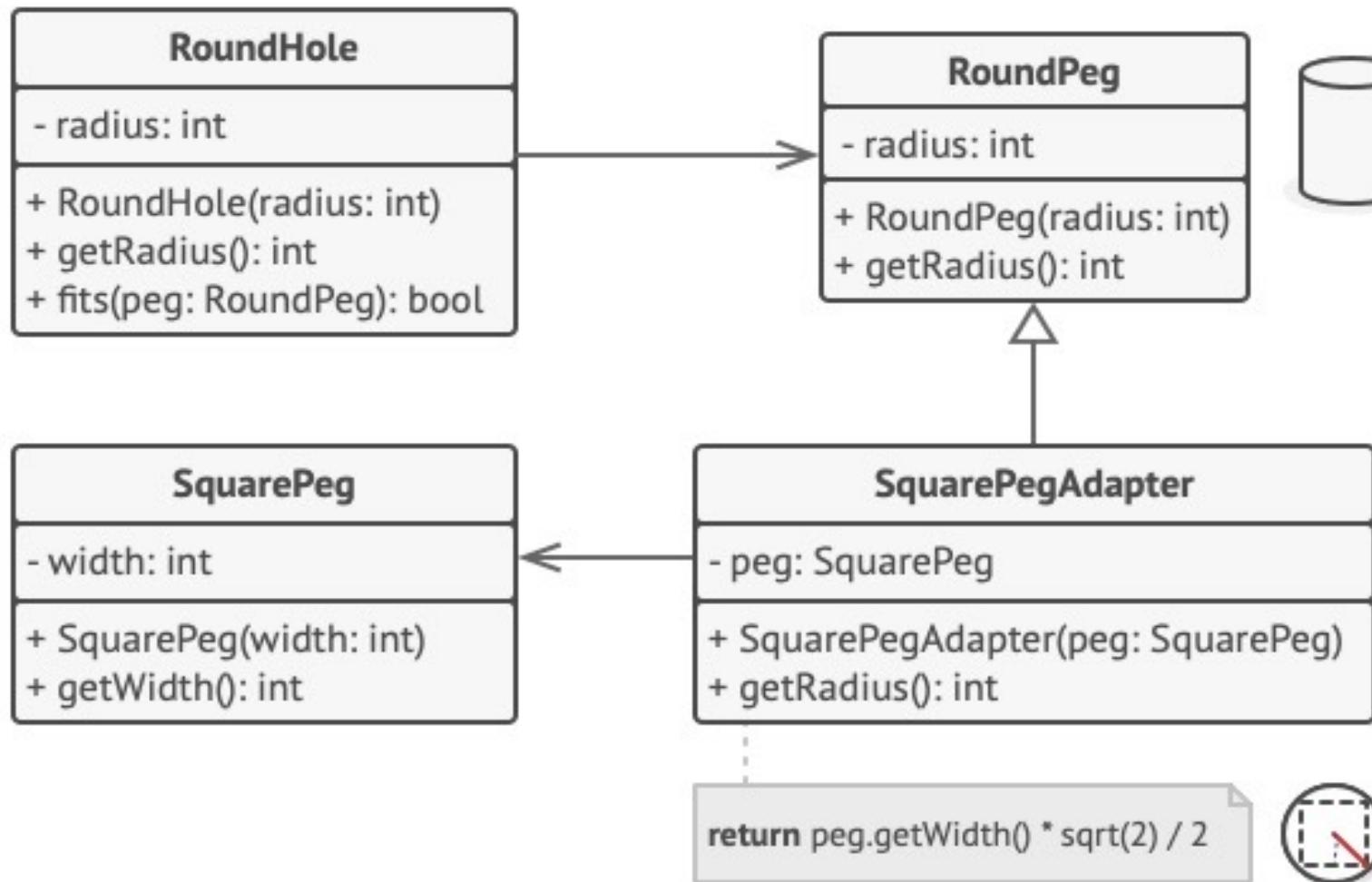
- The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.



Example

- This example of the **Adapter** pattern is based on the classic conflict between square pegs and round holes.
- The Adapter pretends to be a round peg, with a radius equal to a half of the square's diameter (in other words, the radius of the smallest circle that can accommodate the square peg).





```
// Say you have two classes with compatible interfaces:  
// RoundHole and RoundPeg.  
class RoundHole is  
    constructor RoundHole(radius) { ... }  
  
method getRadius() is  
    // Return the radius of the hole.  
  
method fits(peg: RoundPeg) is  
    return this.getRadius() >= peg.getRadius()  
  
class RoundPeg is  
    constructor RoundPeg(radius) { ... }  
  
method getRadius() is  
    // Return the radius of the peg.
```

```
// But there's an incompatible class: SquarePeg.  
class SquarePeg is  
    constructor SquarePeg(width) { ... }  
  
    method getWidth() is  
        // Return the square peg width.  
  
  
    // An adapter class lets you fit square pegs into round holes.  
    // It extends the RoundPeg class to let the adapter objects act  
    // as round pegs.  
class SquarePegAdapter extends RoundPeg is  
    // In reality, the adapter contains an instance of the  
    // SquarePeg class.  
    private field peg: SquarePeg  
  
    constructor SquarePegAdapter(peg: SquarePeg) is  
        this.peg = peg  
  
    method getRadius() is  
        // The adapter pretends that it's a round peg with a  
        // radius that could fit the square peg that the adapter  
        // actually wraps.  
        return peg.getWidth() * Math.sqrt(2) / 2
```

```
// Somewhere in client code.

hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

How to Implement

1. Make sure that you have at least two classes with incompatible interfaces:
 - A useful *service* class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
 - One or several *client* classes that would benefit from using the service class.
2. Declare the client interface and describe how clients communicate with the service.
3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

4. Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.
 5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.
 6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.
-

Pros and Cons

- ✓ *Single Responsibility Principle.* You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle.* You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
- ✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Reference

- <https://refactoring.guru/design-patterns/adapter>