

Software Engineering

mainly based on **Bernd Bruegge's book
Object-Oriented Software Engineering
using UML, Patterns and Java**

Dan CHIOREAN
Babeş-Bolyai University
chiorean@cs.ubbcluj.ro



Software Engineering – the term

- The term "software engineering" was coined by Anthony Oettinger and then was used in 1968 as a title for the world's first conference on software engineering, sponsored and facilitated by NATO. The conference was attended by international experts on software who agreed on defining best practices for software grounded in the application of engineering. The result of the conference is a report that defines how software should be developed. The original report is publicly available.



Fifty-Two Years of Software Engineering

- The *Online Historical Encyclopedia of Programming Languages* now lists no less than **8945** different programming languages [Pigott].
- Given the number and variety of sub-disciplines sheltering under the umbrella term “software engineering”:
“There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years.”

[Pigott, D.]: ‘Online Historical Encyclopedia of Programming Languages’. [<http://hopl.info> (Accessed 23 February 2021)].



Fifty-Two Years of Software Engineering_2

- Quoting Geoffrey Elton, the eminent Cambridge historian:

“The future is dark, the present burdensome. Only the past, dead and finished, bears contemplation. Those who look upon it have survived it; they are its product and its victors. No wonder therefore that men concern themselves with history.”
- Quoting George Santayana:

“Those who cannot remember the past are condemned to repeat it”.
- **Fifty Years of Software Engineering or The View from Garmisch**
- Brian Randell, School of Computing, Newcastle University
<https://arxiv.org/ftp/arxiv/papers/1805/1805.02742.pdf>



Software Engineering Def_Bruegge

- A collection of **principles, methods, techniques, methodologies and tools** that help the production of a high-quality software system:
 - with a given budget,
 - before a given deadline,
 - while changes occurs.

Bernd Bruegge



Software Engineering_Def_IEEE

Software Engineering: Definition 2

- *The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of Software; that is, the application of engineering to software*
[IEEE, ANSI]
- Software engineering spans whole product lifecycle

Peter Müller ETH Zurich – SS-06



Software Engineering Def_Meyer

The production of operational software satisfying defined standards of quality

... includes programming, but is more than programming

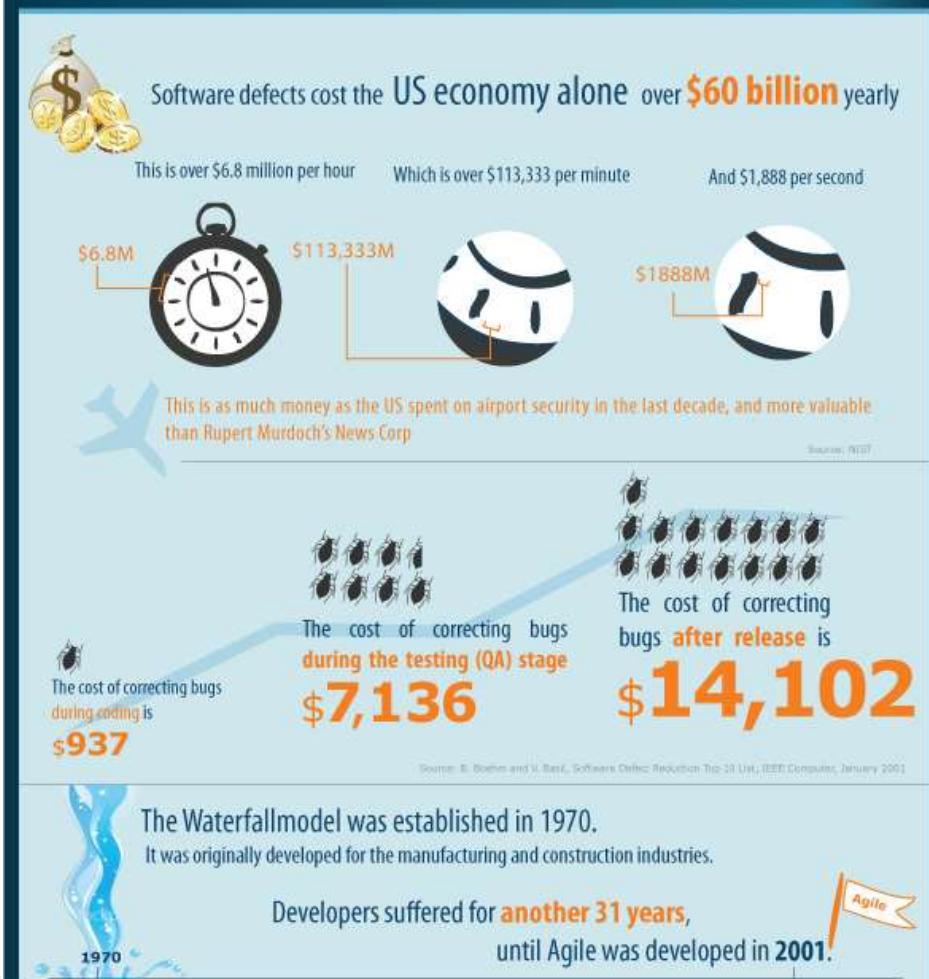
As von Clausewitz did not write: "the continuation of programming through other means".

Bertrand Meyer ETH Zurich – SE



The costs of software errors

The Severity of Bugs: Are We Doomed?

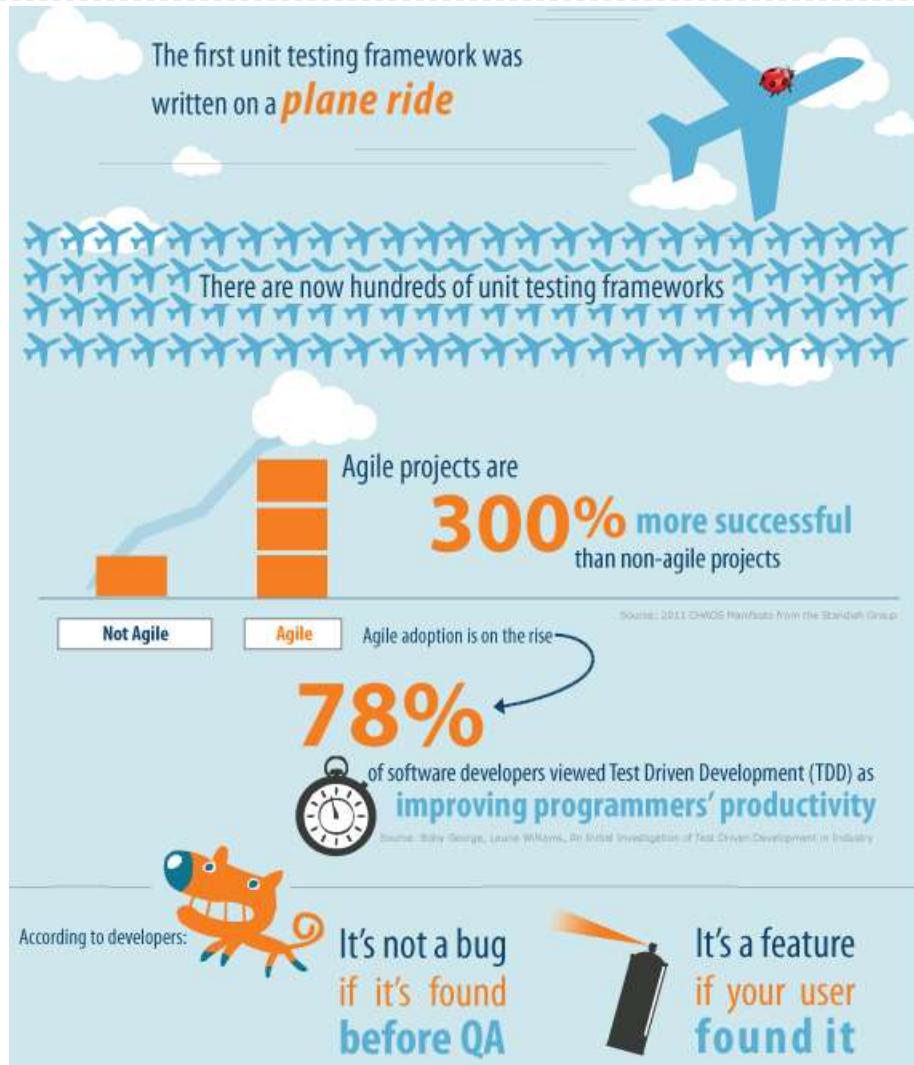


The costs of software errors...update

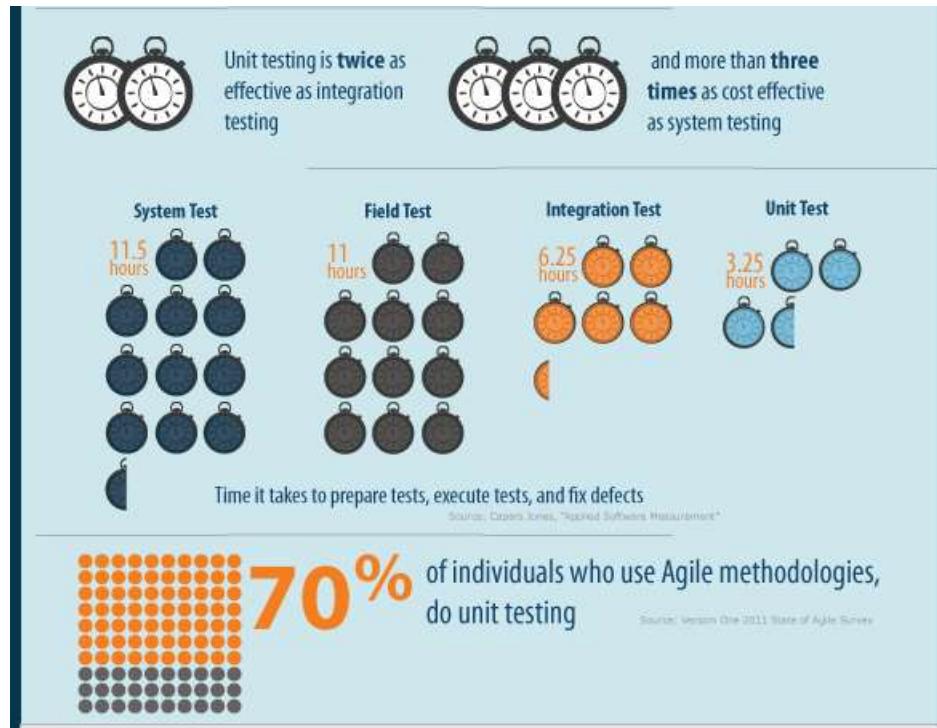
- In 2002, software bugs cost the United States economy approximately \$59.5 billion (Software).
- In 2016, that number jumped to \$1.1 trillion (1 trillion = 1000 billion)
- In summary, the cost of poor-quality software in the US in 2018 is approximately \$2.84 trillion. If we remove the future cost of technical debt, the total becomes \$2.26 trillion.
- The report states that the cost of poor software quality in the U.S. was approximately \$2.08 trillion in 2020
- <https://it-cisq.org/wp-content/uploads/2018/10/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf>
- <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.htm>



The costs of software errors_2



The costs of software errors_3

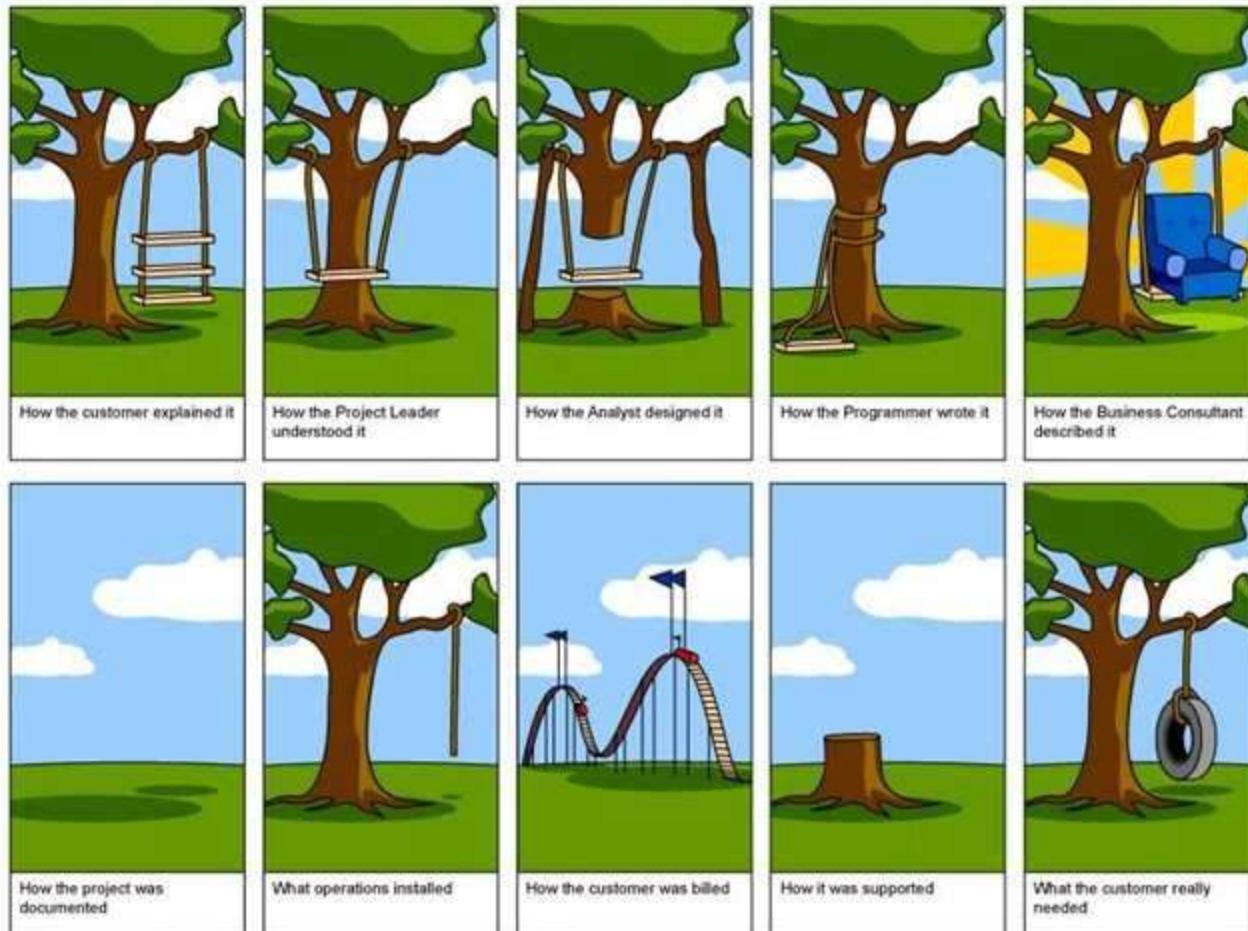


<https://www.typemock.com/software-bugs-infographic/>

<https://www.codeguru.com/blog/category/programming/the-cost-of-bugs.html>

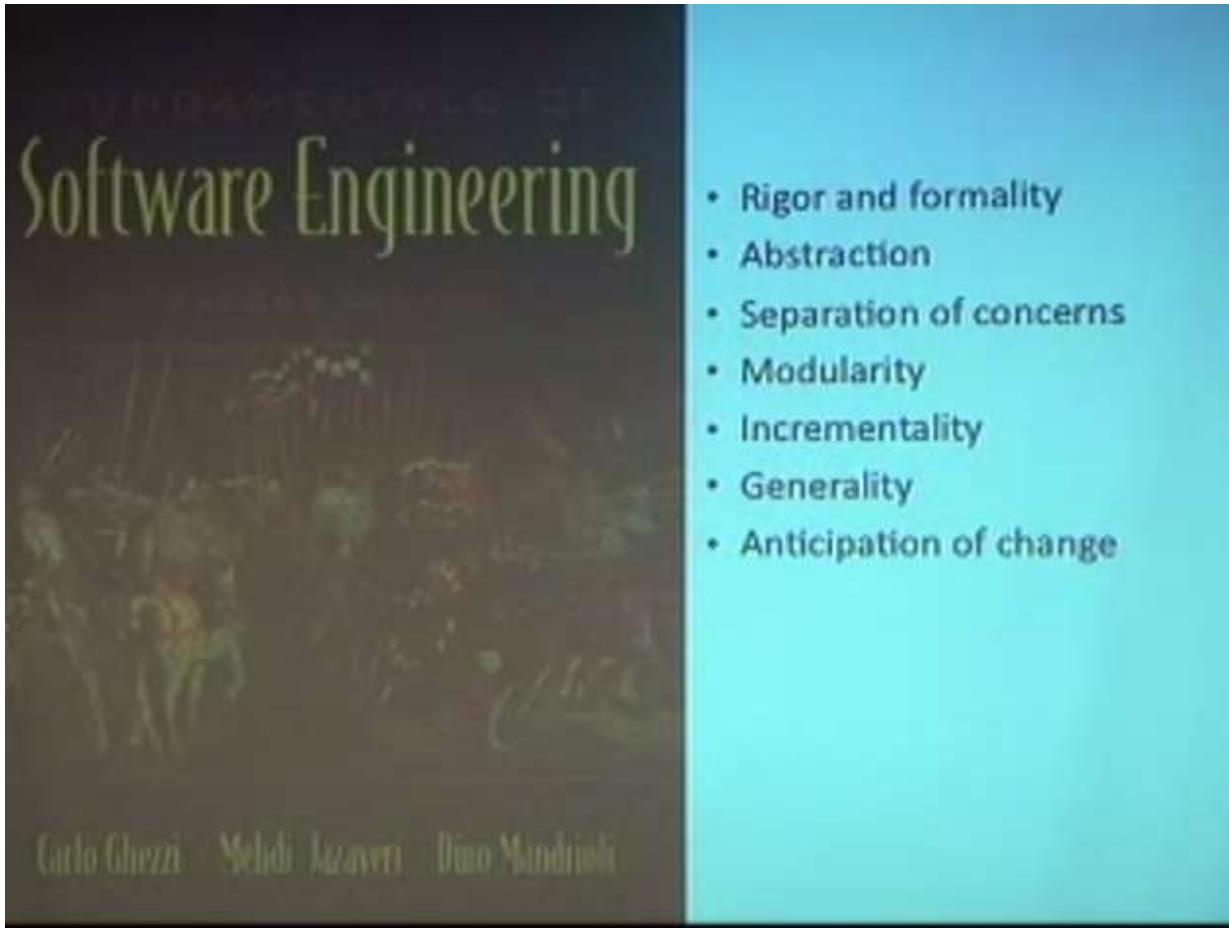


The costs of software errors_4



When model specification process is hasty ... rationales:

Software Engineering Principles



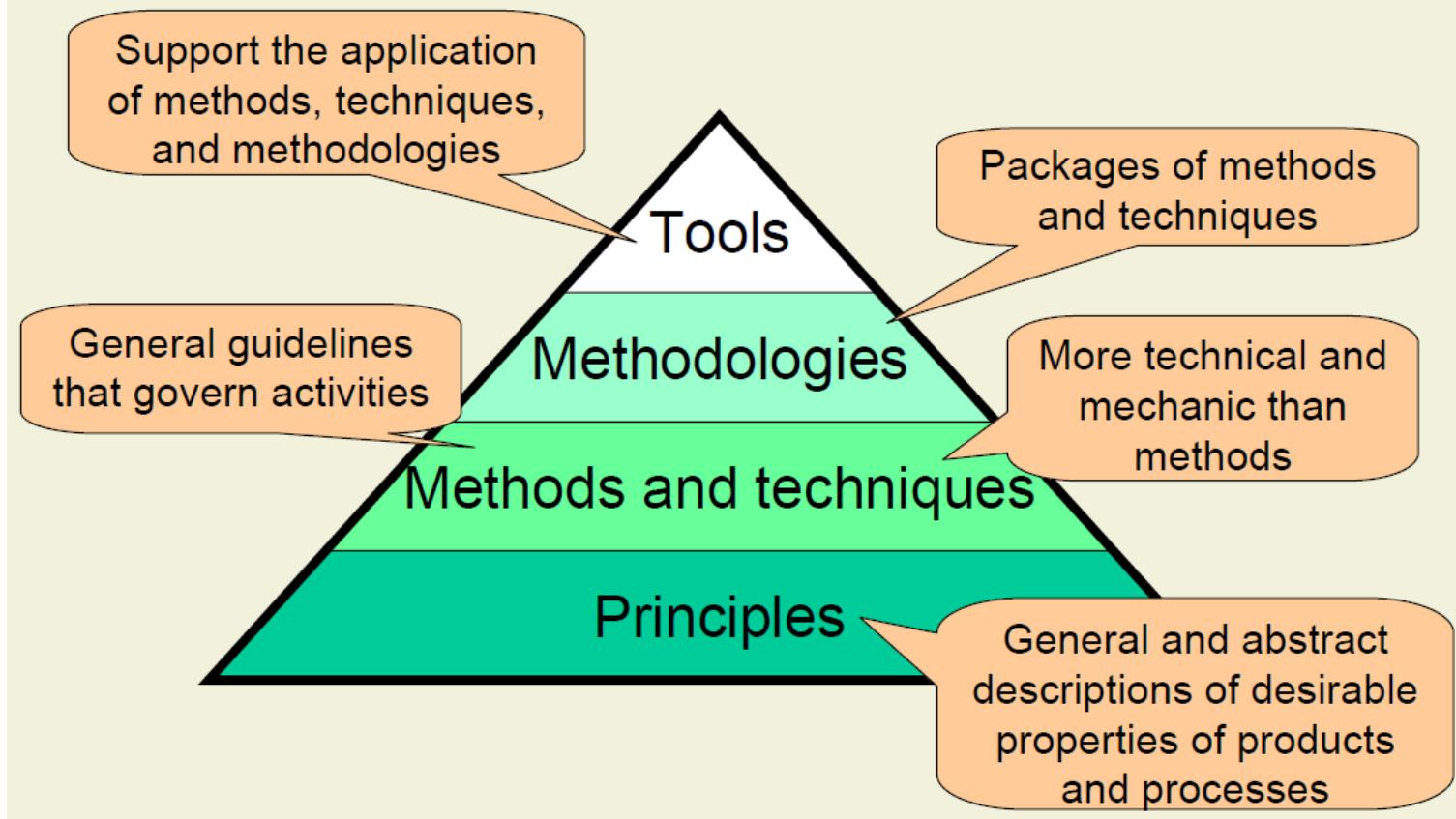
- Rigor and formality
- Abstraction
- Separation of concerns
- Modularity
- Incrementality
- Generality
- Anticipation of change

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli
see <https://www.youtube.com/watch?v=8kG15VoNxhc>.



Software Engineering Principles_2

The Role of Principles



Peter Müller ETH Zurich – SS-06



Software Engineering Principles_3

Important Software Engineering Principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_4

Rigor and Formality

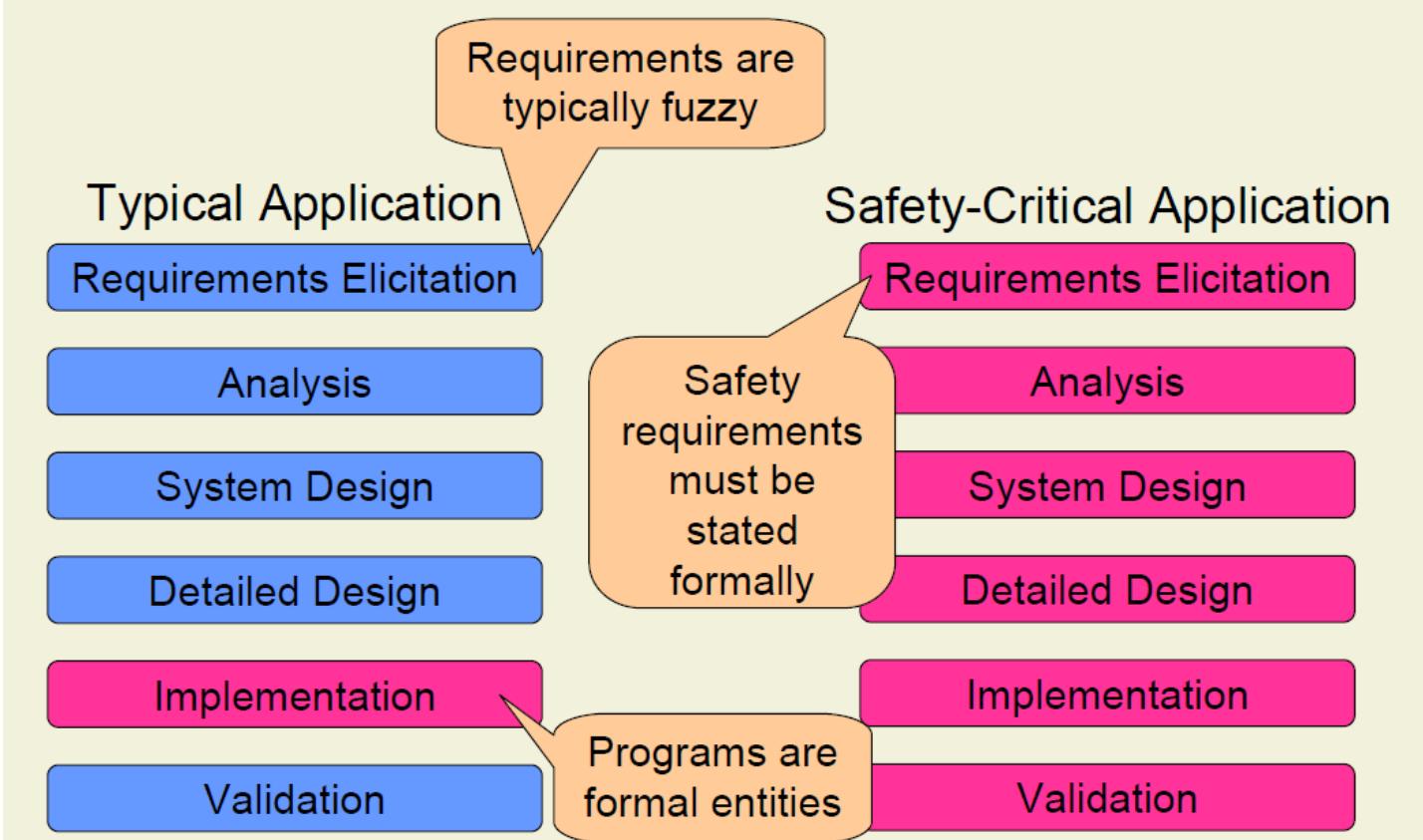
- Rigor means **strict precision**
 - Various degrees of rigor can be achieved
 - Example: mathematical proofs
- Formality is the **highest degree of rigor**
 - Development process driven and evaluated by mathematical laws
 - Examples: refinement
 - Formality enables tool support
- Degree of rigor **depends on application**

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_5

Rigor and Formality: Examples



Peter Müller ETH Zurich – SS-06



Software Engineering Principles_7

Separation of Concerns

- Deal with **different aspects** of a problem **separately**
 - **Reduce complexity**
 - Functionality, reliability, performance, environment, etc.
- Many aspects are **related** and **interdependent**
 - Separate unrelated concerns
 - Consider only the relevant details of a related concern
- **Tradeoff**
 - Risk to miss global optimizations
 - Chance to make optimized decisions in the face of complexity is very limited

https://en.wikipedia.org/wiki/Separation_of_concerns

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_7

Modularity

- Divide system into modules to **reduce complexity**
- **Decompose** a complex system into simpler pieces
- **Compose** a complex system from existing modules
- **Understand** the system in terms of its pieces
- **Modify** a system by modifying only a small number of its pieces

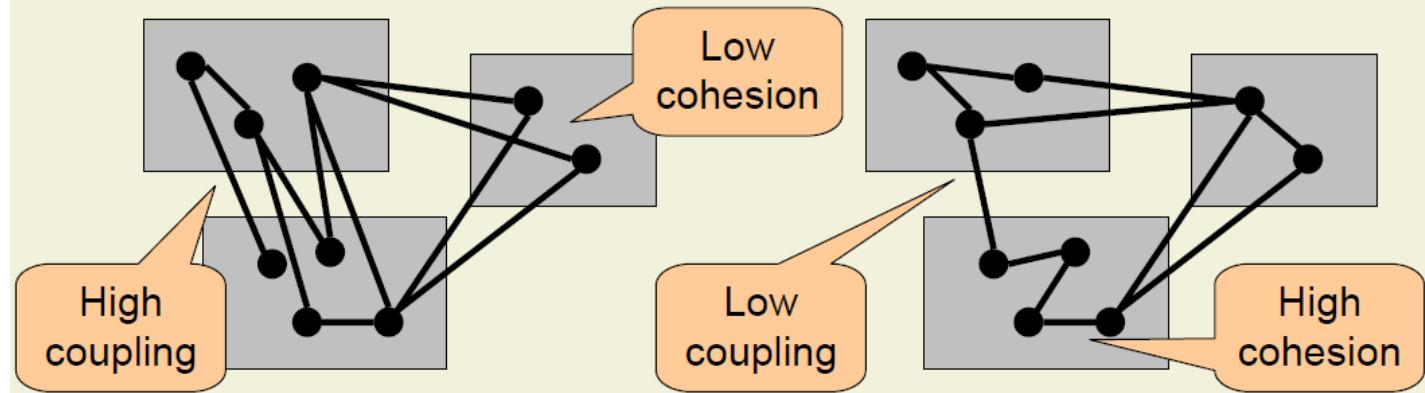
Peter Müller ETH Zurich – SS-06



Software Engineering Principles_8

Cohesion and Coupling

- Cohesion measures **interdependence** of the elements of **one module**
- Coupling measures **interdependence between** different **module**
- Goal: **high cohesion** and **low coupling**



Peter Müller ETH Zurich – SS-06

Software Engineering Principles_9

Abstraction

- Identify the **important aspects** and ignore the **details**
- Abstraction in software engineering
 - **Models** of the real world (omit irrelevant details)
 - **Subtyping** and inheritance (factor out commonalities)
 - Interfaces and **information hiding** (hide implementation details)
 - Parameterization (templates)
 - Structured programming (loops, methods)
 - Layered systems (hide deeper layers in the stack)

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_10

Anticipation of Change

- **Prepare software for changes**
 - **Modularization**: single out elements that are likely to change in the future
 - **Abstraction**: narrow interfaces reduce effects of a change
- Risk: developers spend too much time to make software changeable and reusable
- Software product lines
 - Many similar versions of software (e.g., for different hardware)
 - Examples: software for cell phones, sensors

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_11

Generality

- Attempt to **find more general problem** behind problem at hand
 - Apply **standard solutions and tools**
- A general solution is more likely to be reusable
 - Examples: spreadsheets, database
- General solution may be less efficient
- Example
 - Semantic analysis: Is C a subclass of D?
 - Subclass relation is an acyclic graph
 - Use adjacency matrix and compute transitive closure

Peter Müller ETH Zurich – SS-06



Software Engineering Principles_12

Incrementality

- Characterizes a process which **proceeds in a stepwise fashion**
 - The desired goal is reached by creating successively closer **approximations** to it
- Examples
 - Incremental software life cycles (e.g., spiral model)
 - Prototypes, early feedback
 - Project management is inherently incremental

Peter Müller ETH Zurich – SS-06



What is Software Engineering?

Software engineering is a modeling activity.

Software engineers deal with complexity through modeling, by focusing at any one time on only the relevant details and ignoring everything else.

In the course of development, software engineers build many different models of the system and of the application domain.



What is Software Engineering?_2

SE is a problem-solving activity. Models are used to search for an acceptable solution. This search is driven by experimentation.

Software engineers do not have infinite resources and are constrained by budget and deadlines. Given the lack of a fundamental theory, they often rely on empirical methods to evaluate the benefits of different alternatives.



What is Software Engineering?_3

SE is a knowledge acquisition activity.

In modeling the application and solution domain, software engineers collect data, organize it into information, and formalize it into knowledge.

Knowledge acquisition is not sequential, as a single piece of additional data can invalidate complete models.



What is Software Engineering?_4

SE is a rationale-driven activity.

When acquiring knowledge and making decisions about the system or its application domain, software engineers also need to capture the context in which decisions were made and the rationale behind these decisions. Rationale information, represented as a set of issue models, enables software engineers to understand the implication of a proposed change when revisiting a decision. We assume that change can occur at any time.



Modeling

- One of the basic methods of science.
 - ✓ A model is an abstract representation of a system that enables us to answer questions about the system.
 - ✓ Models are useful when dealing with systems that are too large, too small, too complicated, or too expensive to experience firsthand, but not only. ☺
 - ✓ Models also allow us to visualize and understand systems that either no longer exist or that are only claimed to exist.



Modeling_2

Software engineers need to:

- understand the systems they could build,
- evaluate different solutions and trade-offs.

Most systems are too complex to be understood by any one person, and most systems are expensive to build.

To address these challenges, software engineers describe important aspects of the alternative systems they investigate. In other terms, they need to build a model of the solution domain.



Modeling_3

Object-oriented methods combine the application domain and solution domain modeling activities into one.

- ✓ The application domain is first modeled as a set of objects and relationships. This model is then used by the system to represent the real-world concepts it manipulates. (A train traffic control system includes train objects representing the trains it monitors. A stock trading system includes transaction objects representing the buying and selling of commodities.)
- ✓ Then, solution domain concepts are also modeled as objects.



Modeling_4

- The idea of object-oriented methods is that the solution domain model is a transformation of the application domain model.
- Developing software translates into the activities necessary to identify and describe a system as a set of models that addresses the end user's problem.



Problem solving

Engineering is a problem-solving activity. Engineers search for an appropriate solution, often by:

- trial and error,
- evaluating alternatives empirically,

with limited resources and incomplete knowledge.

In its simplest form, the engineering method includes:

1. Formulate the problem.
2. Analyze the problem.
3. Search for solutions.
4. Decide on the appropriate solution.
5. Specify the solution.



Problem solving_2

Software engineering is an engineering activity, it is not algorithmic. SE requires:

- **experimentation,**
- **the reuse of pattern solutions,**
- **the incremental evolution of the system.**

Object-oriented software development typically includes:

- **requirements elicitation,**
- **analysis,**
- **system design,**
- **object design,**
- **implementation,**
- **testing**



Problem solving_3

Software development also includes activities whose purpose is to evaluate the appropriateness of the respective models.

- During the analysis review, the application domain model is compared with the client's reality, which in turn might change as a result of modeling.
- During the design review, the solution domain model is evaluated against project goals.



Problem solving_4

- During testing, the system is validated against the solution domain model, which might be changed by the introduction of new technologies.
- During project management, managers compare their model of the development process (i.e., the project schedule and budget) against reality (i.e., the delivered work products and expended resources).



Knowledge acquisition

- A common mistake that software engineers and managers make is to assume that the acquisition of knowledge needed to develop a system is linear.
- Knowledge acquisition is a nonlinear process. The addition of a new piece of information may invalidate all the knowledge we have acquired for the understanding of a system. Even if we had already documented this understanding in documents and code (“The system is 90% coded, we will be done next week”), we must be mentally prepared to start from scratch.



Knowledge acquisition_2

- There are several software processes that deal with this problem by avoiding the sequential dependencies inherent in the waterfall model.

Risk-based development attempts to anticipate surprises late in a project by identifying the high-risk components. Issue-based development attempts to remove the linearity altogether.

Any development activity: analysis, system design, object design, implementation, testing, or delivery; can influence any other activity. In issue-based development, all these activities are executed in parallel.



Rationale

- Assumptions that developers make about a system change constantly. Design and implementation faults discovered during testing and usability problems discovered during user evaluation trigger changes to the solution models. Changes can also be caused by new technology.
- A typical task of software engineers is to change a currently operational software system to incorporate this new enabling technology.



Rationale_2

- To change the system, it is not enough to understand its current components and behavior; it is also necessary to capture and understand the context in which each design decision was made.

This additional knowledge is called the rationale of the system.

- Capturing and accessing the rationale of a system is not trivial. For every decision made, several alternatives may have been considered, evaluated, and argued.



Rationale_3

- Rationale represents a much larger amount of information than do the solution models. **Rationale information is often not explicit.**

Developers make many decisions based on their experience and their intuition, without explicitly evaluating different alternatives. When asked to explain a decision, developers may have to spend a substantial amount of time recovering its rationale.

In order to deal with changing systems, software engineers must address the challenges of capturing and accessing rationale.



Software Engineering Concepts

The project purpose is to develop a software system

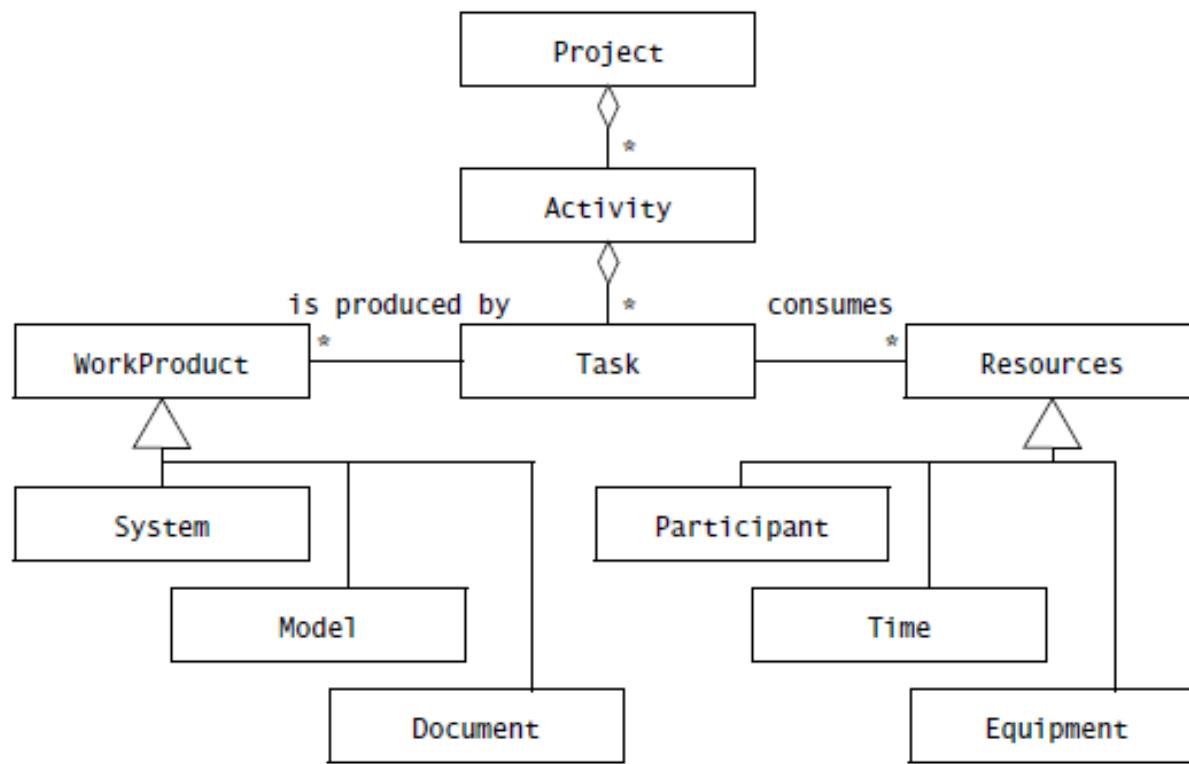


Figure 1-1 Software engineering concepts depicted as a UML class diagram [OMG, 2009].



Participants and Roles

- Developing a software system requires the collaboration of many people with different backgrounds and interests:
 - the client orders and pays for the system,
 - the developers construct the system,
 - the project manager plans and budgets the project and coordinates the developers and the client,
 - the end users are supported by the system.



Participants and Roles_2

- We refer to all the persons involved in the project as **participants**.
- We refer to a set of responsibilities in the project or the system as a **role**.
- A role is associated with a set of tasks and is assigned to a participant. The same participant can fill multiple roles.



Participants and Roles_3

- TicketDistributor is a machine that distributes tickets for trains. Travelers have the option of selecting a ticket for a single trip or for multiple trips, or selecting a timecard for a day or a week. The TicketDistributor computes the price of the requested ticket based on the area in which the trip will take place and whether the traveler is a child or an adult.
- The TicketDistributor must be able to handle several exceptions, such as travelers who do not complete the transaction, travelers who attempt to pay with large bills, and resource outages, such as running out of tickets, change, or power.



Systems & models

- We use the term system as a collection of interconnected parts.

Modeling is a way to deal with complexity by ignoring irrelevant details. We use the term model to refer to any abstraction of the system.

A TicketDistributor for an underground train is a system.

Blueprints for the TicketDistributor, schematics of its electrical wiring, and object models of its software are models of the TicketDistributor



Systems & models_2

- A development project **is itself a system that can be modeled.**
- The project schedule, its budget, and its planned deadlines are **models of the development project.**



Example of roles in Software Engineering

Table 1-1 Examples of roles in software engineering for the *TicketDistributor* project.

Role	Responsibilities	Examples
Client	The client is responsible for providing the high-level requirements on the system and for defining the scope of the project (delivery date, budget, quality criteria).	Train company that contracts the <i>TicketDistributor</i> .
User	The user is responsible for providing domain knowledge about current user tasks. Note that the client and the user are usually filled by different persons.	Travelers
Manager	A manager is responsible for the work organization. This includes hiring staff, assigning them tasks, monitoring their progress, providing for their training, and generally managing the resources provided by the client for a successful delivery.	Alice (boss)
Human Factors Specialist	A human factors specialist is responsible for the usability of the system.	Zoe (Human Computer Interaction specialist)
Developer	A developer is responsible for the construction of the system, including specification, design, implementation, and testing. In large projects, the developer role is further specialized.	John (analyst), Marc (programmer), & Zoe (tester) ^a
Technical Writer	The technical writer is responsible for the documentation delivered to the client. A technical writer interviews developers, managers, and users to understand the system.	John



Work products

- A work product is an artifact that is produced during the development, such as a document or a piece of software for other developers or for the client.
- We refer to a work product for the project's internal consumption as an internal work product.
- We refer to a work product that must be delivered to a client as a deliverable.
Deliverables are generally defined prior to the start of the project and specified by a contract binding the developers with the client.



Work products_2

Table 1-2 Examples of work products for the TicketDistributor project.

Work product	Type	Description
Specification	Deliverable	The specification describes the system from the user's point of view. It is used as a contractual document between the project and the client. The TicketDistributor specification describes in detail how the system should appear to the traveler.
Operation manual	Deliverable	The operation manual for the TicketDistributor is used by the staff of the train company responsible for installing and configuring the TicketDistributor. Such a manual describes, for example, how to change the price of tickets and the structure of the network into zones.
Status report	Internal work product	A status report describes at a given time the tasks that have been completed and the tasks that are still in progress. The status report is produced for the manager, Alice, and is usually not seen by the train company.
Test manual	Internal work product	The test plans and results are produced by the tester, Zoe. These documents track the known defects in the prototype TicketDistributor and their state of repair. These documents are usually not shared with the client.



Activities, Tasks, and Resources

- Activity - a set of tasks that is performed toward a specific purpose. For example:
 - ✓ requirements elicitation is an activity whose purpose is to define with the client what the system will do,
 - ✓ delivery is an activity whose purpose is to install the system at an operational location,
 - ✓ management is an activity whose purpose is to monitor and control the project such that it meets its goals (e.g., deadline, quality, budget).



Activities, Tasks, and Resources_2

- Activities can be composed of other activities. The delivery activity includes a software installation activity and an operator training activity. Activities are also sometimes called phases.
- A task represents an atomic unit of work that can be managed. A manager assigns it to a developer, the developer carries it out, and the manager monitors the progress and completion of the task. Tasks consume resources, result in work products, and depend on work products produced by other tasks.



Activities, Tasks, and Resources_3

Table 1-3 Examples of activities, tasks, and resources for the *TicketDistributor* project.

Example	Type	Description
Requirements elicitation	Activity	The requirements elicitation activity includes obtaining and validating requirements and domain knowledge from the client and the users. The requirements elicitation activity produces the specification work product (Table 1-2).
Develop “Out of Change” test case for <i>TicketDistributor</i>	Task	This task, assigned to Zoe (the tester) focuses on verifying the behavior of the ticket distributor when it runs out of money and cannot give the correct change back to the user. This activity includes specifying the environment of the test, the sequence of inputs to be entered, and the expected outputs.
Review “Access Online Help” use case for usability	Task	This task, assigned to John (the human factors specialist) focuses on detecting usability issues in accessing the online help features of the system.
Tariff Database	Resource	The tariff database includes an example of tariff structure with a train network plan. This example is a resource provided by the client for requirements and testing.



Functional and non Functional Requirements

- Requirements specify a set of features that the system must have.
 - ✓ A functional requirement is a specification of a function that the system must support;
 - ✓ a nonfunctional requirement is a constraint on the operation of the system that is not related directly to a function of the system.



Functional and non Functional Requirements_2

- Functional requirements:
 - ✓ the user must be able to purchase tickets
 - ✓ the user must be able to access tariff information
- Nonfunctional requirements:
 - ✓ the user must be provided feedback in less than one second
 - ✓ the colors used in the interface should be consistent with the company colors



Methods, and Methodologies

- Modeling language/notation
- A method is a repeatable technique that specifies the steps involved in solving a specific problem. Examples:
 - a recipe is a method for cooking a specific dish,
 - a sorting algorithm is a method for ordering elements of a list,
 - rationale management is a method for justifying change,
 - configuration management is a method for tracking change.



Methods, and Methodologies_2

- A methodology is a collection of methods for solving a class of problems and specifies how and when each method should be used.
- A seafood cookbook with a collection of recipes is a methodology for preparing seafood if it also contains advices on how ingredients should be used and what to do if not all ingredients are available.



Software Engineering Development Activities

- Development activities deal with the complexity by constructing and validating models of the application domain or the system.

Development activities include:

- ✓ Requirements Elicitation
- ✓ Analysis
- ✓ System Design
- ✓ Object Design
- ✓ Implementation
- ✓ Testing



Software Engineering Development Activities_2

Requirements Elicitation

- During requirements elicitation, the client and developers define the purpose of the system. The result of this activity is a description of the system in terms of actors and use cases. Actors represent the external entities that interact with the system.



Software Engineering Development Activities_3

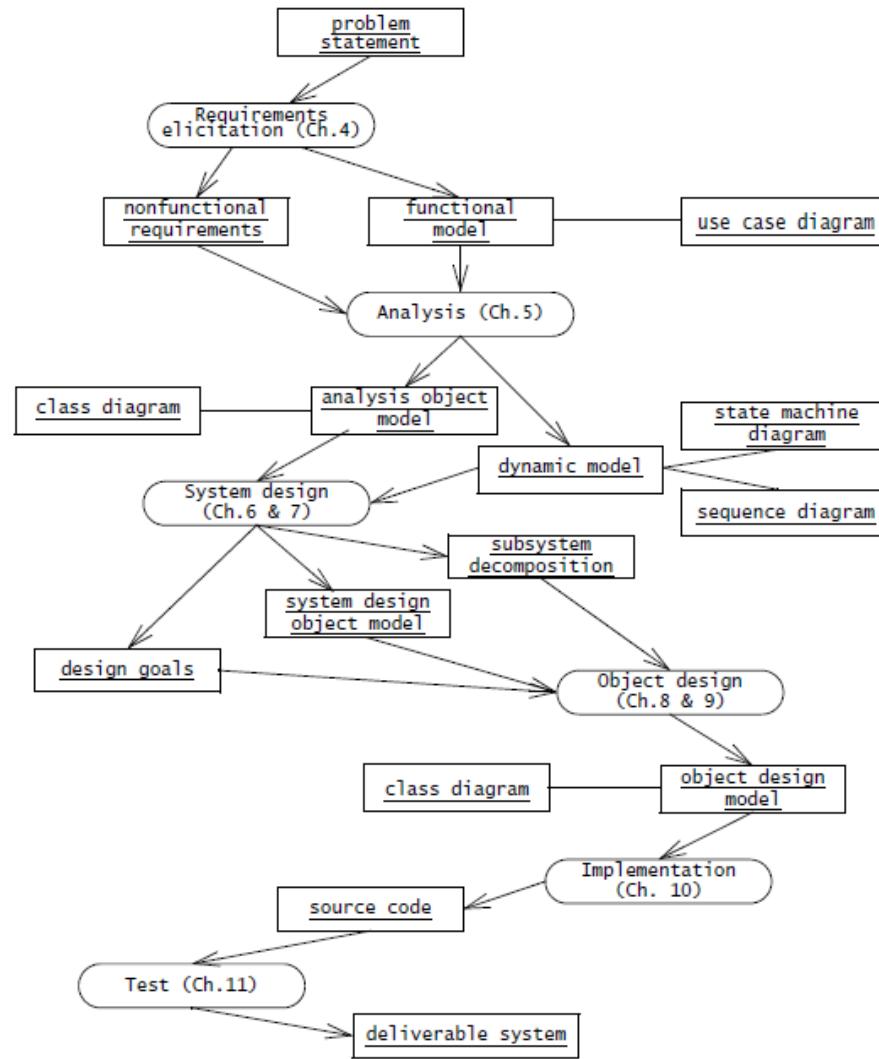


Figure 1-2 An overview of object-oriented software engineering development activities and their products. This diagram depicts only logical dependencies among work products. Object-oriented software engineering is iterative; that is, activities can occur in parallel and more than once.

Software Engineering Development Activities_4

<i>Use case name</i>	PurchaseOneWayTicket
<i>Participating actor</i>	Initiated by Traveler
<i>Flow of events</i>	<ol style="list-style-type: none">1. The Traveler selects the zone in which the destination station is located.2. The TicketDistributor displays the price of the ticket.3. The Traveler inserts an amount of money that is at least as much as the price of the ticket.4. The TicketDistributor issues the specified ticket to the Traveler and returns any change.
<i>Entry condition</i>	The Traveler stands in front of the TicketDistributor, which may be located at the station of origin or at another station.
<i>Exit condition</i>	The Traveler holds a valid ticket and any excess change.
<i>Quality requirements</i>	If the transaction is not completed after one minute of inactivity, the TicketDistributor returns all inserted change.

Figure 1-3 An example of use case, PurchaseOneWayTicket.



Software Engineering Development Activities_5

Analysis

- Developers:

- aim to produce a model of the system that is correct, complete, consistent, and unambiguous,
- transform the use cases produced during requirements elicitation into an object model that completely describes the system,
- discover ambiguities and inconsistencies in the use case model that they resolve with the client.



Software Engineering Development Activities_6

Analysis

- The result of analysis is a system model annotated with attributes, operations, and associations.
- The system model can be described in terms of its structure and its dynamic interoperation.



Software Engineering Development Activities_7

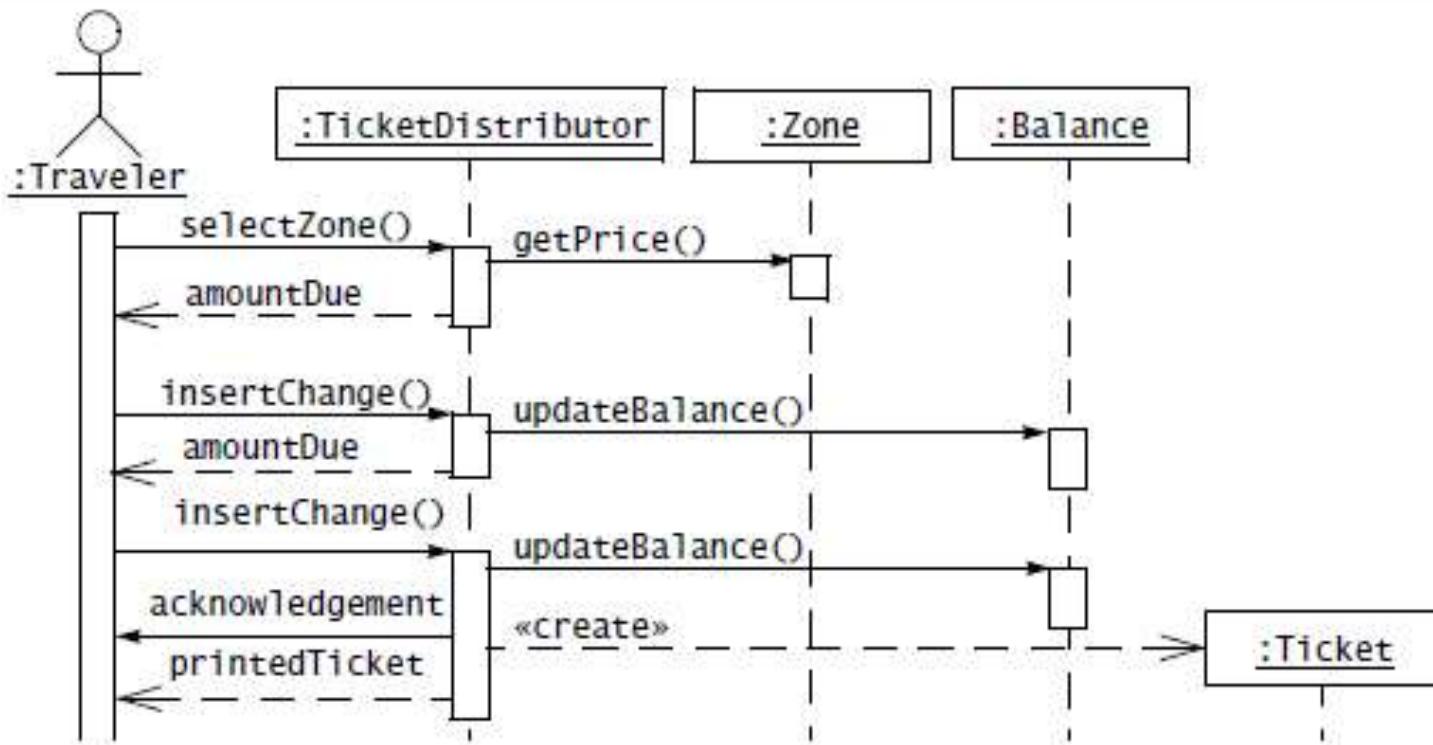


Figure 1-4 A dynamic model for the TicketDistributor (UML sequence diagram). This diagram depicts the interactions between the actor and the system during the PurchaseOneWayTicket use case and the objects that participate in the use case.



Software Engineering Development Activities_8

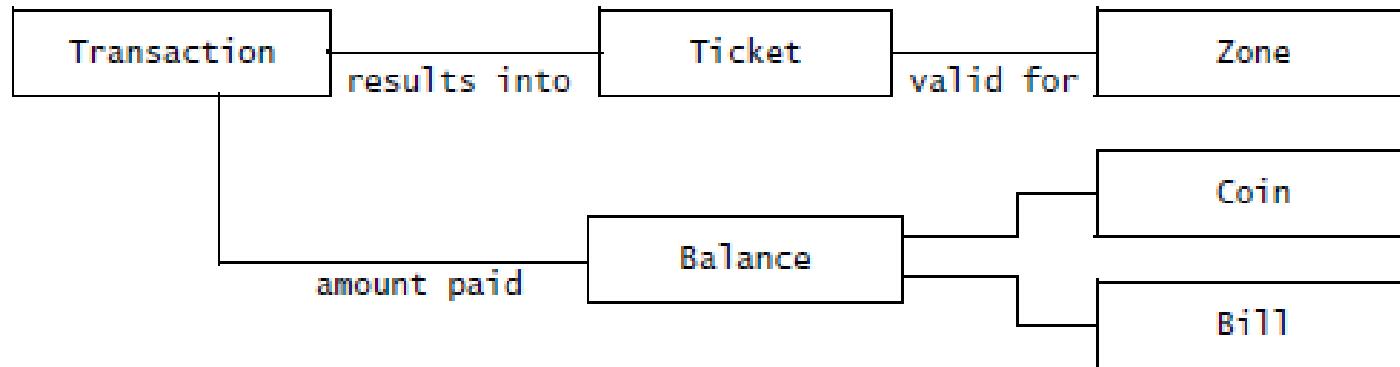


Figure 1-5 An object model for the `TicketDistributor` (UML class diagram). In the `PurchaseOneWayTicket` use case, a `Traveler` initiates a transaction that will result in a `Ticket`. A `Ticket` is valid only for a specified `Zone`. During the `Transaction`, the system tracks the `Balance` due by counting the `Coin`s and `Bill`s inserted.



Software Engineering Development Activities_9

System Design

- Developers:
 - ✓ define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams,
 - ✓ select strategies for building the system, such as the hardware/software platform on which the system will run, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions.



Software Engineering Development Activities_10

System Design

- ⇒ a clear description of each of these strategies, a subsystem decomposition, and a deployment diagram representing the hardware/software mapping of the system.
- Whereas both analysis and system design produce models of the system under construction, only analysis deals with entities that the client can understand.



Software Engineering Development Activities_11

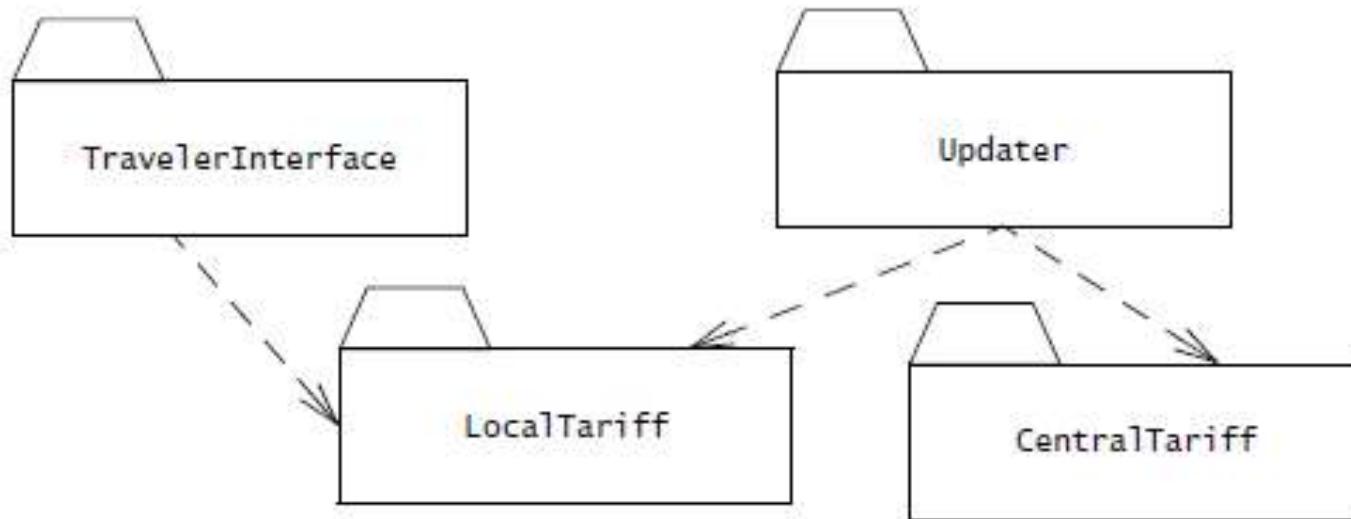


Figure 1-6 A subsystem decomposition for the `TicketDistributor` (UML class diagram, packages represent subsystems, dashed lines represent dependencies). The `TravelerInterface` subsystem is responsible for collecting input from the `Traveler` and providing feedback (e.g., display ticket price, returning change). The `LocalTariff` subsystem computes the price of different tickets based on a local database. The `CentralTariff` subsystem, located on a central computer, maintains a reference copy of the tariff database. An `Updater` subsystem is responsible for updating the local databases at each `TicketDistributor` through a network when ticket prices change.



Software Engineering Development Activities_12

Object Design

- developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform defined during system design. This includes:
 - ✓ precisely describing object and subsystem interfaces,
 - ✓ selecting off-the-shelf components,
 - ✓ restructuring the object model to attain design goals such as extensibility or understandability,
 - ✓ optimizing the object model for performance.



Software Engineering Development Activities_13

- The result of the object design activity is a detailed object model annotated with constraints and precise descriptions for each element.

Implementation

- developers translate the solution domain model into source code. This includes implementing the attributes and methods of each object and integrating all the objects such that they function as a single system.
- spans the gap between the detailed object design model and a complete set of source code files that can be compiled.



Software Engineering Development Activities_14

Testing

- developers find differences between the system and its models by executing the system (or parts of it) with sample input data sets.
 - ✓ unit testing, developers compare the object design model with each object and subsystem.
 - ✓ integration testing, combinations of subsystems are integrated together and compared with the system design model.



Software Engineering Development Activities_15

Testing

- system testing, typical and exception cases are run through the system and compared with the requirements model.
- The goal of testing is to discover as many faults as possible such that they can be repaired before the delivery of the system. The planning of test phases occurs in parallel to the other development activities.



Managing Software Development

- Management activities focus on:

- ✓ planning the project,
- ✓ monitoring its status,
- ✓ tracking changes,
- ✓ coordinating resources

such that a high-quality product is delivered on time and within budget.

- Management activities not only involve managers, but also most of the other project participants as well.



Managing Software Development_2

- Management activities include:
 - ✓ Communication
 - ✓ Rationale Management
 - ✓ Software Configuration Management
 - ✓ Project Management
 - ✓ Software Life Cycle
- As modern software engineering projects become more change driven, the distinction between construction activities and maintenance activities is blurred.



Managing Software Development_3

Communication

- is the most critical and time-consuming activity in software engineering. Misunderstandings and omissions often lead to faults and delays that are expensive to correct later in the development,
- includes the exchange of models and documents about the system and its application domain, reporting the status of work products, providing feedback on the quality of work products, raising and negotiating issues, and communicating decisions.



References

- ▶ Bernd Bruegge & Allen H. Dutoit - Object-Oriented Software Engineering Using UML, Patterns, and Java – Third Edition – Prentice Hall 2010
- ▶ Ian Sommerville – Software Engineering - Ninth Edition – Addison Wesley 2011
- ▶ Ivan Marsic –Software Engineering - Rutgers University, New Brunswick, New Jersey 2012



Evaluation

- ▶ Course - written exam - 60%
- ▶ Optional:
- ▶ Partial (half semester) *evaluation* ?
- ▶ Seminar/lab activities - 40%



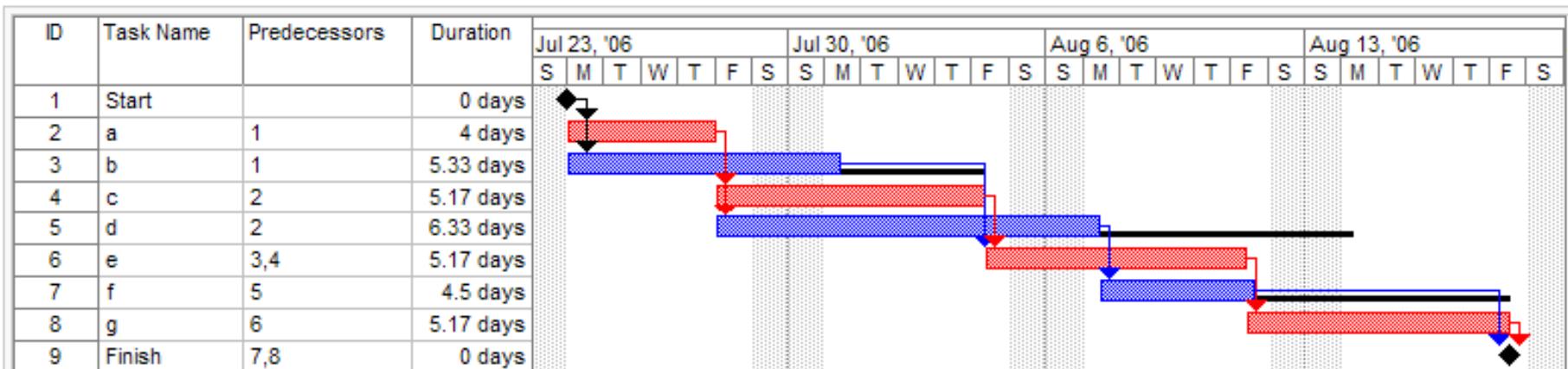
Lab activities - Evaluation

- ▶ **Lab is focused on:**
 - ▶ small teams working to develop software,
 - ▶ usage of models in all phases of software life cycle,
 - ▶ using appropriate tools in model construction, validation and code generation
 - ▶ constructing rigorous models complying with the requirements, design and implementation,
 - ▶ usage of design patterns,
 - ▶ code generation, viewed as model transformation,
 - ▶ system testing,
- ▶ **Teachers will discuss with all team-members about expected deliverables for the current lab**



Program (project) evaluation and review technique, (PERT)

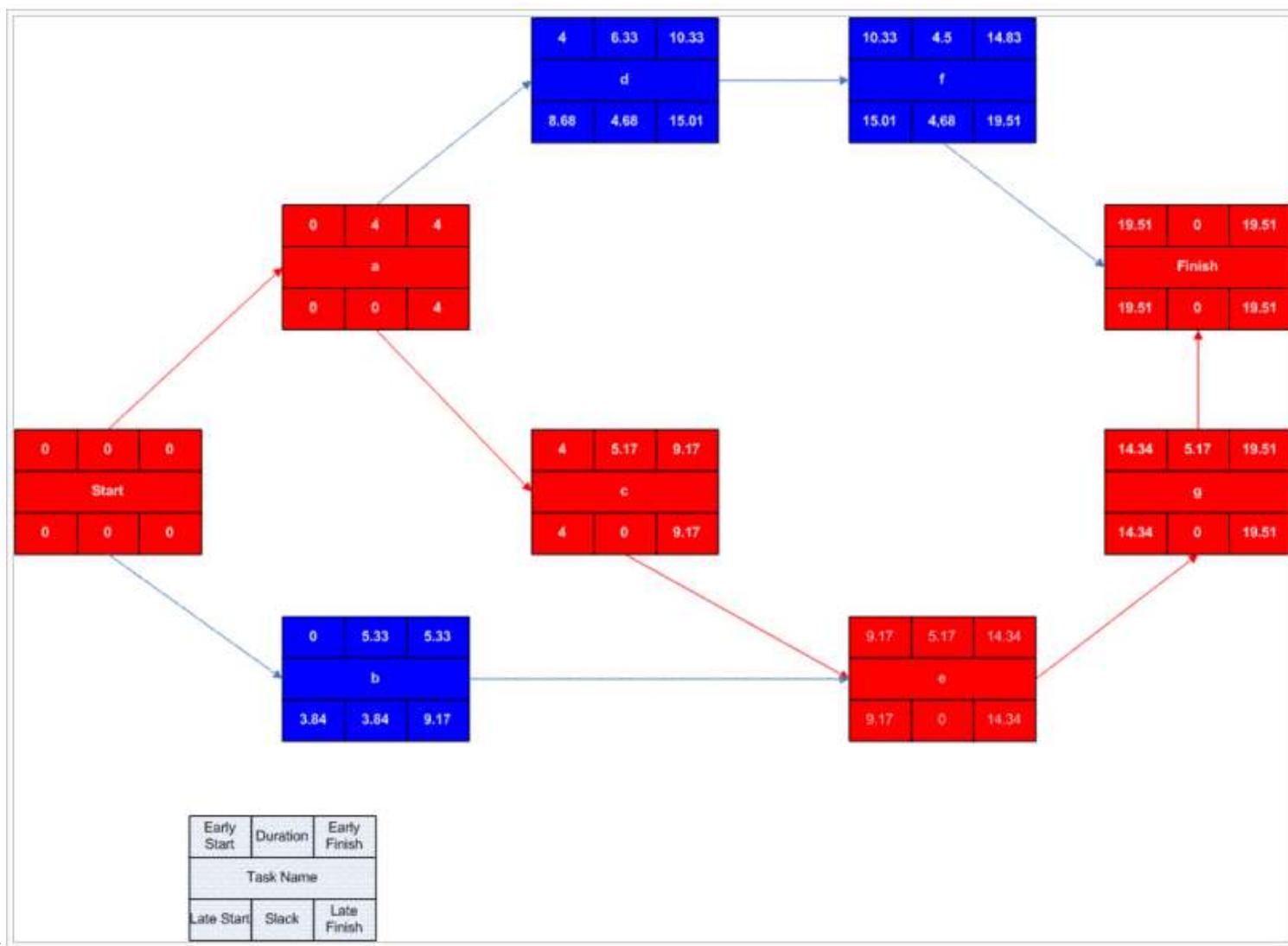
Once this step is complete, one can draw a [Gantt chart](#) or a network diagram.



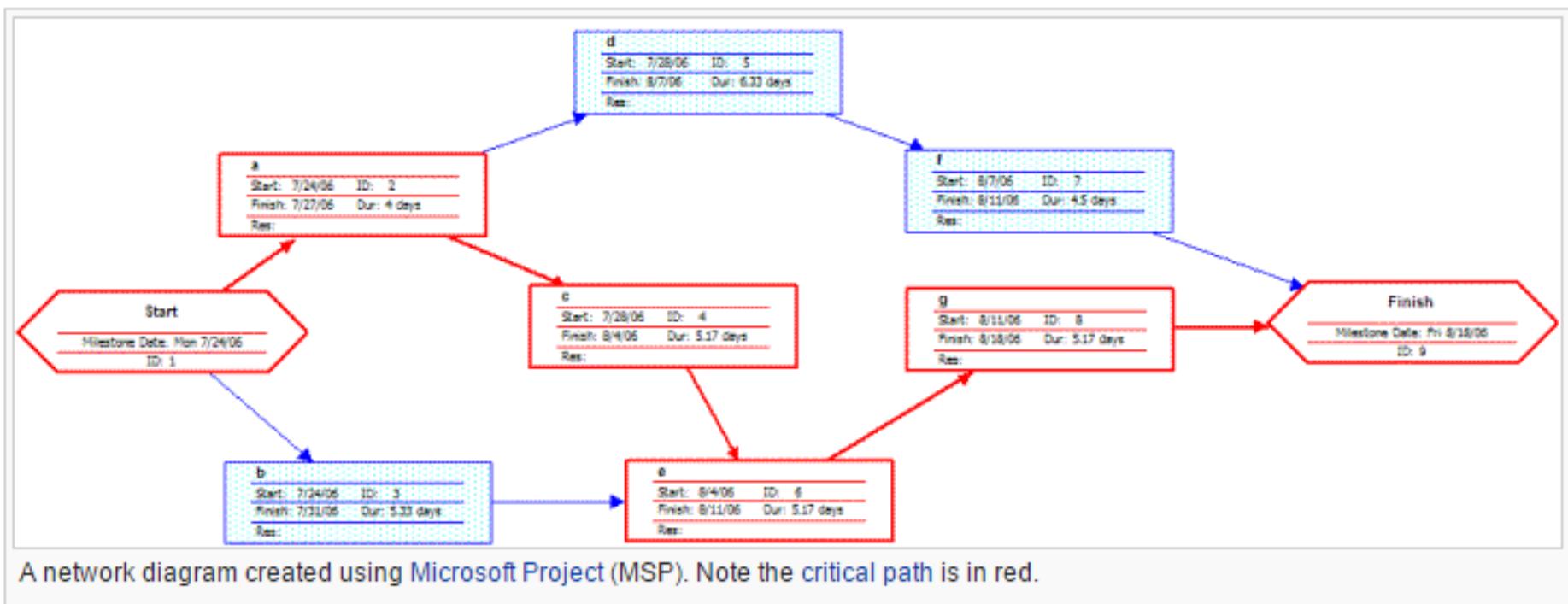
A Gantt chart created using Microsoft Project (MSP). Note (1) the critical path is in red, (2) the slack is the black lines connected to non-critical activities, (3) since Saturday and Sunday are not work days and are thus excluded from the schedule, some bars on the Gantt chart are longer if they cut through a weekend.



Critical Path Method (CPM) – Microsoft Visio



Critical Path Method (CPM) – realized with MSP

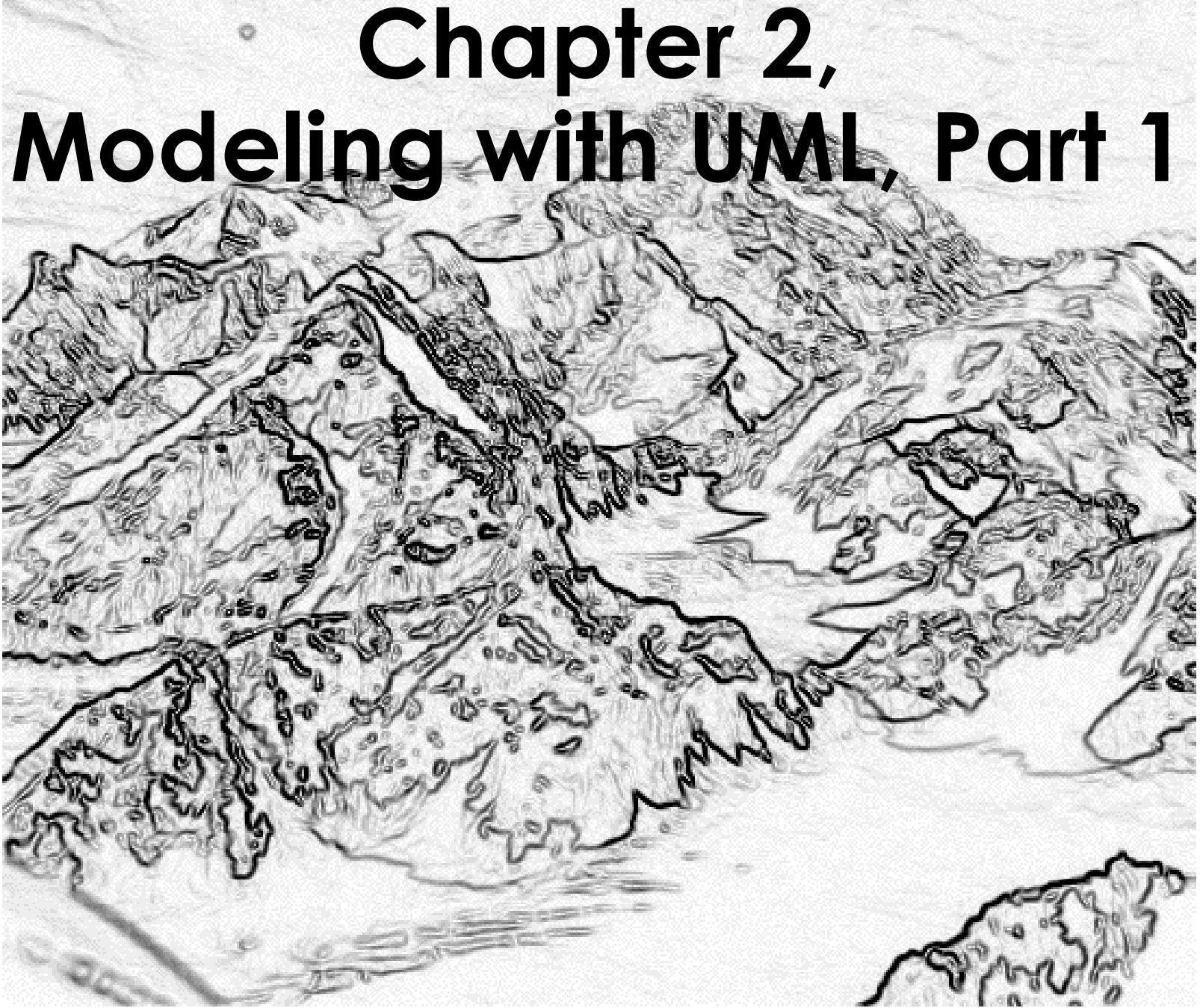


Thanks for your patience!



Object-Oriented Software Engineering

Using UML, Patterns, and Java



Chapter 2, Modeling with UML, Part 1

OMT, OOA&D, OOSE and UML

- The first three methodologies, the modeling language is described by examples in which the concepts and relationships among them is presented by means of the concrete syntax.
- The UML standard description is primarily focused on the abstract syntax, concrete syntax, semantics and at the end on some examples.
- Apart of the above-mentioned methodologies UML is a modeling language including a complementary language, OCL to query the model, to specify different constraints and using the constraint language to define the abstract syntax of UML.

Using models in software development

- For supporting problem and **system** understanding and communications between different participants (the approach described in Bruegge's book and in almost all SE books).
- For supporting all development phases including code generation in a significant manner also for supporting system understanding.
 - Model compilability and compliance with problem requirements
 - The current state of the art concerning RE tools
 - solutions

Overview for the Lecture

- Three ways to deal with complexity
 - ➡ Abstraction and Modeling
 - Decomposition
 - Hierarchy
- Introduction into the UML **notation**
- First pass on:
 - Use case diagrams
 - Class diagrams
 - Sequence diagrams
 - Statechart diagrams
 - Activity diagrams

Abstraction

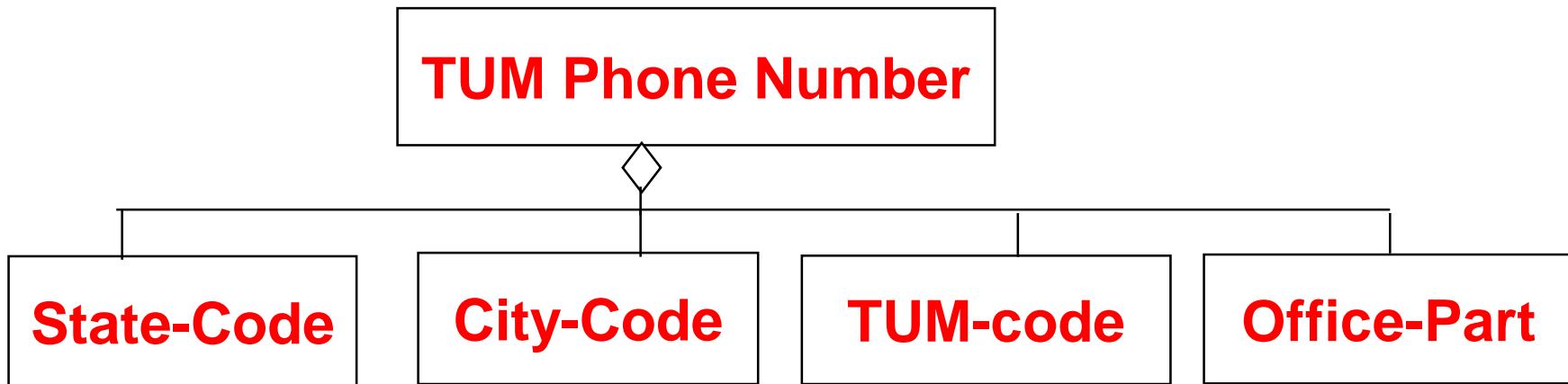
- Complex systems are hard to understand
 - The 7 +- 2 phenomena
 - Our short-term memory cannot store more than 7+-2 pieces at the same time -> **limitation of the brain**
 - TUM Phone Number: 498928918204

Abstraction

- Complex systems are hard to understand
 - The 7 +- 2 phenomena
 - Our short-term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
 - TUM Phone Number: 498928918204
- Chunking:
 - Group collection of objects to reduce complexity
 - 4 chunks:
 - State-code, city-code, TUM-code, Office-Part

Abstraction

- Complex systems are hard to understand
 - The 7 +- 2 phenomena
 - Our short-term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
 - TUM Phone Number: 498928918204
- Chunking:
 - Group collection of objects to reduce complexity
 - State-code, city-code, TUM-code, Office-Part



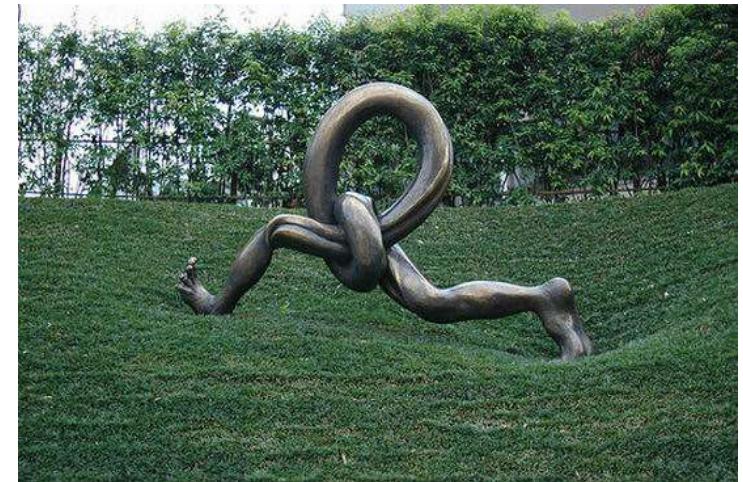
Abstraction

- Abstraction allows us to ignore unessential details
- Two definitions for abstraction:
 - **Abstraction as activity**
 - Abstraction is the *resulting idea* of a thought process
~~where an~~ in which the idea has been distanced from ~~an~~ the object
- **Abstraction as entity**
- ~~Ideas can be expressed by models~~
 - Models are expressed by means of ideas



Model

- A model is an abstraction of a system
 - A system that no longer exists
 - An existing system
 - A future system to be built.

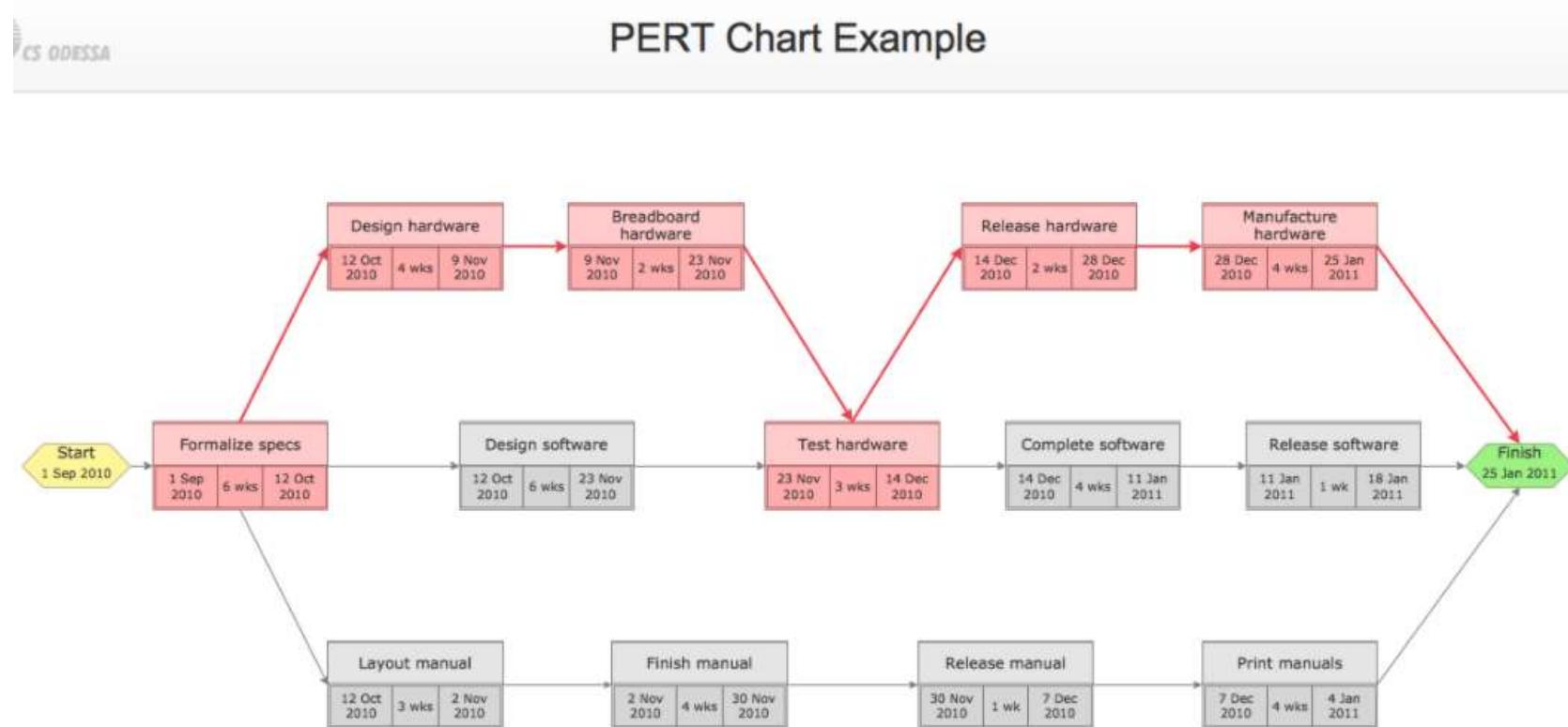


We use Models to describe Software Systems

- Object/Structural model/view: What is the structure of the system?
- Functional model/view: What are the functions of the system?
- Dynamic/Behavioral model/view: How does the system react to external events?
- System/Application Model: Object /Structural model + Functional model + Dynamic model

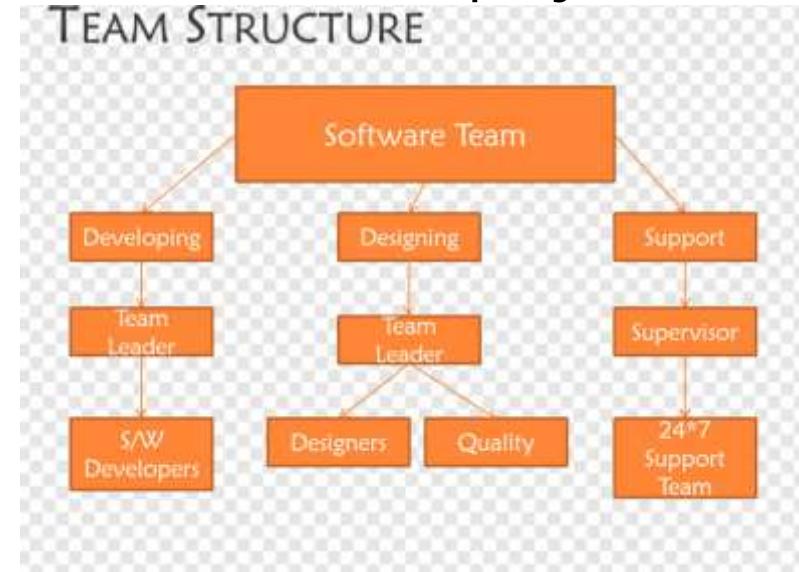
Other models used to describe Software System Development

- Task Model:
 - **PERT Chart:** What are the dependencies between tasks?



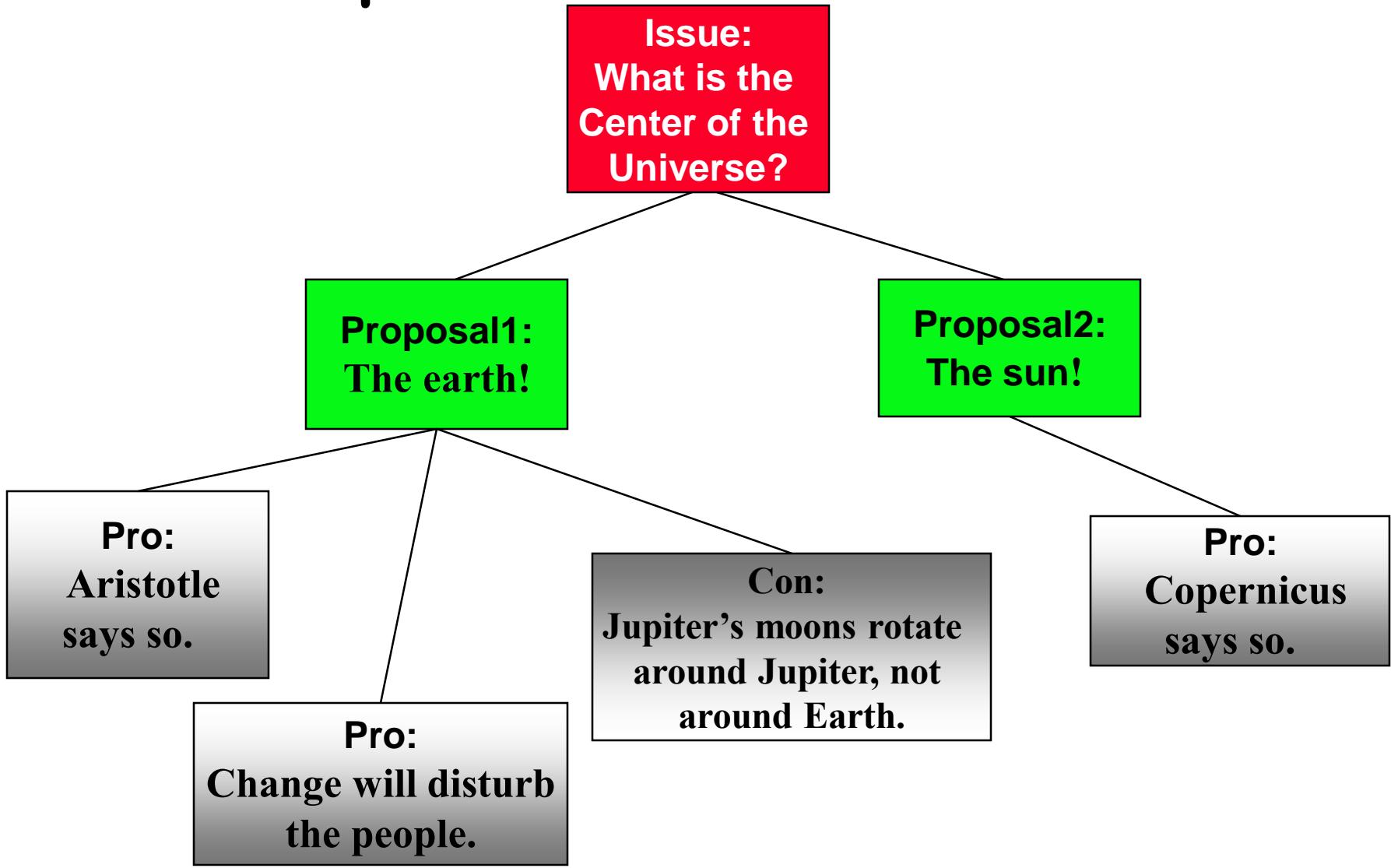
Also, models may be used to describe Software System Development

- Task Model:
 - Organization Chart: What are the roles in the project?

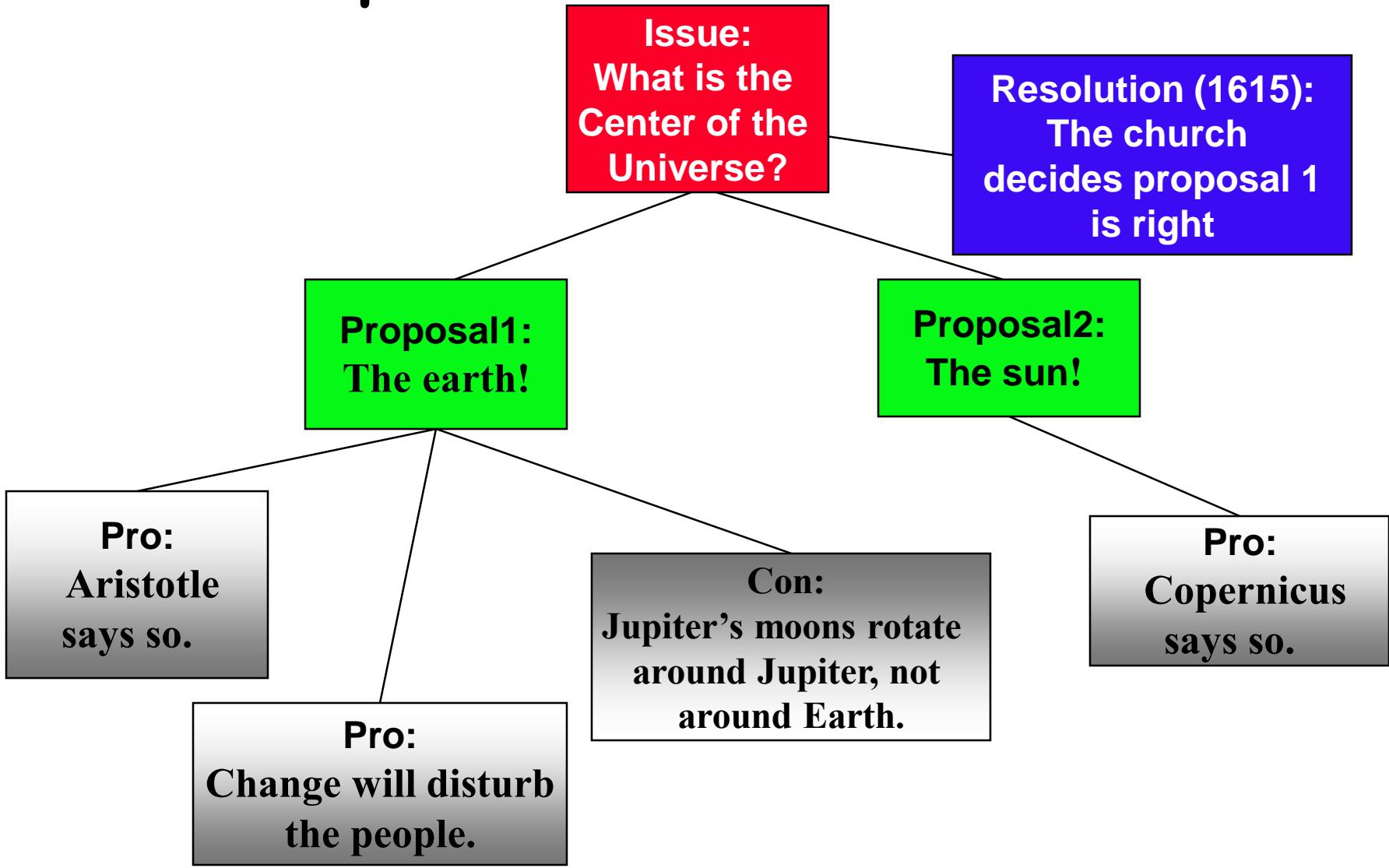


- Issues Model:
 - What are the open and closed issues?
 - What blocks me from continuing?
 - What constraints were imposed by the client?
 - What resolutions were made?
 - These lead to action items

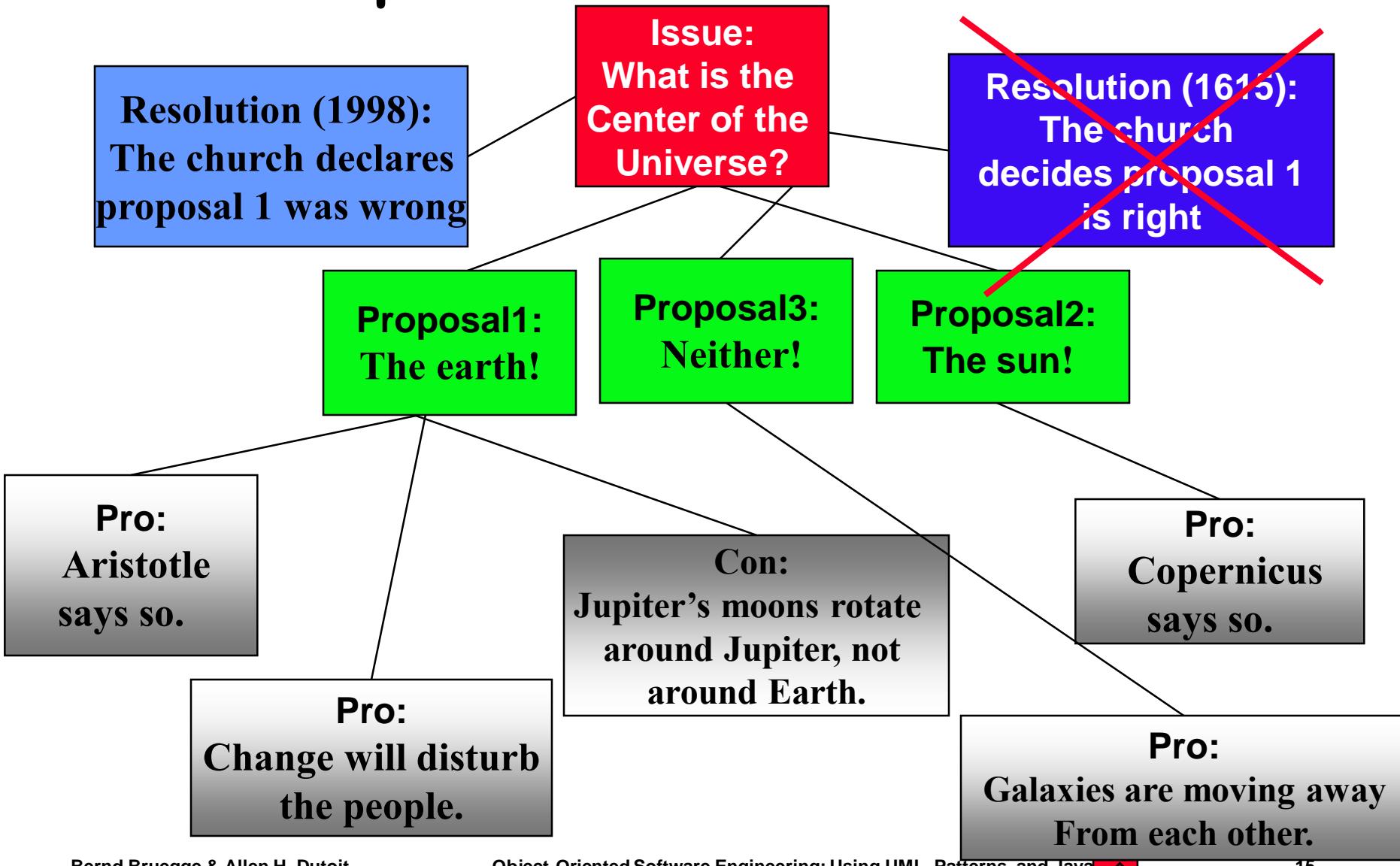
Issue-Modeling - Knowledge acquisition is a nonlinear process



Issue-Modeling - Knowledge acquisition is a nonlinear process



Issue-Modeling - Knowledge acquisition is a nonlinear process



Models must be falsifiable - Knowledge acquisition is a nonlinear process

- Karl Popper ("Objective Knowledge"):
 - There is no absolute truth when trying to understand reality
 - **One can only build theories, that are "true" until somebody finds a counter example**
- **Falsification:** The act of disproving a theory or hypothesis
- The truth of a theory is never certain. We must use phrases like:
 - **"by our best judgement", "using state-of-the-art knowledge"**
- In software engineering any model is a theory:
 - We build models and try to find counter examples by:
 - Requirement's validation, user interface testing, review of the design, source code testing, system testing, etc.
- **Testing: The act of disproving a model.**

2. Technique to deal with Complexity: Decomposition

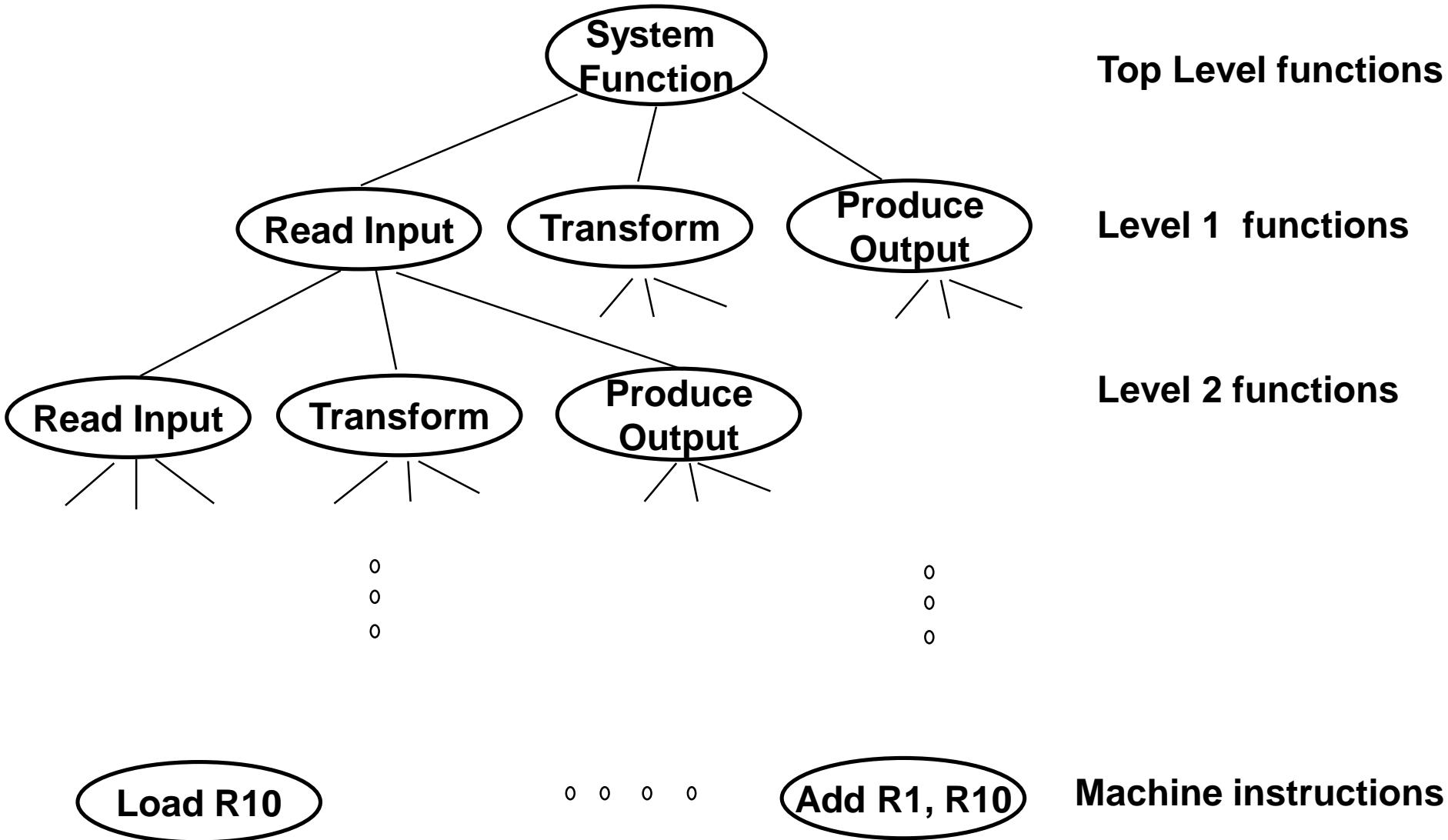
- A technique used to master complexity ("divide and conquer")
- Two major types of decomposition
 - Functional decomposition
 - Object-oriented decomposition
- **Functional decomposition**
 - The system is decomposed into modules/**subsystems**
 - Each module/ **subsystem provides** a major function in the application domain
 - Modules can be decomposed into smaller modules.

Decomposition (cont'd)

- Object-oriented decomposition
 - The system is decomposed into classes ("objects")
 - Each class is a major entity in the application domain
 - Classes can be decomposed into smaller classes
- Object-oriented vs. functional decomposition

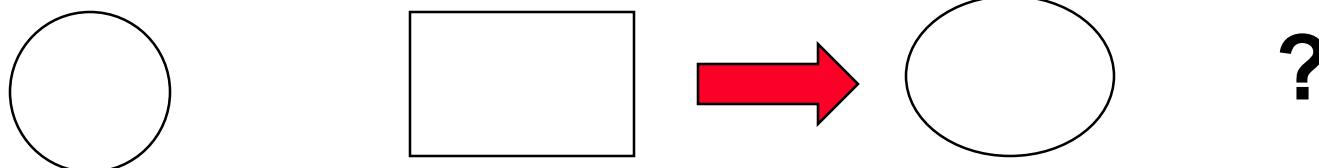
Which decomposition is the right one?

Functional Decomposition

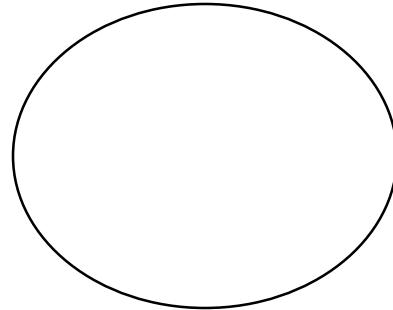


Functional Decomposition

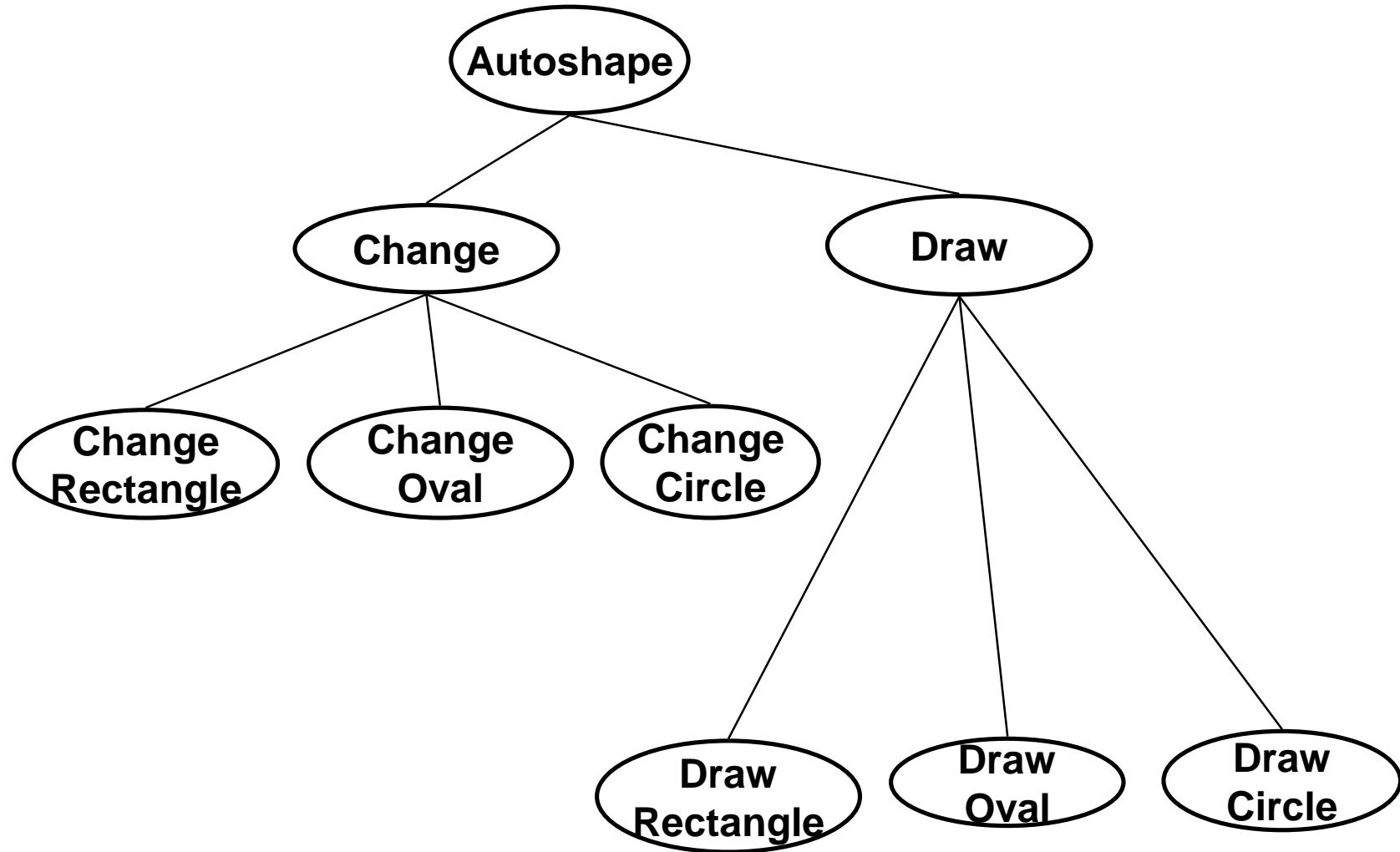
- The functionality is spread all over the system
- Maintainer must understand the whole system to make a single change to the system
- Consequence:
 - Source code is hard to understand
 - Source code is complex and impossible to maintain
 - User interface is often awkward and non-intuitive
- Example: Microsoft Powerpoint's Autoshapes
 - How do I change a square into a circle?



Changing a Square into a Circle



Functional Decomposition: Autoshape



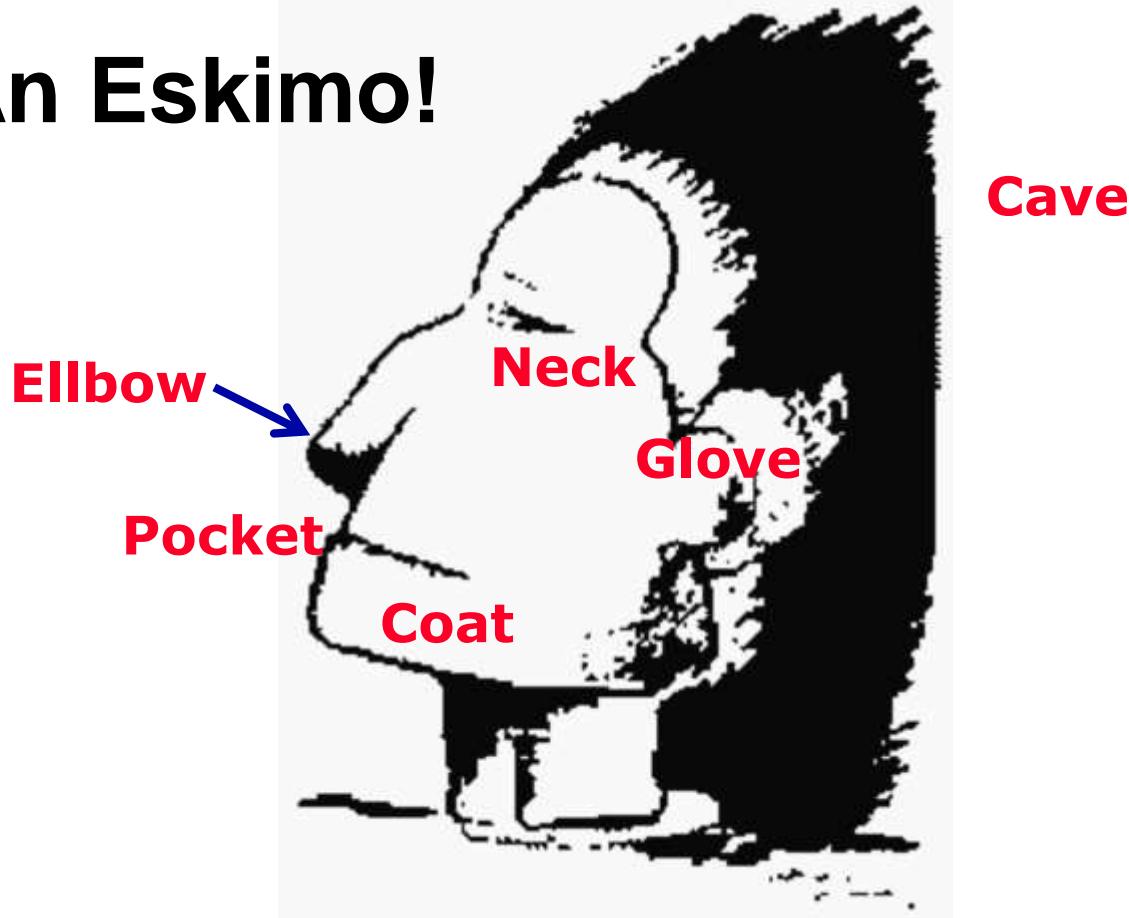
Object-Oriented View

Autoshape

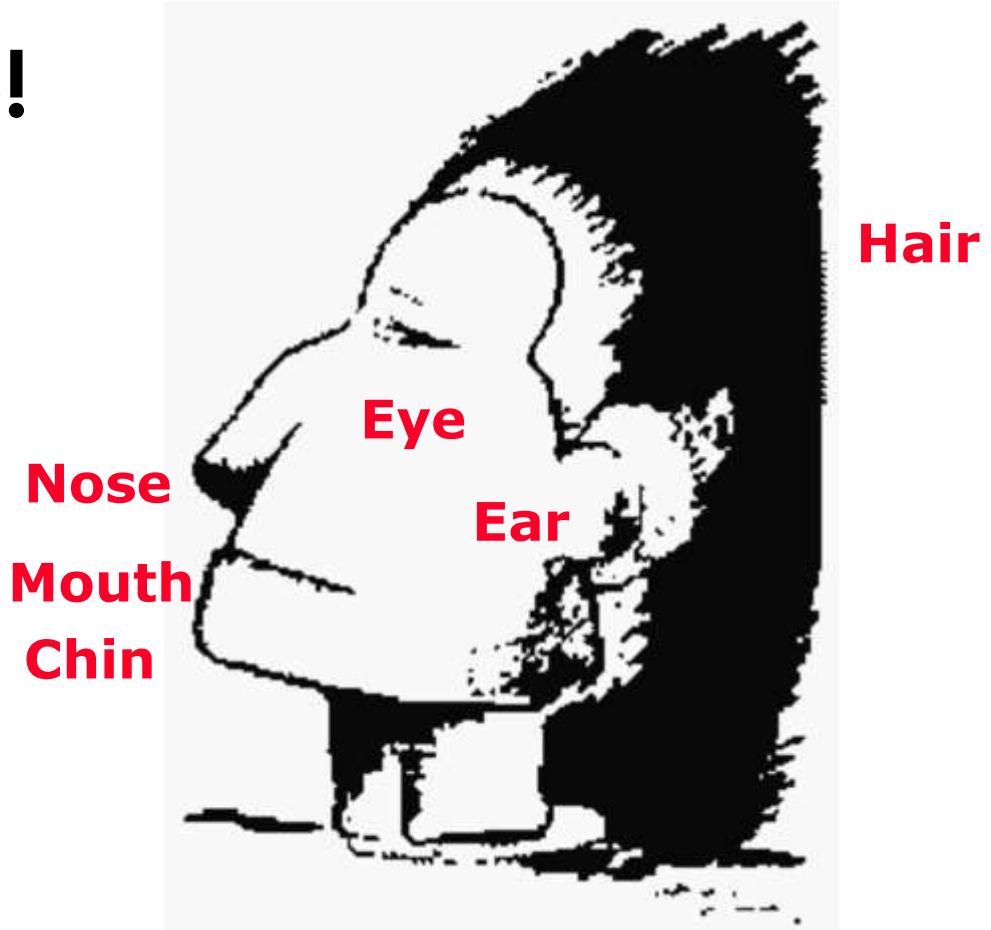
**Draw()
Change()**

What is This?

An Eskimo!

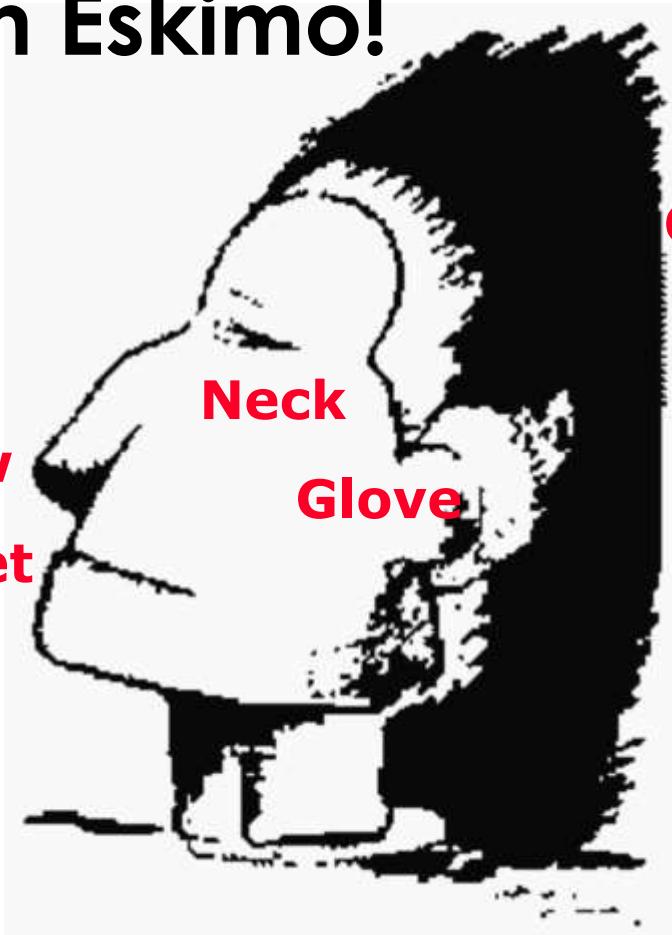


A Face!



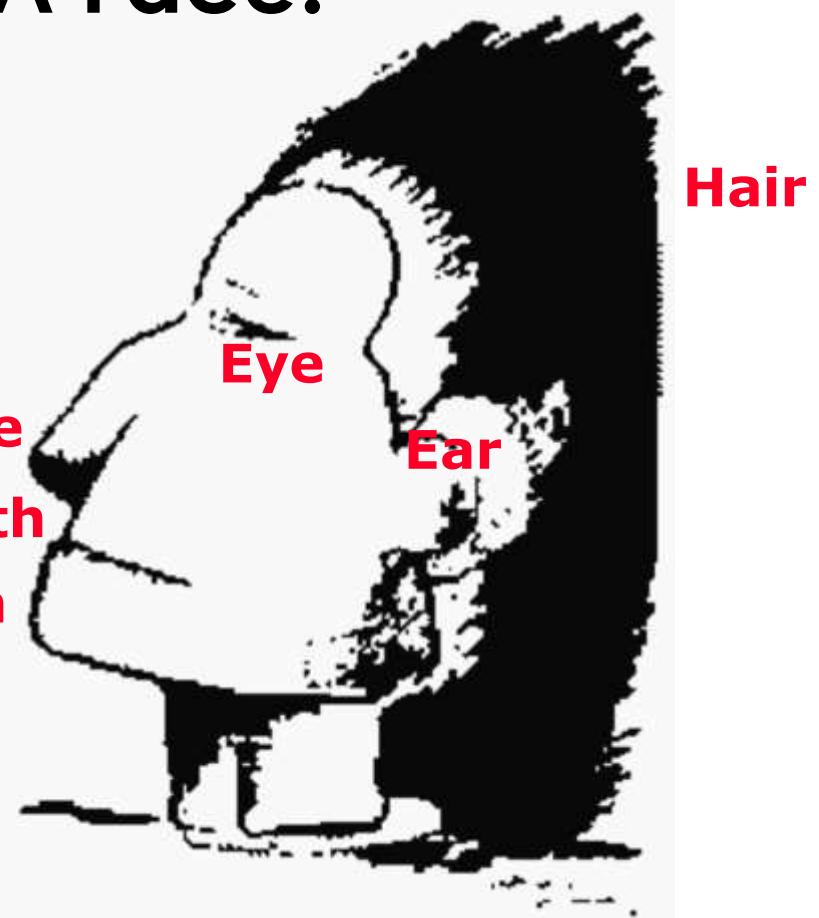
An Eskimo!

Elbow
Pocket
Coat



A Face!

Nose
Mouth
Chin



Class Identification

- **Basic assumptions:**

- We can find the *classes for a new software system*: **Greenfield Engineering**
- We can identify the *classes in an existing system*: **Reengineering**
- We can create a *class-based interface to an existing system*: **Interface Engineering**



Class Identification (cont'd)

- **Why can we do this?**
 - Philosophy, science, experimental evidence
- **What are the limitations?**
 - Depending on the purpose of the system, different objects might be found
- **Crucial**

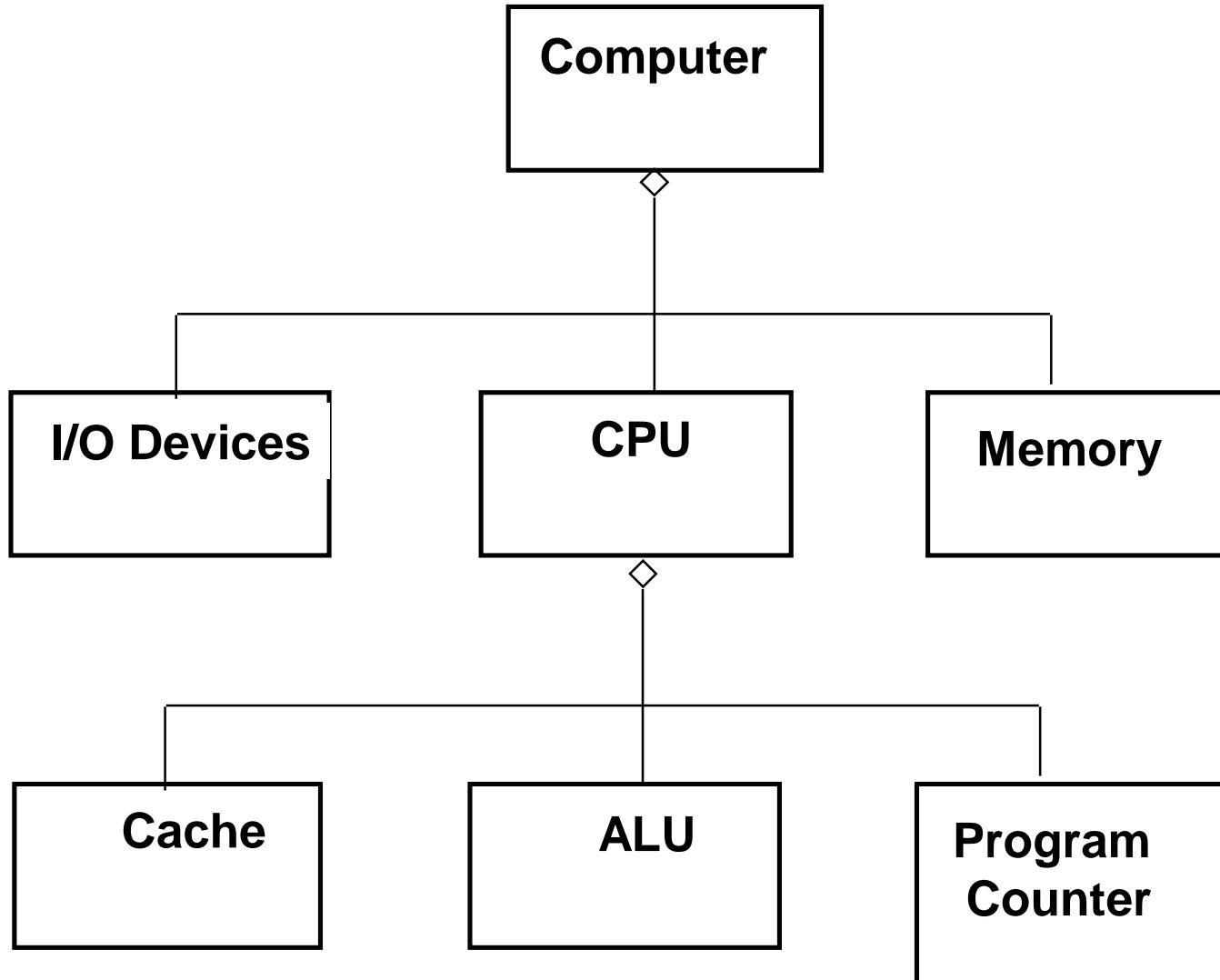
Identify the purpose of a system



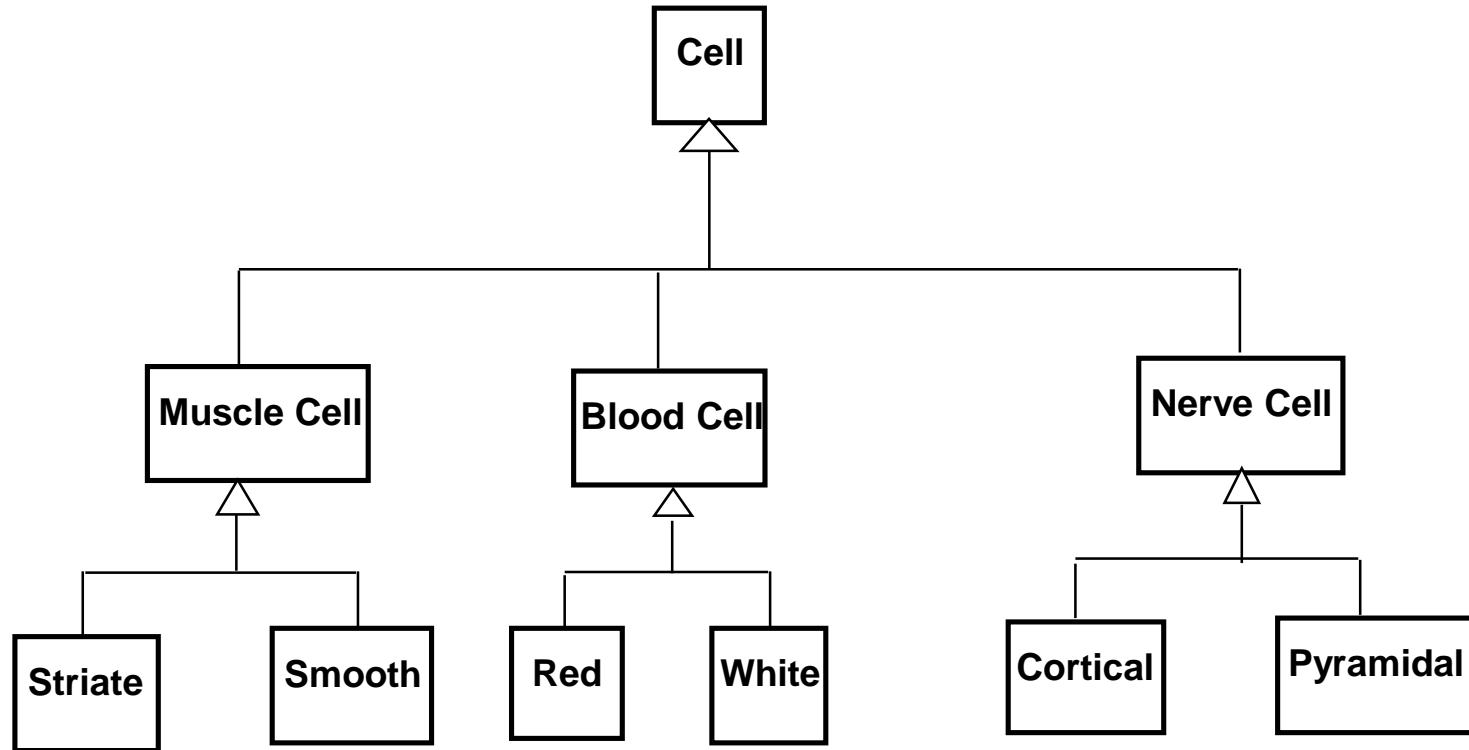
3. Hierarchy

- So far, we got abstractions
 - This leads us to classes and objects
 - “Chunks”
- Another way to deal with complexity is to provide relationships between these chunks
- One of the most important relationships is hierarchy
- 2 special hierarchies
 - "Part-of" hierarchy
 - "Is-kind-of" hierarchy

Part-of Hierarchy (Aggregation)



Is-Kind-of Hierarchy (Taxonomy)



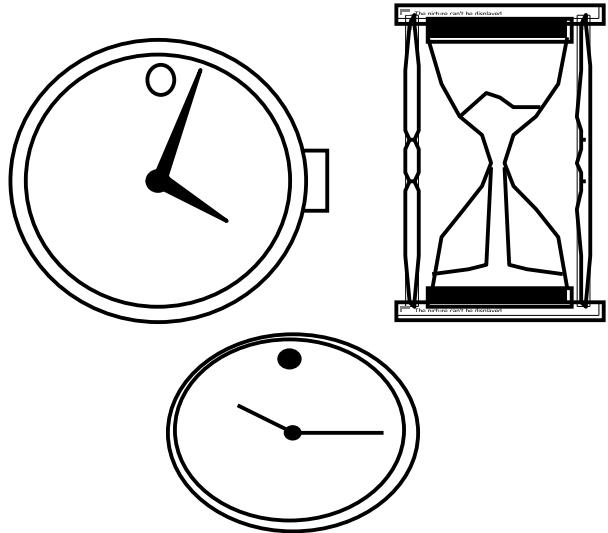
Where are we now?

- Three ways to deal with complexity:
 - Abstraction, Decomposition, Hierarchy
- Object-oriented decomposition is good
 - Unfortunately, depending on the purpose of the system, different objects can be found (an eskimo or a far-west Indian)
- How can we do it right?
 - Start with a description of the functionality of a system
 - Then proceed to a description of its structure
- Ordering of development activities
 - Software lifecycle

Concepts and Phenomena

- **Phenomenon**
 - An object in the world of a domain as you perceive it
 - Examples: This lecture at 9:35, my black watch
- **Concept**
 - The description of the common properties of phenomenon
 - Example: All lectures on software engineering
 - Example: All black watches
- **A Concept is a 3-tuple:**
 - **Name:** The name distinguishes the concept from other concepts
 - **Purpose:** Properties that determine if a phenomenon is a member of a concept
 - **Members:** The set of phenomena which are part of the concept.

Concepts, Phenomena, Abstraction and Modeling

Name	Purpose	Members
Watch	A device that measures time.	 A collection of three line-art illustrations. At the top left is a round wristwatch with a small second hand at the top. To its right is a simple clock face with two hands. To the right of the clock is an hourglass with sand falling from the top bulb into the bottom one. Above the hourglass is a small horizontal bar with the text "Time cannot be measured". Below the hourglass is another small horizontal bar with the text "Time is only measured".

Definition Abstraction:

- Classification of phenomena into concepts

Definition Modeling:

- Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Abstract Data Types & Classes

- **Abstract data type**

- A type whose implementation is hidden from the rest of the system

- **Class:**

- An abstraction in the context of object-oriented languages
- A class encapsulates state **structure** and behavior **specification**
 - Example: Watch

Inheritance

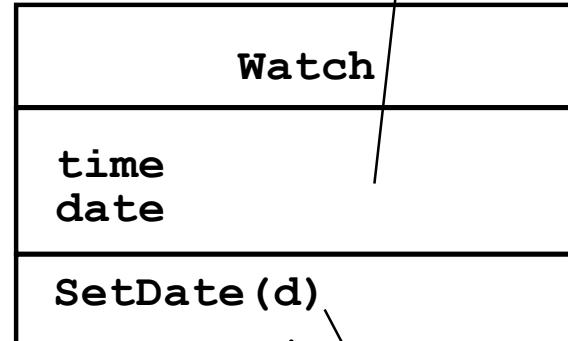
Unlike abstract data types, subclasses can be defined in terms of other classes using inheritance

- Example: CalculatorWatch

Subclass/Child

Superclass/Parent

State/Class Attributes



Watch

time
date

SetDate (d)

Behavior

CalculatorWatch

calculatorState

EnterCalcMode ()
InputNumber (n)

Type and Instance

- **Type:**
 - A concept in the context of programming languages
 - Name: int
 - Purpose: integral number
 - Members: 0, -1, 1, 2, -2, ...
- **Instance:**
 - Member of a specific type
- The type of a variable represents all possible instances of the variable

The following relationships are **similar/related**:

Type <-> Variable

Concept <-> Phenomenon

Class <-> Object

Systems

- A *system* is an organized set of communicating parts
 - **Natural system:** A system whose ultimate purpose is not known
 - **Engineered system:** A system which is designed and built by engineers for a specific purpose
- The parts of the system can be considered as systems again
 - In this case we call them *subsystems*

Examples of natural systems:

- Universe, earth, ocean

Examples of engineered systems:

- Airplane, watch, GPS

Examples of subsystems:

- Jet engine, battery, satellite.

Systems, Models and Views

- A **model** is an abstraction describing a system or a subsystem
- A **view** depicts selected aspects of a model
- A **notation** is a set of graphical or textual rules for depicting models and views:
 - formal notations, “napkin designs”

System: Airplane

Models:

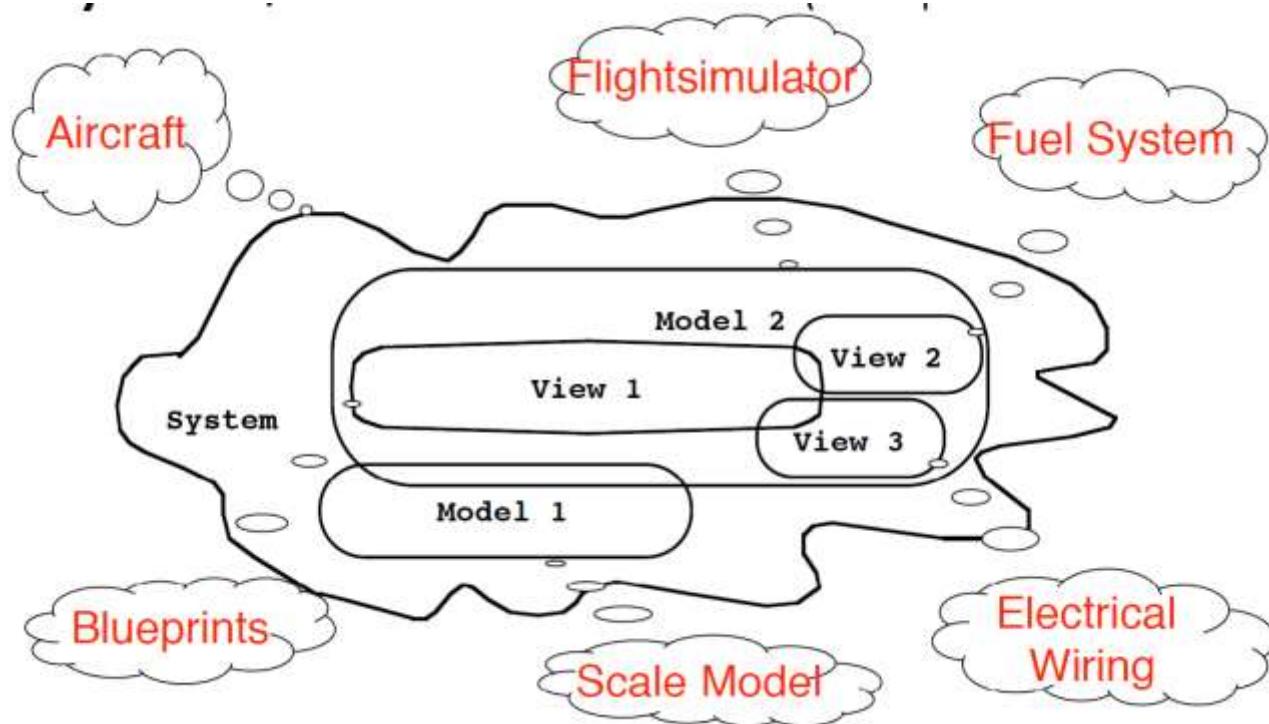
Flight simulator
Scale model

Views:

Blueprint of the airplane components
Electrical wiring diagram, Fuel system
Sound wave created by airplane



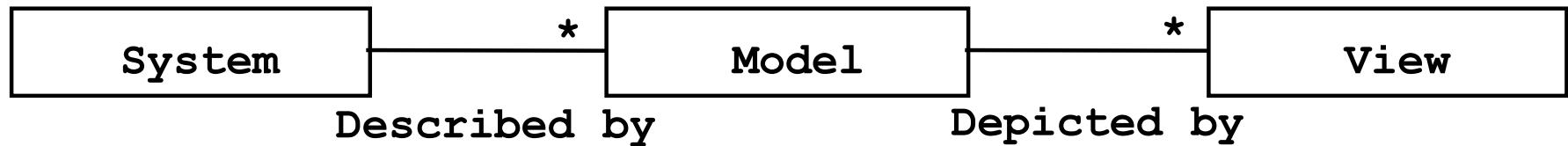
Systems, Models and Views (“Napkin” Notation)



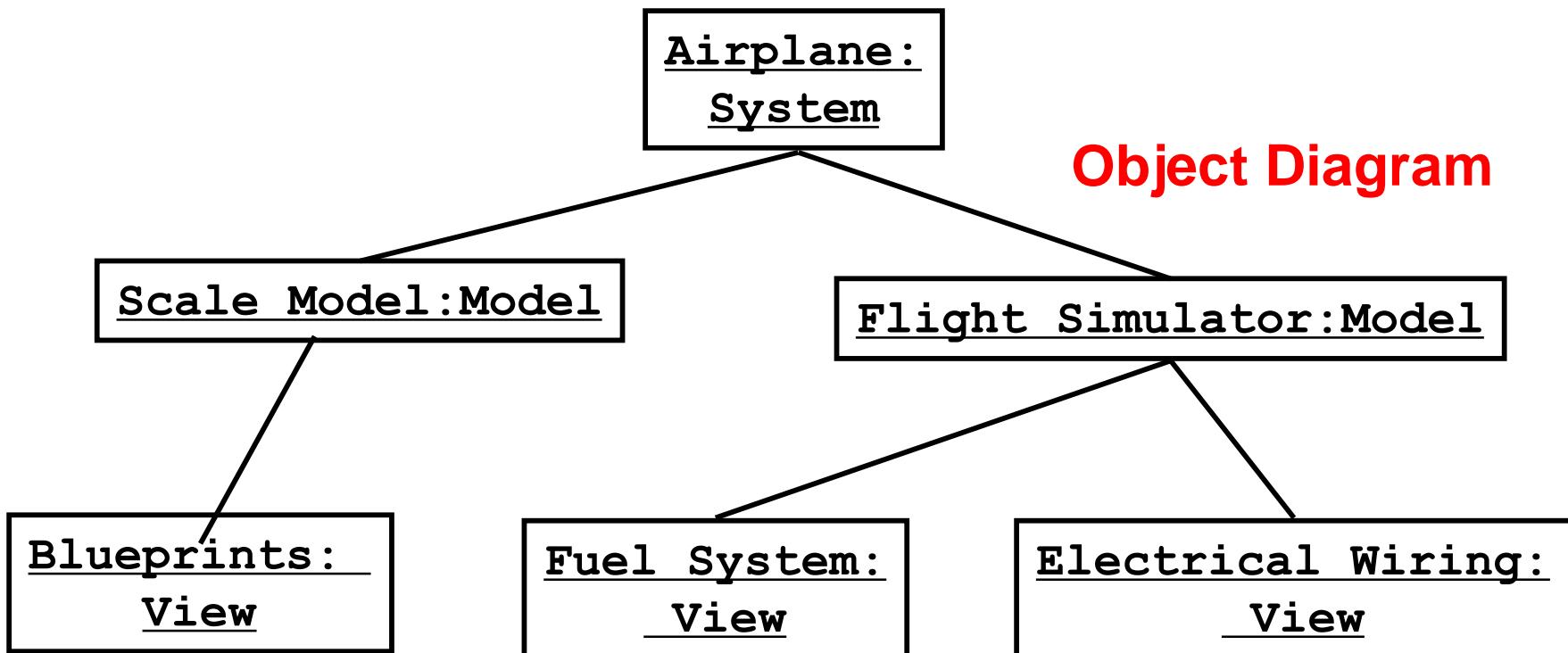
Views and models of a complex system usually overlap

Systems, Models and Views (UML Notation)

Class Diagram



Object Diagram



Model-Driven Development

1. Build a platform-independent model of an applications functionality and behavior
 - a) Describe model in modeling notation (UML)
 - b) Convert model into platform-specific model
2. Generate executable from platform-specific model

Advantages:

- Code is generated from model ("mostly")
- Portability and interoperability
- Model Driven Architecture effort:
 - <http://www.omg.org/mda/>
- OMG: Object Management Group

Model-driven Software Development

Reality: A stock exchange lists many companies. Each company is identified by a ticker symbol

Analysis results in analysis object model (UML Class Diagram):



Implementation results in source code (Java):

```
public class StockExchange {  
    public m_Company = new Vector();  
};  
public class Company {  
    public int m_tickerSymbol;  
    public Vector m_StockExchange = new Vector();  
};
```

OCLE Code Sample for StockExchange model

```
* Velocity Template Engine 1.3rc1</a>
*/
import java.util.LinkedHashSet;
import java.util.Set;
import ro.ubbcluj.lci.codegen.framework.ocl.BasicConstraintChecker;

/**
 *
 * @author unascribed
 */
public class Company {

    public final Set getStockExchanges() {
        if (stockExchanges == null) {
            return java.util.Collections.EMPTY_SET;
        }
        return java.util.Collections.unmodifiableSet(stockExchanges);
    }

    public final void addStockExchanges(StockExchange arg) {
        if (arg != null) {
            if (stockExchanges == null) stockExchanges = new LinkedHashSet();
            if (stockExchanges.add(arg)) {
                arg.addCompanies(this);
            }
        }
    }

    public final void removeStockExchanges(StockExchange arg) {
        if (stockExchanges != null && arg != null) {
            if (stockExchanges.remove(arg)) {
                arg.removeCompanies(this);
            }
        }
    }
}
```

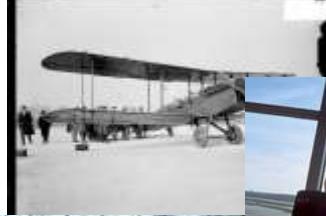
OCL Code Sample for StockExchange model

```
public Company() {  
}  
  
public class ConstraintChecker extends BasicConstraintChecker {  
  
    public void checkConstraints() {  
  
        super.checkConstraints();  
        check_Company_test();  
  
    }  
  
    public void check_Company_test() {  
  
        int nTickerSymbol = Company.this.tickerSymbol;  
        boolean bGreater = nTickerSymbol > 0;  
        if (!bGreater) {  
            System.err.println("invariant 'test' failed for object "+Company.this);  
        }  
  
    }  
}  
  
public int tickerSymbol;  
  
public Set stockExchanges;  
}
```

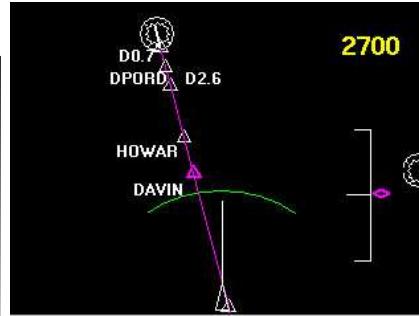
Application vs Solution Domain

- Application Domain (Analysis):
 - The environment in which the system is operating
- Solution Domain (Design, Implementation):
 - The technologies used to build the system
- Both domains contain abstractions that we can use for the construction of the system model.

Object-oriented Modeling

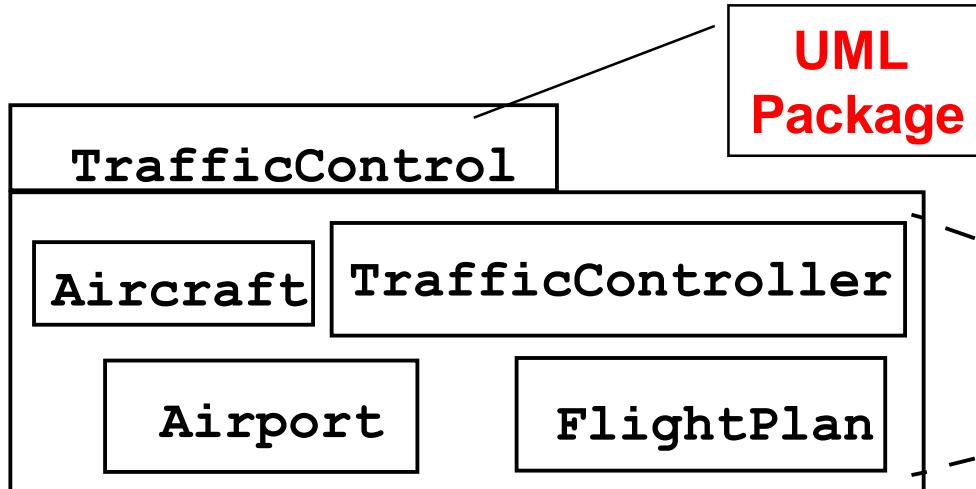


Application Domain
(Phenomena)

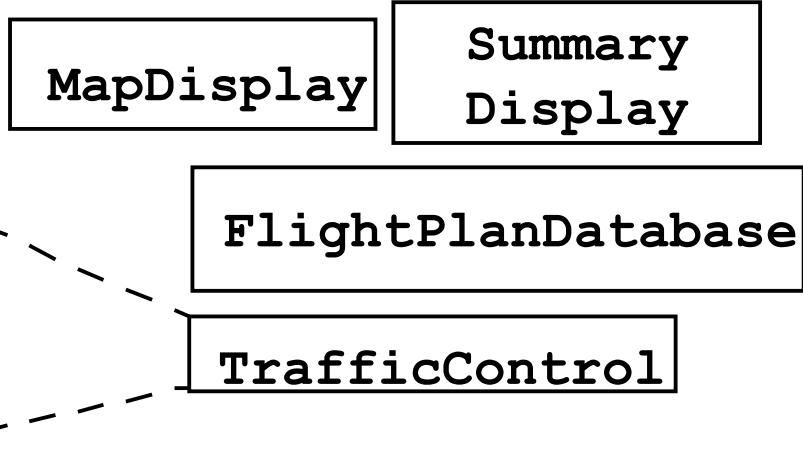


Solution Domain
(Phenomena)

System Model (Concepts)*(Analysis)*



System Model (Concepts)*(Design)*



What is UML?

- UML (Unified Modeling Language)
 - Nonproprietary standard for modeling software systems, OMG
 - Convergence of notations used in object-oriented methods
 - OMT (James Rumbaugh and colleagues)
 - Booch (Grady Booch)
 - OOSE (Ivar Jacobson)
- Current Version: UML 2.5.1 -
<https://www.omg.org/spec/UML/2.5.1>
 - Information at the OMG portal <http://www.uml.org/>

UML tools

- **IBM Rational Software Architect Designer** - This trial is fully functional. Download the trial for an **evaluation period of 60 days**. (See notes for extensions and limitations.)
https://www.ibm.com/developerworks/downloads/r/architect/index.html?mhsrc=ibmsearch_a&mhq=ibm%20rational%20architect
- **ENTERPRISE ARCHITECT** - Trial Free For 30 Days!
<https://sparxsystems.com/products/ea/>
- **MagicDraw** - <https://www.nomagic.com/products/magicdraw> offer trial version
- **Visual Paradigm** - <https://www.visual-paradigm.com/> - No registration. 30-day FREE Trial
- **UMLe** - <https://cruise.umle.org/umle/>

UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We teach you those 20% - for using UML to support communication and understanding
- 80-20 rule: Pareto principle
 - http://www.ephorie.de/hindle_pareto-prinzip.htm

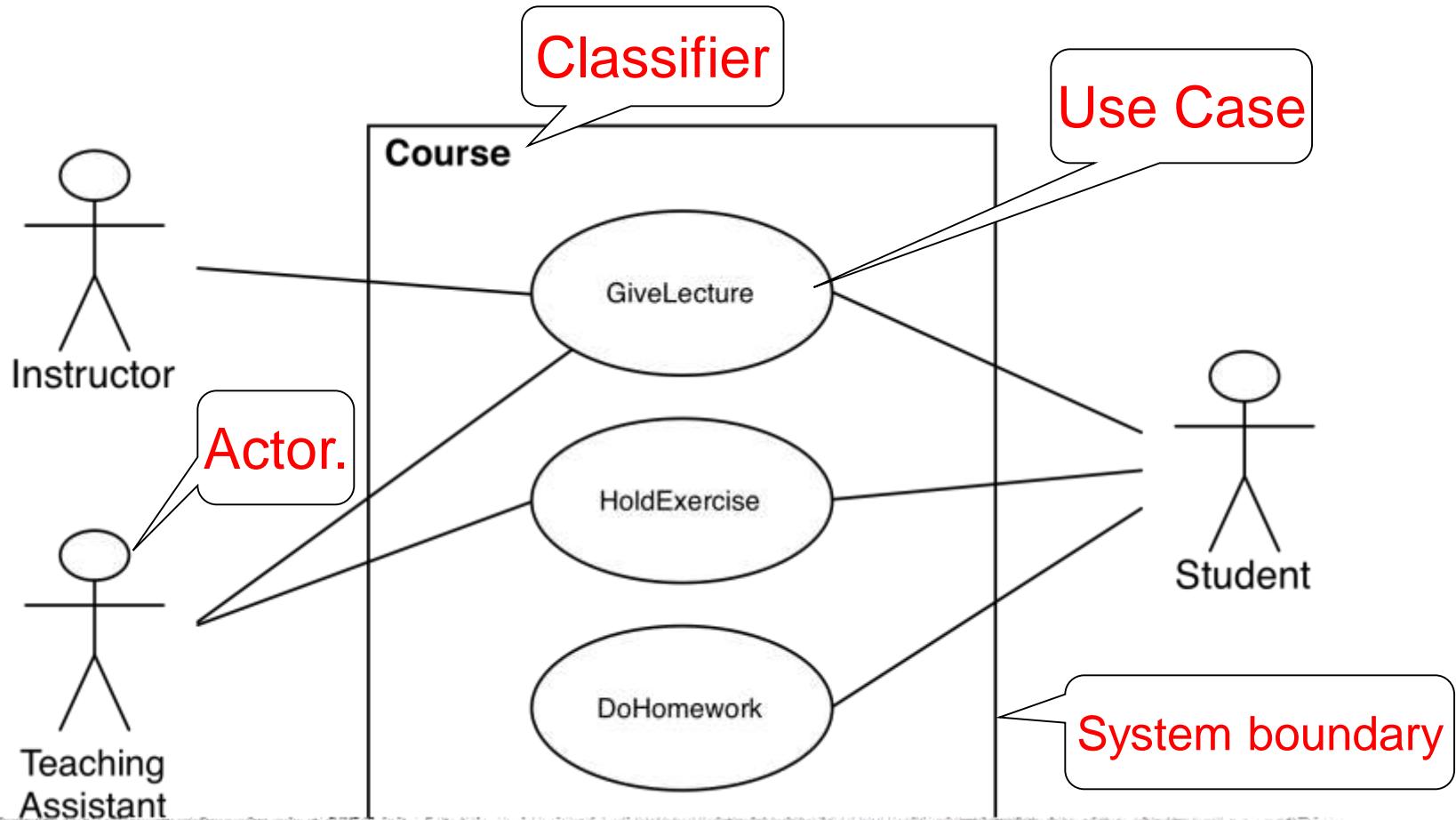
UML First Pass

- Use case diagrams
 - Describe the functional behavior of the system as seen by the user
- Class diagrams
 - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
 - Describe the dynamic behavior between objects of the system
- Statechart diagrams
 - Describe the dynamic behavior of an individual object
- Activity diagrams
 - Describe the dynamic behavior of a system, in particular the workflow.

UML Core Conventions

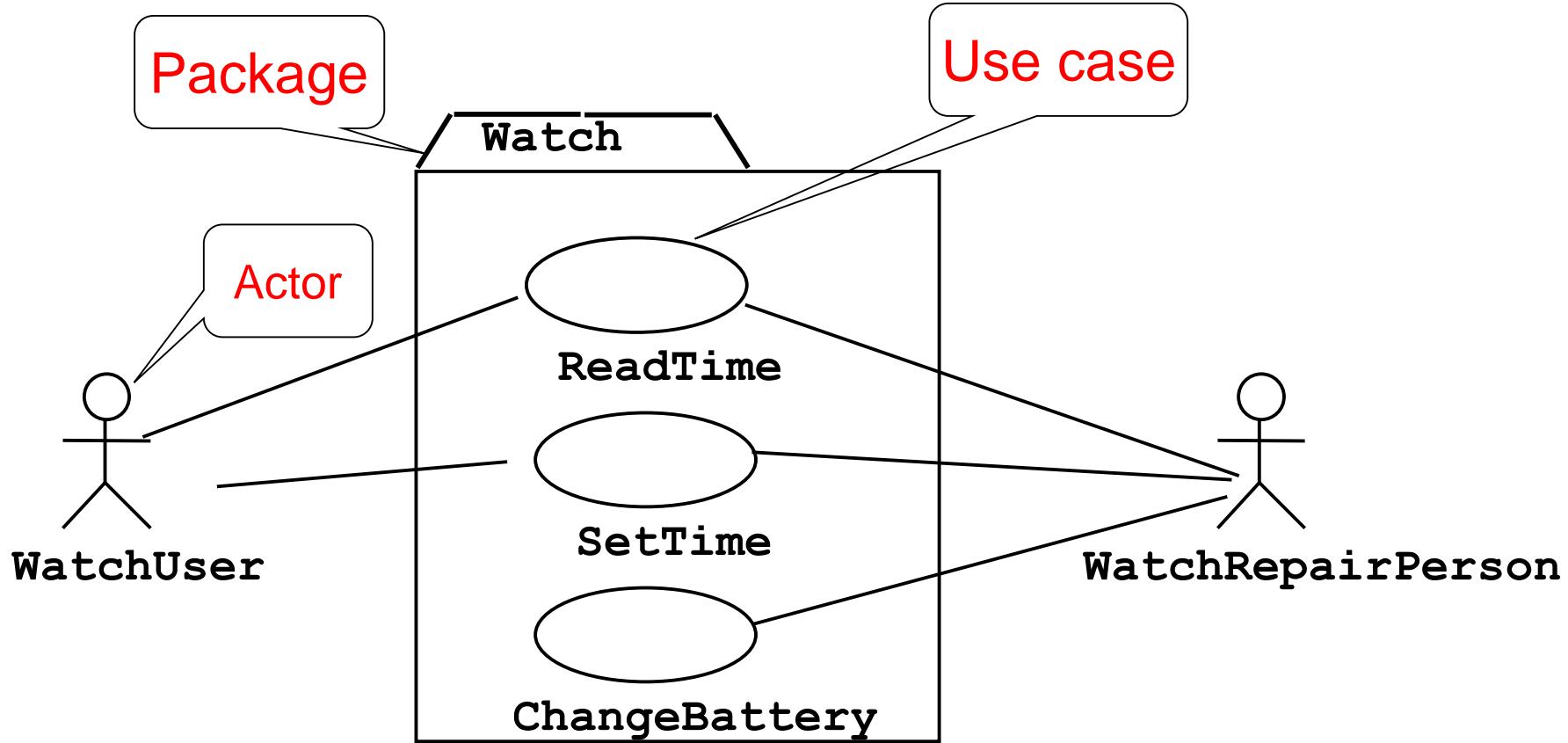
- All UML Diagrams denote graphs of nodes and edges
 - Nodes are entities and drawn as rectangles or ovals
 - Rectangles denote classes or instances
 - Ovals denote functions
- Names of Classes are not underlined
 - SimpleWatch
 - Firefighter
- Names of Instances are **underlined**
 - myWatch:SimpleWatch
 - Joe:Firefighter
- An edge between two nodes denotes a relationship between the corresponding entities

UML first pass: Use case diagrams



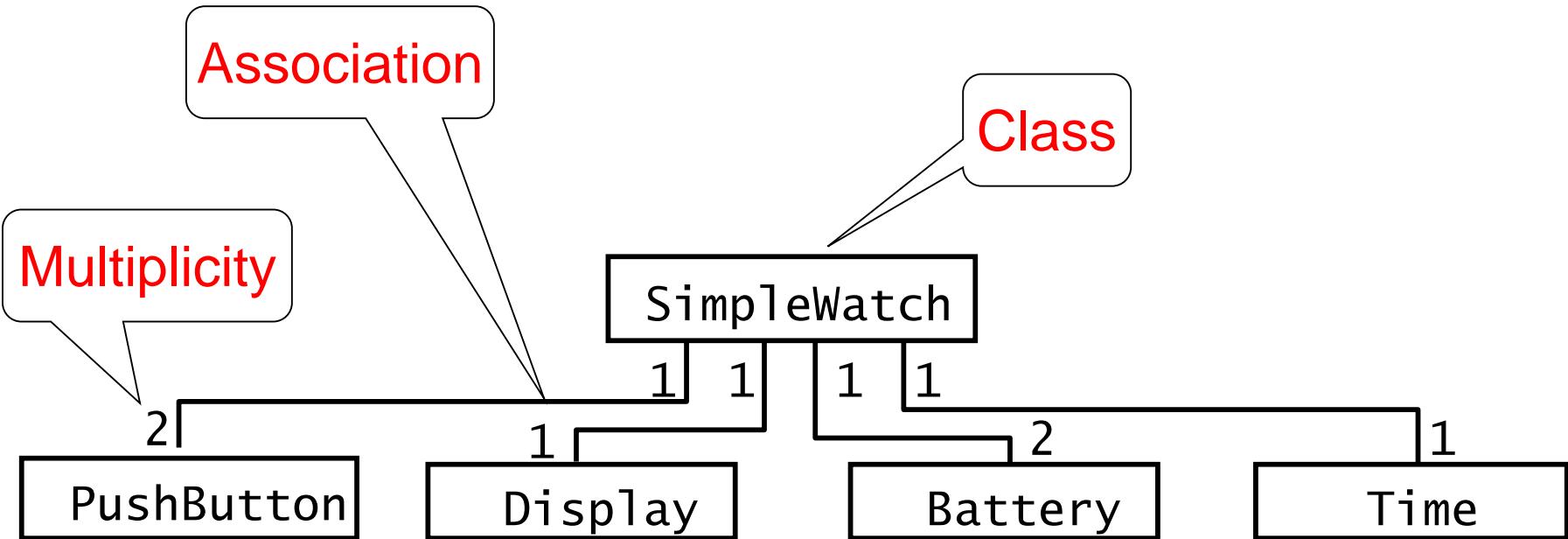
Use case diagrams represent the functionality of the system from user's point of view

Historical Remark: UML 1 used packages



Use case diagrams represent the functionality of the system from user's point of view

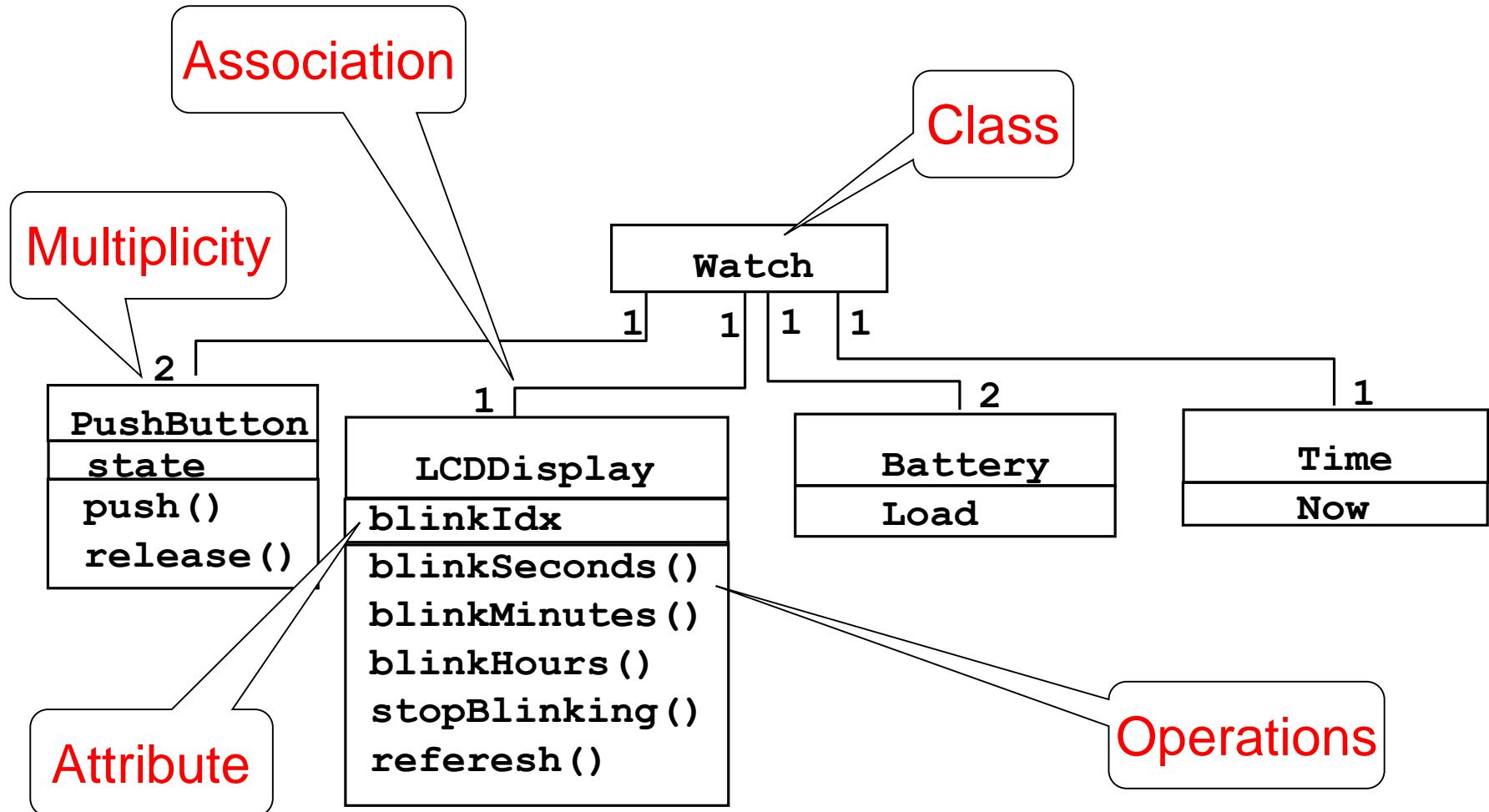
UML first pass: Class diagrams



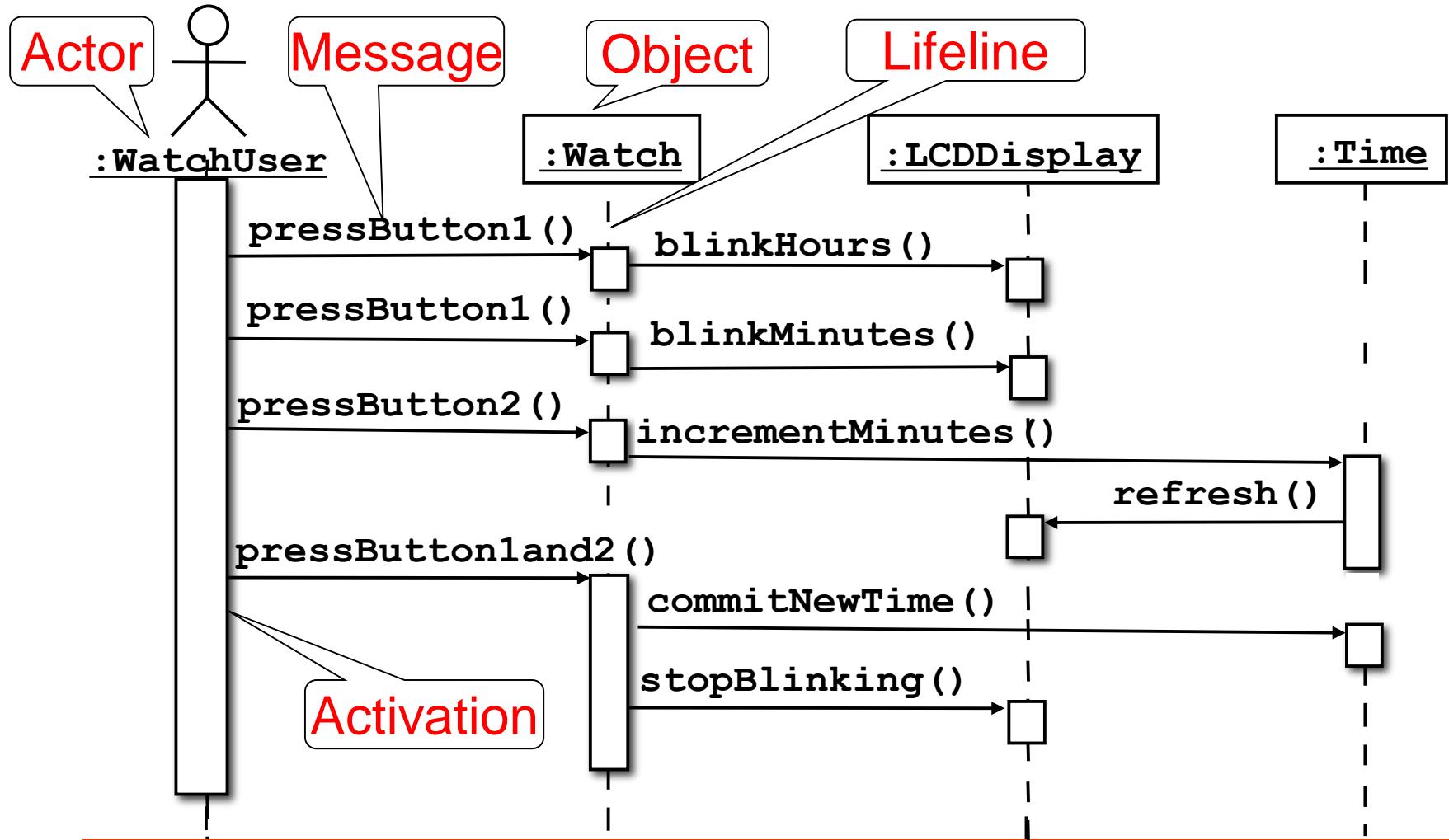
Class diagrams represent the structure of the system

UML first pass: Class diagrams

Class diagrams represent the structure of the system

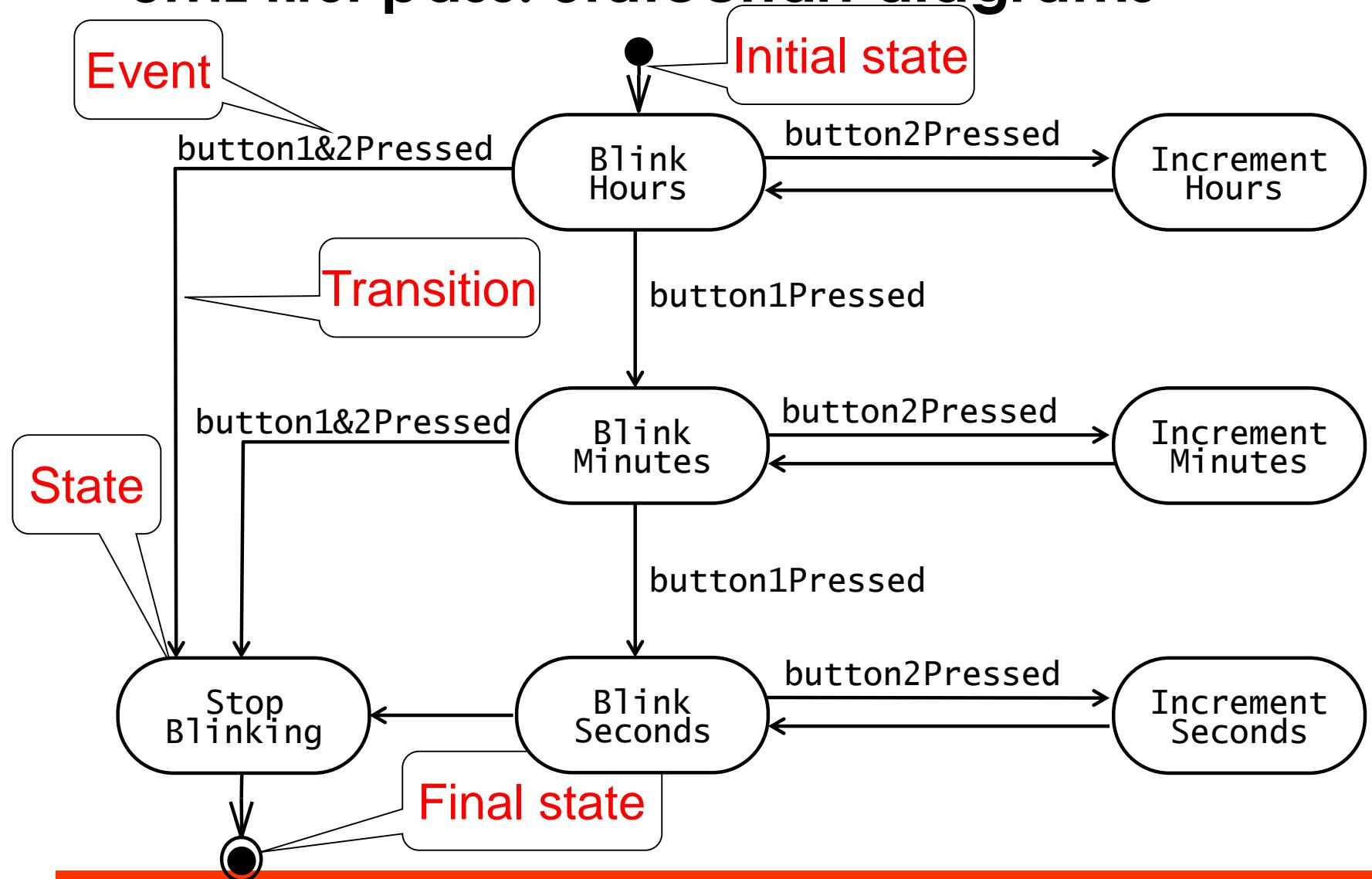


UML first pass: Sequence diagram



Sequence diagrams represent the behavior of a system as messages (“interactions”) between *different objects*

UML first pass: Statechart diagrams



Represent behavior of a *single object* with interesting dynamic behavior.

Other UML Notations

UML provides many other notations, for example

- Deployment diagrams for modeling configurations
 - Useful for testing and for release management
- We introduce these and other notations as we go along in the lectures
 - OCL: A language for constraining UML models

What should be done first? Coding or Modeling?

- It all depends....
- **Forward Engineering**
 - Creation of code from a model
 - Start with modeling
 - Greenfield projects
- **Reverse Engineering**
 - Creation of a model from existing code
 - Interface or reengineering projects
- **Roundtrip Engineering**
 - Move constantly between forward and reverse engineering
 - Reengineering projects
 - Useful when requirements, technology and schedule are changing frequently.

UML Basic Notation Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- Today we concentrated on a few notations:
 - Functional model: Use case diagram
 - Object model: Class diagram
 - Dynamic model: Sequence diagrams, statechart

Additional References

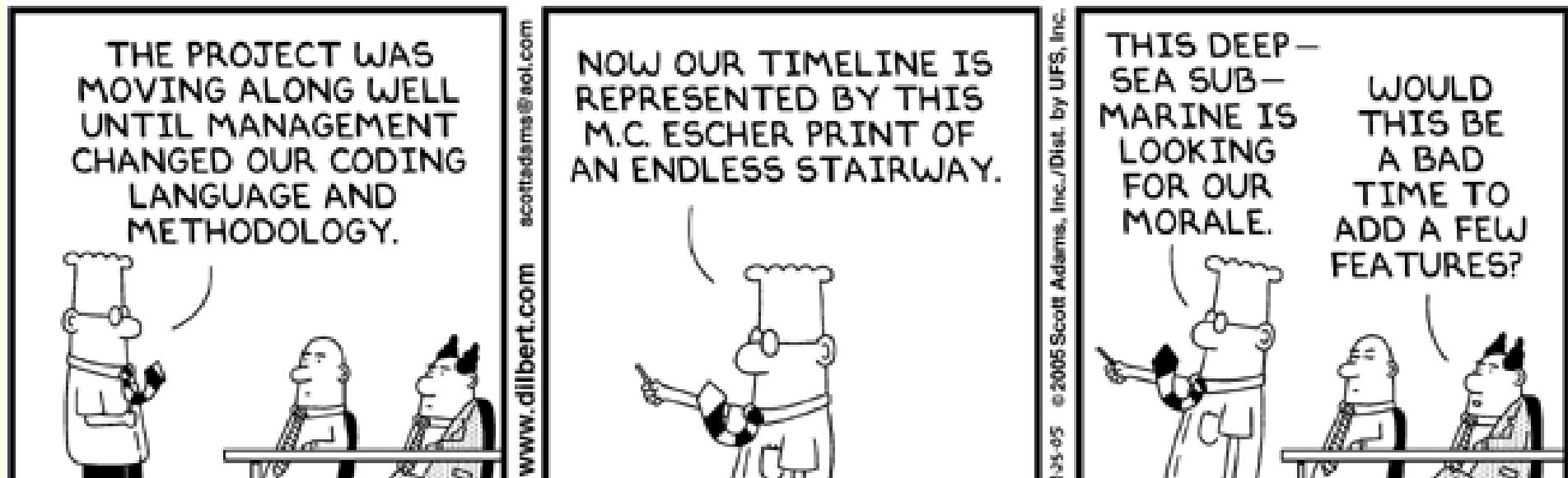
- Martin Fowler
 - **UML Distilled: A Brief Guide to the Standard Object Modeling Language**, 3rd ed., Addison-Wesley, 2003
- Grady Booch, James Rumbaugh, Ivar Jacobson
 - **The Unified Modeling Language User Guide**, Addison Wesley, 2nd edition, 2005
 - **The Unified Modeling Language Reference Manual**, 2nd Edition, Addison Wesley, 2004
 - **The Unified Software Development Process**, Addison-Wesley 1st Edition, 1999
- Martina Seidl, Marion Scholz, Christian Huemer, Gerti Kappel
UML @ Classroom - An Introduction to Object-Oriented Modeling, Springer 2014

UML / UML 2.0 tutorial

Ileana Ober
IRIT – UPS, Toulouse, France
<http://www.irit.fr/~Ileana.Ober>

Modeling in the '80 – '90s

- Lots of (slightly different) languages and design techniques
 - OMT
 - Coad & Yourdon
 - BON
 - SDL
 - ROOM
 - Shlaer Mellor
- ... Quite a mess



© Scott Adams, Inc./Dist. by UFS, Inc.

In use with permission from PIB Copenhagen A/S, obtained august 2005

UML

- Sought as a solution to the OOA&D mess
 - Aims at
 - Unifying design languages
 - Being a general purpose modeling language
- Lingua franca of modeling

Overview

- What is UML?
- Structure description
- Behavior description
- OCL
- UML and tools

Overview

- What is UML?
- Structure description
- Behavior description
- OCL
- UML and tools

UML (Unified Modeling Language)

- Goal: lingua franca in modeling
- Definition driven by **consensus** rather than **innovation**
- Standardized by the OMG
- Definition style:
 - Described by a **meta-model** (abstract syntax)
 - **Well formedness rules** in OCL
 - **Textual description**
 - static and dynamic semantics
(in part already described by WFRs)
 - notation description
 - usage notes

Overview of the 13 diagrams of UML

Structure diagrams

1. Class diagram
2. Composite structure diagram (*)
3. Component diagram
4. Deployment diagram
5. Object diagram
6. Package diagram

Behavior diagrams

7. Use-case diagram
8. State machine diagram
9. Activity diagram

Interaction diagrams

10. Sequence diagram
11. Communication diagram
12. Interaction overview diagram (*)
13. Timing diagram (*)

(*) not existing in UML 1.x, added in UML 2.0

UML principle: diagram vs. model

- Different diagrams describe various **facets** of the model
- Several diagrams of the same kind may coexist
- Each diagram shows a *projection* of the model
- *Incoherence* between diagrams (of the same or of different kind(s)) correspond to an *ill-formed* model
- The **coherence rules** between different kinds of diagrams is **not fully stated**

This tutorial looks closer at ...

- Use case diagram
- Class diagram
- Composite structure diagram
- Component/deployment diagram
- State machine diagram
- Activity diagram
- Interaction diagrams

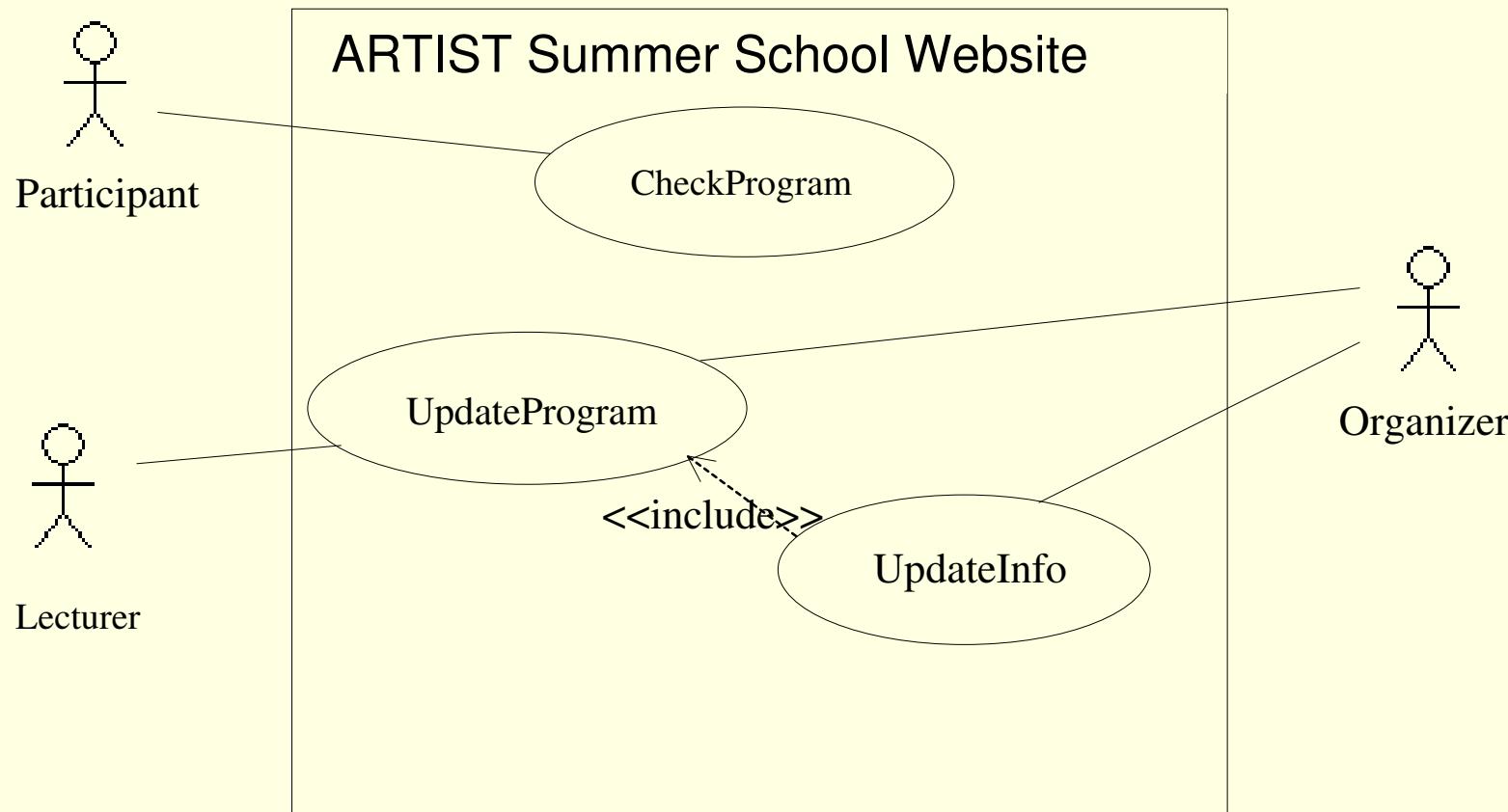
Overview

- What is UML?
- Structure description
 - Use case diagram
 - Class diagram
 - Composite structure diagram
 - Communication principles in UML
 - System initialization
- Behavior description
- OCL
- UML and tools

Use case diagram

- Displays the relationship among *actors* and *use cases*, in a given *system*
- Main concepts:
 - System – the system under modeling
 - Actor – external “user” of the system
 - Use case – execution scenario, observable by an actor

Use case diagram example



Use case diagram – final remarks

- Widely used in real-life projects
- Used at:
 - Exposing requirements
 - Communicate with clients
 - Planning the project
- Additional textual notes are often used/required
- User-centric, non formal notation
- Few constraints in the standard

Further reading:

- D. Rosenberg, K.Scott Use Case Driven Object Modeling with UML : A Practical Approach, Addison-Wesley Object Technology Series, 1999
- I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Professional, 1999
- Writing Effective Use Cases Alistair Cockburn Addison-Wesley Object Technology Series, 2001

Overview

- What is UML?
- Structure description
 - Use case diagram
 - Class diagram
 - Composite structure diagram
 - Communication principles in UML
 - System initialization
- Behavior description
- OCL
- UML and tools

Class diagram

- The *most known* and the *most used* UML diagram
- Gives information on model's **structural elements**
- Main concepts involved
 - Class - Object
 - Inheritance
 - Association (various kinds of)

Let's start with ... object orientation

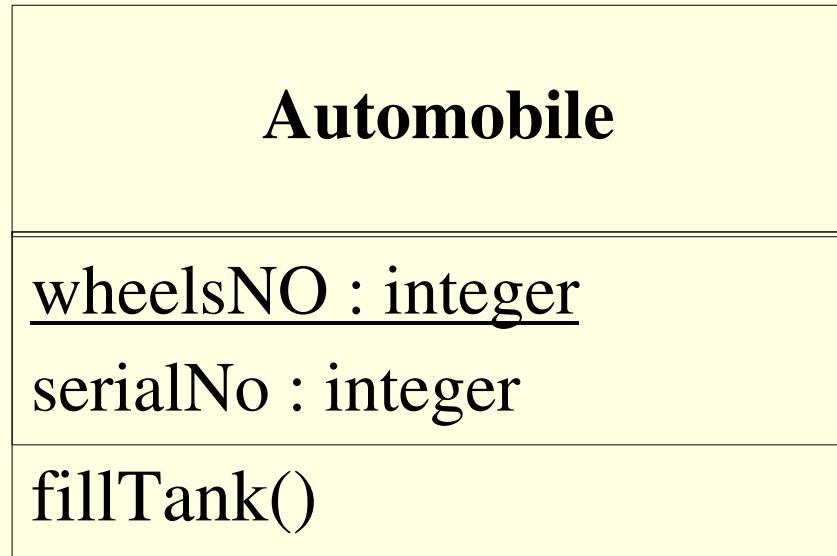
- Why OO?
 - In the first versions, UML was described as addressing the needs of modeling systems in a OO manner
 - Statement not any longer maintained, however the OO inspiration for some key concepts is still there
- Main concepts:
 - **Object** – individual unit capable of *receiving/sending messages*, processing data
 - **Class** – pattern giving an abstraction for a set of objects
 - **Inheritance** – technique for reusability and extendibility

Further reading:

Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 2000

UML Class

- Gives the **type** of a **set of objects** existing at run-time
- Declares a **collection of methods and attributes** that describe the structure and behavior of its objects
- Basic notation:



Properties of UML classes

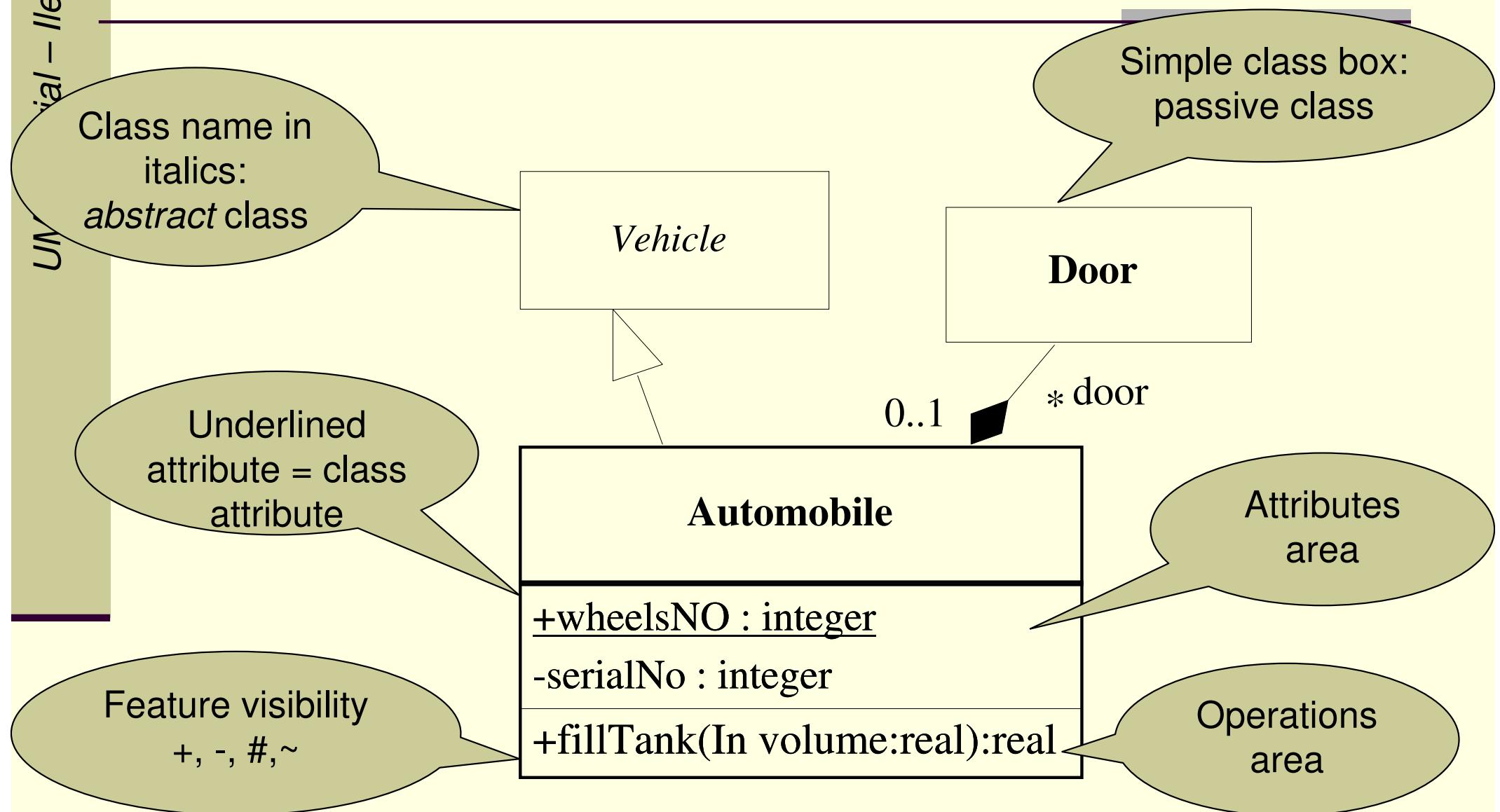
- May own *features*
 - *Structural* (data related) : attributes
 - *Behavioral* : operations
- May own *behavior* (state machines, interactions, ...)
- May be instantiated
 - except for **abstract classes** that *can NOT be directly instantiated* and exist only for the inheritance hierarchy

Class features – characterized by

- Signature
- Visibility (public, private, protected, package)
- Changeability (changeable, frozen, addOnly)
- Owner scope (class, instance) – equivalent to `static` clause in programming languages
- Invariant constraint

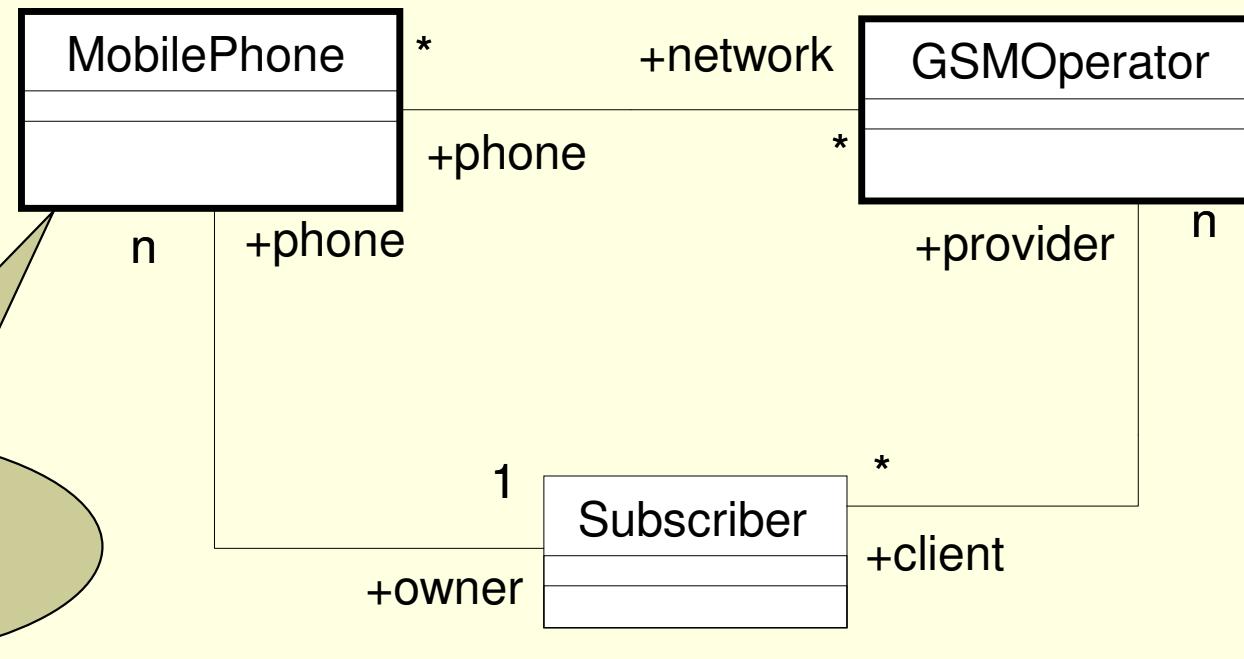
- Additionally, operations are characterized by
 - *concurrency kind*: sequential, guarded, *concurrent*
 - *pre* or *post* conditions
 - body (state machine or action description)

Decode class symbol adornments



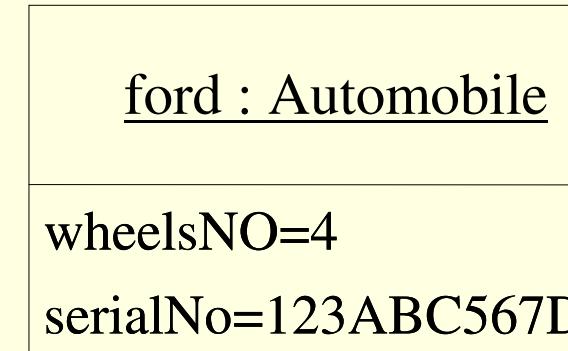
Active / passive classes

- specifies the *concurrency model* for classes
- specifies whether an **Object** of the **Class** maintains its own thread of control and runs concurrently with other active **Objects** (active)



Object

- Instance of a class
- Can be shown in a class diagram
- Notation



Inheritance

- A.k.a. generalization (specialization)
- Applies mainly on classes
- Other UML model elements can be subject to inheritance (e.g. interface)
(if you want the exact list go check the UML metamodel for kinds of *GeneralizableElements*)
- Allows for polymorphism

Inheritance/polymorphism example

Animal a;

Cow cw;

Cat ct;

.....

if (<condition>)

 a:= cw

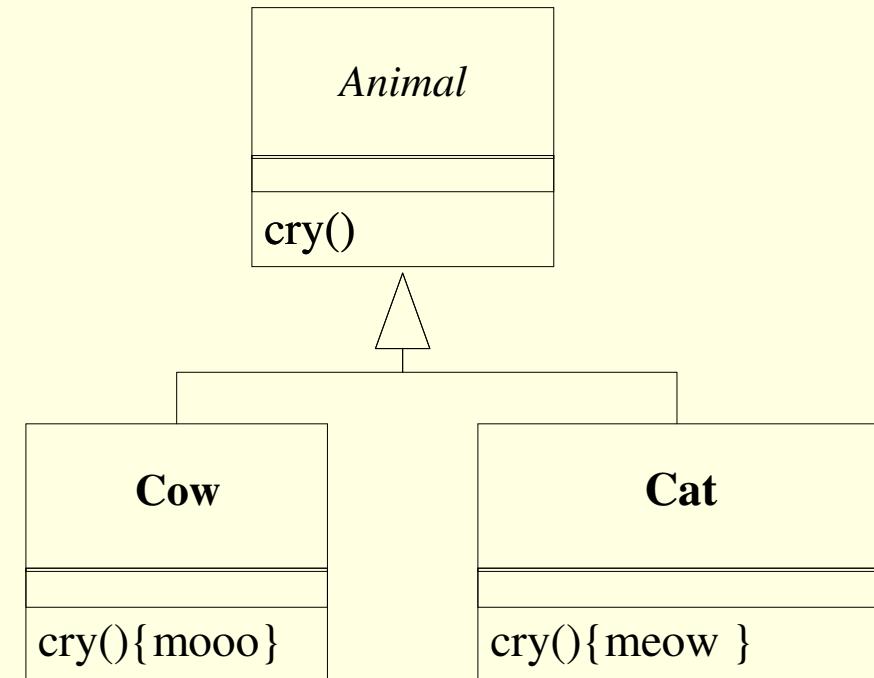
else

 a:= ct

endif

a.cry()

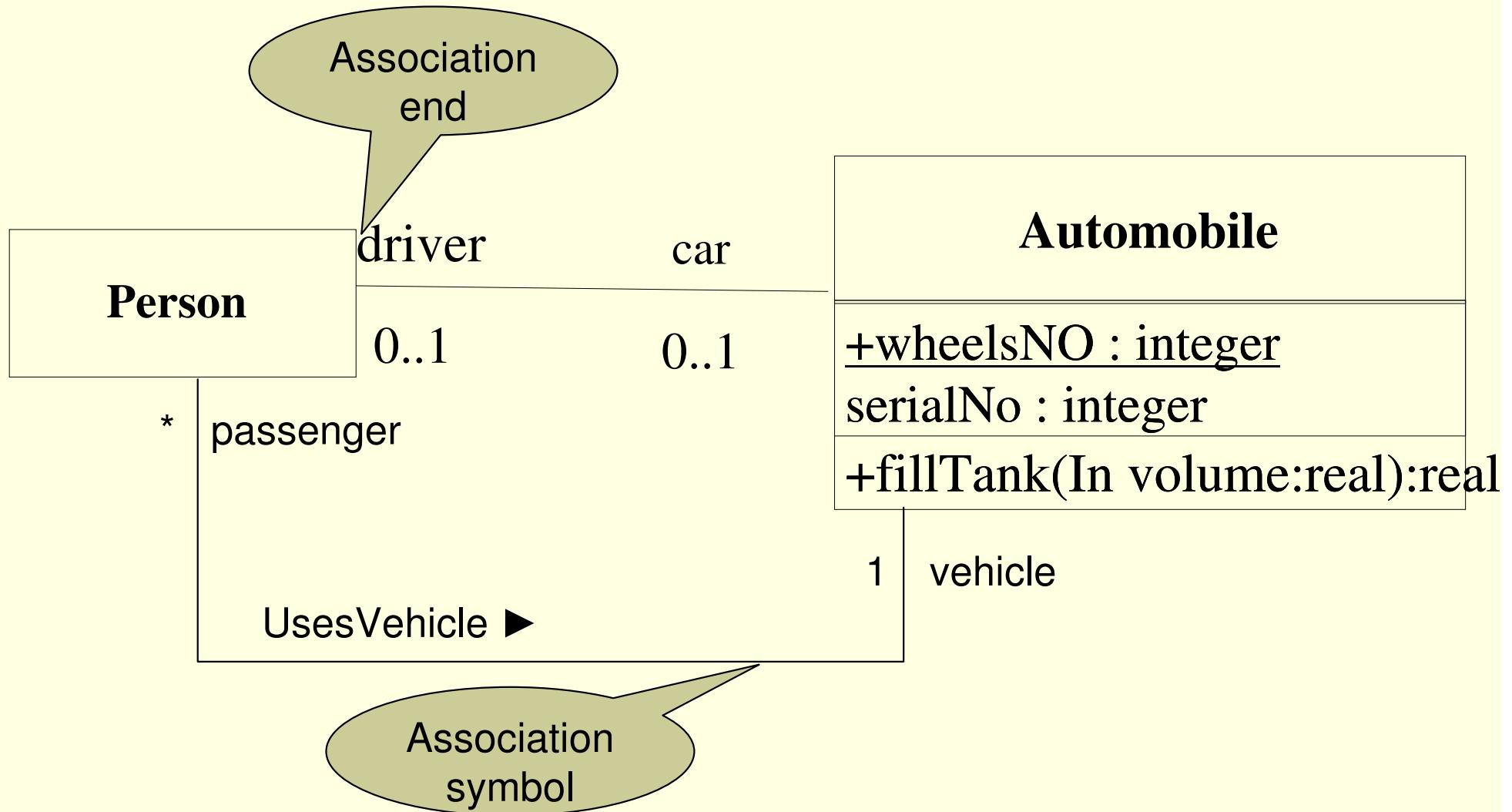
--- should be a moo or a meow
depending on the <condition>



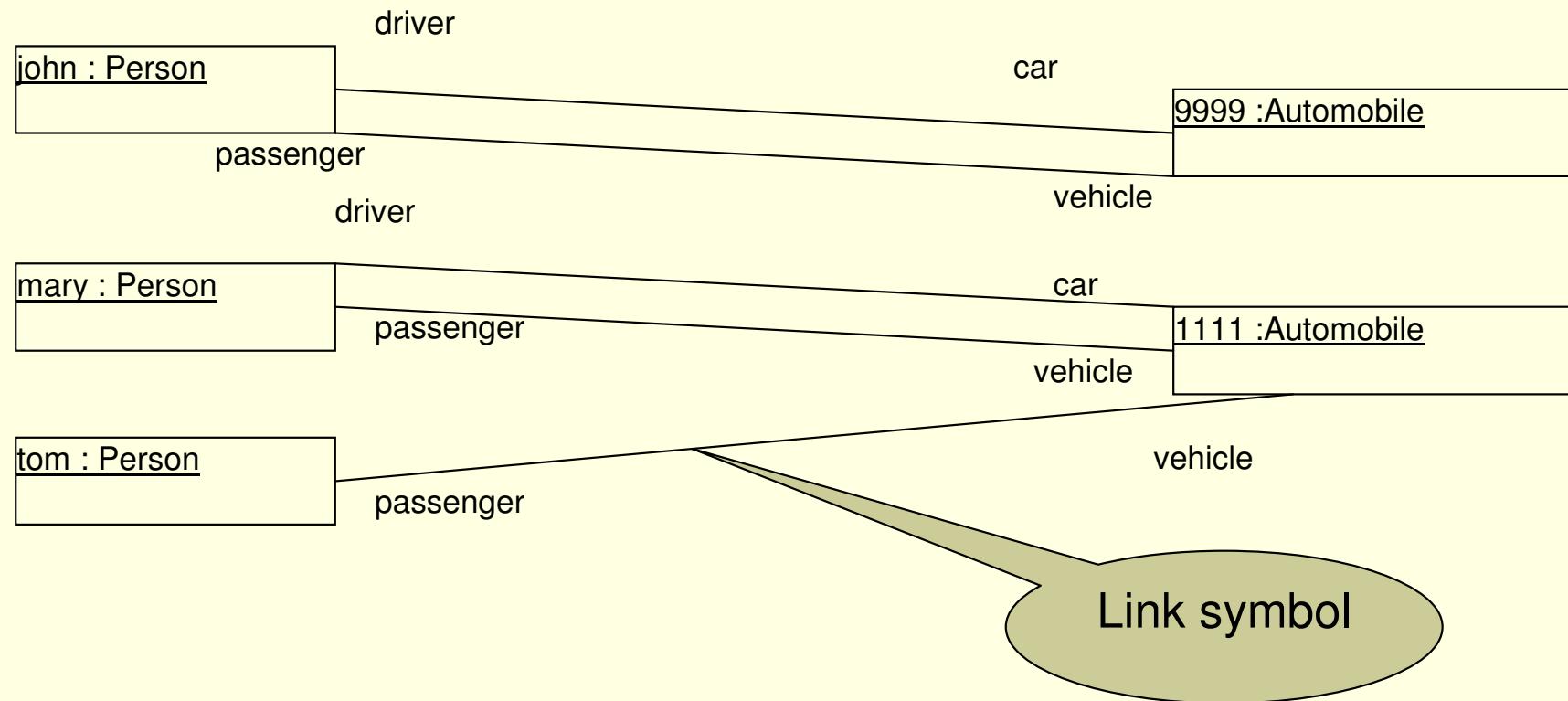
Association

- Concept with **no direct equivalent** in common programming languages
- Is defined as a **semantic relationship** between classes, that can materialize at runtime
- The *instance* of an association is a *set of tuples relating instances* of the classes
- Its actual nature may vary, in terms of code, they may correspond to
 - Attributes, pointers
 - Operations
 - Nothing (i.e. graphical comments)

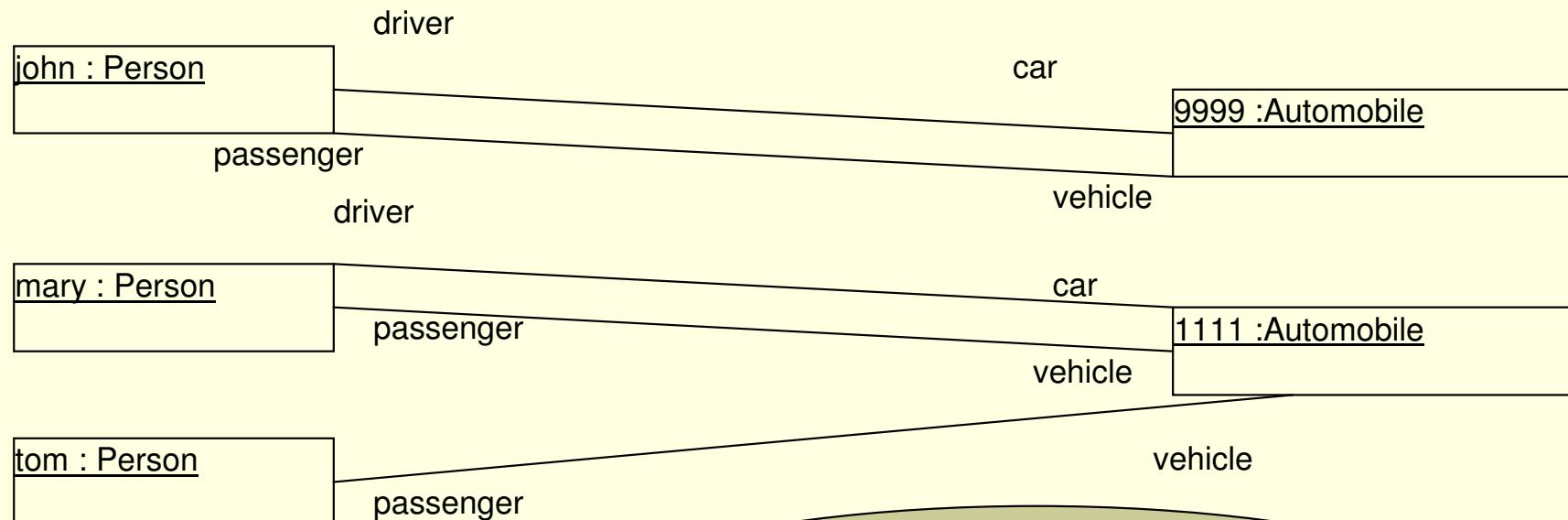
Example



Example – at instance level



Example – at instance level



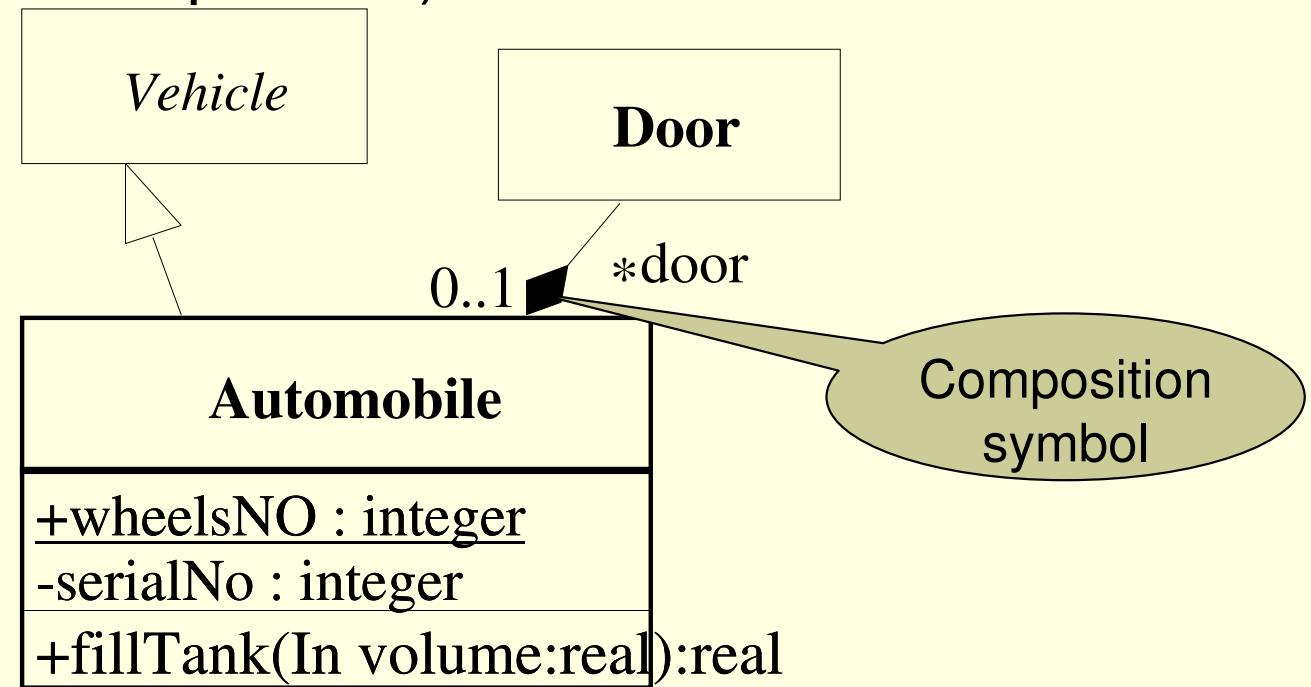
Note on style in UML diagrams:
Instance level names: lower case
Type level names: upper case

Association end

- Endpoint of an association
- Characterized by a set of properties contributing to the association definition
 - Multiplicity (ex: 1, 2..7, *, 4..*)
 - Ordering ordered/unordered
 - Visibility +,-,#, ~
 - Aggregation...

Various kinds of associations (1/2)

- Regular association
- Composition: one class *is owned (composed in)* the associated class
Composition implies lifetime responsibility (based on association end multiplicities)

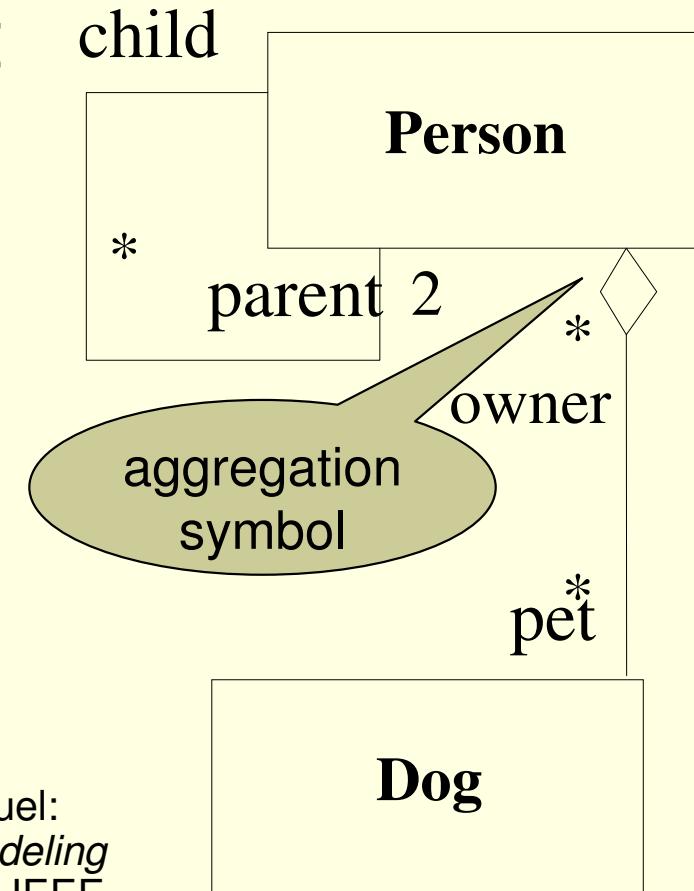


Various kinds of associations (2/2)

■ Aggregation

“light” composition, semantics left open, to be accommodated to user needs

As it is, it has no particular meaning...

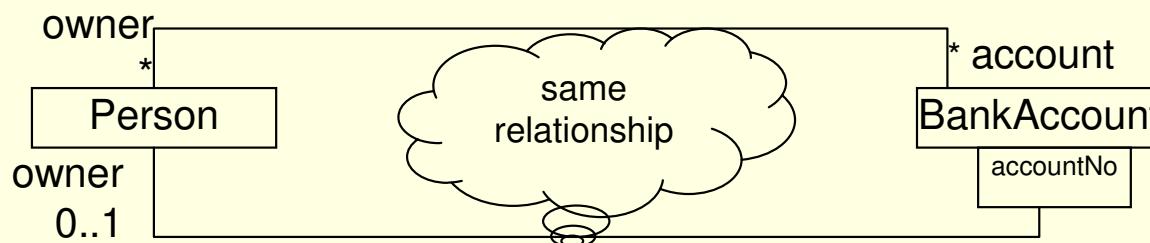


Further reading:

F.Barbier, B.Henderson-Sellers, A.Le Parc-Lacayrelle, J.-M.Bruel:
Formalization of the Whole-Part Relationship in the Unified Modeling Language, IEEE Transactions on Software Engineering, 29(5), IEEE Computer Society Press, pp. 459-470, 2003

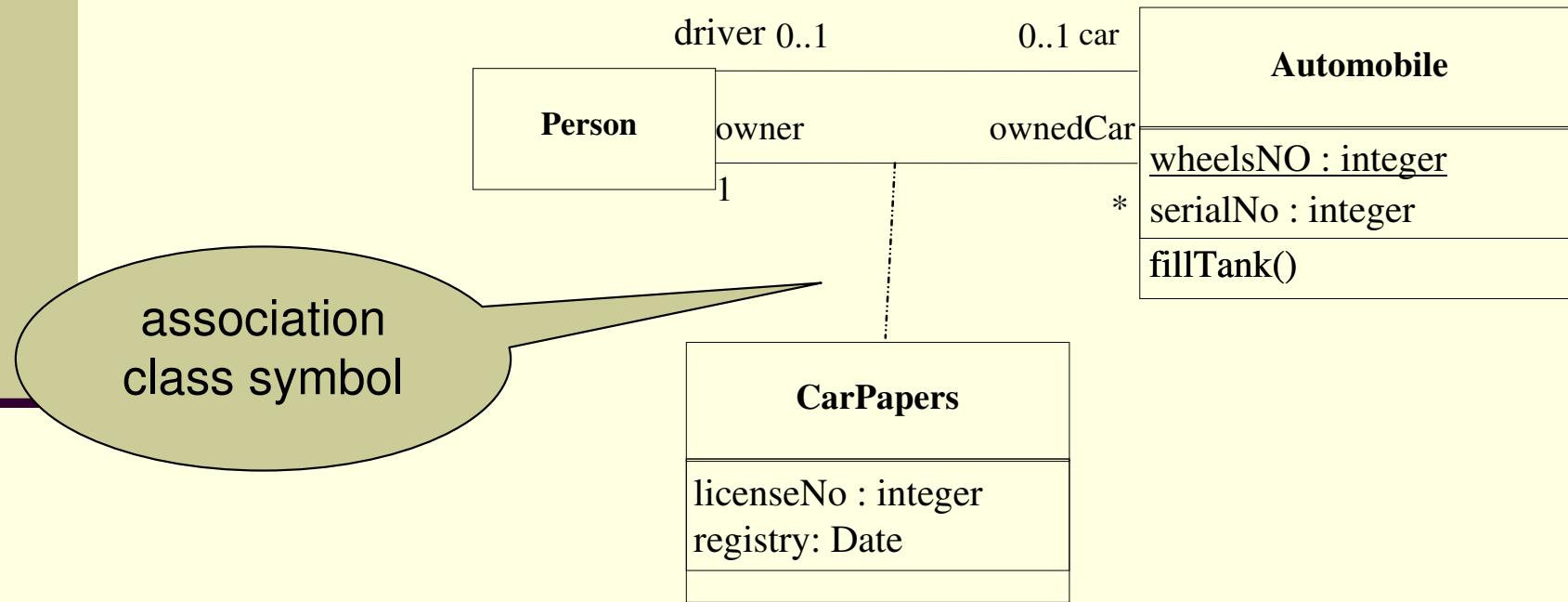
More on associations...

- Associations may be **n-ary** ($n > 2$)
- **Qualifiers** – partition the set of objects that may participate in an association



Association class

- An association that is also a class.
- It defines a set of *features* that belong to the *relationship* itself and not any of the classifiers.



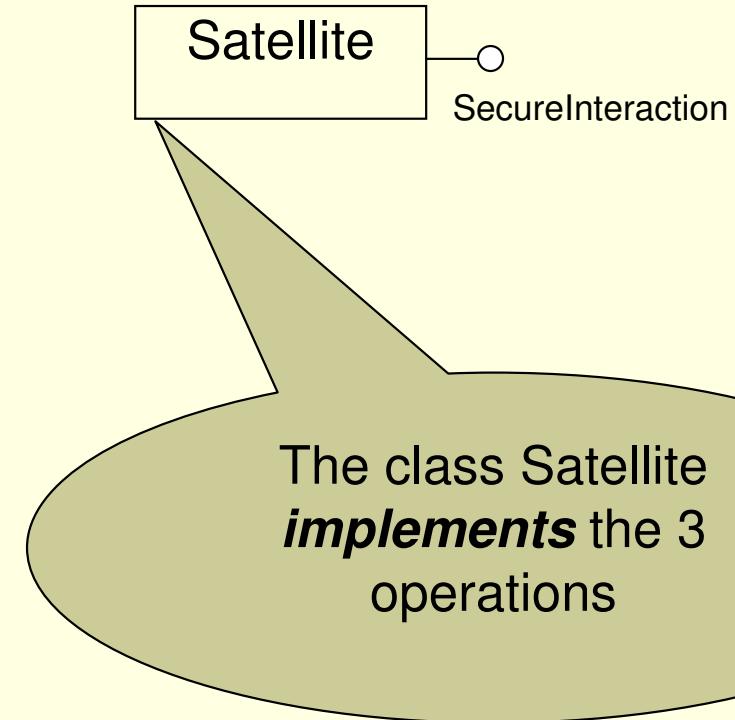
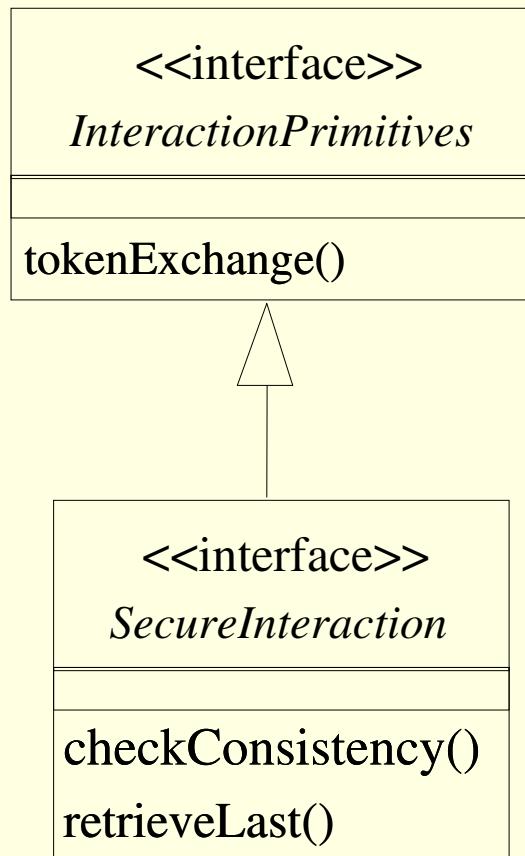
Other elements of class diagrams

- Interface (definition and use)
- Templates
- Comments

Interface

- Declares **set of public features and obligations**
- **Specifies a contract**, to be fulfilled by classes implementing the interface
- Not instantiable, required or **provided** by a class
- Its specification **can be realized** by 0, 1 or several classes
 - the class presents a **public facade that conforms to the interface specification**
(e.g. interface having an attribute does not imply attribute present in the instance)
 - a class may implement several interfaces
- Interfaces hierarchies can be defined through inheritance relationships

Interface definition and use examples



Means to specify the interface contract

- Invariant conditions
- Pre and post conditions (e.g. on operations)
- Protocol specifications

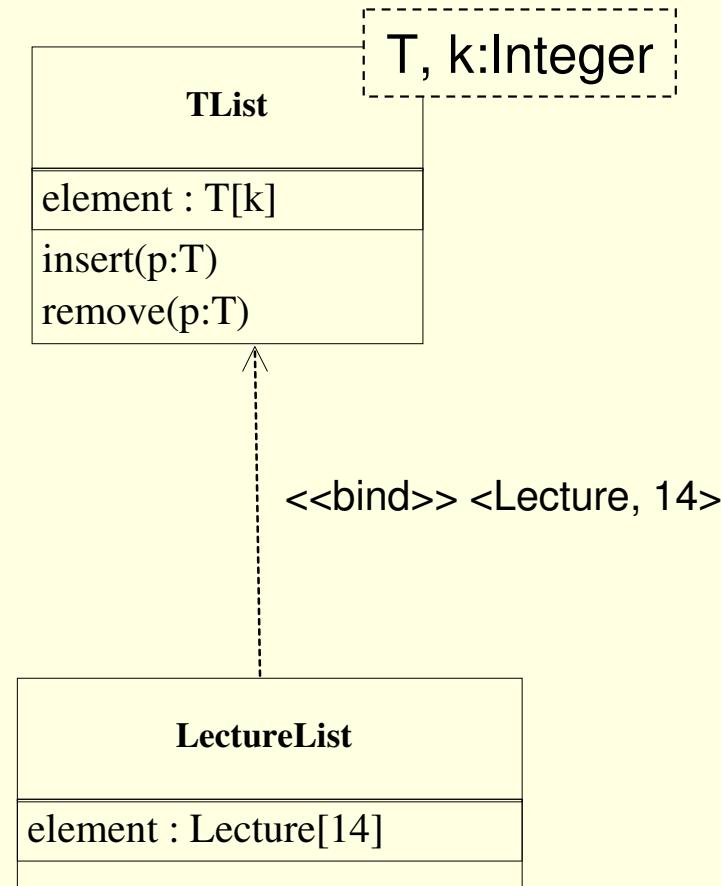
which may impose ordering restrictions on interactions through the interface
for this one may use *protocol state machines*

Templates

- Mechanism for defining **patterns** whose **parameters** represent **types**
- It applies to **classifiers**, **packages**, **operations**

- A **template class** is a template definition
 - Cannot be instantiated directly, since it is not a real type
 - Can be **bound** to an **actual class** by specifying its parameters
- A **bound class** is a real type, which can be instantiated

Template example



Class diagram summary

- The most used diagrams
- Describes the static structure of the system in terms of classes and their relationships (associations, inheritance)
- Offers connection points with the UML behavior description means

Overview

- What is UML?
- Structure description
 - Use case diagram
 - Class diagram
 - Components
 - Communication principles in UML
 - System initialization
- Behavior description
- OCL
- UML and tools

Component

- Its definition evolves from UML 1.x to UML 2.0
- In UML 1.x - deployment artifacts



- In UML 2.0 – structured classes

Component in UML 2.0

- Modular part of a system encapsulating its content
- Defines its behavior in terms of provided and required interfaces, and associated contracts
- Defines a type. Type conformance is defined on the basis of conformance to provided / required interfaces
- Main property: substitutability = ability to transparently replace the content (implementation) of a component, provided its interfaces and interface contracts are not modified

Component examples (1/2)

■ Algorithmic calculus component

■ Interface:

- Offered: provided mathematical calculus functions
- Required : logarithm value calculus

■ Contract

- Expected behavior
- Constraints on unauthorized values

Component examples (2/2) : mobile phone logical network

Sample component: virtual cell manager

■ Interface:

- Manage reachable mobile phones
- Forward message calls
- ...

■ Contracts:

- Functional
 - Fulfill expected behavior
 - Protocol describing authorized message exchange:
(e.g. first identify)
- Non-functional
 - Net load capacity, reactivity time, electromagnetic interference...

Component related concepts

- Class
- Package
- (Library)

The exact relationship between all these concepts is not completely clear (neither in UML, nor in the literature)

UML offers a unifying concept... classifier

- Generalization of the class concept
- Gives a type for a collection of instances sharing common properties

- Interfaces, classes, data types, components

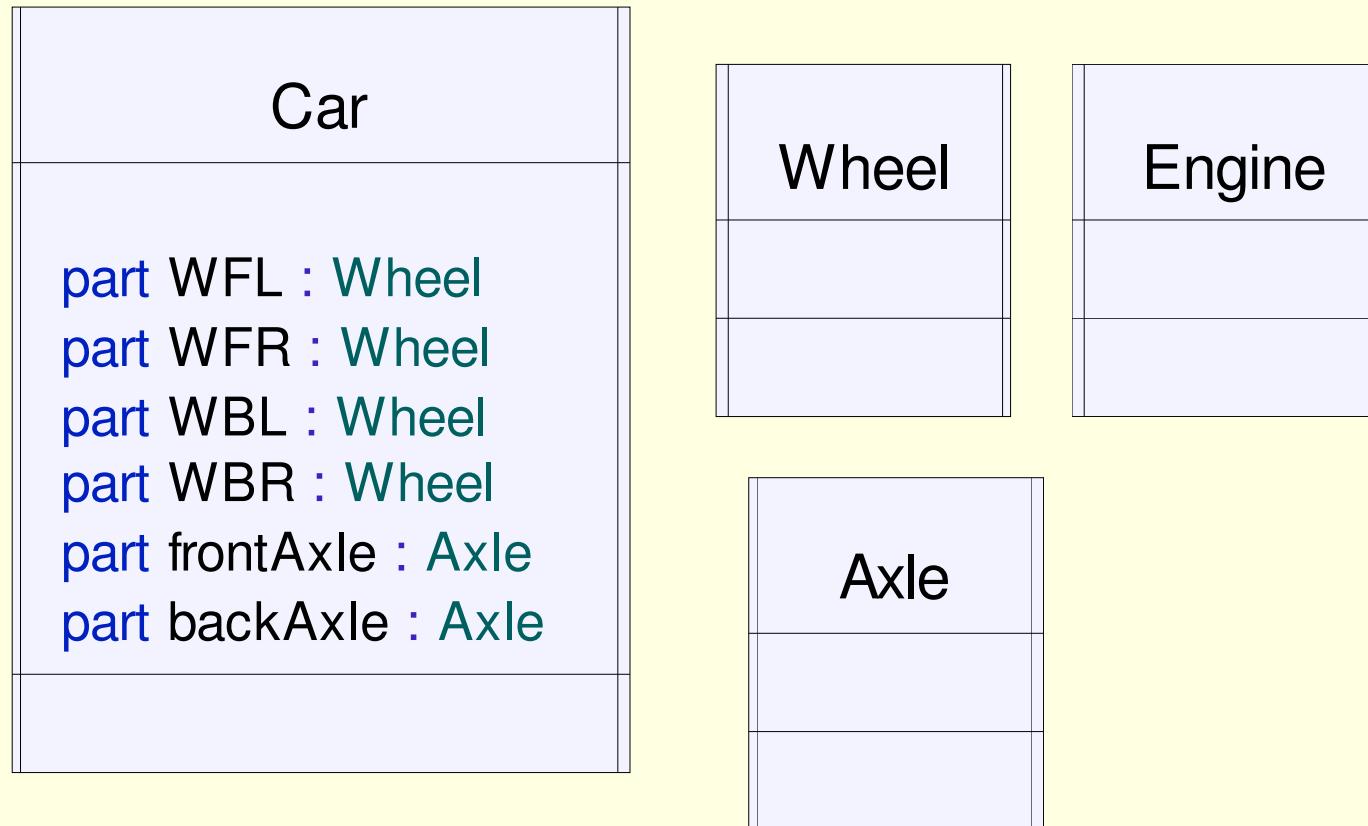
Composite structure diagram (a.k.a. architecture diagram)

- Added in UML 2.0
- Depicts
 - The internal structure of a classifier
 - Interaction points to other parts of the system
 - Configuration of parts that perform together the behavior of the containing classifier
- Concepts involved:
 - Classifier
 - Interface
 - Connection
 - Port
 - Part

Part

- Element representing a (set of) instance owned by a classifier
- Semantics close to the one of attributes or composed classes
 - May specify a multiplicity
 - At parent creation time, parts may need to be created also
 - When the parent is destroyed, parts may need to be destroyed also

Example



Abstraction level for part

- Somewhere between instance and type...
- WFL characterizes the wheels front left, owned by Car instances
- *Given a Car class instance, the part WFL is an instance of its front right wheel*
- *If no Car class instance is fixed, the part WFL is an instance abstraction generically characterizing front right wheels of Cars*

Port

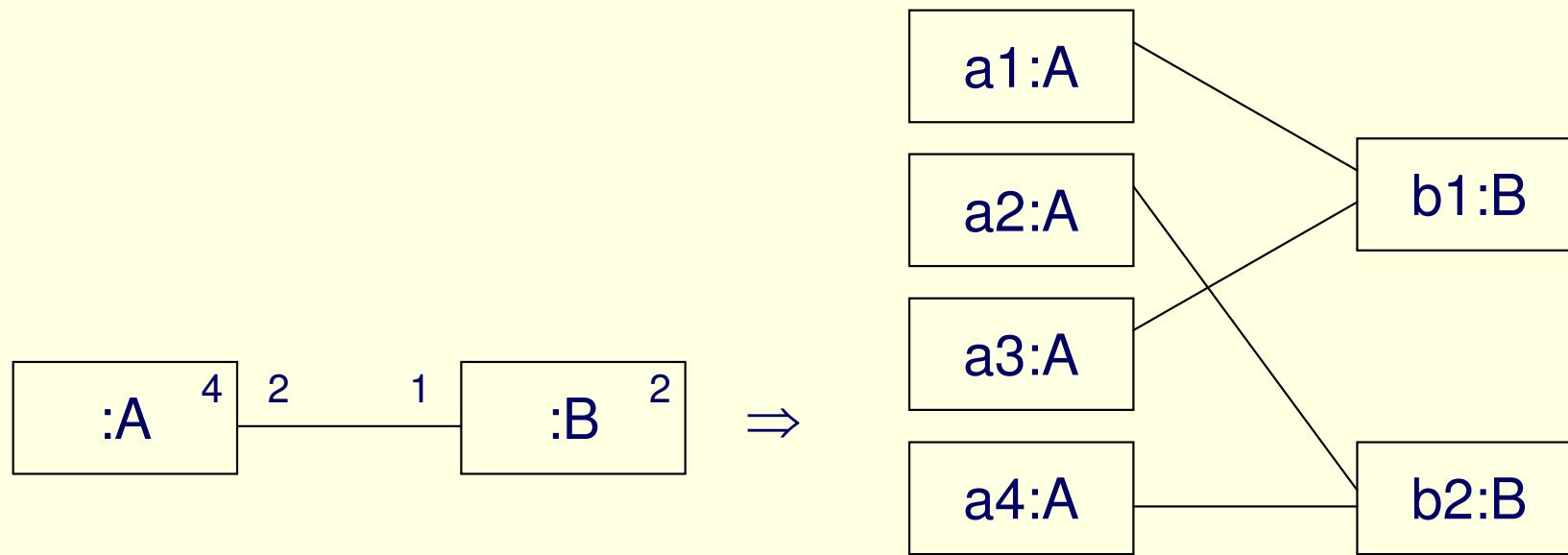
- feature of a classifier specifying a distinct **interaction point**
 - between that classifier and its environment (**service port**)
 - between the behavior of the classifier and its internal parts (**behavior port**)
- characterized by a list of **required** and **provided interfaces**
 - **Required interfaces** describe services the owning classifiers *expect* from environment and *may access via this interaction point*
 - **Provided interfaces** describe services the owning classifiers *offer* to its environment *via this interaction point*
- *an instance may differentiate between invocations of a same operation received through different ports*

Connector

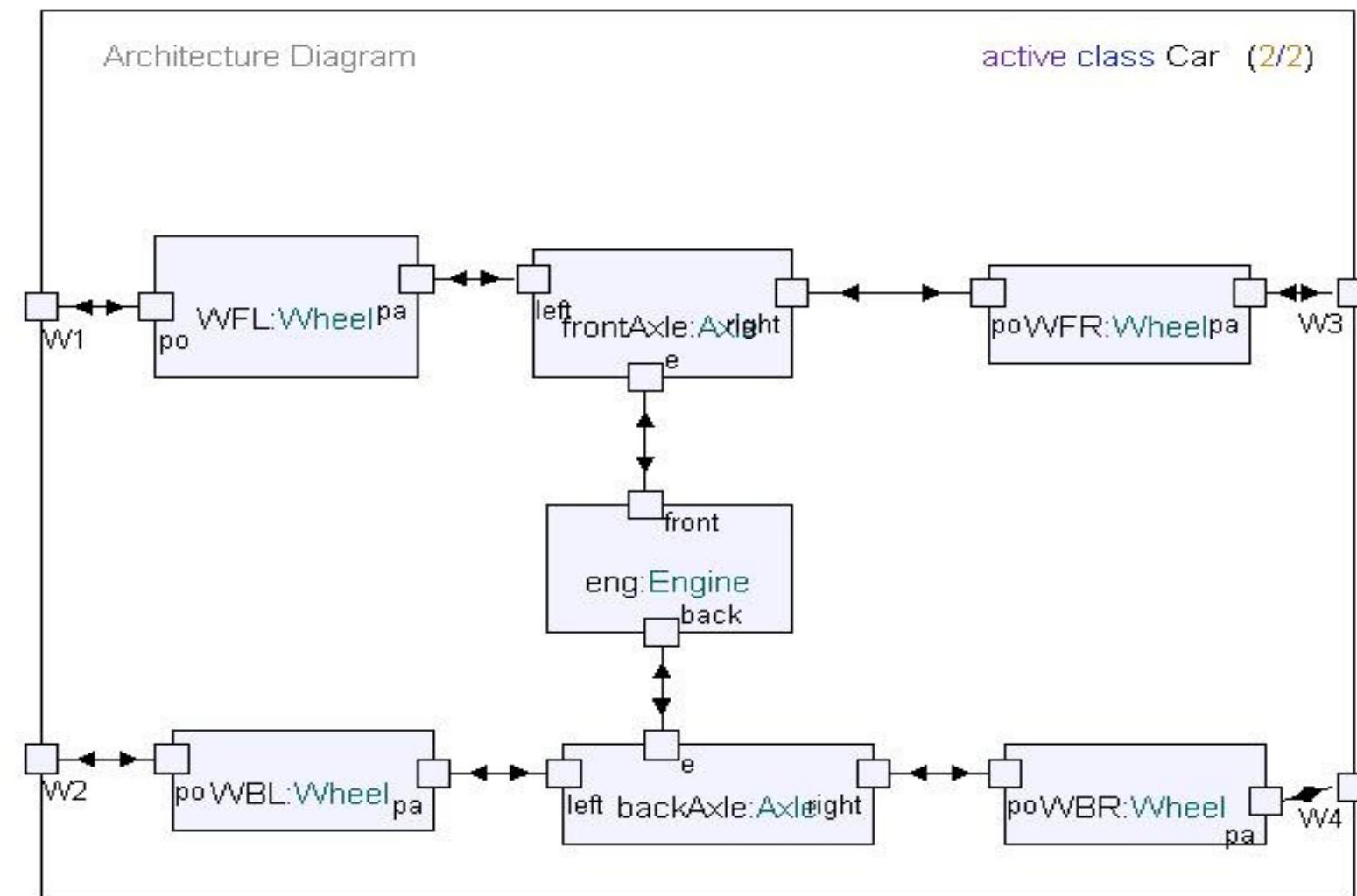
- Link enabling communication btw instances
- Its actual realization is not specified
(simple pointer, network connection, ...)
- It has two connector ends, each playing a distinct role
- The communication realized over a connector may be constrained
(type constraint, logical constraint in OCL, etc)

Communication architecture

- complex multiplicity → need for initialization rules



Example: composite structure diagram



Port vs. interface

- Interface – signature
- Port – interaction point

- Interfaces describe what happens at a port
- The same interface may be attached to several ports of a component

Port constraints vs. interface constraints

- Constraints may be attached to both ports and interfaces
- For both, constraints can take the form of pre and post conditions, invariants, protocol constraints
- Nothing is stated on how constraints at various levels should be composed
- By default, constraint conjunction
- More elaborated constraint handling schemes may be imposed by the methodology

Connector vs. link

- Link = association instance
 - Data oriented
 - May be attached to any instance of the corresponding classifier
- Connector
 - Behavior (communication) oriented
 - Can only be connected to particular instances
 - Instance to which it applies are depicted in the composite structure diagram

Overview

- What is UML?
- Structure description
 - Use case diagram
 - Class diagram
 - Components
 - Communication principles in UML
 - System initialization
- Behavior description
- OCL
- UML and tools

Communication

- Communication primitives
- Communication schema

Communication primitives

■ Signal

- One way
- Asynchronous communication primitive
- May carry data
- It is defined independently of the classifiers handling it

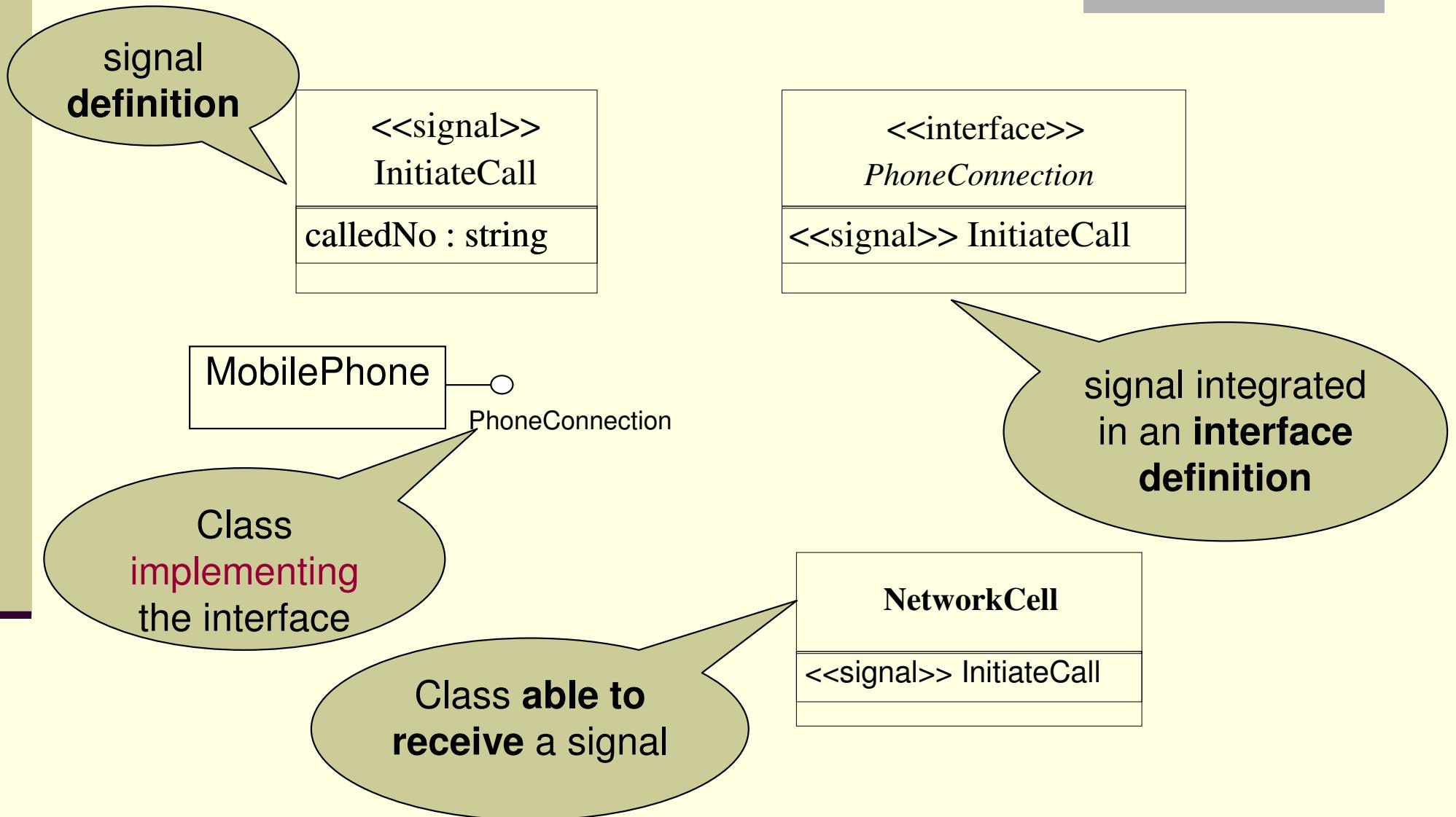
■ Operation call

- Two-way communication primitive (call-reply)
- The caller is blocked
- May carry data
- Typically, it has a target object

■ Queue

- Communication buffer
- May be attached to instances
- Management policy not constrained

Signal definition and use examples



Communication schema in UML

- If the *model says noting on communication* (i.e. no connectors exist)
 - **Point to point**: between objects knowing their ID (due to existing associations, passed as parameter in some operation, etc)
 - **Broadcast**: to listening and accessible objects
- If a *communication structure is stated* (architecture diagram) - the communication obeys its constraints
 - communication paths, connectors chain, conveyed messages, port constraints etc...

Overview

- What is UML?
- Structure description
 - Use case diagram
 - Class diagram
 - Components
 - Communication principles in UML
 - System initialization
- Behavior description
- OCL
- UML and tools

System initialization

■ What it is?

- The mechanism that gives the **initial status of the system**

■ How it can be done?

- Using a **God** object that **creates the whole system**
- Using an **initialization script**
- Based on a particular **object diagram** giving the snapshot of the system at initialization time

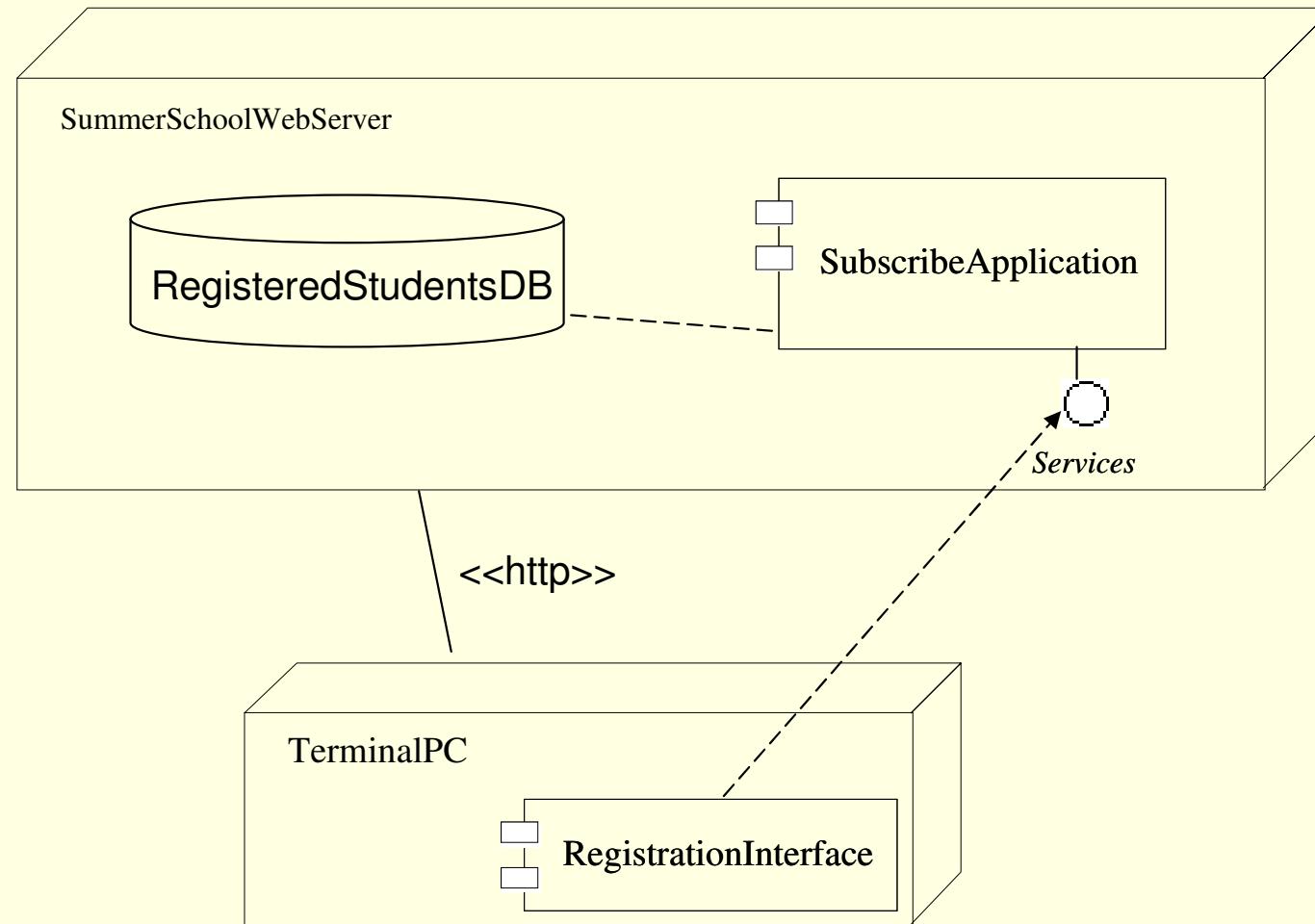
■ How it is in the standard?

- **No standard mechanism exists**

Going forward in component based modeling ...

- The actual “wiring” of components is designed using *component* and *deployment diagrams*
- Component diagrams
 - Models **business and technical software architecture**
 - Uses **components** defined in the **composite structure diagrams**, in particular their **ports** and **interfaces**
- Deployment diagrams
 - Models the **physical software architecture**, including issues such as the **hardware**, the **software installed** on it and the **middleware**
 - Gives a **static view of the run-time configuration** of **processing nodes** and the **components** that run on those nodes

Deployment diagram example



Overview

- What is UML?
- Structure description
- Behavior description
 - State machine diagrams
 - Protocol state machine
 - Activity diagrams
 - Sequence diagrams
 - Timing diagrams
- OCL
- UML and tools

Specifying behavior in UML

	specification	description
System	Use case Sequence diagram Invariants	State machine
Class	Sequence diagram Invariants Protocol state machine	State machine
Operation	Pre-condition Post-condition Invariants Protocol state machine	State machine Actions

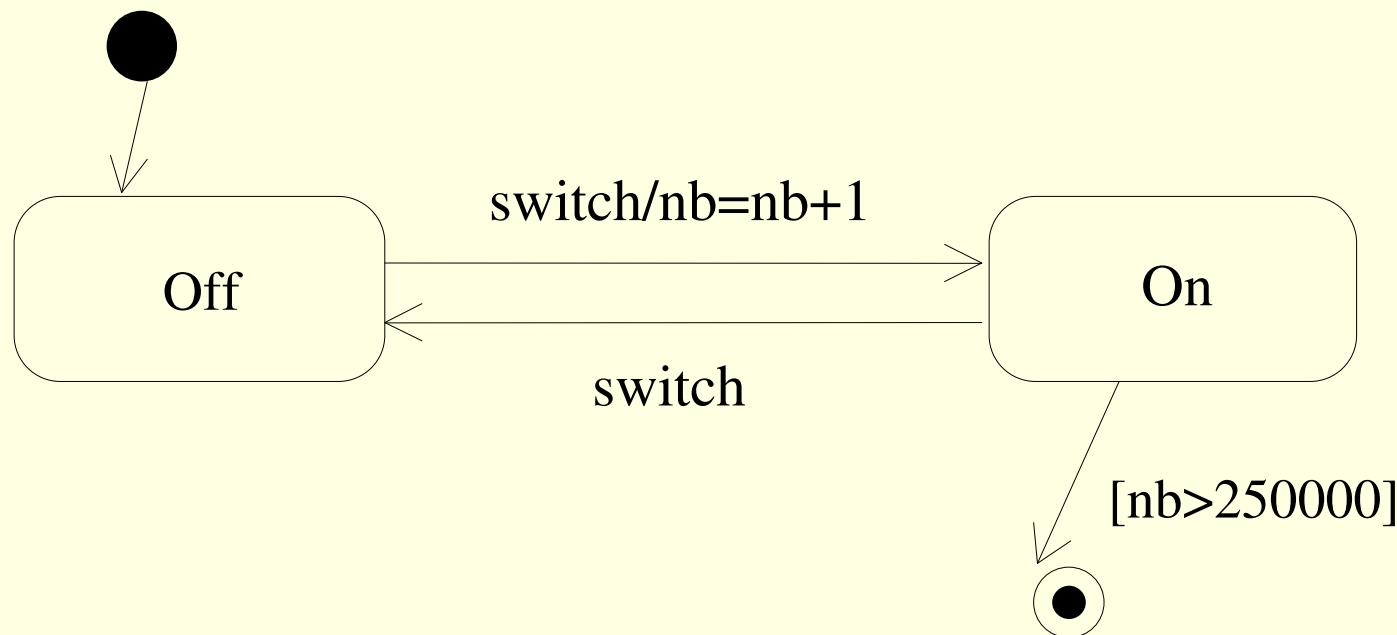
State machine

- UML finite state automaton
- Behavior description mechanism
- Describes the behavior for:
 - System
 - Class
 - Operation

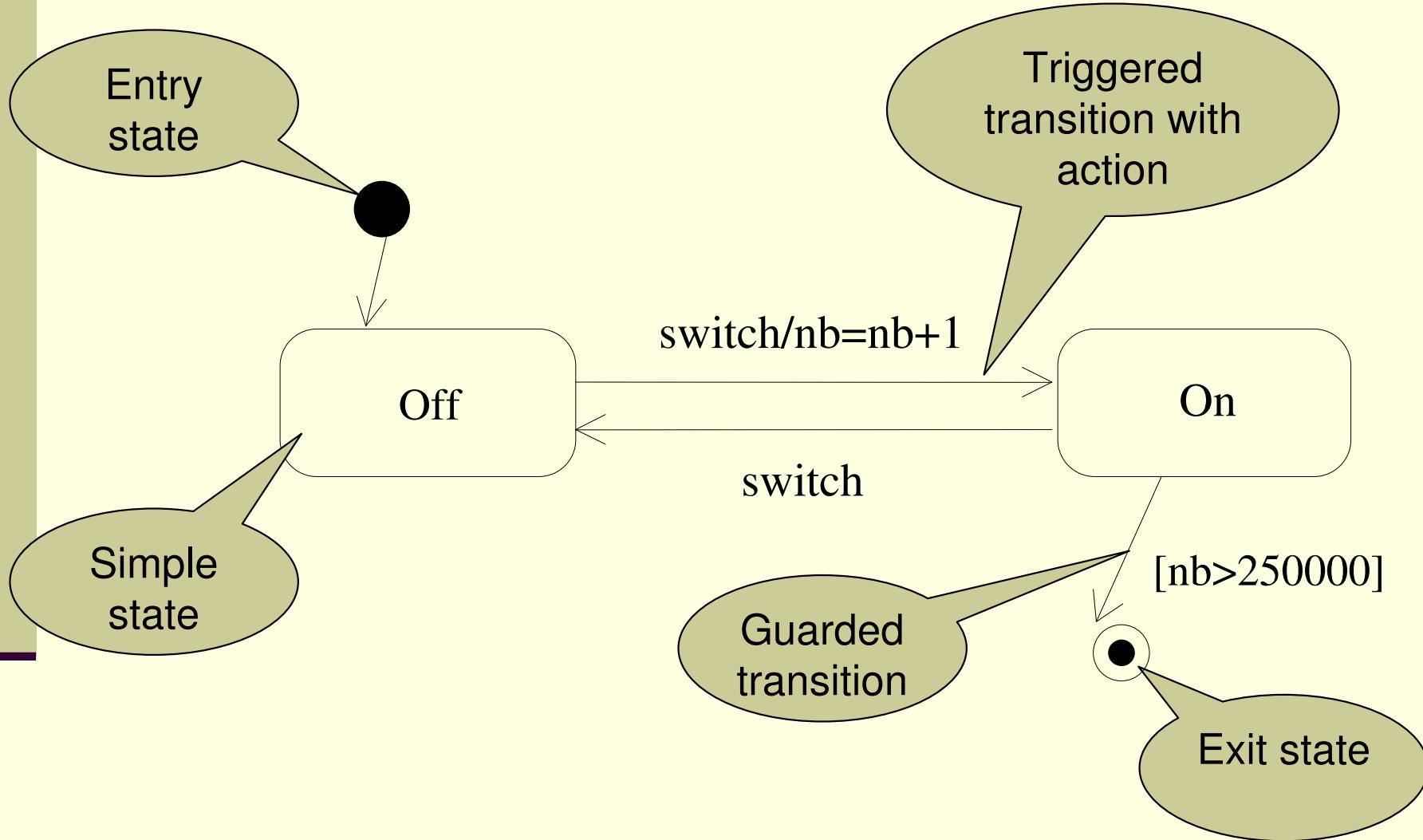
Main concepts

- **State** – stores information of the system (encodes the past)
Particular states
 - Initial state (?)
 - Final state
- **Transition** – describes a state change
 - Can be **triggered** by an event
 - Can be **guarded** by a condition
- **Actions** – behavior performed at a given moment
 - **Transition action** : action performed at transition time
 - **Entry action** : action performed when entering a state
 - **Exit Action** : action performed when exiting a state
 - **Do Action** : action performed while staying in a state

Simple state machine example



Simple state machine example



Event

- “Specification of some **occurrence** that may potentially trigger effects by an object”

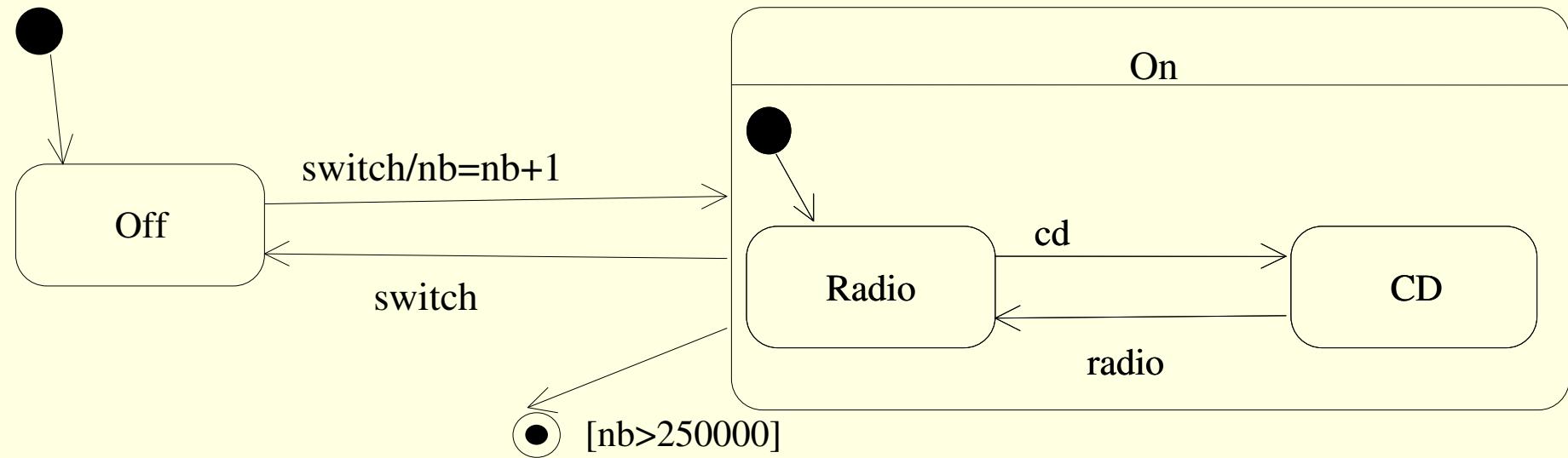
Typically used in StateMachines as triggers on transitions

- Examples (as defined in the standard): SignalEvent, CallEvent, ChangeEvent, TimeEvent, etc.
- Notion refined in the SPT profile

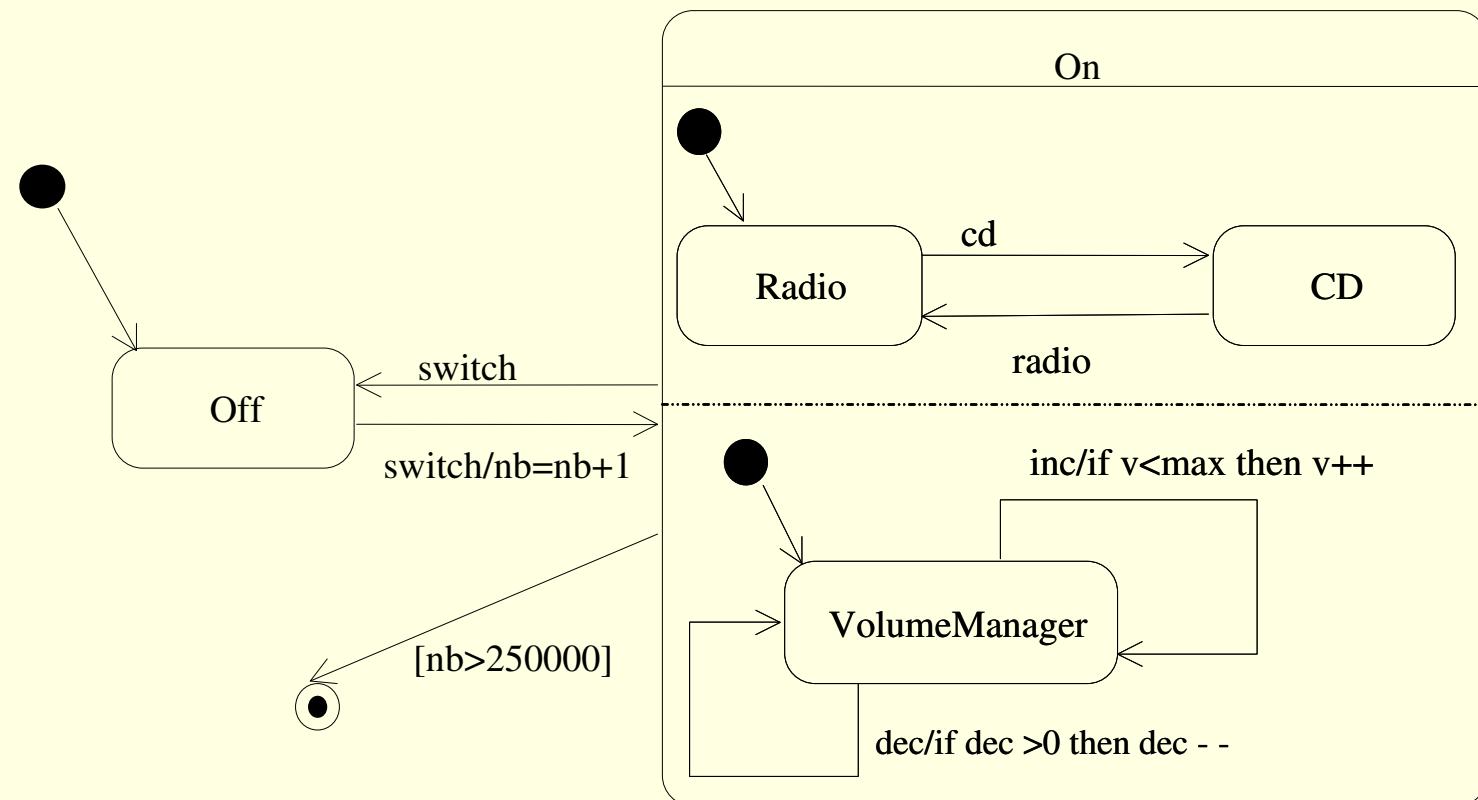
Hierarchical states

- All states are at the same level => the design does not capture the commonality that exists among states
- Solution: Hierarchical states – described by sub-state machine(s)
- Two kinds of hierarchical states:
 - And-states (the contained sub-states execute in parallel)
 - Or-states (the contained sub-states execute sequentially)

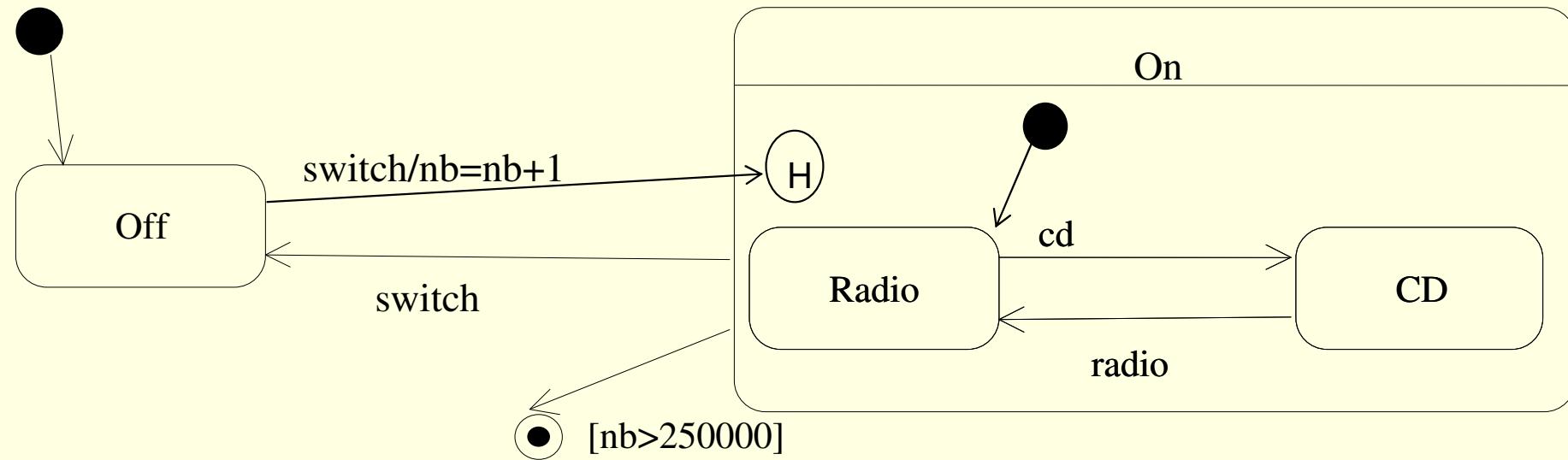
Hierarchical OR-state machine example



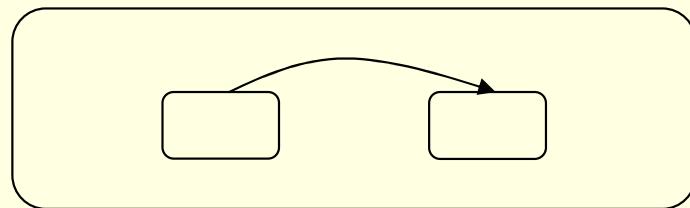
Hierarchical AND-state machine example



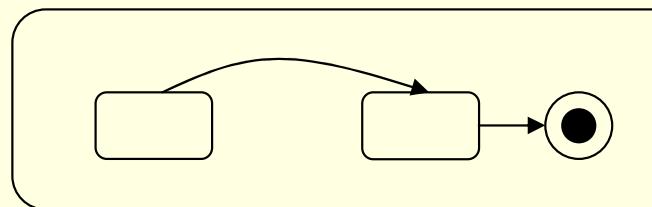
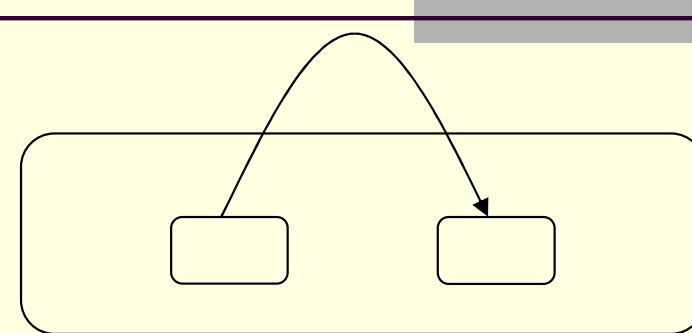
History sub-states



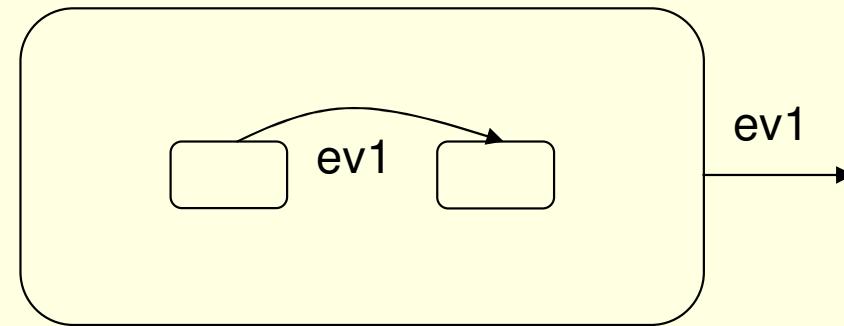
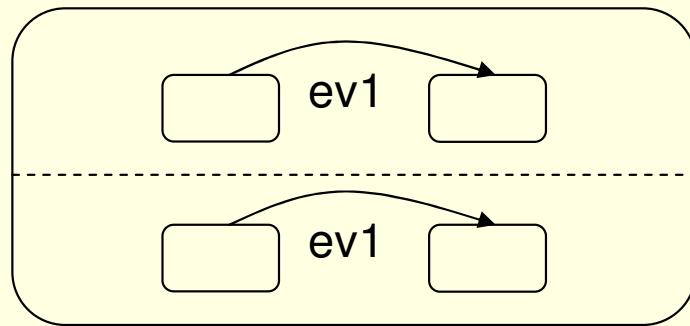
Semantic nuances in state machine diagrams



vs.



/* no trigger here */



When to use state machines?

- For reactive systems
- Why use them?
 - If properly used
 - easy to read
 - nice verification results
 - the tools can generate code more efficient than if hand-written
- Open questions:
 - state machine inheritance...
 - consensual semantics

Further reading:

Harel, David and Eran Gery, "Executable Object Modeling with Statecharts", *IEEE Computer*, July 1997, pp. 31-42.

Harel, David, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, 1987, pp. 231-274.

Overview

- What is UML?
- Structure description
- Behavior description
 - State machine diagrams
 - Protocol state machine
 - Activity diagrams
 - Sequence diagrams
 - Timing diagrams
- OCL
- UML and tools

Protocol state machines

- Particular state machines used to impose sequencing constraints
- Can be attached to interfaces, components, ports, classes
- Express
 - Usage protocols
 - Lifecycles for objects
 - Constrain the order of invocation for its operations
- Do not preclude any specific behavior description
- Protocol **conformance** must apply between the **protocol state machine** and the **actual implementation**
- A classifier may own several state machines (ex. due to inheritance)

Syntactic constraints on protocol state machines

- No entry, exit, do action on states
- No action on its transitions
- If a transition is triggered by an operation call, then that operation should apply to the context classifier

Protocol state machine interpretations

■ Declarative

- Specifies legal transitions for each operation
- The actual legal transitions for operations are not specified
- Defines the contract for the user of the context classifier

■ Executable

- Specifies all events that an object may receive and handle, plus the implied transitions
- Legal transitions for operations are the triggered transitions

Protocol state machine example

- Notation: {protocol} mark should be placed close to the state machine name

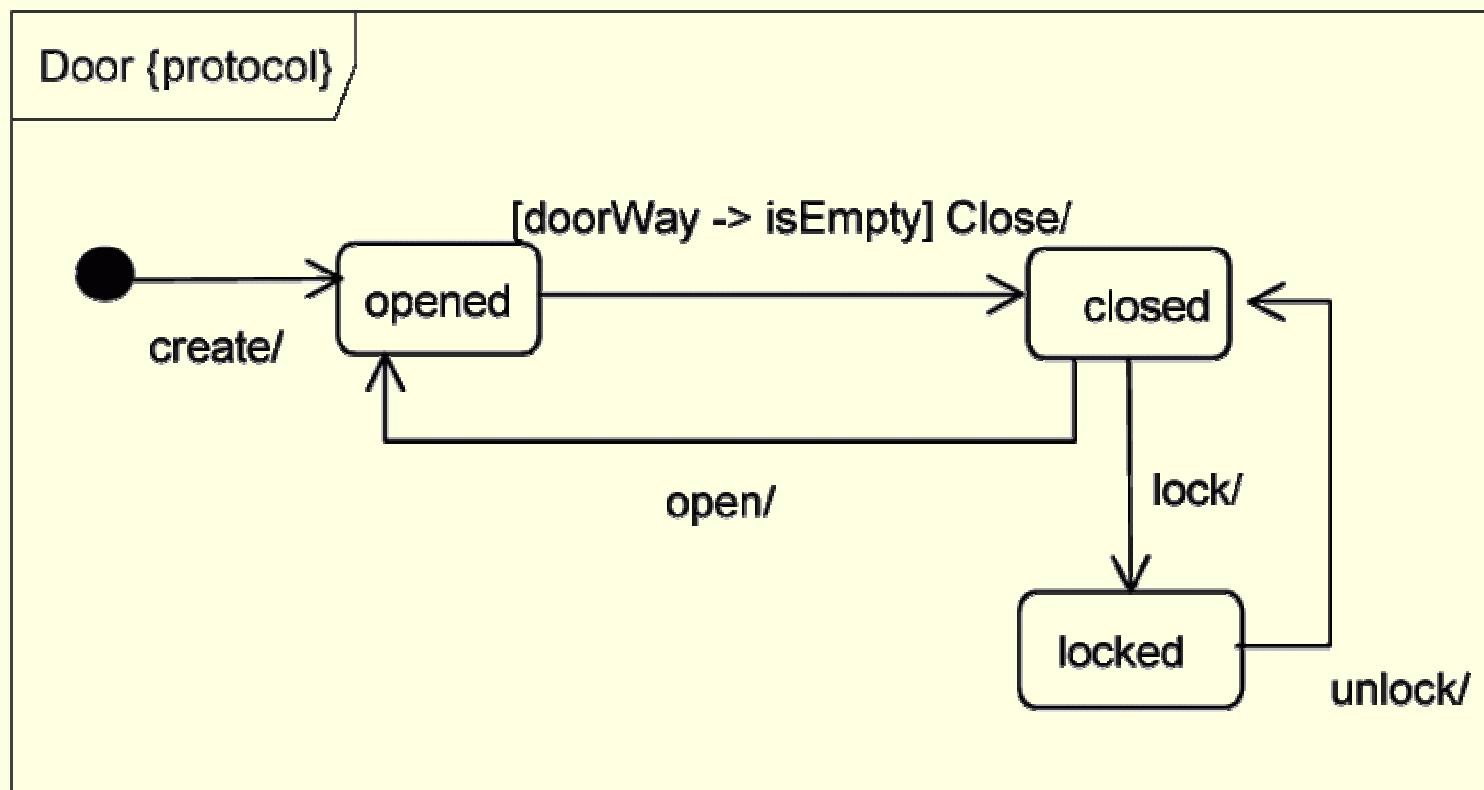


Figure 364 - Protocol state machine

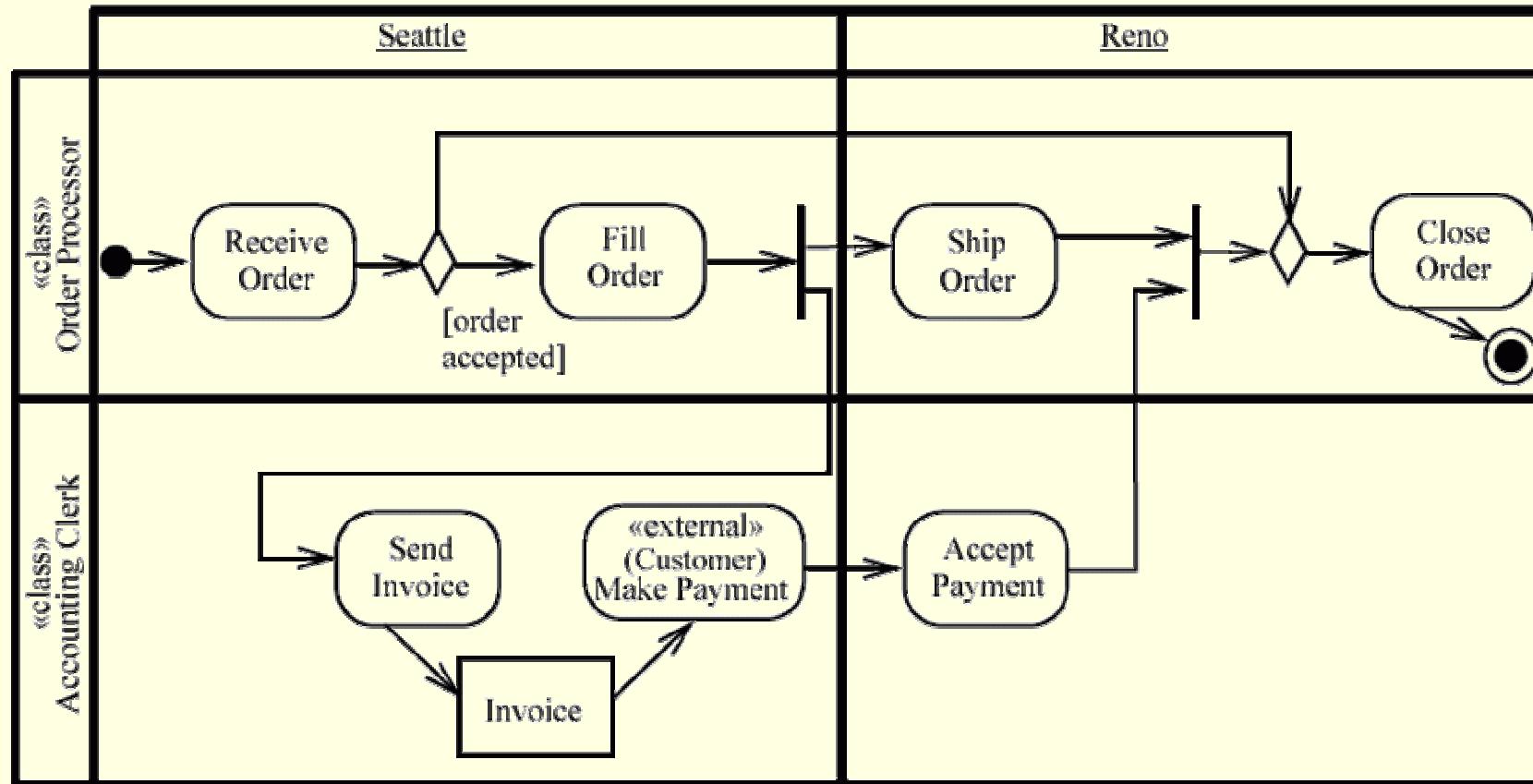
Overview

- What is UML?
- Structure description
- Behavior description
 - State machine diagrams
 - Protocol state machine
 - Activity diagrams
 - Sequence diagrams
 - Timing diagrams
- OCL
- UML and tools

Activity diagrams

- Related to state machine diagrams
 - State diagrams – focus on the execution of a single object
 - Activity diagram – focus on the behavior of a set of objects
- Purpose
 - Models high-level business processes, including data flow,
 - Models the logic of complex logic within a system
- Concurrency model based on Petri Nets

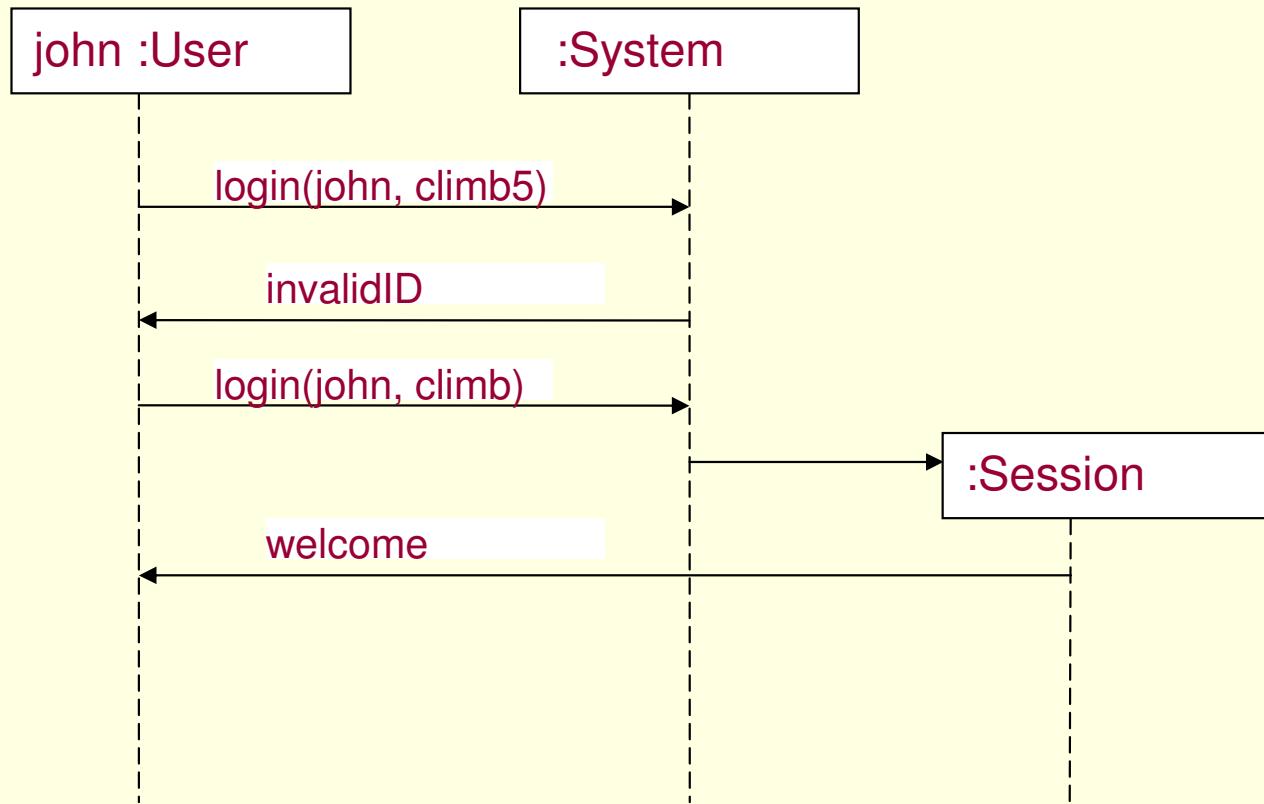
Activity diagram example



Sequence diagram

- Shows a concrete execution scenario, involving:
objects, actors, generic system
- Highlights the lifelines of the participating instances
- Focuses on *interaction, exchanged messages and their ordering*
- Give *instances of (cooperating) state machine executions*
- Can address various levels of abstraction:
 - System level
 - Object sets level
 - Object level
 - Method level

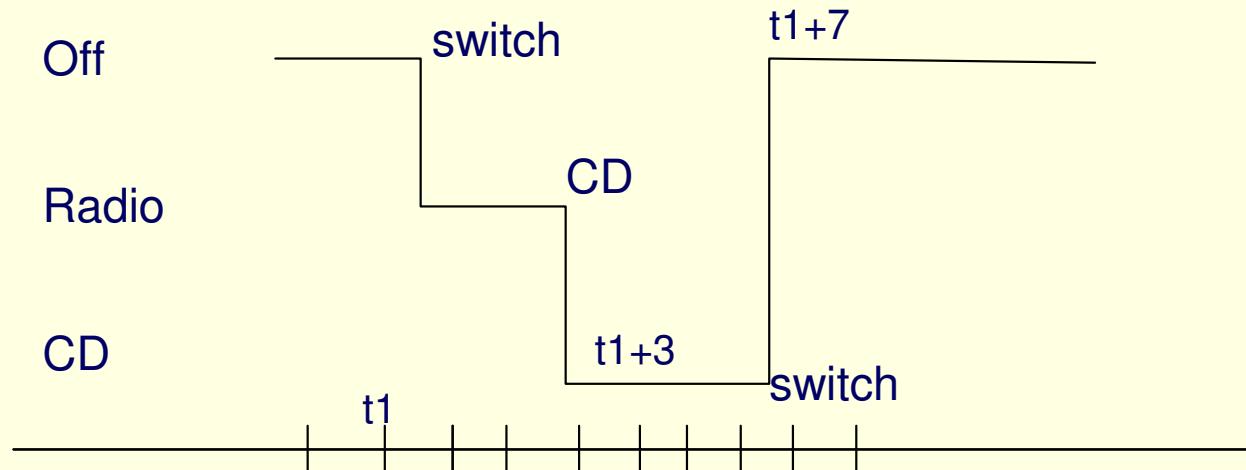
Example



Timing diagram

- used to explore the behaviors of one or more objects throughout a given time interval
- relevant for systems with time sensitive behavior

w:Walkman



Although universal, UML can't contain everything...

- Extension mechanisms
 - **Stereotype**
 - mechanism allowing to **specialize** particular UML concepts
 - allows to use **platform or domain specific terminology**
 - e.g. Class stereotyped *reactive* if it has a state machine
 - **Tagged values** – allows to attach **information** to UML model elements
 - Profile - a **stereotyped package** containing **model elements that have been customized** (e.g. for a specific domain) using stereotypes, tagged definitions and constraints
- e.g. SPT, UML profile for EDOC, ...

Overview

- What is UML?
- Structure description
- Behavior description
 - State machine diagrams
 - Protocol state machine
 - Activity diagrams
 - Sequence diagrams
 - Timing diagrams
- OCL
- UML and tools

OCL – Object Constraint Language

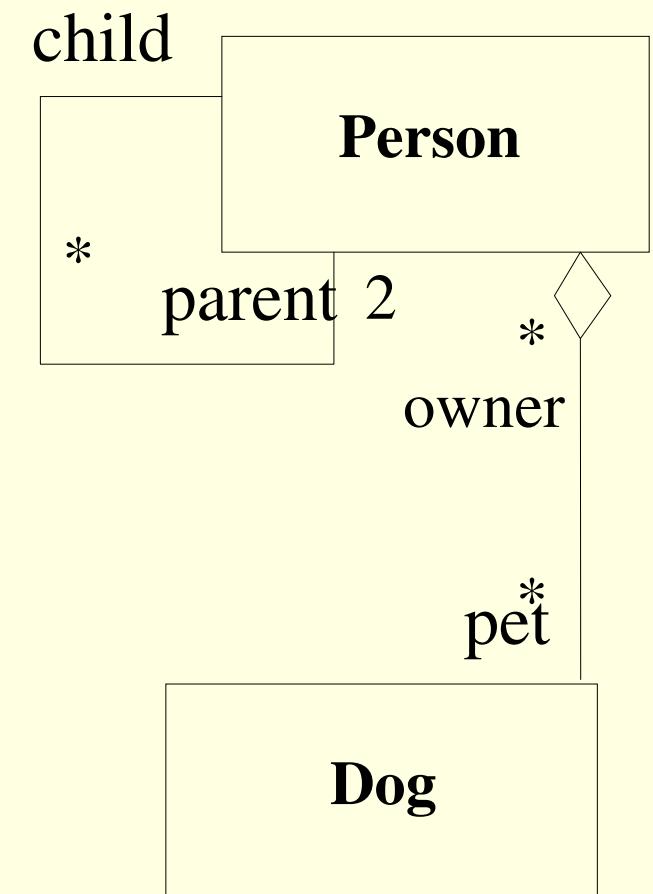
- Constraint language integrated in the UML standard
- Aims to fill the gap between mathematical rigor and business modeling
- Recommended for:
 - Constraints: pre and post conditions, invariants
 - Boolean expressions: guards, query body specification
 - Defining initial and derived values of features
- UML meta-model WFRs written in OCL

Example 1 – (all kinds of) invariants

No grandchild may not have more than 2 pet dogs:

contex Person

inv: self.child.child.pet \rightarrow size()<2



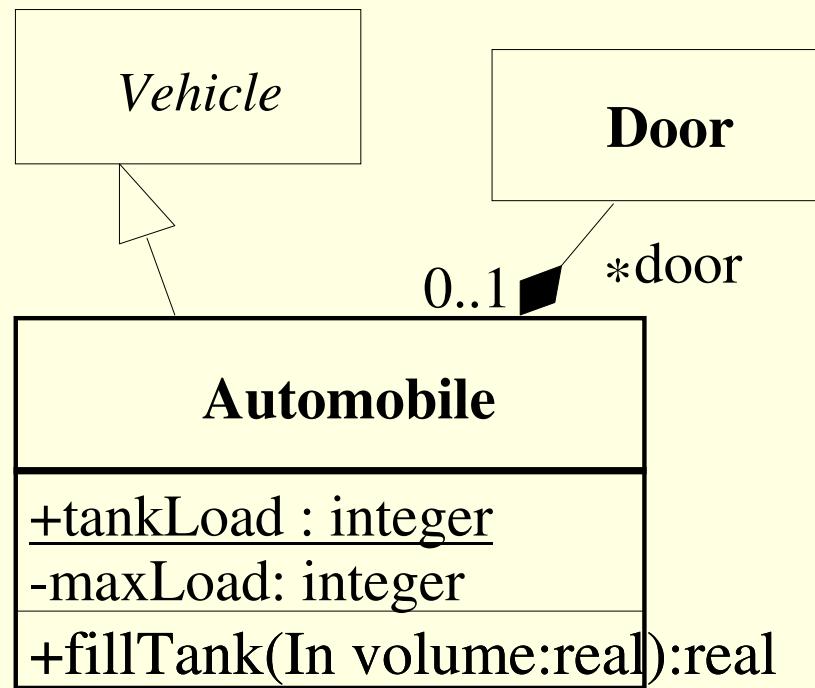
Example 2 – pre and post conditions

context Automobile::fillTank (in volume:real):real

pre: volume>0

pre: tankLoad + volume < maxLoad

post: tankLoad = tankLoad@pre + volume



Overview

- What is UML?
- Structure description
- Behavior description
- OCL
- UML and tools

Tool support for UML

- UML can only live if tool builders support it
Just think of a programming language with no compiler...
- Tool builders are de facto deciders of live and dead parts of the languages
- There is no UML tool that offers all the functionalities one can think of
- This part is not a presentation of tools, rather a list a functionalities offered by various tools

Functionalities

- Editing support
- Documentation generation
- Syntax check
- Static semantic check

- Code generation
- Symbolic execution / simulation
- Formal verification

- Support for tests on model
- Test case generation

- Reverse engineering
- Model transformation and translations to other formalisms

- ...

Model interchange

- The need
 - A single tool does not offer all the functionalities
 - Avoid user kidnapping
- The solution
 - XMI: standardized model interchange format
 - Offers an XML DTD schema of the metamodel, to be used by tools
- The reality
 - Commercial tools offer limited support (why?)
 - The complexity of the UML metamodel often leaves place to interpretations => incompatibilities
 - Until UML 2.0 no diagram interchange

Conclusions – UML summary

- UML – modeling language to be used throughout the entire software lifecycle
- Capture requirements
 - Use cases
 - Sequence diagrams
- Structure aspects
 - OO inspired definition
 - Component support
- Behavior aspects
 - State machines – for reactive behavior
 - Actions – in general
- Deployment aspects
 - Component/deployment diagrams

UML summary (2/2)

- To be as flexible as possible
 - UML offers extension mechanisms, profiles
 - Using profile UML can be transformed in a DSL

- Tool support
 - Lots of commercial/non-commercial tools exist
 - Various functionalities offered
 - Tool interchange exists, but lots are still to be done

Impact on research activity

Researchers attitude evolved:

- Hostility: received with skepticism, and (violent) critics
- Resign: very used in research papers, projects, books
- Pragmatism: taken as it is, used as a bridge with the industrial world
- Often the main focus of conferences, workshops, basic research, more as a means than as a goal

The bad news is that ...

- The various notations within UML are not perfectly coordinated
- Often, different tools interpret the UML standard differently
- The unique modeling language is in fact a set of dialects

The good news is that ...

- We have a language allowing to design and model various aspects of systems
- This language is standardized and supported by various tools

- The tool support and interoperability improves in time, as UML, OCL, and XML are still relatively young standards

Requirements Elicitation

Use Case & Scenarios

These slides are based on the Chapter 4 of Bruegge's book
OOSE using UML, Patterns and Java

Requirement, requirements engineering, requirements elicitation and analysis

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams, in *Mostly Harmless*

- A **requirement** is a feature that the system must have or a constraint that it must satisfy to be accepted by the client.
- **Requirements engineering** aims at defining the requirements of the system under construction. **Requirements engineering** includes two main activities:
- **requirements elicitation**, which results in the specification of the system that the client understands, and
- **analysis**, which results in an analysis model that the developers can unambiguously interpret.

Requirement, requirements engineering, requirements elicitation and analysis_2

- **Requirements elicitation** is the more challenging of the two because it requires the collaboration of several groups of participants with different backgrounds.
 - On the one hand, the client and the **users are experts in their domain** and **have a general idea of what the system should do**, but they often **have little experience in software development**.
 - On the other hand, the **developers have experience in building systems**, but often have **little knowledge of the everyday environment of the users**.
- **Scenarios and use cases** provide **tools for bridging this gap**.
- A **scenario** describes an example of system use in terms of a series of interactions between the user and the system.
- A **use case** is an abstraction that describes a **group of related scenarios**. Both scenarios and use cases are written in **natural language**, a form that is understandable to the user.

Requirements elicitation - definition:

- In **requirements engineering**, **requirements elicitation** is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders. The practice is also sometimes referred to as "**requirement gathering**" [Wikipedia].
- Commonly used **elicitation processes** are the stakeholder meetings or interviews.

First step in identifying the Requirements: System identification

- **Two questions need to be answered:**
 1. How can we identify the purpose of a system?
 2. What is inside, what is outside the system?
- These two questions are answered during requirements elicitation and analysis
- **Requirements elicitation:**
 - Definition of the system in terms understood by the customer ("Requirements specification")
- **Analysis:**
 - Definition of the system in terms understood by the developer (Technical specification, "Analysis model")
- **Requirements Process:** Contains the activities Requirements Elicitation and Analysis.

Types of Requirements

- **Functional requirements**
 - Describe the interactions between the system and its environment independent from the implementation
“An operator must be able to define a new game.”
- **Nonfunctional requirements**
 - Aspects not directly related to functional behavior.
“The response time must be less than 1 second”
- **Constraints**
 - Imposed by the client or the environment
 - “The implementation language must be Java”
 - Called “**Pseudo requirements**” in the textbook.

Functional vs. Nonfunctional Requirements

Functional Requirements

- Describe user tasks that the system needs to support
- Phrased as actions
 - “Advertise a new league”
 - “Schedule tournament”
 - “Notify an interest group”

Nonfunctional Requirements

- Describe properties of the system or the domain
- Phrased as constraints or negative assertions
 - “All user inputs should be acknowledged within 1 second”
 - “A system crash should not result in data loss”.

Types of Nonfunctional Requirements

Quality requirements

- **Usability** - the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component.
- **Reliability** - the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Recently, this category is often replaced by **dependability**, which is the property of a computer system such that reliance can justifiably be placed on the service it delivers. **Dependability** includes **robustness** and **safety**.
- **Performance requirements** - are concerned with quantifiable attributes of the system, such as: **Response time**, **Throughput**, **Availability**

Types of Nonfunctional Requirements_2

Quality requirements

- **Supportability requirements** - are concerned with the ease of changes to the system after deployment, including for example, **adaptability** (the ability to change the system to deal with additional application domain concepts), **maintainability** (the ability to change the system to deal with new technology or to fix defects), and internationalization.

Constraints (Pseudo requirements)

- **Implementation requirements** - are constraints on the implementation of the system, including the use of specific tools, programming languages, or hardware platforms.

Types of Nonfunctional Requirements_3

Constraints (Pseudo requirements)

- **Implementation requirements** - are constraints on the implementation of the system, including the use of specific tools, programming languages, or hardware platforms.
- **Interface requirements** - are constraints imposed by external systems, including legacy systems and interchange formats.
- **Operations requirements** - are constraints on the administration and management of the system in the operational setting.
- **Packaging requirements** - are constraints on the actual delivery of the system (e.g., constraints on the installation media for setting up the software).

Types of Nonfunctional Requirements_4

**Constraints
(Pseudo
requirements)**

Legal requirements - are concerned with licensing, regulation, and certification issues. An example of a legal requirement is that software developed for the U.S. federal government must comply with Section 508 of the Rehabilitation Act of 1973, requiring that government information systems must be accessible to people with disabilities.

Nonfunctional Requirements (Quality): Examples

- “Spectators must be able to watch a match without prior registration and without prior knowledge of the match.”

➤ *Usability Requirement*

- “The system must support 10 parallel tournaments”

➤ *Performance Requirement*

- “The operator must be able to add new games without modifications to the existing system.”

What should not be in the Requirements?

- System structure, implementation technology
 - Development methodology
 - Development environment
 - Implementation language
 - Reusability
-
- It is desirable that none of these above are constrained by the client. Fight for it!

Requirements Validation

Requirements' validation is a quality assurance step, usually performed after requirements elicitation or after analysis

- **Correctness:**
 - The requirements represent the client's view
- **Completeness:**
 - All possible scenarios, in which the system can be used, are described
- **Consistency:**
 - There are no requirements that contradict each other.

Requirements Validation (2)

- Clarity:
 - Requirements can only be interpreted in one way
- Realism:
 - Requirements can be implemented and delivered
- Traceability:
 - Each system behavior can be traced to a set of functional requirements
- Problems with requirements validation:
 - Requirements change quickly during requirements elicitation
 - Inconsistencies are easily added with each change
 - Tool support is needed!

We can specify Requirements for “Requirements Management”

- **Functional requirements:**
 - Store the requirements in a shared repository
 - Provide multi-user access to the requirements
 - Automatically create a specification document from the requirements
 - Allow change management of the requirements
 - Provide traceability of the requirements throughout the artifacts of the system.

Different Types of Requirements Elicitation

- **Greenfield Engineering**
 - Development starts from scratch, no prior system exists, requirements come from end users and clients
 - Triggered by user needs
- **Re-engineering**
 - Re-design and/or re-implementation of an existing system using newer technology
 - Triggered by technology enabler
- **Interface Engineering**
 - Provision of existing services in a new environment
 - Triggered by technology enabler or new market needs

Prioritizing requirements

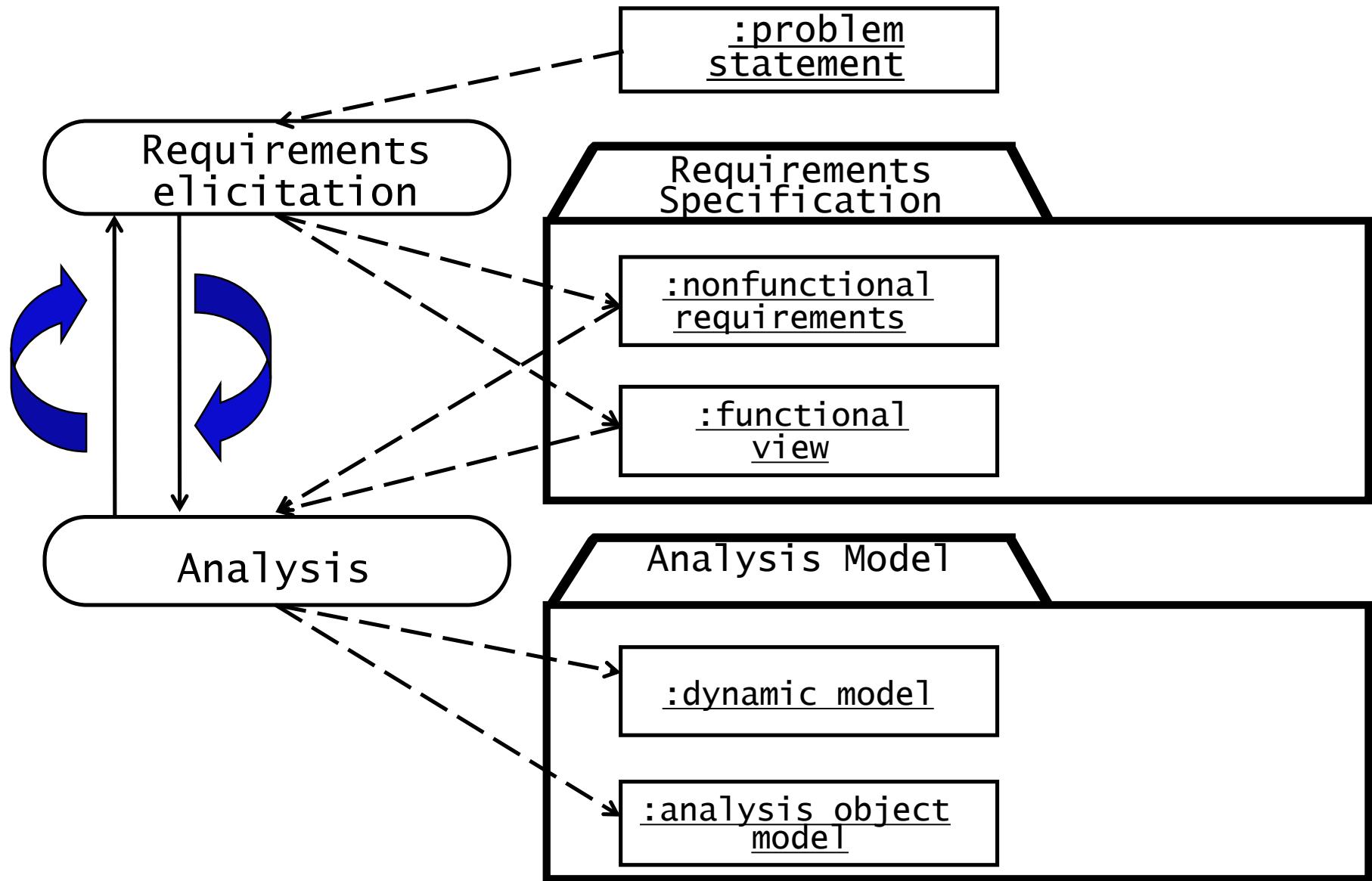
- **High priority**
 - Addressed during analysis, design, and implementation
 - A high-priority feature must be demonstrated
- **Medium priority**
 - Addressed during analysis and design
 - Usually demonstrated in the second iteration
- **Low priority**
 - Addressed only during analysis
 - Illustrates how the system is going to be used in the future with not yet available technology

Requirements Specification vs Analysis Model

Both focus on the requirements from the user's view of the system

- The **requirements specification** uses natural language (derived from the problem statement)
- The **analysis model** uses a formal or semi-formal notation (we use UML)

Requirements Process



Techniques to elicit Requirements

- Bridging the gap between end user and developer:
 - **Questionnaires:** Asking the end user a list of pre-selected questions
 - **Task Analysis:** Observing end users in their operational environment
 - **Scenarios:** Describe the use of the system as a series of interactions between a concrete end user and the system
 - **Use cases:** Abstractions that describe a class/group/category of scenarios.

Scenario-Based Design

Scenario-Based Design: The use of scenarios in a software lifecycle activity

Scenarios can have many different uses during the software lifecycle:

- ***Requirements Elicitation:*** As-is scenario, visionary scenario
- ***Client Acceptance Test:*** Evaluation scenario
- ***System Deployment:*** Training scenario

Types of Scenarios

- **As-is scenario:**
 - Describes a current situation. Usually used in re-engineering projects. The user describes the system
- **Visionary scenario:**
 - Describes a future system. Usually used in Greenfield engineering and reengineering projects
 - Can often not be done by the user or developer alone

Additional Types of Scenarios (2)

- Evaluation scenario:
 - Description of a user task against which the system is to be evaluated.
- Training scenario:
 - A description of the step-by-step instructions that guide a novice user through a system

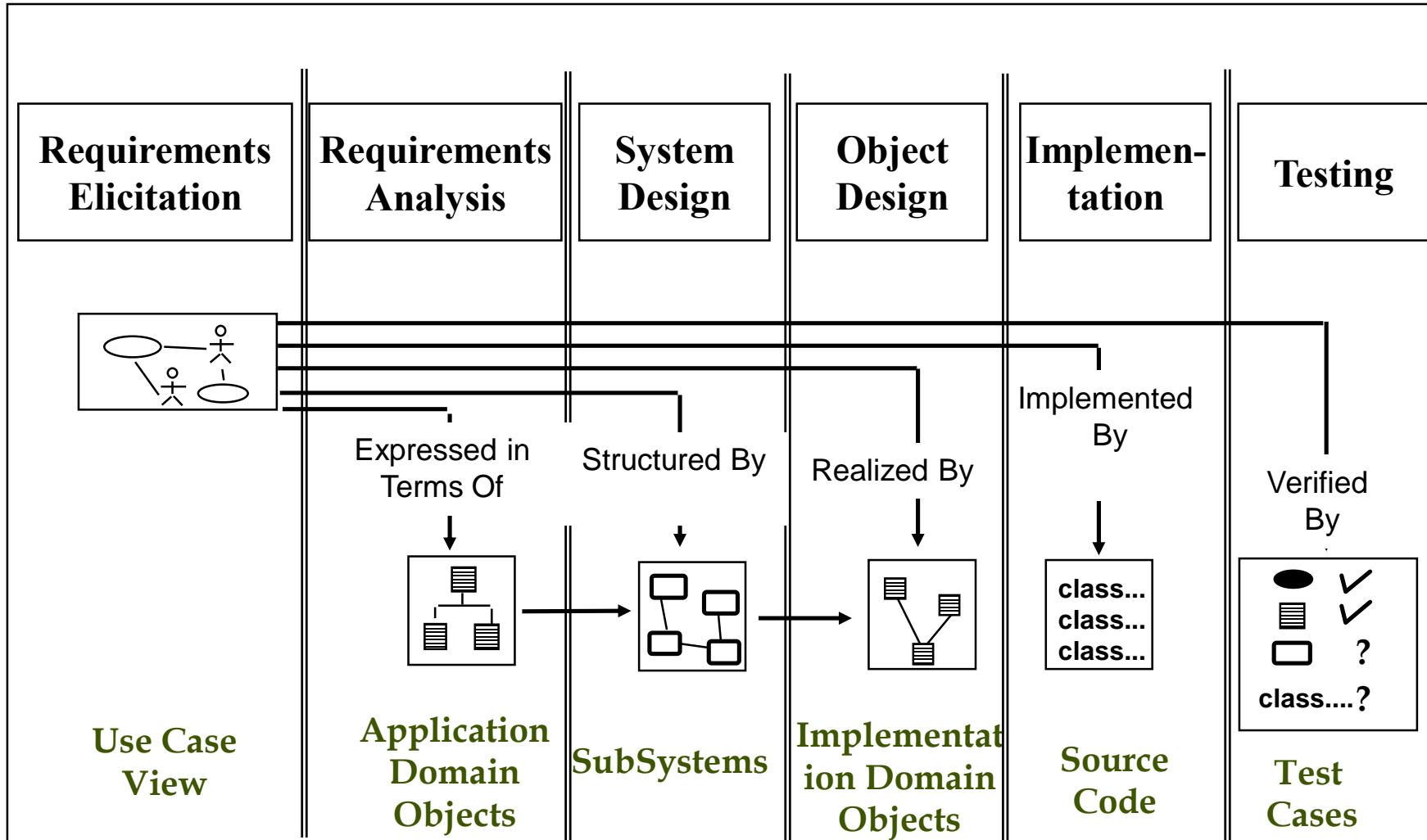
How do we find scenarios?

- Don't expect the client to be verbal if the system does not exist
 - Client understands problem domain, not the solution domain.
- Don't wait for information even if the system exists
 - "What is obvious does not need to be said"
- Engage in a dialectic approach
 - You help the client to formulate the requirements
 - The client helps you to understand the requirements
 - The requirements evolve while the scenarios are being developed

Heuristics for finding scenarios

- Ask yourself or the client the following questions:
 - What are the primary tasks that the system needs to perform?
 - What data will the actor create, store, change, remove or add in the system?
 - What external changes does the system need to know about?
 - What changes or events will the actor of the system need to be informed about?
- However, don't rely on questions and questionnaires alone
- Insist on task observation if the system already exists (interface engineering or reengineering)
 - Ask to speak to the end user, not just to the client
 - Expect resistance and try to overcome it.

Software Lifecycle Activities



Requirements Analysis Document Template

1. Introduction
2. Current system
3. Proposed system
 - 3.1 Overview
 - 3.2 Functional requirements
 - 3.3 Nonfunctional requirements
 - 3.4 Constraints ("Pseudo requirements")
 - 3.5 System models
 - 3.5.1 Scenarios
 - 3.5.2 Use case model
 - 3.5.3 Object model
 - 3.5.3.1 Data dictionary
 - 3.5.3.2 Class diagrams
 - 3.5.4 Dynamic models
 - 3.5.5 User interface
 4. Glossary



Requirement Document Desiderata

- **Correct**
 - The client agrees that it represents the reality
- **Complete**
 - The client agrees that all relevant scenarios are described
- **Consistent**
 - No two requirements of the specification contradict each other.
- **Unambiguous**
 - There is only 1 way to interpret the specification
- **Realistic**
 - All features can be implemented subject to all constraints
- **Verifiable**
 - Requirements & constraints are testable
- **Traceable**
 - For each system function there is some requirement[s]
 - For each requirement there is some system function[s]

Identify Scenarios

Scenario

- A description of what actors do as they use the system
- For each scenario, there is a:
 - Use case
 - Acceptance test case

Questions for identifying scenarios:

- What **tasks** want actors to perform the system?
- What **data** does the actor need?
 - Who creates, modifies, removes that data?
- Which **external** changes affect the **system's state**?
 - How is the change communicated to the system? (what actor?)
 - Under what circumstances?

Identify Scenarios - Warehouse on Fire

Scenario name warehouseOnFire

Participating actor instances bob, alice:FieldOfficer
john:Dispatcher

- Flow of events**
1. Bob, driving down main street in his patrol car, notices smoke coming out of a warehouse. His partner, Alice, activates the "Report Emergency" function from her FRIEND laptop.
 2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene, given that the area appears to be relatively busy. She confirms her input and waits for an acknowledgment.

Identify Scenarios - cont

Flow of events

3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice.
4. Alice receives the acknowledgment and the ETA.

Identify Actors

Questions whose answers identify the actors:

- Which user groups does the system support to **do their work?**
- Which user groups execute the system's **primary functions?**
- Which user groups execute the system's **secondary functions?**
 - E.g., maintain or administer the system
- With which **external systems** does the system interact?
 - E.g., hardware or software

Identifying Use Cases

Use case name

ReportEmergency

Participating Actors

Initiated by FieldOfficer
Communicates with Dispatcher

Flow of events

1. The FieldOfficer activates the “Report Emergency” function of her terminal.
2. FRIEND responds by presenting a form to the FieldOfficer.
3. The FieldOfficer completes the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form.
4. FRIEND receives the form and notifies the Dispatcher.

Identifying Use Cases_2

Flow of events

5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report.
6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.

Entry condition

The FieldOfficer is logged into FRIEND.

Exit conditions

- The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR
- The FieldOfficer has received an explanation indicating why the transaction could not be processed

Identifying Use Cases_3

Quality requirements

- The FieldOfficer's report is acknowledged within 30 seconds.
- The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Simple Use Case Writing Guide

- Use cases should be named with verb phrases. The name of the use case should indicate what the user is trying to accomplish (e.g., ReportEmergency, OpenIncident).
- Actors should be named with noun phrases (e.g., FieldOfficer, Dispatcher, Victim).
- The boundary of the system should be clear. Steps accomplished by the actor and steps accomplished by the system should be distinguished (e.g., system actions are indented to the right).
- Use case steps in the flow of events should be phrased in the active voice. This makes it explicit who accomplished the step.
- The causal relationship between successive steps should be clear.

Simple Use Case Writing Guide_cont

- A use case should describe a complete user transaction (e.g., the ReportEmergency use case describes all the steps between initiating the emergency reporting and receiving an acknowledgment).
- Exceptions should be described separately.
- A use case should not describe the user interface of the system. This takes away the focus from the actual steps accomplished by the user and is better addressed with visual mock-ups (e.g., the ReportEmergency only refers to the “Report Emergency” function, not the menu, the button, nor the actual command that corresponds to this function).

Simple Use Case Writing Guide_cont

- A use case description should not exceed two or three pages in length. Otherwise, use include and extend relationships to decompose it in smaller use cases.

Refine the Use case view

Iterate the following steps:

- Define a **horizontal** slice (i.e., many high-level scenarios) to define the scope of the system.
- **Validate** the use case view with the user.
- Detail a **vertical** slice.
- **Validate** with the user.
- In extreme programming, you design, implement, test [and integrate] this vertical slice, before tackling the next vertical slice.

Identify Relationships Among Actors & Use cases

- **Access control** – which actors' access which functionality – is represented with use cases.
- **Extend relation**
 - ConnectionDown use case:
 - Entry condition:
 - Connection between FieldOfficer & Dispatcher fails before EmergencyReport is submitted
 - Event flow:
 - FieldOfficer notifies Dispatcher via voice channel
- **Heuristics:**
 - Use **extend** relation for **exceptions, optional, or rare behavior**.
 - Use **include** relation for behavior **common to 2 or more use cases**.

Identify Initial Analysis Objects

It may be one of the system objects if it is a:

- Term developers/users need to clarify to understand a use case;
- Recurring noun in the use case model;
- Real-world object the system needs to track (e.g., FieldOfficer)
- Real-world process the system needs to track (e.g., EmergencyOperationPlan)
- Use cases (e.g., ReportEmergency)
- Application domain term

Identify Nonfunctional Requirements

Investigate the following:

- User interface – level of expertise of user
- Documentation – User manual? The system design should be documented. What about development process?
- Hardware considerations – What assumptions are made?
- Error handling – philosophy (e.g., tolerates failure of any single system component).
- Physical environment – what assumptions are made about power supply, cooling, etc.
- Physical security
- Resource – constraints on power, memory, etc.

Example of questions for eliciting nonfunctional requirements

Supportability

(including maintainability and portability)

- What are the foreseen extensions to the system?

- Who maintains the system?

- Are there plans to port the system to different software or hardware environments?

- Are there constraints on the hardware platform?

- Are constraints imposed by the maintenance team?

- Are constraints imposed by the testing team?

- Should the system interact with any existing systems?

- How are data exported/imported into the system?

- What standards in use by the client should be supported by the system?

Implementation

Interface

Example of questions for eliciting nonfunctional requirements

Operation

- Who manages the running system?

Packaging

- Who installs the system?
- How many installations are foreseen?
- Are there time constraints on the installation?
- How should the system be licensed?
- Are any liability issues associated with system failures?
- Are any royalties or licensing fees incurred by using specific algorithms or components?

In practice, analysts use a taxonomy of nonfunctional requirements (e.g., the FURPS+) to generate check lists of questions to help the client and the developers focus on the nonfunctional aspects of the system. **Functionality, Usability, Reliability, Performance, and Supportability**

To Do List

- Project definition
 - Complete a project description that the customer agrees to.
- Research
 - Initial identification of actors & event flows
 - Document unresolved issues/ambiguities
- Create a draft Requirements Analysis Document (RAD)
- While (there are unresolved items) do:
 - Resolve an item on list of issues/ambiguities;
 - Modify RAD accordingly;
 - Insert new, more detailed unresolved items;

Requirements Elicitation: Difficulties and Challenges

- Communicate accurately about the domain and the system
 - People with different backgrounds must collaborate to bridge the gap between end users and developers
 - Client and end users have application domain knowledge
 - Developers have solution domain knowledge
- Identify an appropriate system (Defining the system boundary)
- Provide an unambiguous specification
- Leave out unintended features

Example of an Ambiguous Specification

During a laser experiment, a laser beam was directed from earth to a mirror on the Space Shuttle Discovery

The laser beam was supposed to be reflected back towards a mountain top 10,023 feet high

The operator entered the elevation as "10023"

The light beam never hit the mountain top
What was the problem?

Example of an Ambiguous Specification

During a laser experiment, a laser beam was directed from earth to a mirror on the Space Shuttle Discovery

The laser beam was supposed to be reflected back towards a mountain top 10,023 feet high

The operator entered the elevation as "10023"

The light beam never hit the mountain top
What was the problem?

The computer interpreted the number in miles...

Example of an Unintended Feature

From the News: London underground train leaves station without driver!

What happened?

- A passenger door was stuck and did not close
- The driver left his train to close the passenger door
 - He left the driver door open
 - He relied on the specification that said the train does not move if at least one door is open
- When he shut the passenger door, the train left the station without him



Example of an Unintended Feature

From the News: London underground train leaves station without driver!

What happened?

- A passenger door was stuck and did not close
- The driver left his train to close the passenger door
 - He left the driver door open
 - He relied on the specification that said the train does not move if at least one door is open
- When he shut the passenger door, the train left the station without him
- The driver door was not treated as a door in the source code!



Scenario example from earlier: Warehouse on Fire

- Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.
- Alice enters the address of the building into her wearable computer , a brief description of its location (i.e., north west corner), and an emergency level.
- She confirms her input and waits for an acknowledgment.
- John, the dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and sends the **estimated arrival time (ETA)** to Alice.
- Alice received the acknowledgment and the ETA.

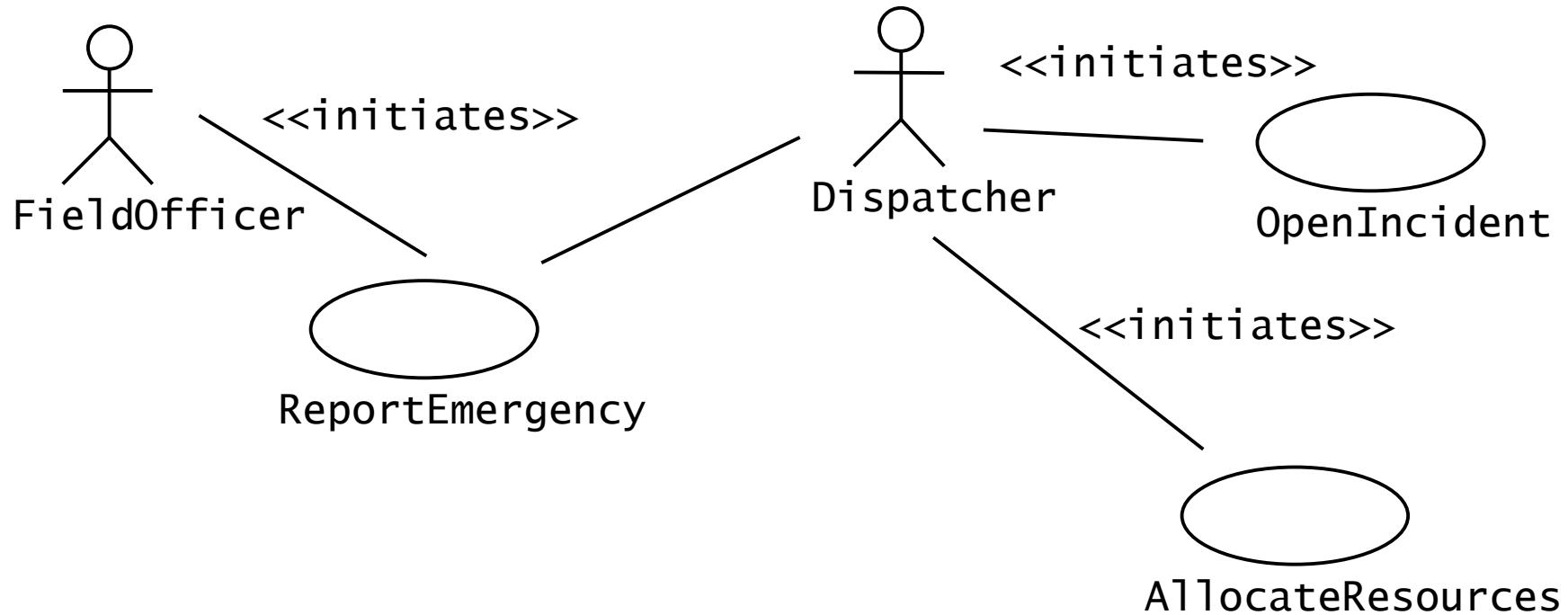
Observations about Warehouse on Fire Scenario

- Concrete scenario
 - Describes a single instance of reporting a fire incident.
 - Does not describe all possible situations in which a fire can be reported.
- Participating actors
 - Bob, Alice and John

After the scenarios are formulated

- Find all the use cases in the scenario that specify all instances of how to report a fire
 - Example: “Report Emergency” in the first paragraph of the scenario is a candidate for a use case
- Describe each of these use cases in more detail
 - Participating actors
 - Describe the entry condition
 - Describe the flow of events
 - Describe the exit condition
 - Describe exceptions
 - Describe nonfunctional requirements

Use Case View for Incident Management



How to find Use Cases

- Select a narrow vertical slice of the system (i.e. one scenario)
 - Discuss it in detail with the user to understand the user's preferred style of interaction
- Select a horizontal slice (i.e. many scenarios) to define the scope of the system.
 - Discuss the scope with the user
- Use illustrative prototypes (mock-ups) as visual support
- Find out what the user does
 - Task observation (Good)
 - Questionnaires (Bad)

Use Case Example: ReportEmergency

- Use case name: ReportEmergency
- Participating Actors:
 - Field Officer (Bob and Alice in the Scenario)
 - Dispatcher (John in the Scenario)
- Exceptions:
 - The FieldOfficer is notified immediately if the connection between terminal and central is lost.
 - The Dispatcher is notified immediately if the connection between a FieldOfficer and central is lost.
- Flow of Events: **on next slide.**
- Special Requirements:
 - The Field Officer's report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Use Case Example: ReportEmergency

Flow of Events

1. The **FieldOfficer** activates the “Report Emergency” function of her terminal. FRIEND responds by presenting a form to the officer.
2. The FieldOfficer fills the form, by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes a response to the emergency situation. Once the form is completed, the FieldOfficer submits the form, and the **Dispatcher** is notified.
3. The Dispatcher creates an Incident in the database by invoking the OpenIncident use case. He selects a response and acknowledges the report.
4. The FieldOfficer receives the acknowledgment and the selected response.

Order of steps when formulating use cases

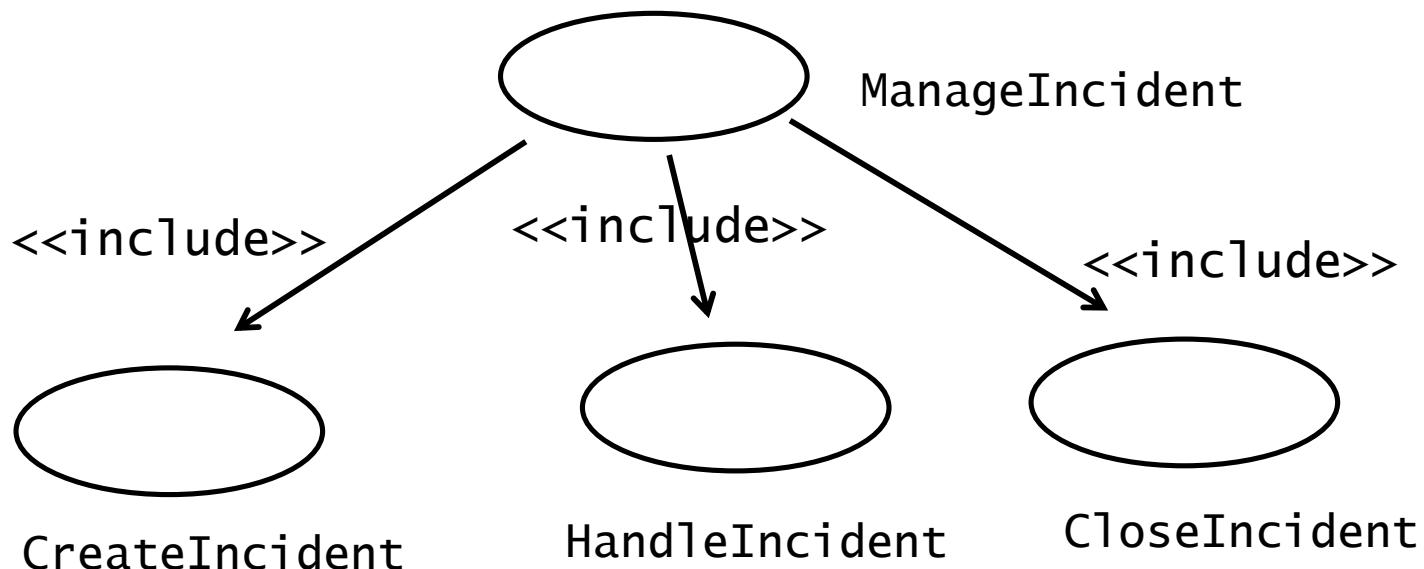
- First step: Name the use case
 - Use case name: ReportEmergency
- Second step: Find the actors
 - Generalize the concrete names ("Bob") to participating actors ("Field officer")
 - Participating Actors:
 - Field Officer (Bob and Alice in the Scenario)
 - Dispatcher (John in the Scenario)
- Third step: Concentrate on the flow of events
 - Use informal natural language

Use Case Associations

- Dependencies between use cases are represented with use case associations
- Associations are used to reduce complexity
 - Decompose a long use case into shorter ones
 - Separate alternate flows of events
 - Refine abstract use cases
- Types of use case associations
 - Includes
 - Extends
 - Generalization

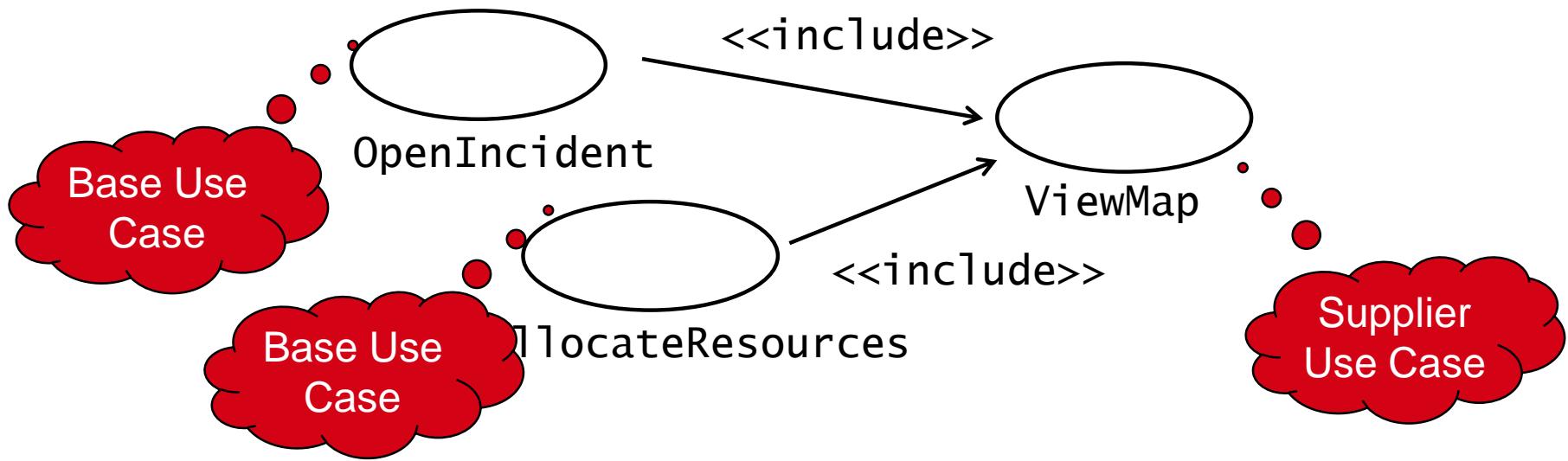
<<include>>: Functional Decomposition

- Problem:
 - A function in the original problem statement is too complex
- Solution:
 - Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into shorter use cases



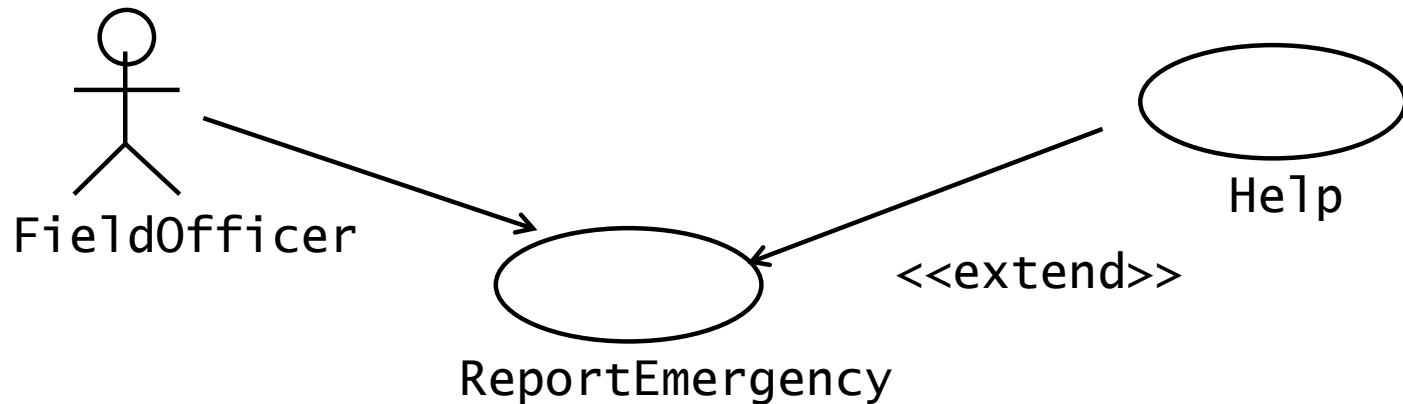
<<include>>: Reuse of Existing Functionality

- **Problem:** There are overlaps among use cases. How can we *reuse* flows of events instead of duplicating them?
- **Solution:** The *includes association* from use case A to use case B indicates that an instance of use case A performs all the behavior described in use case B ("A delegates to B")
- **Example:** Use case "ViewMap" describes behavior that can be used by use case "OpenIncident" ("ViewMap" is factored out)



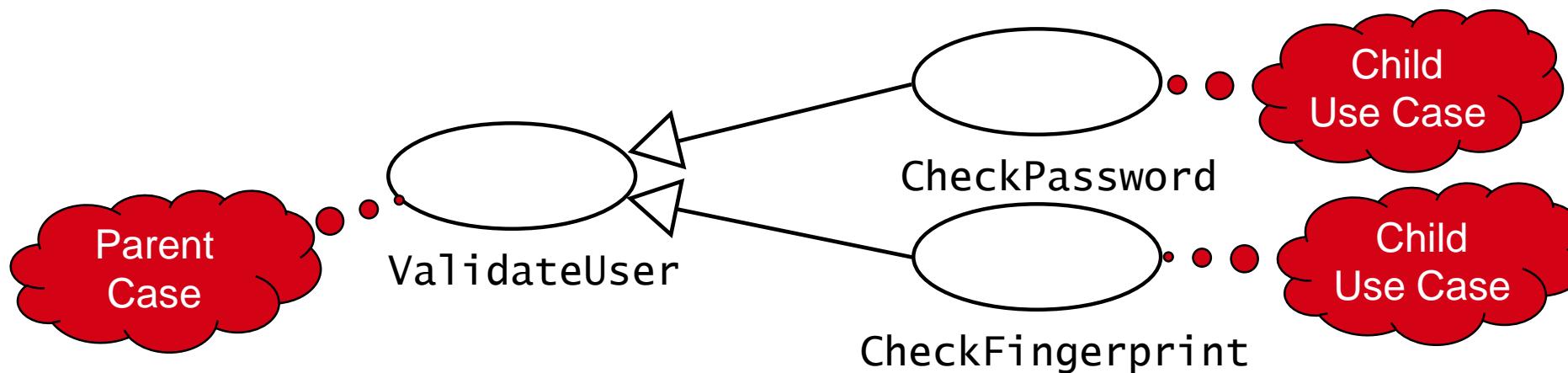
<<extend>> Association for Use Cases

- **Problem:** The functionality in the original problem statement needs to be extended.
- **Solution:** An *extend association* from use case A to use case B
- **Example:** “ReportEmergency” is complete by itself, but **may** be extended by use case “Help” for a scenario in which the user requires help



Generalization in Use Cases

- **Problem:** We want to factor out common (but not identical) behavior.
- **Solution:** The child use cases inherit the behavior and meaning of the parent use case and add or override some behavior.
- **Example:** “ValidateUser” is responsible for verifying the identity of the user. The customer might require two realizations: “CheckPassword” and “CheckFingerprint”



Another Use Case Example

Actor Bank Customer

- Person who owns one or more Accounts in the Bank.

Withdraw Money

- The Bank Customer specifies an Account and provides credentials to the Bank proving that s/he is authorized to access the Bank Account.
- The Bank Customer specifies the amount of money s/he wishes to withdraw.
- The Bank checks if the amount is consistent with the rules of the Bank and the state of the Bank Customer's account. If that is the case, the Bank Customer receives the money in cash.

Use Case Attributes

Use Case Withdraw Money Using ATM

Initiating actor:

- Bank Customer

Preconditions:

- Bank Customer has opened a Bank Account with the Bank **and**
- Bank Customer has received an ATM Card and PIN

Postconditions:

- Bank Customer has the requested cash **or**
- Bank Customer receives an explanation from the ATM about why the cash could not be dispensed

Use Case Flow of Events

Actor steps

1. The Bank Customer inputs the card into the ATM.
3. The Bank Customer types in PIN.
5. The Bank Customer selects an account.
7. The Bank Customer inputs an amount.

System steps

2. The ATM requests the input of a four-digit PIN.
4. If several accounts are recorded on the card, the ATM offers a choice of the account numbers for selection by the Bank Customer
6. If only one account is recorded on the card or after the selection, the ATM requests the amount to be withdrawn.
8. The ATM outputs the money and a receipt and stops the interaction.

Use Case Exceptions

Actor steps

1. The Bank Customer inputs her card into the ATM. **[Invalid card]**
3. The Bank Customer types in PIN. **[Invalid PIN]**
5. The Bank Customer selects an account .
7. The Bank Customer inputs an amount. **[Amount over limit]**

[Invalid card]

The ATM outputs the card and stops the interaction.

[Invalid PIN]

The ATM announces the failure and offers a 2nd try as well as canceling the whole use case. After 3 failures, it announces the possible retention of the card. After the 4th failure it keeps the card and stops the interaction.

[Amount over limit]

The ATM announces the failure and the available limit and offers a second try as well as canceling the whole use case.

Guidelines for Formulation of Use Cases (1)

- **Name**
 - Use a verb phrase to name the use case.
 - The name should indicate what the user is trying to accomplish.
 - Examples:
 - “Request Meeting”, “Schedule Meeting”, “Propose Alternate Date”
- **Length**
 - A use case description should not exceed 1-2 pages. If longer, use include relationships.
 - A use case should describe a complete set of interactions.

Guidelines for Formulation of Use Cases (2)

Flow of events:

- Use the active voice. Steps should start either with “The Actor” or “The System ...”.
- The causal relationship between the steps should be clear.
- All flow of events should be described (not only the main flow of event).
- The boundaries of the system should be clear. Components external to the system should be described as such.
- Define important terms in the glossary.

Example of a **badly written** Use Case

“The driver arrives at the parking gate, the driver receives a ticket from the distributor, the gate is opened, the driver drives through.”

- What is wrong with this use case?

Example of a **badly written** Use Case

“The driver arrives at the parking gate, the driver receives a ticket from the distributor, the gate is opened, the driver drives through.”

It contains no actors

It is not clear which action triggers the ticket being issued

Because of the passive form, it is not clear who opens the gate (The driver? The computer? A gate keeper?)

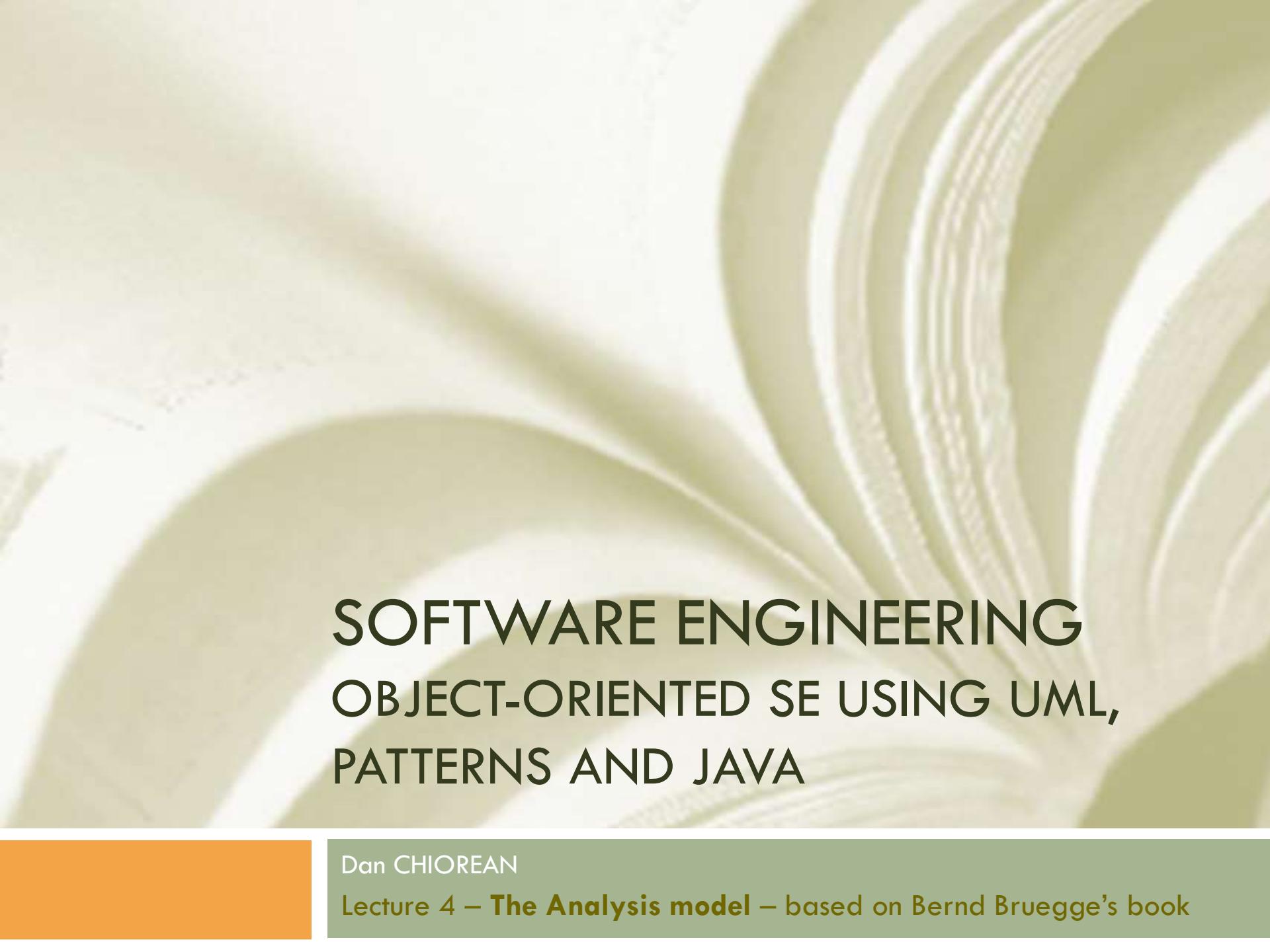
It is not a complete transaction. A complete transaction would also describe the driver paying for the parking and driving out of the parking lot.

How to write a use case (Summary)

- Name of Use Case
- Actors
 - Description of Actors involved in use case
- Entry condition
 - “This use case starts when...”
- Flow of Events
 - Free form, informal natural language
- Exit condition
 - “This use cases terminates when...”
- Exceptions
 - Describe what happens if things go wrong
- Special Requirements
 - Nonfunctional Requirements, Constraints

Summary

- Scenarios:
 - Great way to establish communication with client
 - Different types of scenarios: As-Is, visionary, evaluation and training
- Use cases
 - Abstractions of scenarios
- Use cases bridge the transition between functional requirements and objects.



SOFTWARE ENGINEERING OBJECT-ORIENTED SE USING UML, PATTERNS AND JAVA

Dan CHIOREAN

Lecture 4 – **The Analysis model** – based on Bernd Bruegge's book

An Overview of Analysis

- Analysis focuses on producing a model of the system, called **the analysis model, which is correct, complete, consistent and verifiable.**
- Analysis is different from requirements elicitation in that developers focus on structuring and formalizing the requirements elicited from users.
- The formalization leads to new insights and to discovery of errors in requirements.

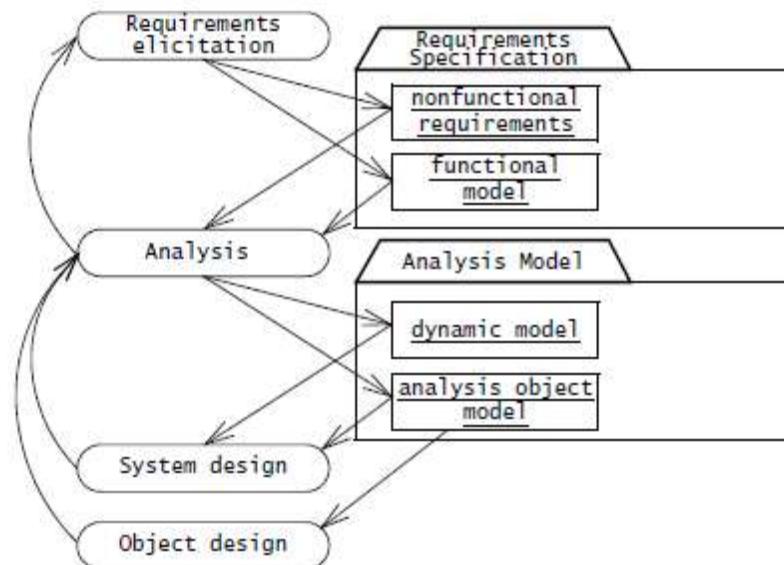


Figure 5-2 Products of requirements elicitation and analysis (UML activity diagram).

Challenges in Model Analysis

- Formalization helps identify areas of ambiguity as well as inconsistencies and omissions in a requirements specification. Once developers identify problems with the specification, they address them by eliciting more information from the users and the client. Requirements elicitation and analysis are iterative and incremental activities that occur concurrently

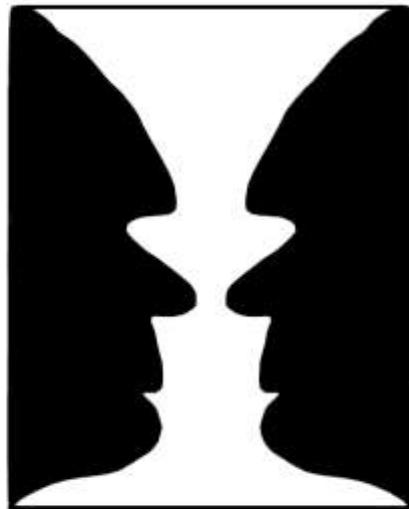
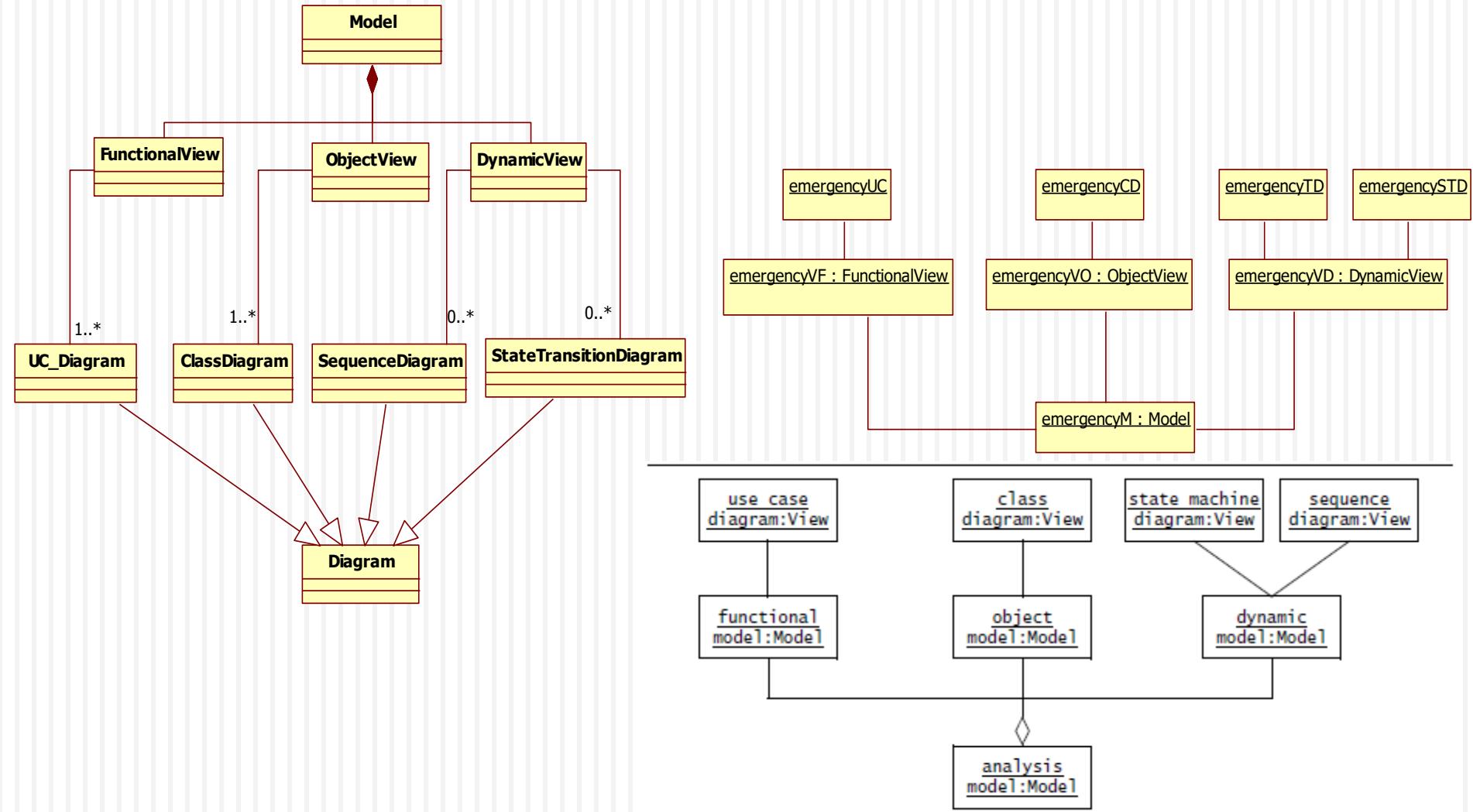


Figure 5-1 Ambiguity: what do you see?

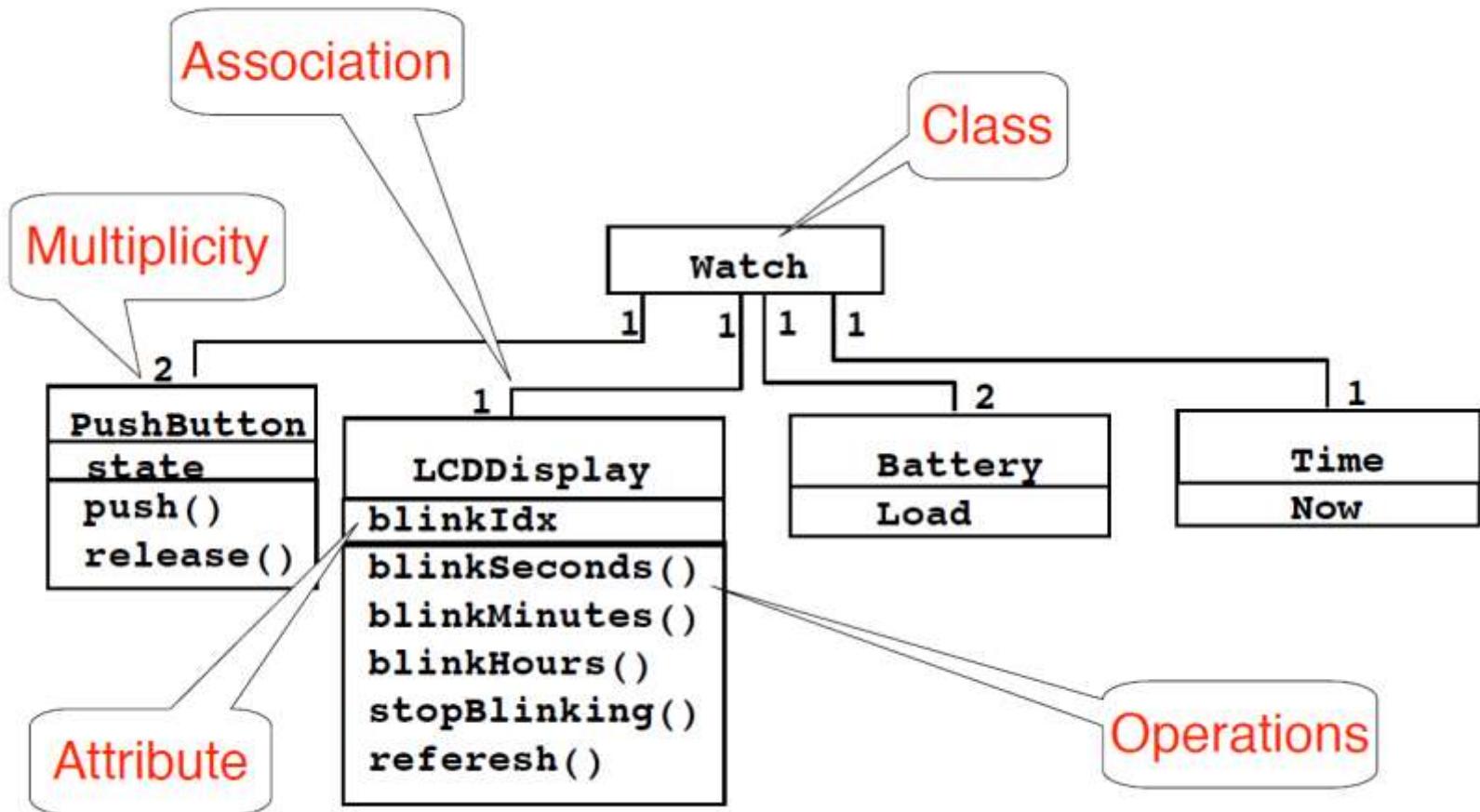
The Analysis Model



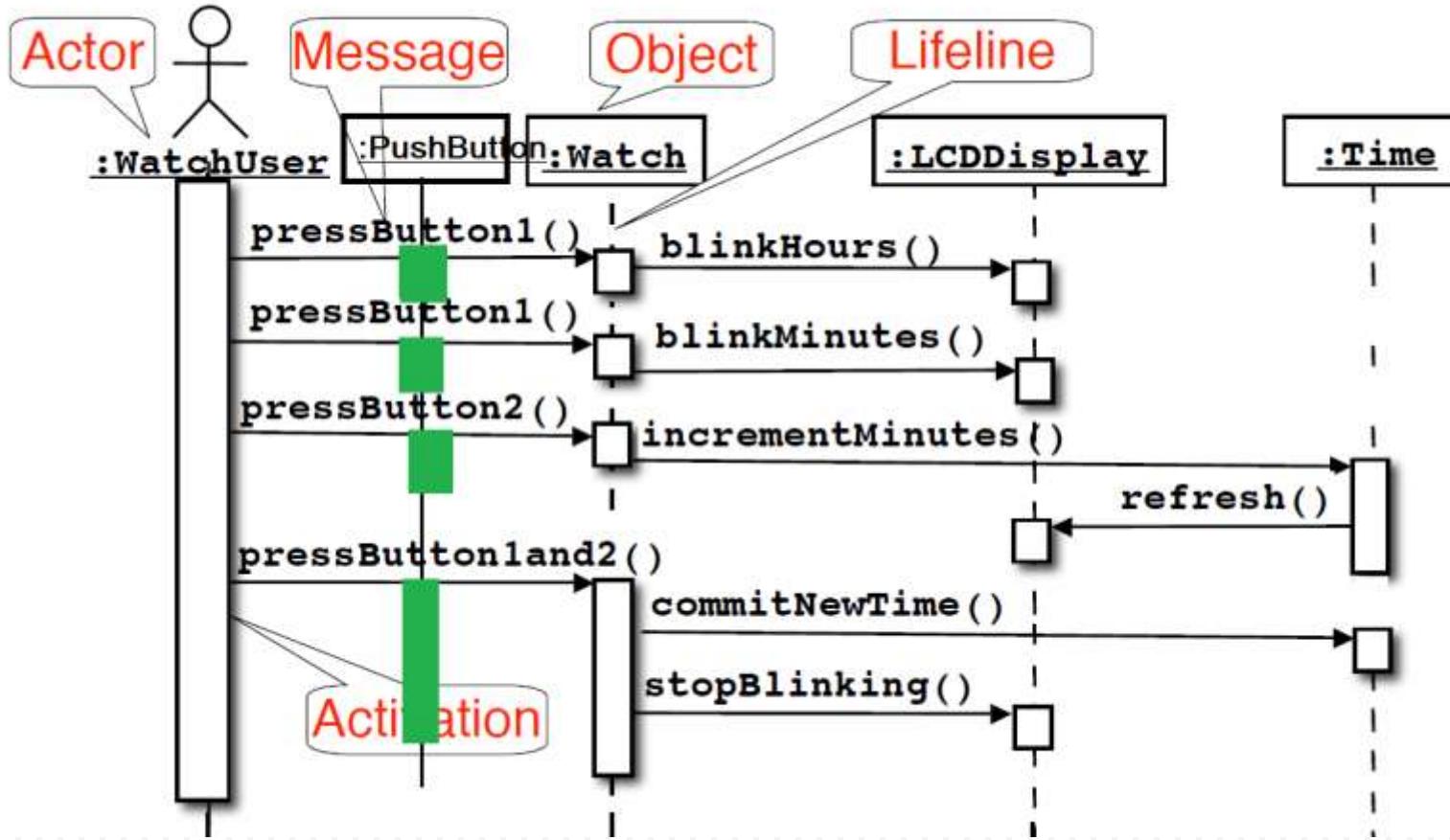
Analysis Object and Dynamic Views

- The Analysis Object View consists of **entity**, **boundary** and **control objects**
 - ▣ **Entity objects** represent the persistent information tracked by the system
 - ▣ **Boundary objects** represent the interaction between the actors and the system
 - ▣ **Control objects** are in charge of realizing use cases
- Example: In the 2Bwatch example, **Time** is the **entity object**, **PushButton** and **LCDDisplay** are **boundary objects**, **Watch** is a **control object** that represents the activity of changing the time by pressing combination of buttons

Analysis Object and Dynamic Views



Analysis Object and Dynamic Views



Analysis Object and Dynamic Views/2

- To distinguish between different types of objects, UML provides the stereotype mechanism to enable the developer to attach such meta-information to modeling elements.



Figure 5-5 Analysis classes for the 2Bwatch example.

Stereotype an UML Extension Mechanisms

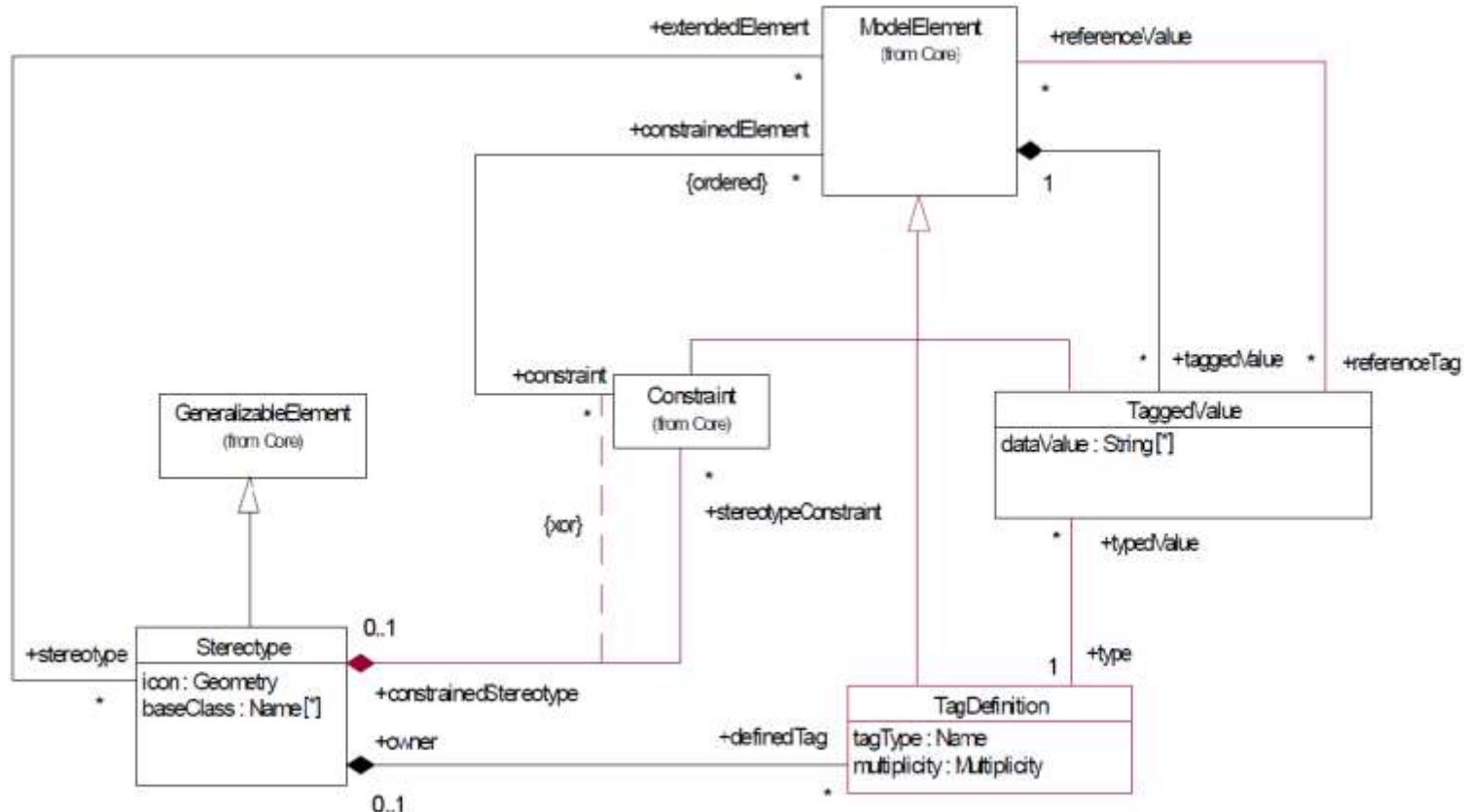
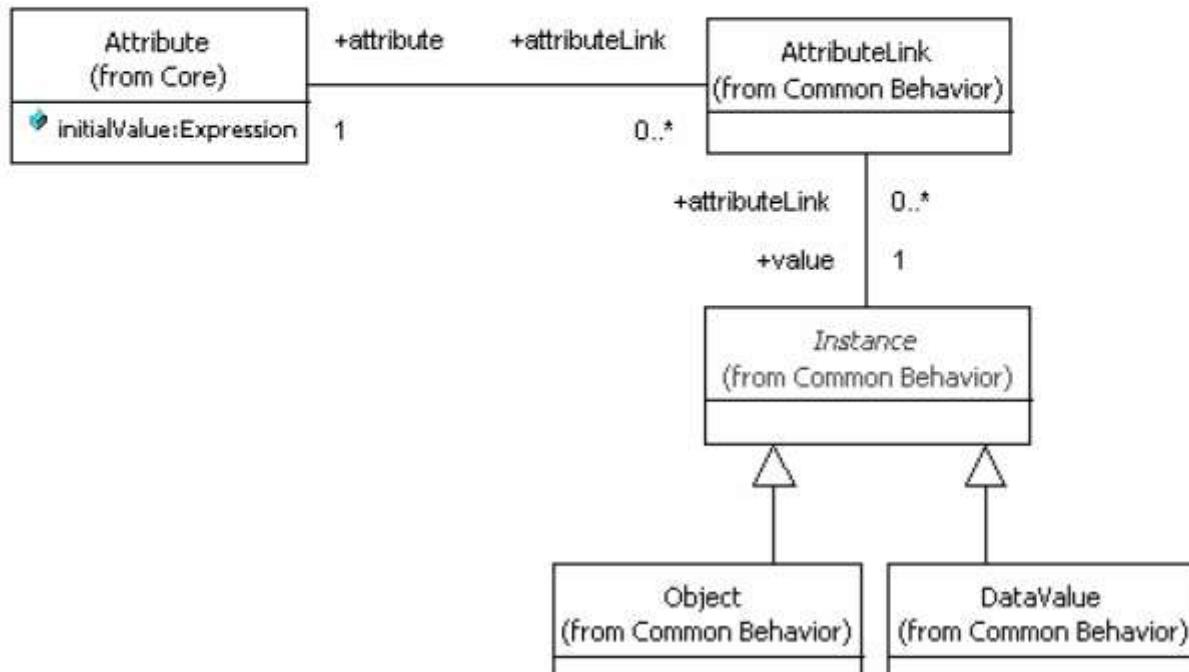


Figure 2-10 Extension Mechanisms

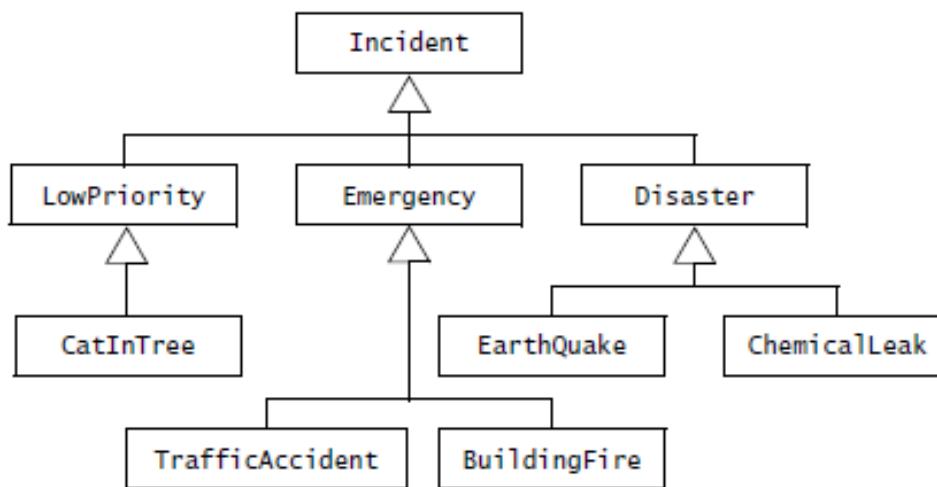
Managing Stereotypes in OCLE

```
context Attribute
  inv REQUIRED:
    if self.stereotype.name->includes ('REQUIRED')
      then self.attributeLink.value->any(v | v.isUndefined)->isEmpty
      else true
    endif
```



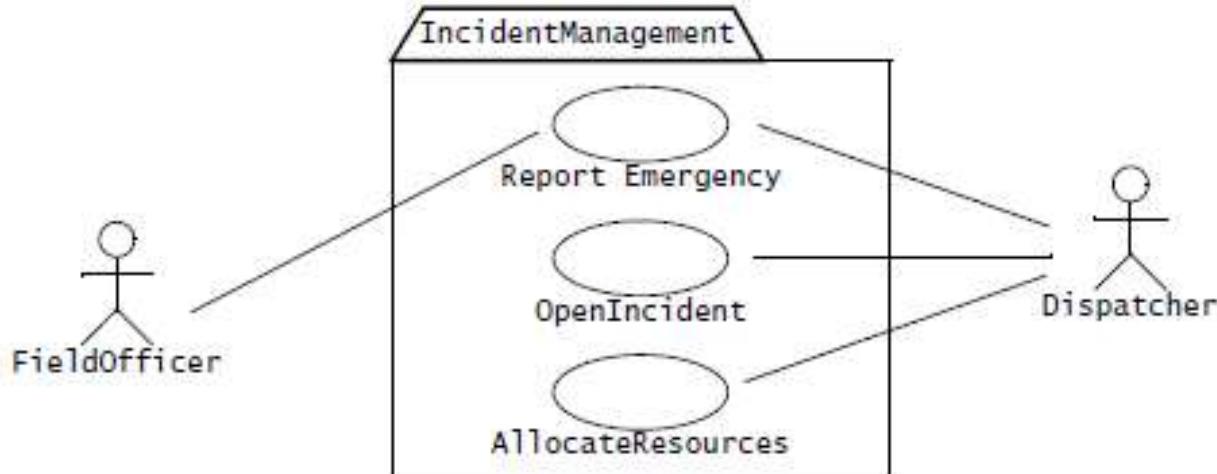
Analysis Object and Dynamic Views – Generalization and Specialization

- In both cases, generalization and specialization result in the specification of **inheritance relationships** between concepts. In some instances, modelers call inheritance relationships **generalization-specialization relationships**. In Bruegge's book, **inheritance** is used to denote the relationship and terms "**generalization**" and "**specialization**" to denote activities that find inheritance relationships.



The FRIEND system (book pp. 68)

- FRIEND – an accident management system in which field officers such as police officer or fire fighter have access to a wireless computer that enable them to interact with a dispatcher. The dispatcher in turn can visualize the current state of all its resources such as police cars or trucks, on a computer screen and dispatch a resource by issuing commands from a workstation.



The FRIEND system - cont

<i>Use case name</i>	ReportEmergency
<i>Participating actors</i>	Initiated by FieldOfficer Communicates with Dispatcher
<i>Flow of events</i>	<ol style="list-style-type: none">1. The FieldOfficer activates the “Report Emergency” function of her terminal.2. FRIEND responds by presenting a form to the FieldOfficer.3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form.4. FRIEND receives the form and notifies the Dispatcher.5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report.6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
<i>Entry condition</i>	<ul style="list-style-type: none">• The FieldOfficer is logged into FRIEND.
<i>Exit condition</i>	<ul style="list-style-type: none">• The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR• The FieldOfficer has received an explanation indicating why the transaction could not be processed.
<i>Quality requirements</i>	<ul style="list-style-type: none">• The FieldOfficer’s report is acknowledged within 30 seconds.• The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Figure 2-14 An example of a use case, ReportEmergency.

Analysis Activities: From Use Cases to Objects

- Identifying Entity Objects,
- Identifying Boundary Objects,
- Identifying Control Objects,
- Mapping Use Cases to Objects with Sequence Diagrams
- Mapping Interactions among Objects with CRC Cards
- Identifying Associations
- Identifying Aggregates
- Identifying Attributes
- Modeling State-Dependent Behavior of Individual Objects
- Modeling Inheritance Relationships
- Reviewing the Analysis Model

Abbot's heuristics - mapping parts of speech to model components

Table 5-1 Abbott's heuristics for mapping parts of speech to model components [Abbott, 1983].

Part of speech	Model component	Examples
Proper noun	Instance	Alice
Common noun	Class	Field officer
Doing verb	Operation	Creates, submits, selects
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation	Has, consists of, includes
Modal verb	Constraints	Must be
Adjective	Attribute	Incident description

Analysis Activities: Identifying Entity Objects

Heuristics for identifying entity objects

- Terms that developers or users need to clarify in order to understand the use case
- Recurring nouns in the use case (e.g. Incident)
- Real-world entities that the system needs to track (e.g. FieldOfficer, Dispatcher, Resource)
- Real-world activities that the system needs to track (e.g. EmergencyOperationsPlan)
- Data sources or sinks (e.g. Printer)

Analysis Activities: Identifying Entity Objects/2

Table 5-2 Entity objects for the ReportEmergency use case.

Dispatcher	Police officer who manages Incidents. A Dispatcher opens, documents, and closes Incidents in response to Emergency Reports and other communication with FieldOfficers. Dispatchers are identified by badge numbers.
EmergencyReport	Initial report about an Incident from a FieldOfficer to a Dispatcher. An EmergencyReport usually triggers the creation of an Incident by the Dispatcher. An EmergencyReport is composed of an emergency level, a type (fire, road accident, other), a location, and a description.
FieldOfficer	Police or fire officer on duty. A FieldOfficer can be allocated to, at most, one Incident at a time. FieldOfficers are identified by badge numbers.
Incident	Situation requiring attention from a FieldOfficer. An Incident may be reported in the system by a FieldOfficer or anybody else external to the system. An Incident is composed of a description, a response, a status (open, closed, documented), a location, and a number of FieldOfficers.

Analysis Activities: Identifying Boundary Objects

Heuristics for identifying boundary objects

- Identify user interface controls that the user needs to initiate the use case (e.g., ReportEmergencyButton)
- Identify forms the users need to enter data into the system (e.g., EmergencyReportForm)
- Identify notices and messages the system uses to respond to the user (e.g. AcknowledgmentNotice)

Analysis Activities: Identifying Boundary Objects/2

Heuristics for identifying boundary objects

- When multiple actors are involved in a use case, identify actor terminals (e.g., DispatcherStation) to refer to the user interface under consideration
- Do not model the visual aspects of the interface with boundary objects (user mock-ups are better suited for that)
- Always use the end user's terms for describing interfaces; do not use terms from the solution or implementation domains.

Analysis Activities: Identifying Boundary Objects /3

Table 5-3 Boundary objects for the ReportEmergency use case.

AcknowledgmentNotice	Notice used for displaying the Dispatcher's acknowledgment to the FieldOfficer.
DispatcherStation	Computer used by the Dispatcher.
ReportEmergencyButton	Button used by a FieldOfficer to initiate the ReportEmergency use case.
EmergencyReportForm	Form used for the input of the ReportEmergency. This form is presented to the FieldOfficer on the FieldOfficerStation when the "Report Emergency" function is selected. The EmergencyReportForm contains fields for specifying all attributes of an emergency report and a button (or other control) for submitting the completed form.
FieldOfficerStation	Mobile computer used by the FieldOfficer.
IncidentForm	Form used for the creation of Incidents. This form is presented to the Dispatcher on the DispatcherStation when the EmergencyReport is received. The Dispatcher also uses this form to allocate resources and to acknowledge the FieldOfficer's report.

Analysis Activities: Identifying Control Objects

Heuristics for identifying control objects

- Identify one control object per use case
- Identify one control object per actor in the use case
- The life span of a control object should cover the extent of the use case or the extent of a user session. If it is difficult to identify the beginning and the end of a control object activation, the corresponding use case probably does not have well-defined entry and exit conditions

Analysis Activities: Identifying Control Objects/2

Table 5-4 Control objects for the ReportEmergency use case.

ReportEmergencyControl	Manages the ReportEmergency reporting function on the FieldOfficerStation. This object is created when the FieldOfficer selects the “Report Emergency” button. It then creates an EmergencyReportForm and presents it to the FieldOfficer. After submitting the form, this object then collects the information from the form, creates an EmergencyReport, and forwards it to the Dispatcher. The control object then waits for an acknowledgment to come back from the DispatcherStation. When the acknowledgment is received, the ReportEmergencyControl object creates an AcknowledgmentNotice and displays it to the FieldOfficer.
ManageEmergencyControl	Manages the ReportEmergency reporting function on the DispatcherStation. This object is created when an EmergencyReport is received. It then creates an IncidentForm and displays it to the Dispatcher. Once the Dispatcher has created an Incident, allocated Resources, and submitted an acknowledgment, ManageEmergencyControl forwards the acknowledgment to the FieldOfficerStation.

Mapping Use Cases to Objects with Sequence Diagrams

- A **sequence diagram** ties use cases with objects. It shows how the behavior of a use case (or scenario) is distributed among participating objects.

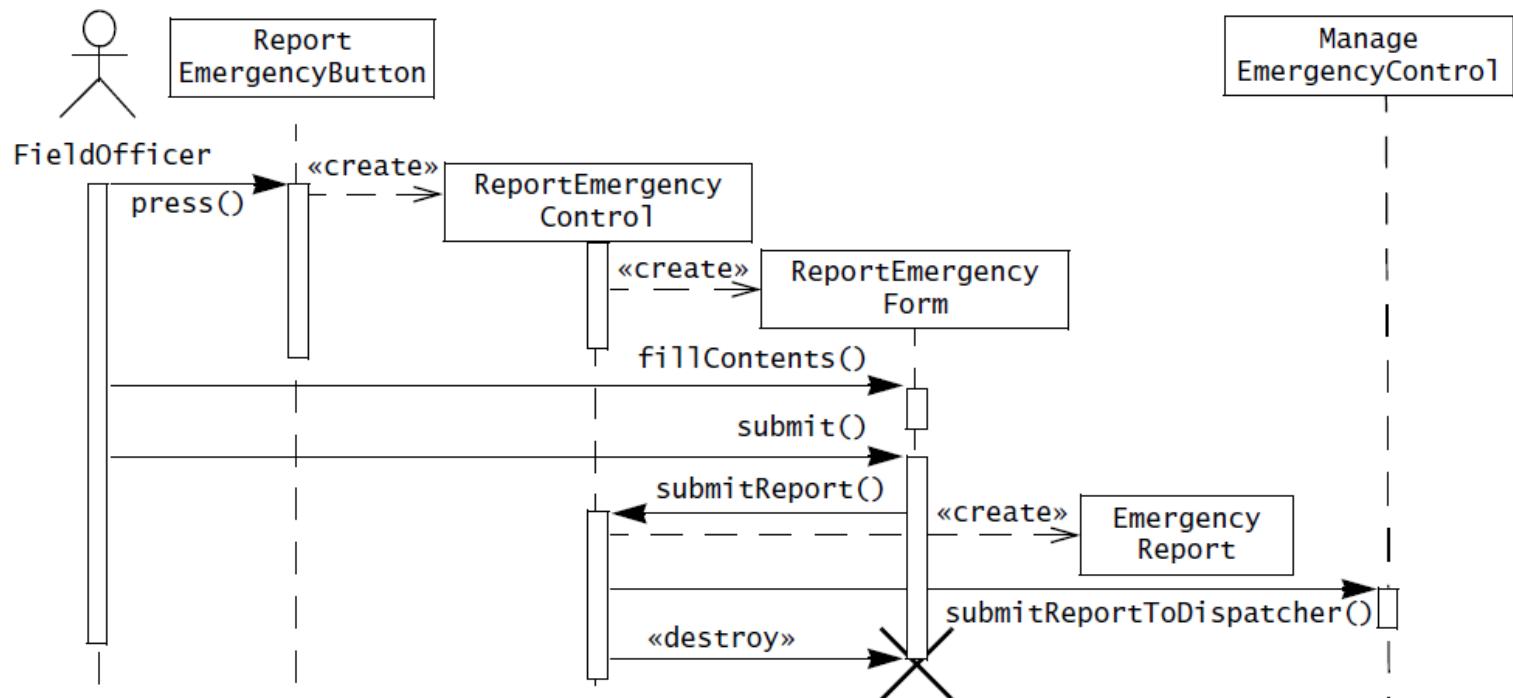


Figure 5-8 Sequence diagram for the ReportEmergency use case.

Mapping Use Cases to Objects with Sequence Diagrams_2

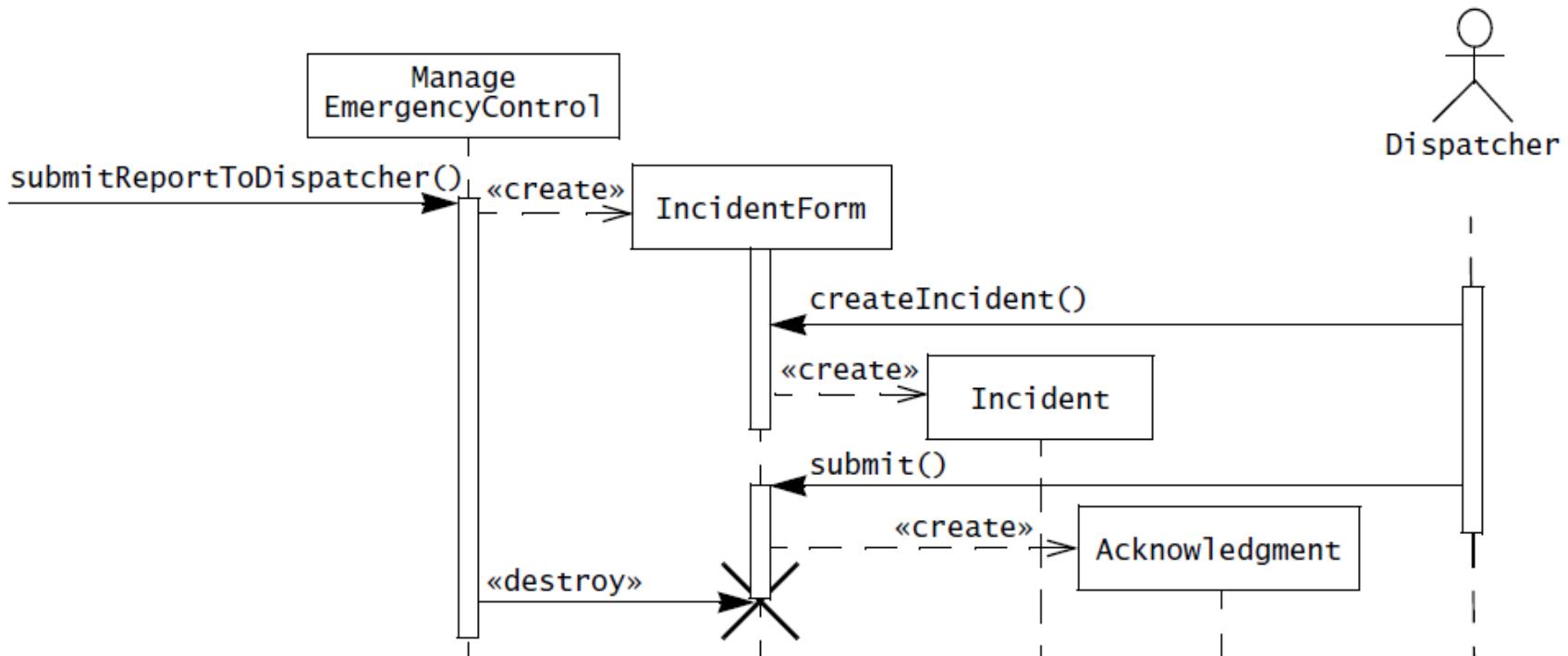


Figure 5-9 Sequence diagram for the ReportEmergency use case (continued from Figure 5-8).

Mapping Use Cases to Objects with Sequence Diagrams_3

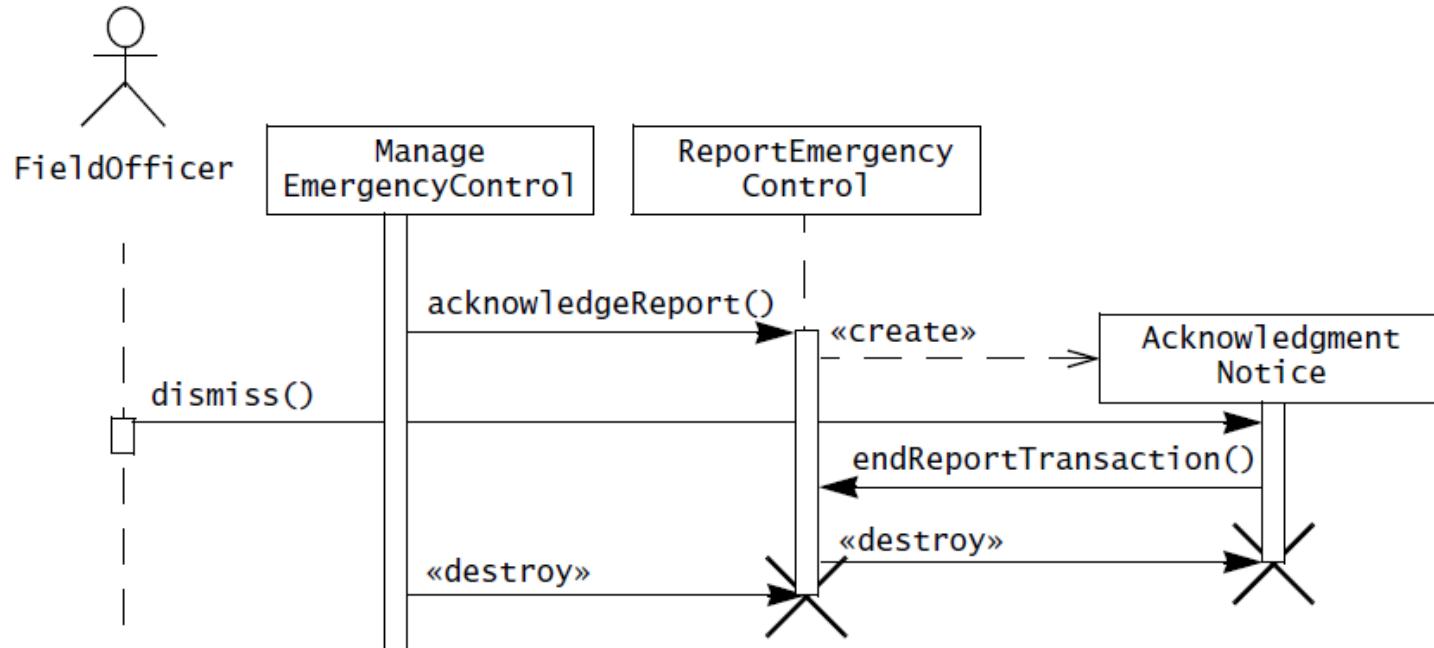


Figure 5-10 Sequence diagram for the ReportEmergency use case (continued from Figure 5-9).

Sequence diagrams allow the developers to find missing objects or grey areas in the requirements specification.

Mapping Use Cases to Objects with Sequence Diagrams

- **Heuristics for drawing sequence diagrams**
- The **first column** should correspond to the **actor** who initiated the use case.
- The **second column** should be a **boundary object** (that the actor used to initiate the use case).
- The **third column** should be the **control object** that manages the rest of use case.

Mapping Use Cases to Objects with Sequence Diagrams/2

- **Heuristics for drawing sequence diagram**
- **Control objects** are created by **boundary objects** initiating **use cases**.
- **Boundary objects** are created by **control objects**.
- **Entity objects are accessed by** control and boundary objects.
- **Entity objects never access** boundary or control objects; this makes it easier to share entity objects across use cases.

Modeling interactions among Objects with CRC Cards

- An alternative for identifying interactions among objects are CRC cards [Beck & Cunningham, 1989]. CRC cards (class, responsibilities, collaborators) were initially introduced as a tool for teaching object-orientation concepts to novices and to experienced developers unfamiliar with object-orientation.

Modeling interactions among Objects with CRC Cards/2

- Each class is represented with an index card. The name of the class is depicted on the top, its responsibilities on the left column, and the name of the classes it needs to accomplish the responsibilities are depicted on the right column.
- CRC cards and sequence diagrams are two different representations for supporting the same type of activity.

Modeling interactions among Objects with CRC Cards/3

- Sequence diagrams are a better tool for a single modeler or for documenting a sequence of interactions because are more precise and compact. **CRC cards are a better tool for a group of developers refining and iterating over an object structure** during a brainstorming session because are easier to create and to modify.

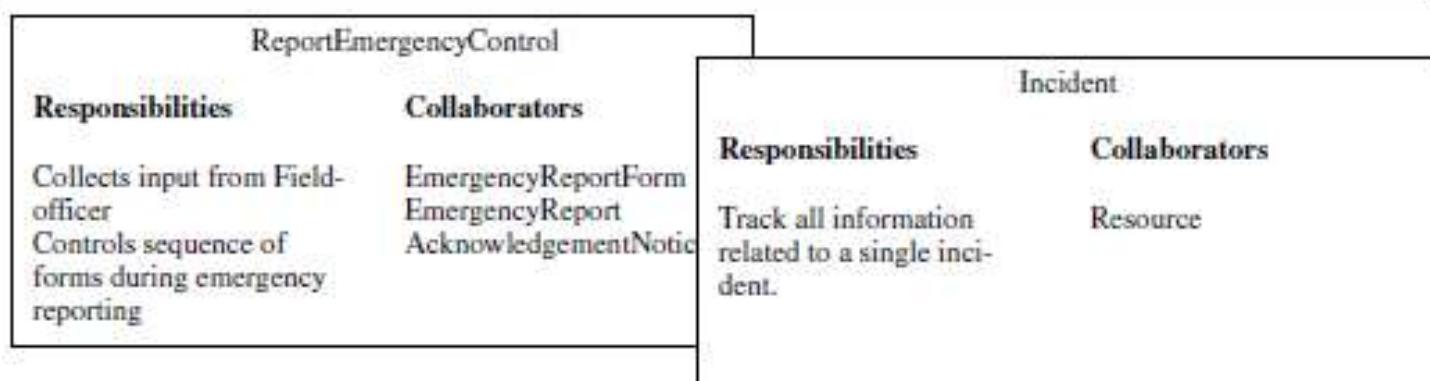


Figure 5-12 Examples of CRC cards for the ReportEmergencyControl and the Incident classes.

Identifying associations

- An association shows a relationship between two or more different objects instantiated by the same class (recursive association) or by different classes.
- Identifying associations has two advantages:
 - ▣ It clarifies the analysis model by making relationships between objects explicit (e.g., an EmergencyReport can be created by a FieldOfficer or a Dispatcher).
 - ▣ It enables the developer to discover **boundary cases** associated with links. **Boundary cases** are exceptions that must be clarified in the model.

Identifying associations/2

- Association have several properties:
 - **Associations names are optional** and need not to be unique globally.
 - A **role** at each end, identifying the function of each class' instances with respect to the instances of the opposite end (e.g., author is the role played by FieldOfficer in relation with EmergencyReport).
 - A **multiplicity** at each end, identifying the possible number of instances.

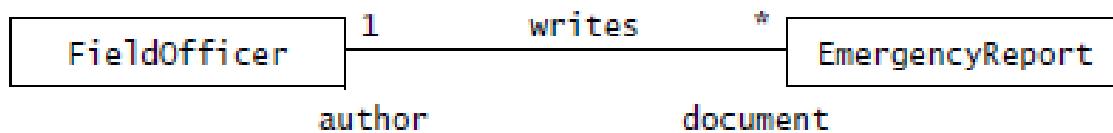


Figure 5-13 An example of association between the `EmergencyReport` and the `FieldOfficer` classes.

Identifying associations/3

Heuristics for identifying associations

- Examine verb phrases
- Name associations and roles as suggestive as possible
- Use qualifiers as often as possible to identify **namespaces and key attributes**
- Eliminate any association that can be derived from other associations
- Do not worry about multiplicity until the set of associations is stable
- Too many associations make a model unreadable

Identifying associations/4

Identifying Aggregates

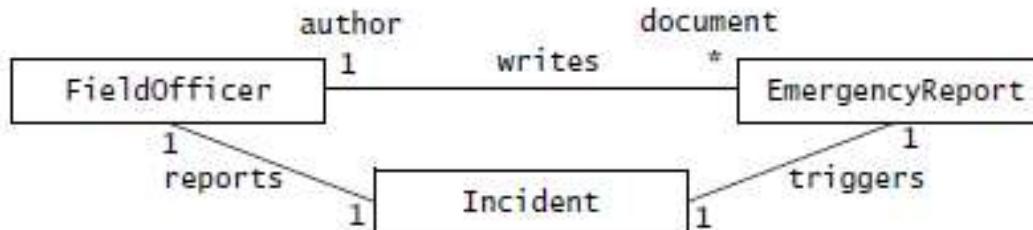


Figure 5-14 Eliminating redundant association. The receipt of an EmergencyReport triggers the creation of an Incident by a Dispatcher. Given that the EmergencyReport has an association with the FieldOfficer that wrote it, it is not necessary to keep an association between FieldOfficer and Incident.

- Aggregations are special types of associations denoting a whole-part relationship.
- There are two types of aggregations: composition and shared.

Identifying associations/5

Identifying Aggregates

A solid diamond denotes composition. A composition indicates that the existence of the parts depends on the whole

A shared aggregation relationship indicate that the whole and the parts can exist independently

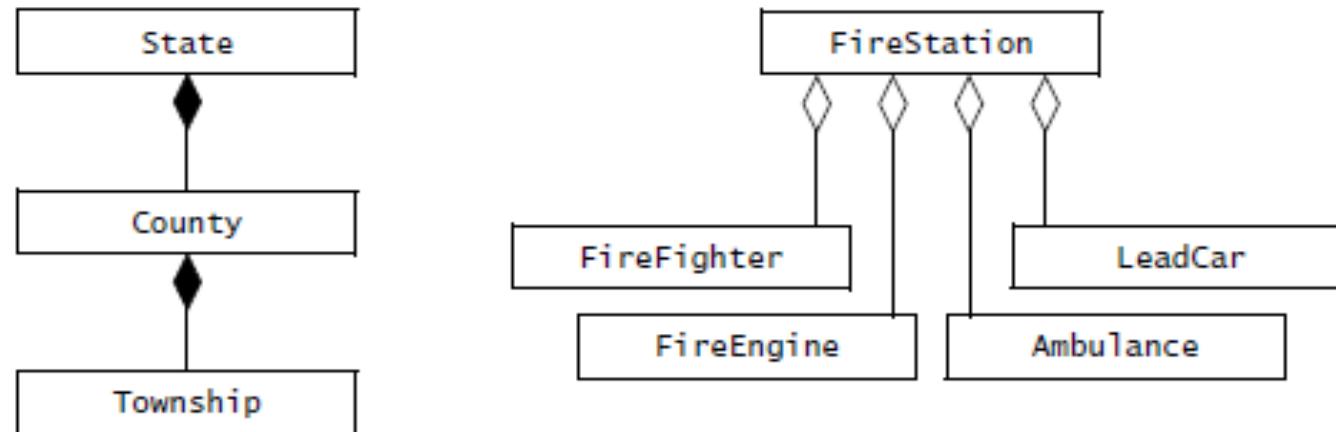


Figure 5-15 Examples of aggregations and compositions (UML class diagram). A State is composed of many Counties, which in turn is composed of many Townships. A FireStation includes FireFighters, FireEngines, Ambulances, and a LeadCar.

Identifying Attributes

- **Attributes are properties of individual objects**
- When identifying properties of objects, only the attributes relevant to the system should be considered.
- Properties that are represented by objects are not attributes
- Developers should identify as many associations as possible before identifying attributes to avoid confusing attributes and objects.

Identifying Attributes/2

- Attributes have:
 - A **name** identifying them within an **object**
 - A **brief description**
 - A **type** describing the legal values it can take

Identifying Attributes /3

Heuristics for identifying attributes

- Examine possessive phrases.
- Represent stored state as an attribute of the entity object.
- Describe each attribute.
- Do not represent an attribute as an object; use an association instead
- Do not waste time describing fine details before the object structure is stable.

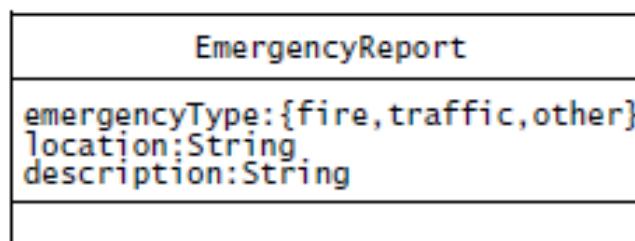


Figure 5-16 Attributes of the EmergencyReport class.

Modeling State-Dependent Behavior of Individual Objects

- Statechart diagrams represent behavior from the perspective of a single object. Viewing behavior from the perspective of each object enable the developer to build a more formal description of the behavior of the object, and consequently, to identify missing use cases.
- By focusing on individual states, developers may identify new behavior. By examining each transition in the statechart diagram that is triggered by a user action that triggers the transition.

Modeling State-Dependent Behavior of Individual Objects_2

- Only objects with an extended lifespan and state-dependent behavior are worth considering. This is almost always the case for control objects, less often for entity objects, and almost never for boundary objects.

Modeling State-Dependent Behavior of Individual Objects/3

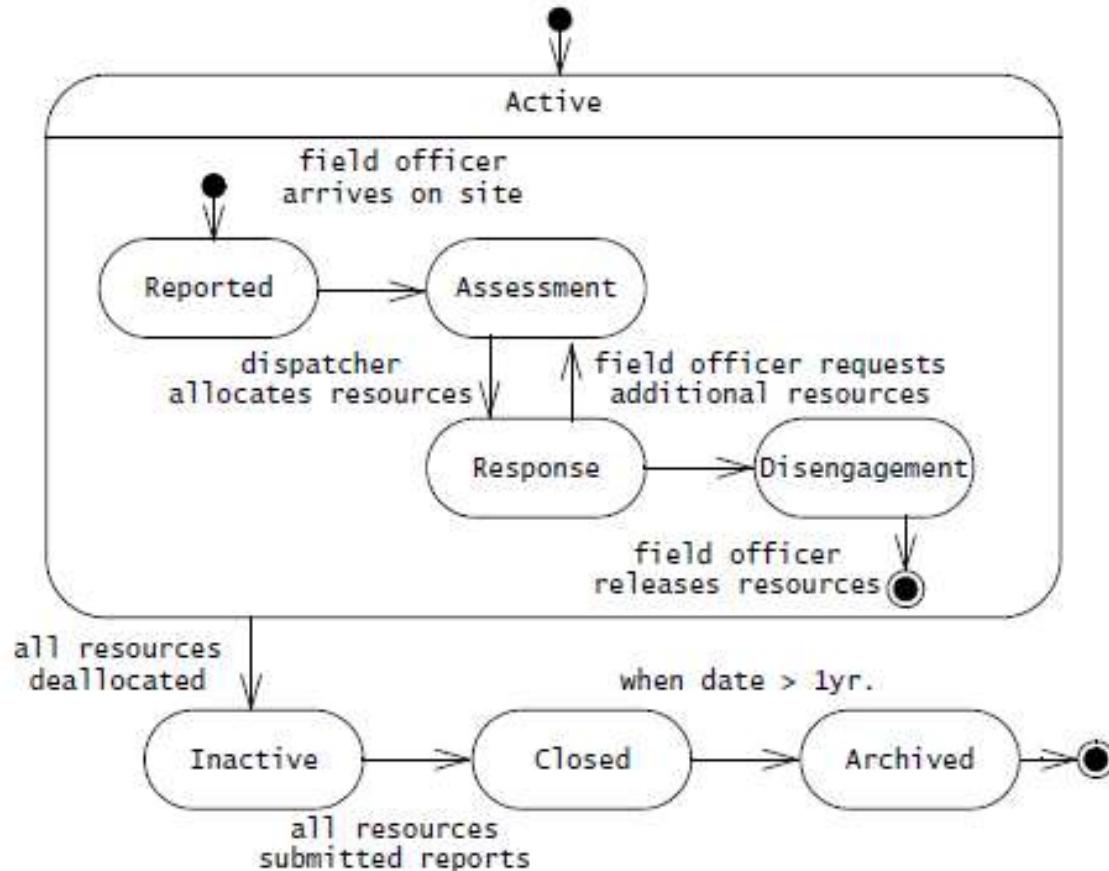


Figure 5-17 UML state machine for Incident.

Modeling Inheritance Relationships between Objects

- Generalization is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behavior, the similarities are consolidated into an ascendent class. Both Dispatchers and FieldOfficers have a badgeNumber attribute that serves to identify them within a city. To model this similarity, we introduce an abstract PoliceOfficer class.

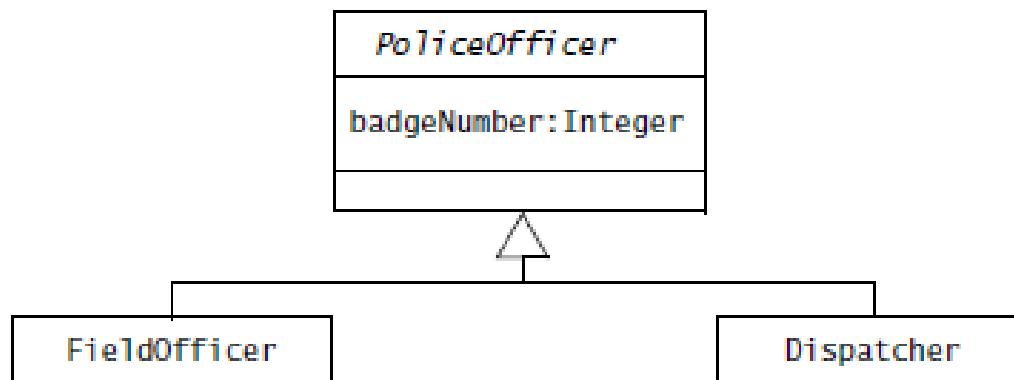


Figure 5-18 An example of inheritance relationship (UML class diagram).

Reviewing the Analysis Model

- The analysis model is built incrementally and iteratively. This model is seldom correct or even complete on the first pass. Before the analysis model converges toward a correct specification usable by the developers for design and implementation, several iterations with the client and the user are necessary.
- Once the analysis model becomes stable (i.e., when the number of changes to the models are minimal and the scope of the changes localized), the analysis model is reviewed, first by the developers (i.e., internal reviews), then jointly by the developers and the client.

Reviewing the Analysis Model/2

- The goal of the review is to make sure that the requirements specification is correct, complete, consistent and unambiguous.
- Developers should be prepared to discover errors downstream and make changes to the specification.

Reviewing the Analysis Model/3

- To ensure that **the model is correct**, the following questions should be asked:
 - Is the glossary of entity objects understandable by the user?
 - Do abstract classes correspond to user-level concepts?
 - Are all descriptions in accordance with the users' definitions?

Reviewing the Analysis Model/4

- To ensure that **the model is correct**, the following questions should be asked:
 - Do all entity and boundary objects have meaningful noun phrases as names?
 - Do all use cases and control objects have meaningful verb phrases as names?
 - Are all error cases described and handled?

Reviewing the Analysis Model/5

- To ensure that **the model is complete**, the following questions should be asked:
 - For each object: Is it needed by any use case? In which use case is it created? modified? Destroyed? Can it be accessed from a boundary object?
 - For each attribute: When is it set? What is its type? Should it be a qualifier?
 - For each association: When is it traversed? Why was the specific multiplicity chosen? Can associations with one-to-many multiplicities qualified?

Reviewing the Analysis Model/6

- To ensure that **the model is complete**, the following questions should be asked:
 - For each control object: Does it have the necessary associations to access the objects participating in its corresponding use case?

Reviewing the Analysis Model / 7

- To ensure that **the model is consistent**, the following questions should be asked:
 - Are there multiple classes or use cases with the same name?
 - Do entities (e.g., use cases, classes, attributes) with similar names denote similar concepts?
 - Are there objects with similar attributes and associations that are not in the same generalization hierarchy?

Reviewing the Analysis Model/8

- To ensure that the system described by **the analysis model is realistic**, the following questions should be asked:
 - ▣ Are there any novel features in the system? Were any studies or prototypes built to ensure their feasibility?
 - ▣ Can the performance and reliability requirements be met? Were these requirements verified by any prototypes running on the selected hardware?

Analysis Summary

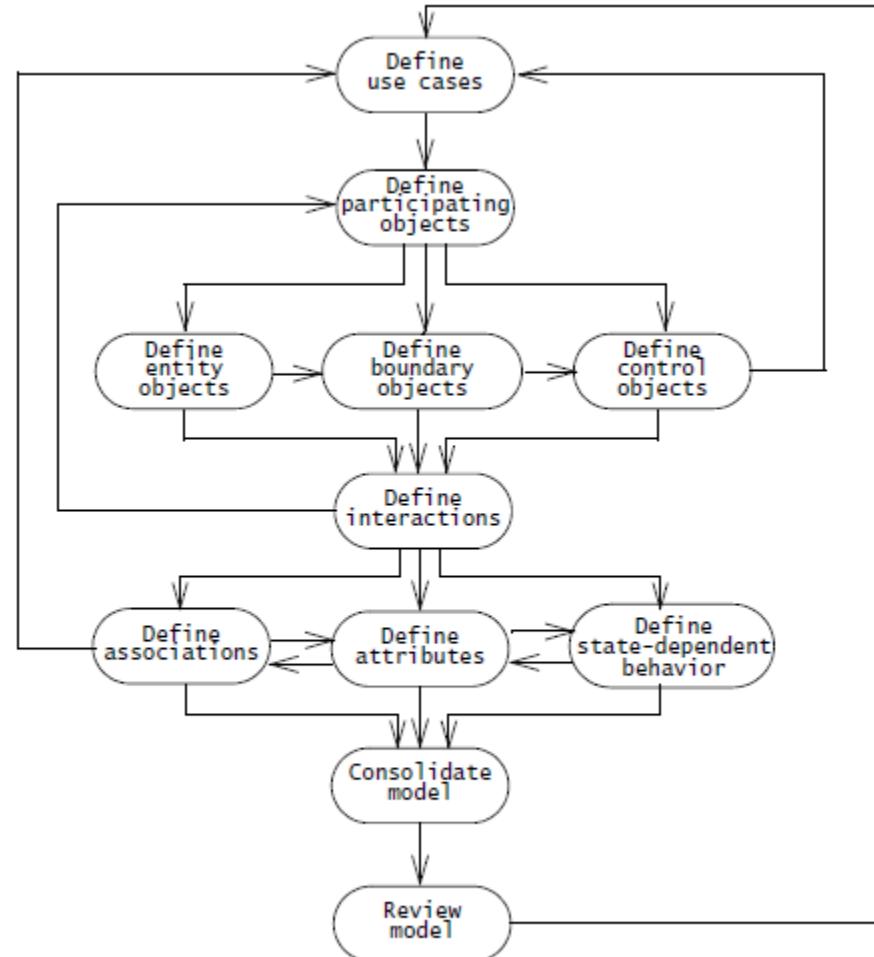


Figure 5-19 Analysis activities (UML activities diagram).

Managing Analysis – Documenting Analysis

- The primary challenge in managing requirements is to maintain consistency while using so many resources.
- The requirements elicitation and analysis activities are documented in the **Requirements Analysis Document (RAD)**. During analysis the sections of the above document are revised as ambiguities and new functionality are discovered.

Managing Analysis – Documenting Analysis/2

- The **object model** documents in detail all the objects identified with their attributes and associations.

- The **dynamic model** documents the behaviour of the object model by means of statechart diagrams and sequence diagrams.

- The revision history of the **RAD** will provide a history of changes, including the author responsible for each change.

Managing Analysis – Documenting Analysis_2

Requirements Analysis Document

1. Introduction
2. Current system
3. Proposed system
 - 3.1 Overview
 - 3.2 Functional requirements
 - 3.3 Nonfunctional requirements
 - 3.4 System models
 - 3.4.1 Scenarios
 - 3.4.2 Use case model
 - 3.4.3 Object model
 - 3.4.3.1 Data dictionary
 - 3.4.3.2 Class diagrams
 - 3.4.4 Dynamic models
 - 3.4.5 User interface—navigational paths and screen mock-ups
 4. Glossary

Figure 5-20 Overview outline of the Requirements Analysis Document (RAD). See Figure 4-16 for a detailed outline.

Assigning Responsibilities

- Analysis requires the participation of a wide range of individuals:
 - ▣ The target user provides application domain knowledge.
 - ▣ The client funds the project and coordinates the user side of the effort.
 - ▣ The analyst elicits application domain knowledge and formalizes it.
 - ▣ Developers provide feedback on feasibility and cost.
 - ▣ The project manager coordinates the effort on the development side.

Assigning Responsibilities/2

- For large systems, many users, analysts and developers may be involved, introducing additional challenges during integration and communication requirements. **These challenges can be met by assigning well-defined roles and scopes to individuals.**
- There are **three main types of roles:**
 - generation of information,
 - **integration** and
 - **review.**

Client sign-off



Represents the acceptance of the analysis model (as documented by the requirements analysis document) by the client. The client and the developers converge on a single idea and agree about the functions and features that the system will have.

In addition, they agree on:

- a list of priorities
- a revision process
- a schedule and a budget.
- a list of criteria that will be used to accept or reject the system

Client sign-off/2

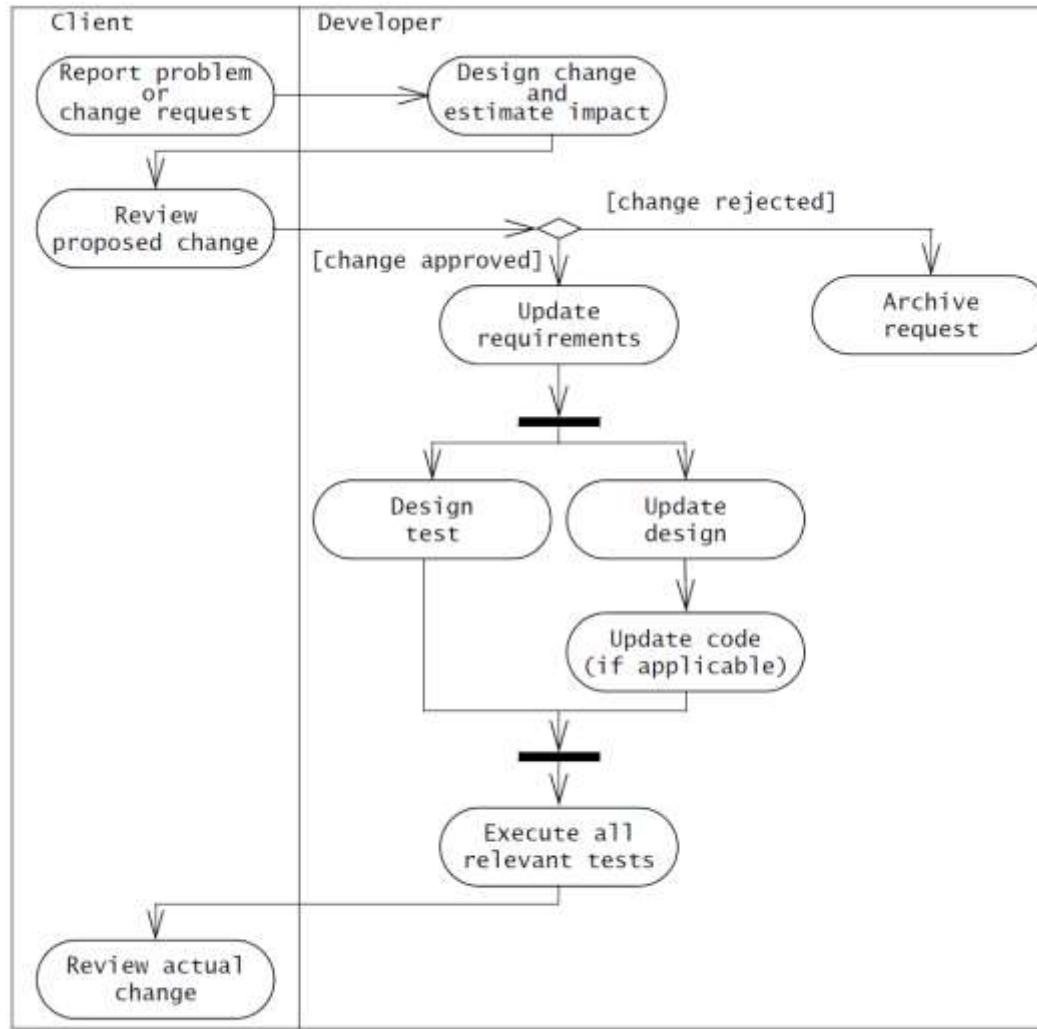


Figure 5-22 An example of a revision process (UML activity diagram).

SE is ... more than programming

Software engineering



... includes programming, but is more than programming

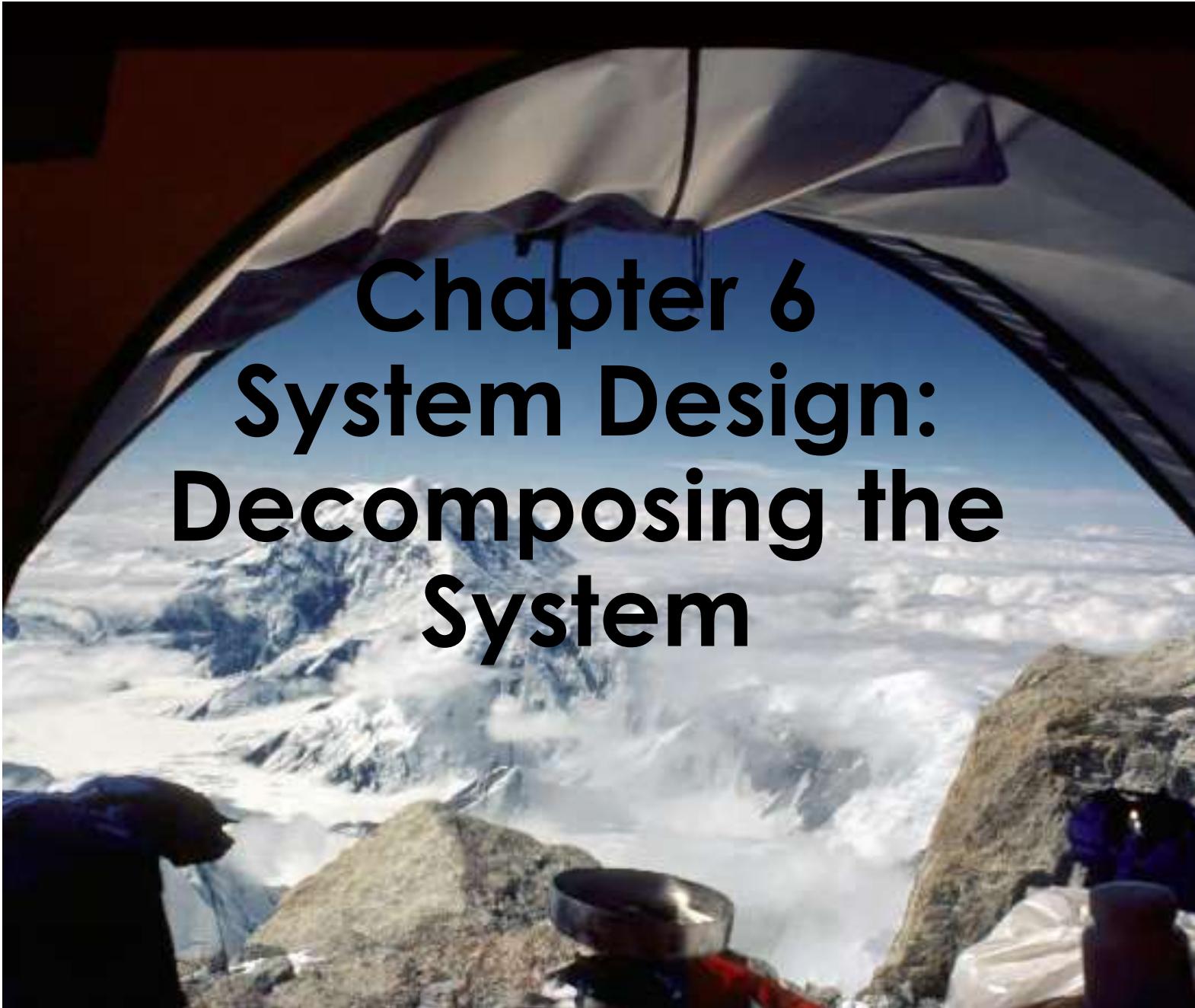
As von Clausewitz did not write: "the continuation of programming through other means".

Object-Oriented Software Engineering

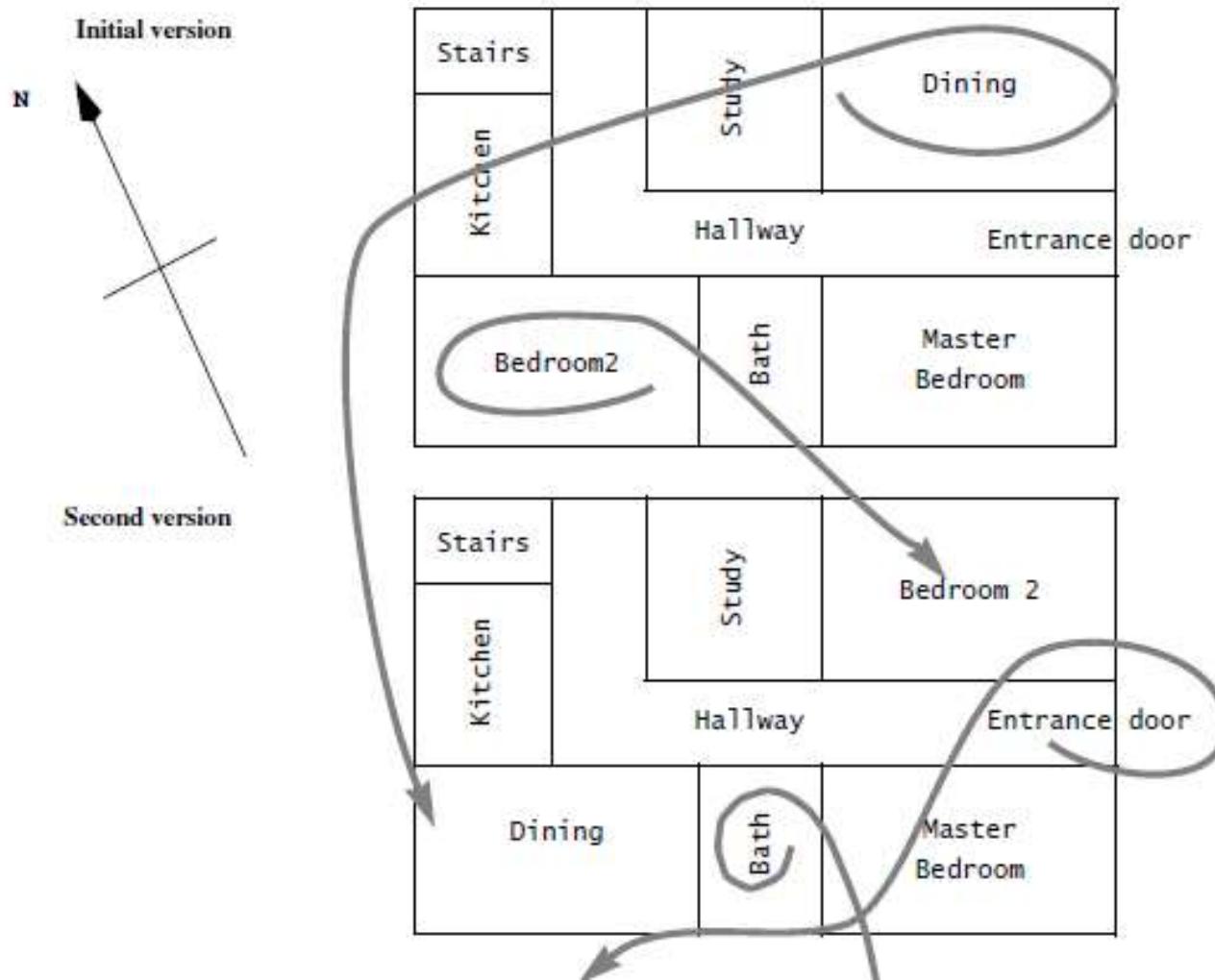
Using UML, Patterns, and Java

Chapter 6

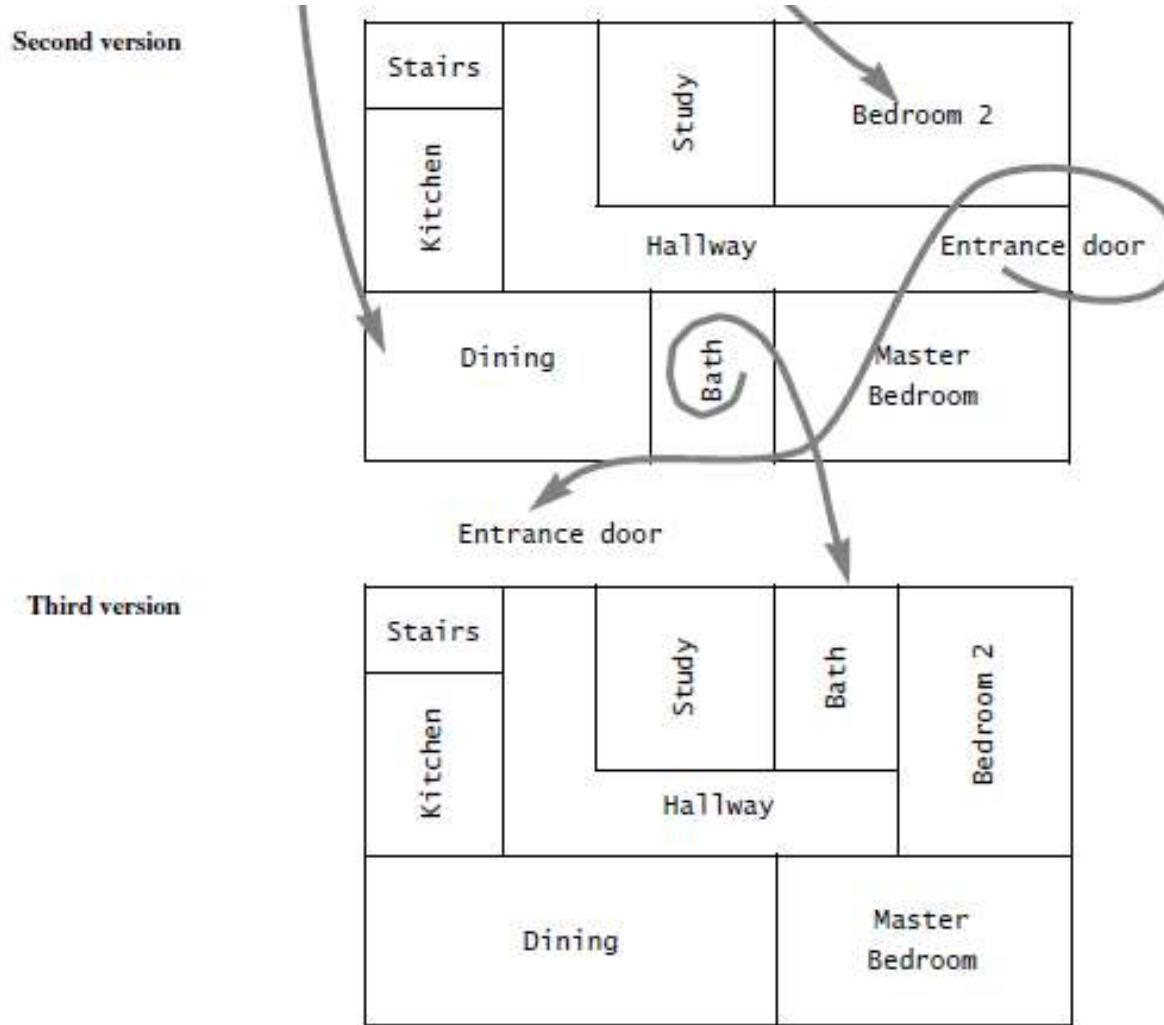
System Design: Decomposing the System



A floor plan example



A floor plan example cont



Design is Difficult

- There are two ways of constructing a software design (Tony Hoare):
 - One way is to make it so simple that there are obviously no deficiencies
 - The other way is to make it so complicated that there are no obvious deficiencies.”
- Corollary (Jostein Gaarder):
 - If our brain would be so simple that we can understand it, we would be too stupid to understand it.



Sir Antony Hoare, *1934

- Quicksort
- Hoare logic for verification
- CSP ([Communicating Sequential Processes](#)): modeling language for concurrent [processes](#) (basis for Occam).



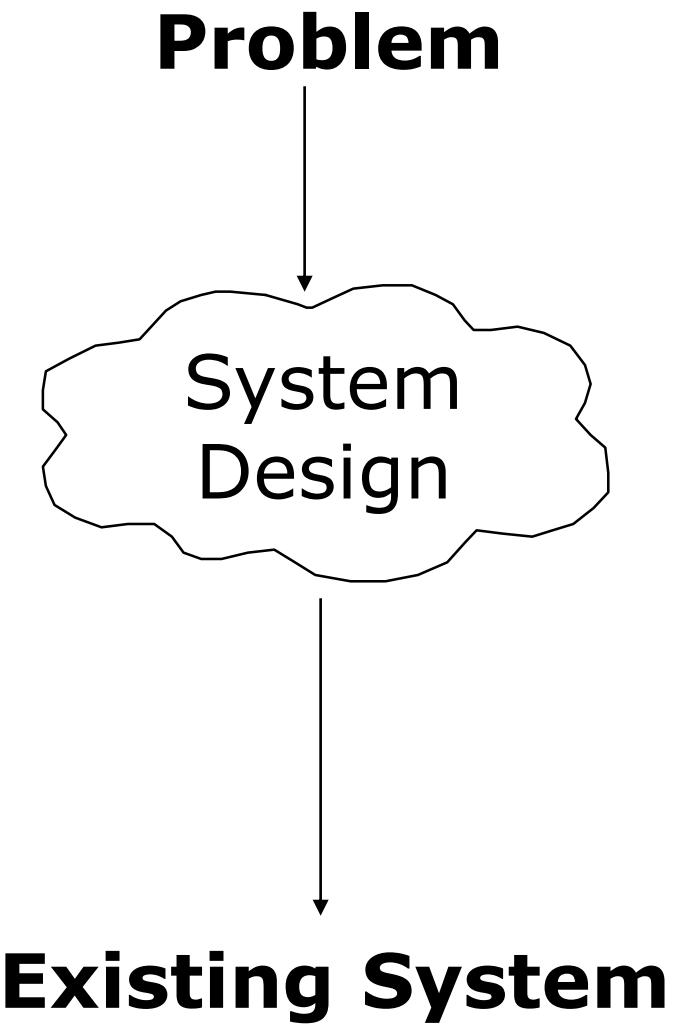
Jostein Gardner, *1952, writer
Uses metafiction in his stories:
Fiction which uses the device of fiction
- Best known for: „Sophie’s World“.

Why is Design so Difficult?

- **Analysis:** Focuses on the application domain
- **Design:** Focuses on the solution domain
 - The solution domain is changing very rapidly
 - Halftime knowledge in software engineering: About 3-5 years
 - Cost of hardware rapidly sinking
 - Design knowledge is a moving target
- **Design window:** Time in which design decisions have to be made.

The Scope of System Design

- Bridge the gap
 - between a problem and an existing system in a manageable way
- How?
- Use Divide & Conquer:
 - 1) Identify design goals
 - 2) Model the new system design as a set of subsystems
 - 3-8) Address the major design goals.



The Activities of System Design

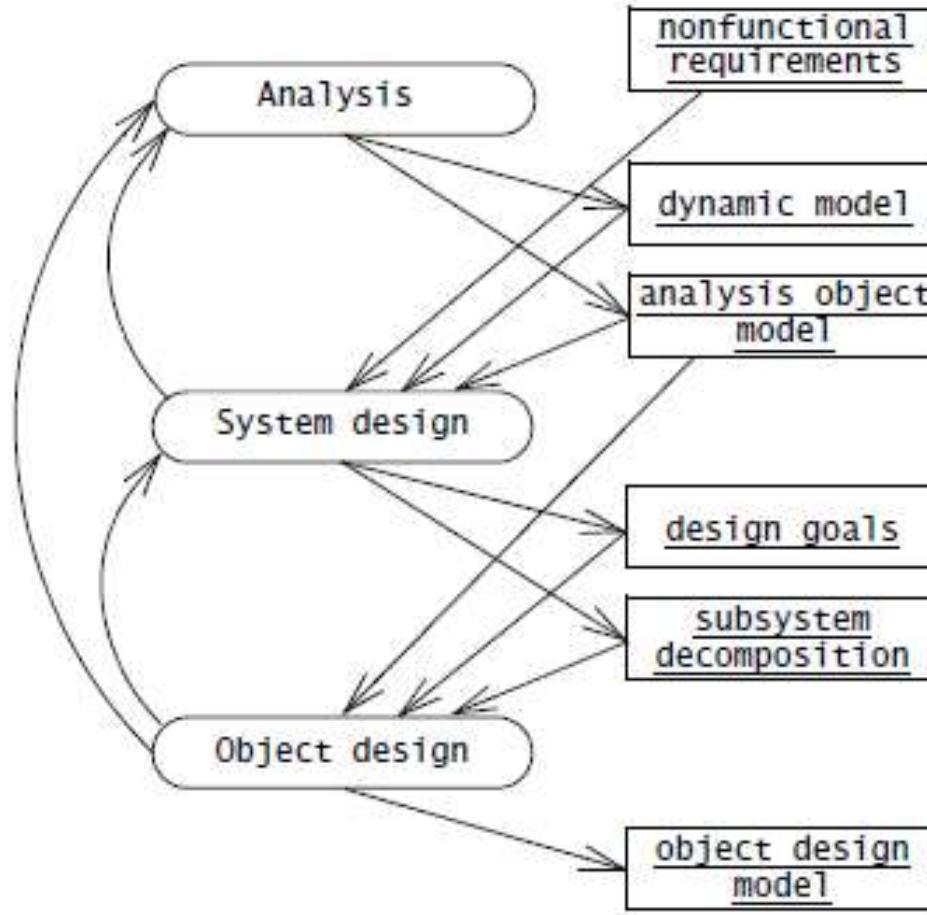
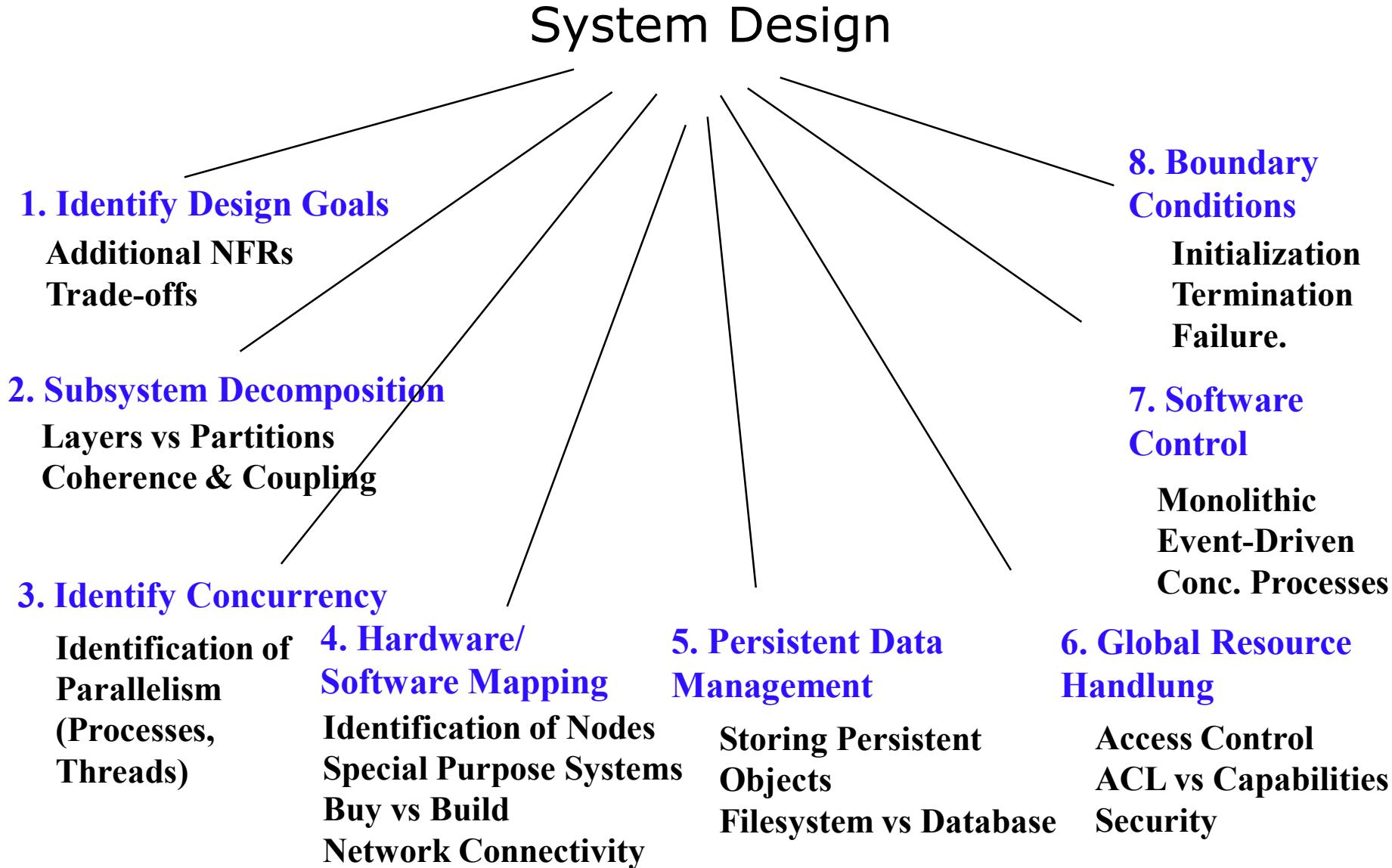


Figure 6-2 The activities of system design (UML activity diagram).

System Design: Eight Issues



Subsystem Decomposition

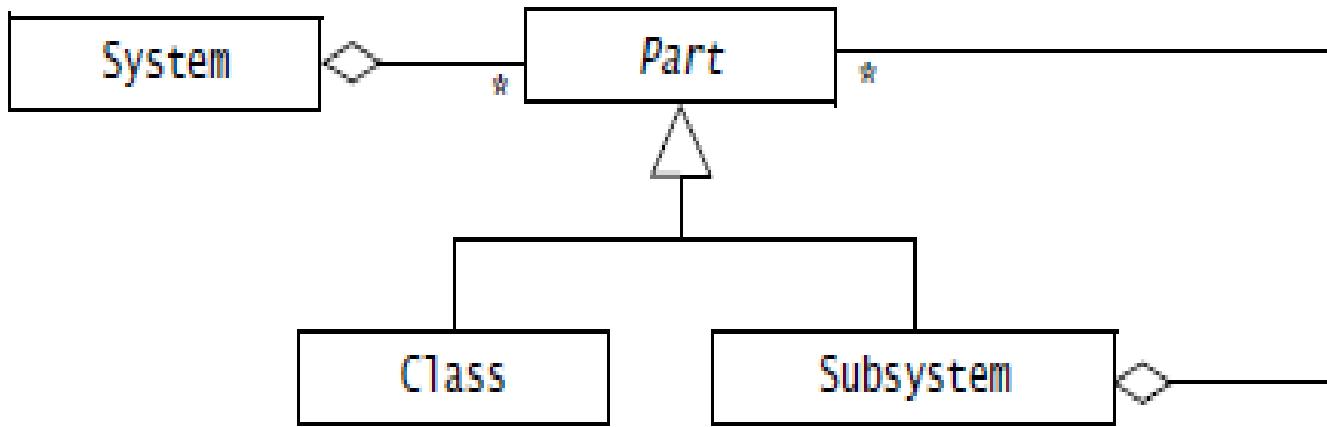


Figure 6-3 Subsystem decomposition (UML class diagram).

Subsystem Decomposition

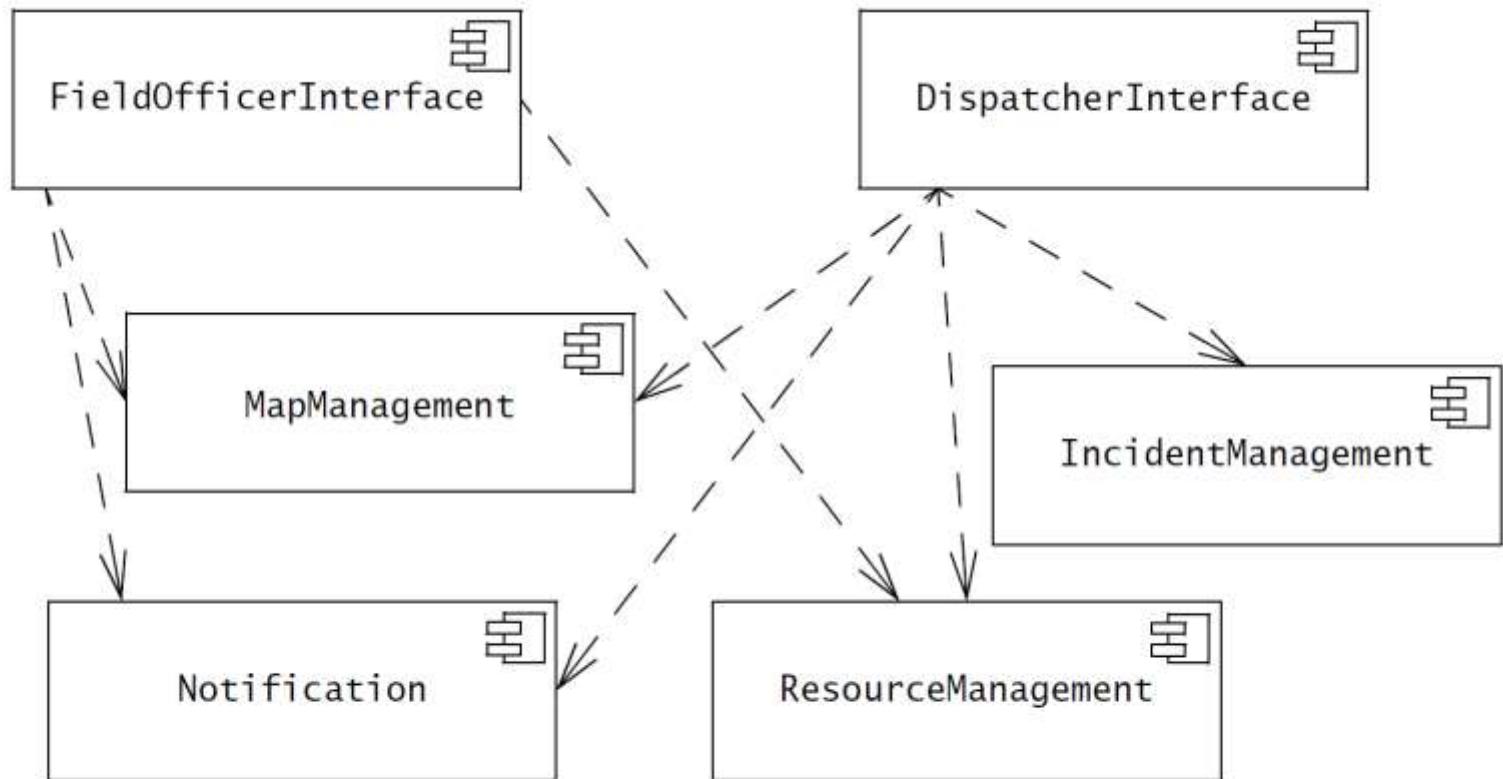


Figure 6-4 Subsystem decomposition for an accident management system (UML component diagram). Subsystems are shown as UML components. Dashed arrows indicate dependencies between subsystems.

Subsystem Decomposition_cont

A subsystem is characterized by the services it provides to other subsystems. A **service** is a set of related operations that share a common purpose.

The set of operations of a subsystem that are available to other subsystems form the **Subsystem interface**.

Provided and required interfaces can be depicted in UML with **assembly connectors**, also called **ball-and-socket connectors**.

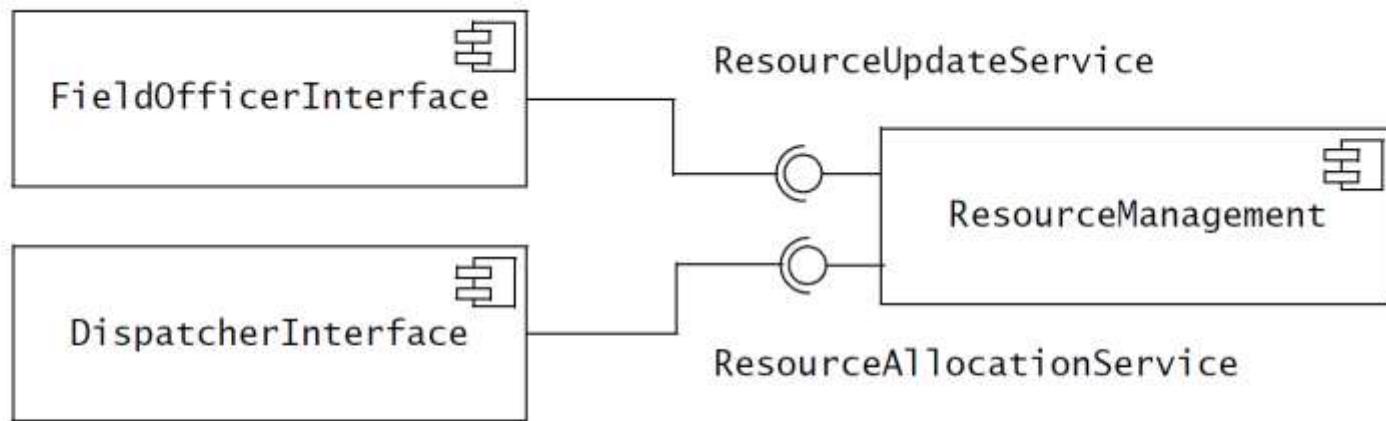
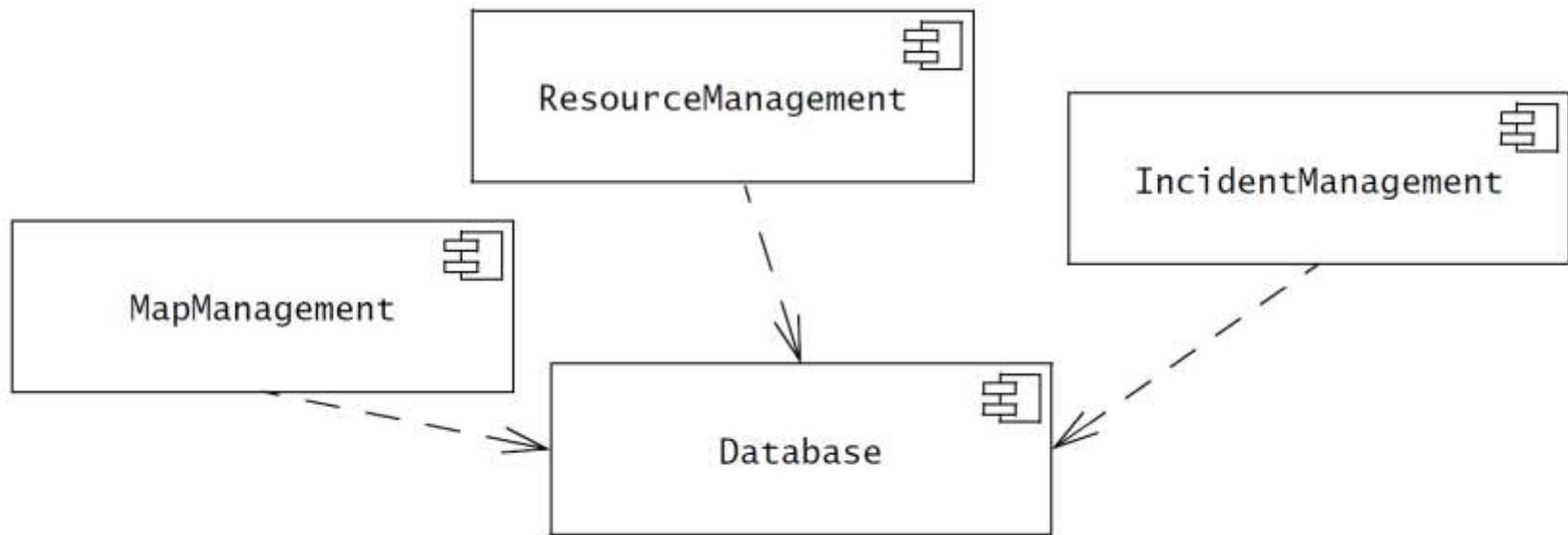


Figure 6-5 Services provided by the ResourceManagement subsystem (UML component diagram, ball-and-socket notation depicting provided and required interfaces).

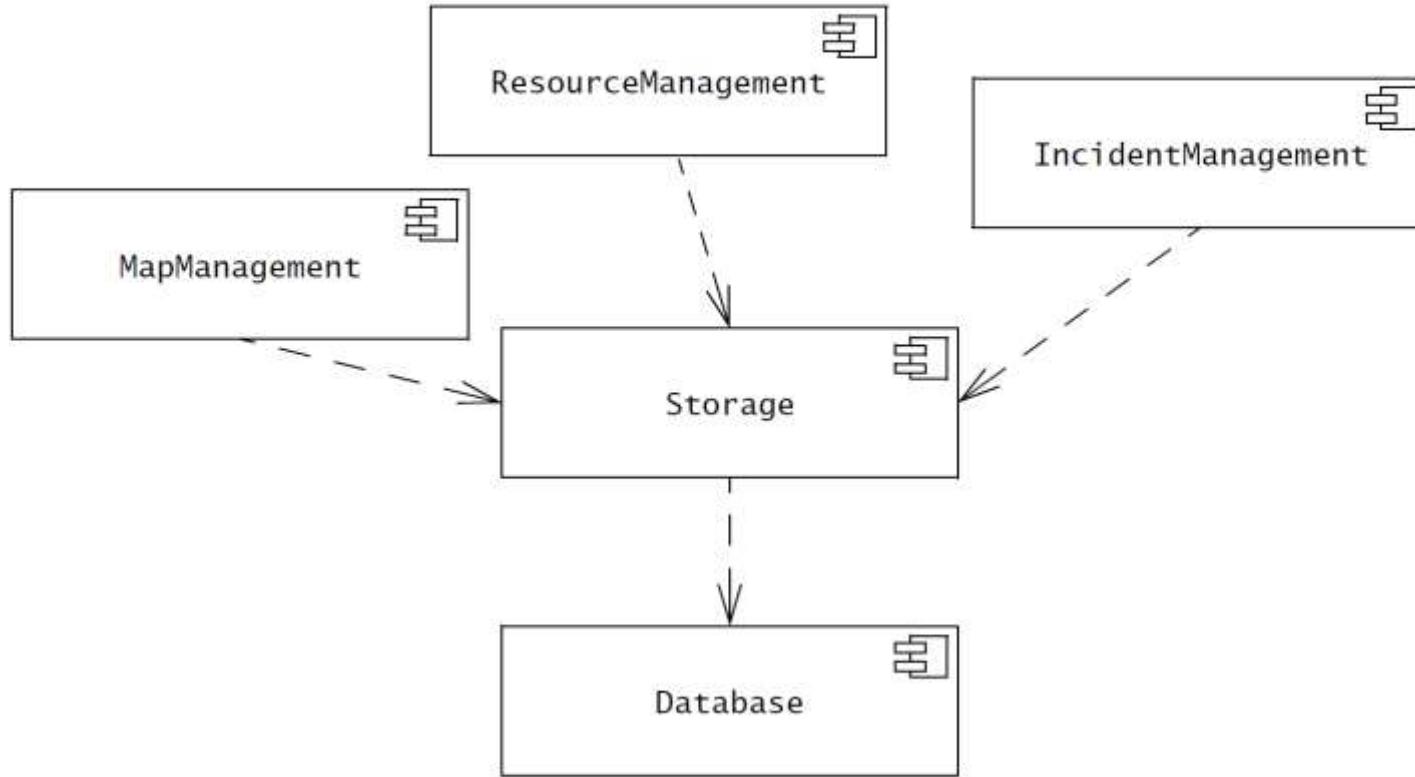
Subsystem Coupling

Alternative 1: Direct access to the Database subsystem



Subsystem Coupling

Alternative 2: Indirect access to the Database through a Storage subsystem



Example of reducing the coupling of subsystems (UML component diagram, subsystems FieldOfficerInterface, DispatcherInterface, and Notification omitted for clarity).

How the Analysis Models influence System Design

- Nonfunctional Requirements
 - => Definition of Design Goals
- Functional model
 - => Subsystem Decomposition
- Object model
 - => Hardware/Software Mapping, Persistent Data Management
- Dynamic model
 - => Identification of Concurrency, Global Resource Handling, Software Control
- Finally: Hardware/Software Mapping
 - => Boundary conditions

Coupling and Cohesion

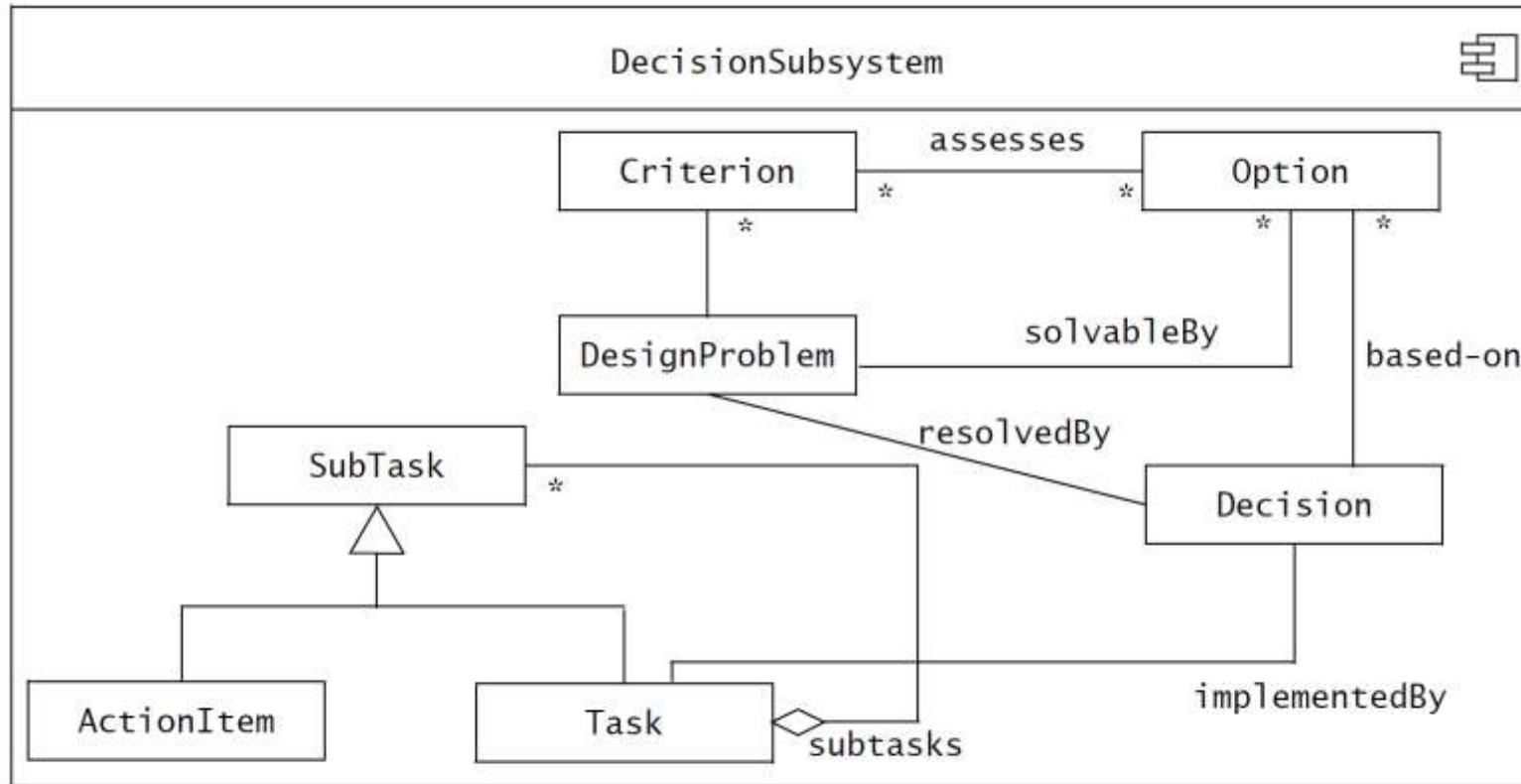
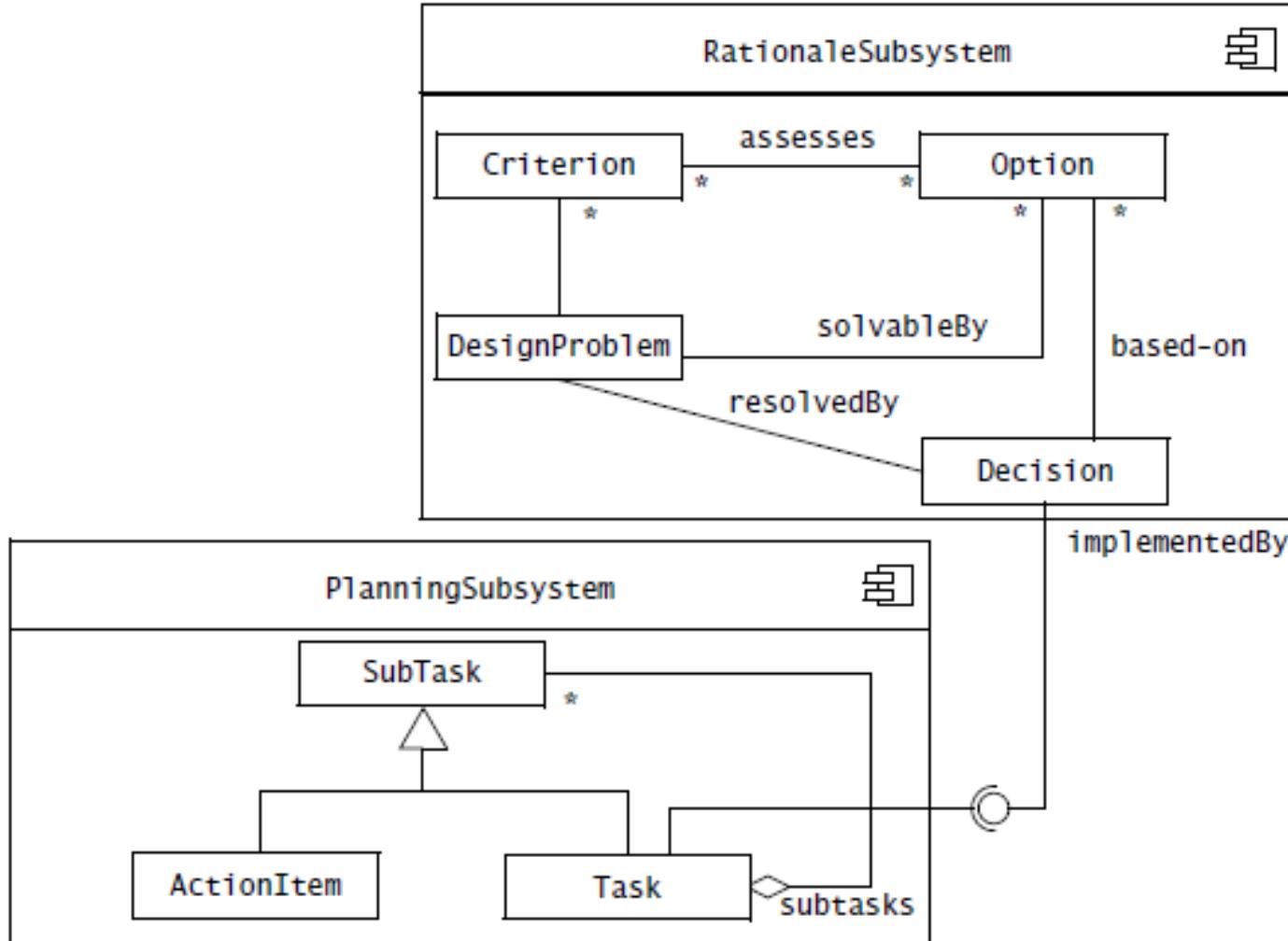


Figure 6-7 Decision tracking system (UML component diagram). The **DecisionSubsystem** has a low cohesion: The classes **Criterion**, **Option**, and **DesignProblem** have no relationships with **Subtask**, **ActionItem**, and **Task**.

Coupling and Cohesion_cont



Typical Design Trade-offs

- Functionality v. Usability
- Cost v. Robustness
- Efficiency v. Portability
- Rapid development v. Functionality
- Cost v. Reusability
- Backward Compatibility v. Readability

Subsystem Decomposition

- **Subsystem**
 - Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
 - The objects and classes from the object model are the “seeds” for the subsystems
 - In UML subsystems are modeled as packages
- **Service**
 - A set of named operations that share a common purpose
 - The origin (“seed”) for services are the use cases from the functional model
- **Services are defined during system design.**

Subsystem Interfaces vs API

- **Subsystem interface:** Set of fully typed UML operations
 - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
 - Refinement of service, should be well-defined and small
 - Subsystem interfaces are defined during object design
- **Application programmer's interface (API)**
 - The API is the specification of the subsystem interface in a specific programming language
 - APIs are defined during implementation
- The terms subsystem interface and API are often confused with each other
 - *The term API should not be used during system design and object design, but only during implementation.*

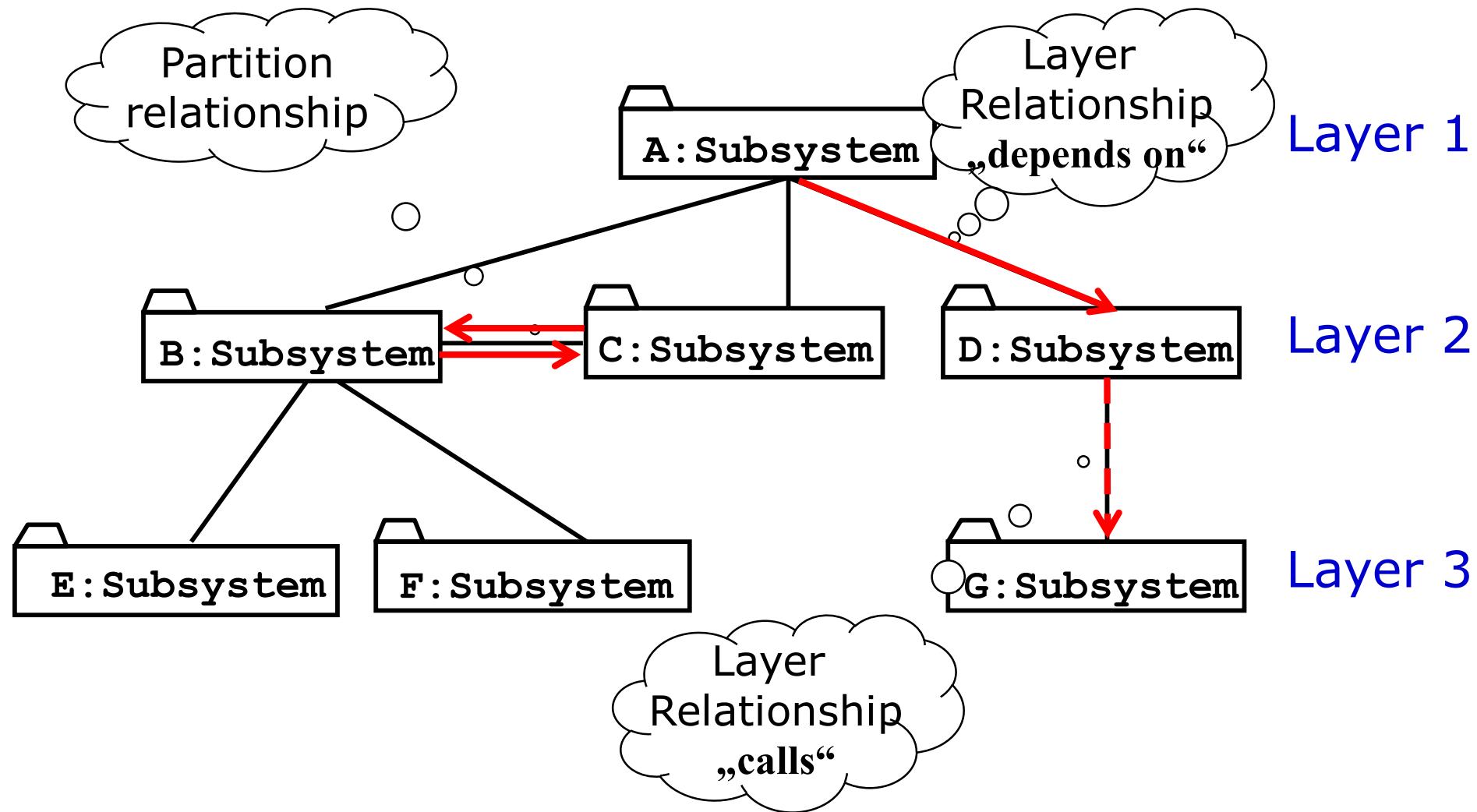
Properties of Subsystems: Layers and Partitions

- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions:
 - A layer only depends on services from lower layers
 - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called **partitions**
 - Partitions provide services to other partitions on the same layer
 - Partitions are also called “weakly coupled” subsystems.

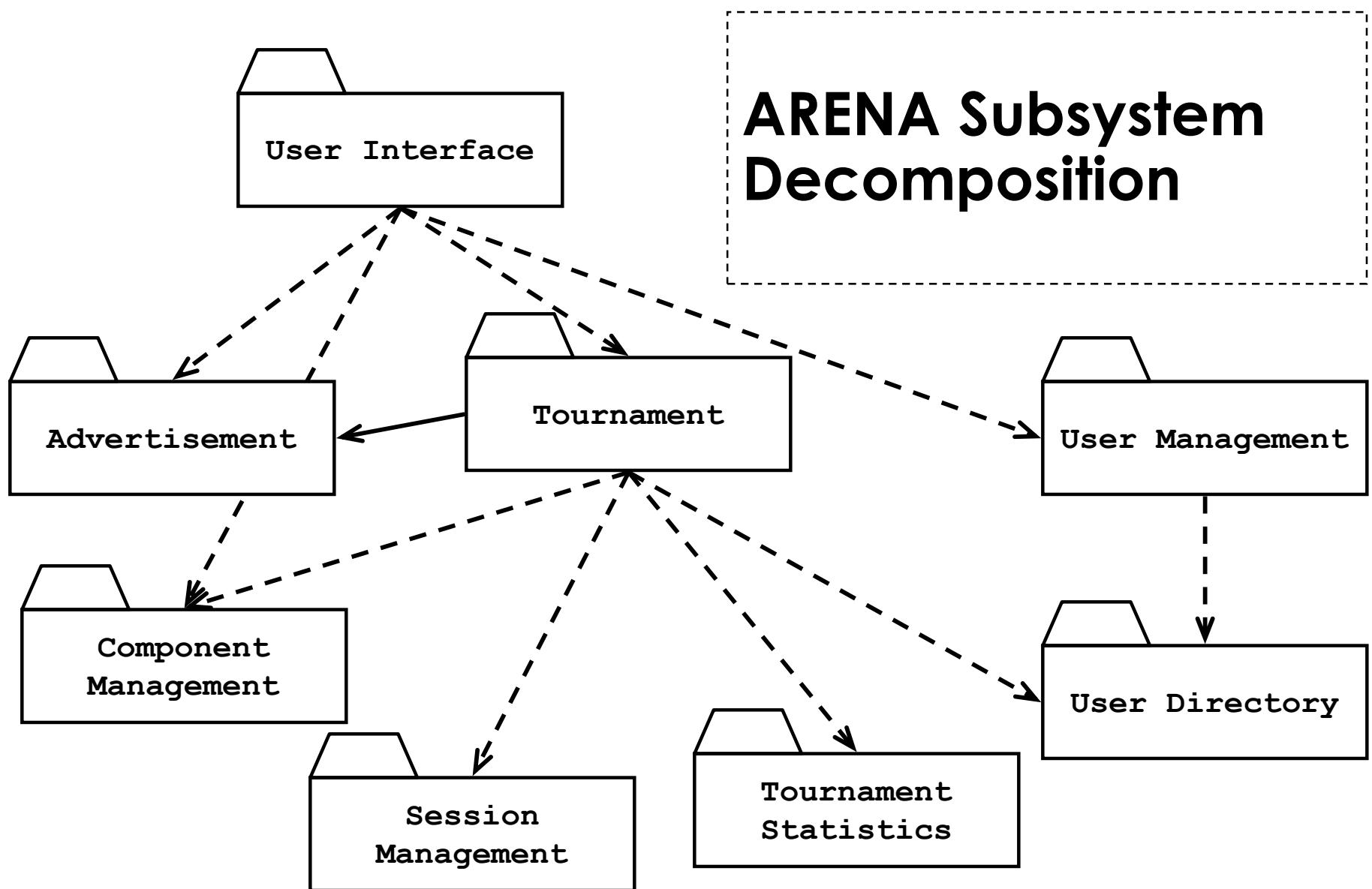
Relationships between Subsystems

- Two major types of Layer relationships
 - Layer A “depends on” Layer B (compile time dependency)
 - Example: Build dependencies (make, ant, maven)
 - Layer A “calls” Layer B (runtime dependency)
 - Example: A web browser calls a web server
 - Can the client and server layers run on the same machine?
 - Yes, they are layers, not processor nodes
 - Mapping of layers to processors is decided during the Software/hardware mapping!
- Partition relationship
 - The subsystems have mutual knowledge about each other
 - A calls services in B; B calls services in A ([Peer-to-Peer](#))
- UML convention:
 - Runtime dependencies are associations with dashed lines
 - Compile time dependencies are associations with solid lines.

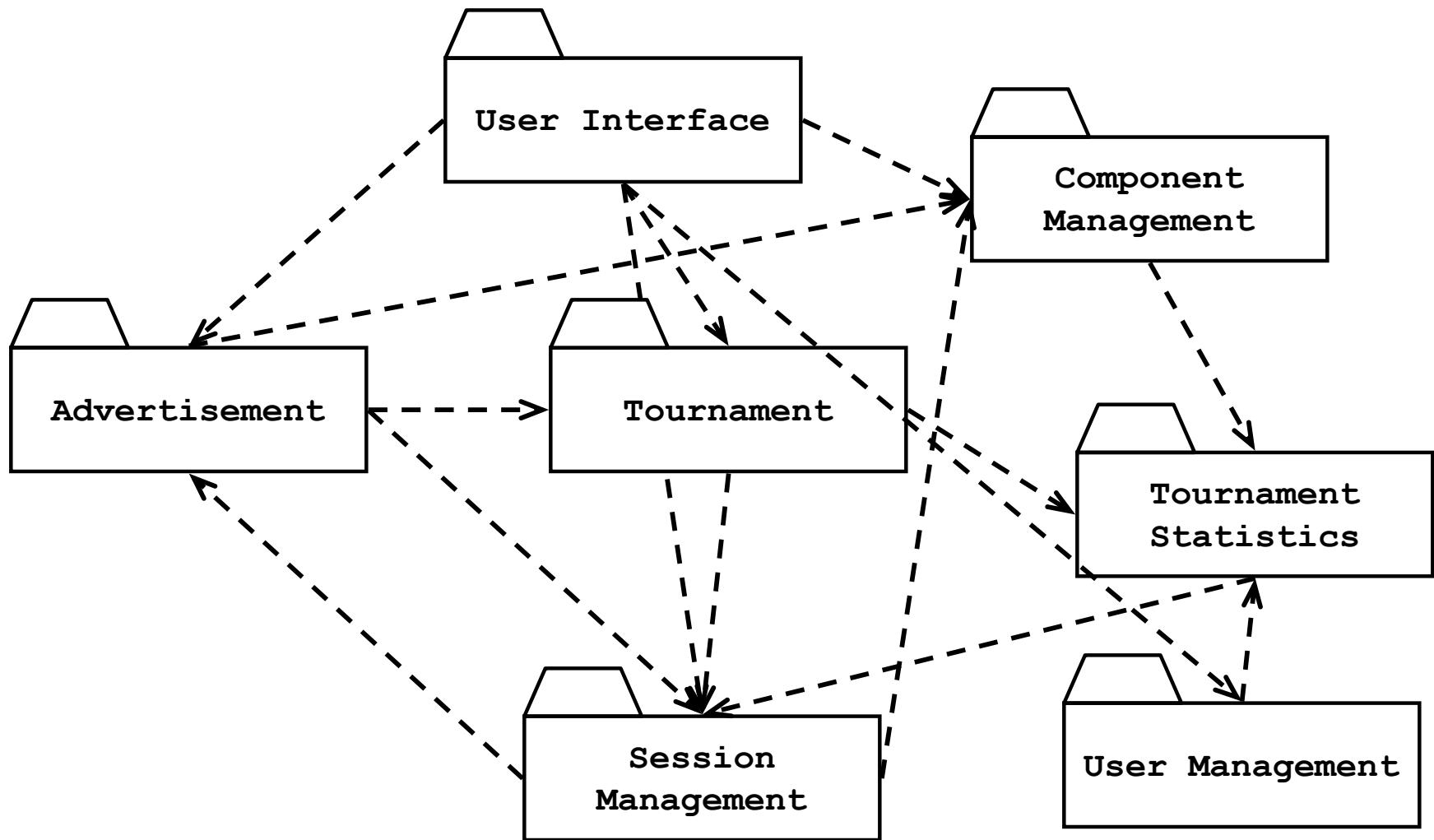
Example of a Subsystem Decomposition



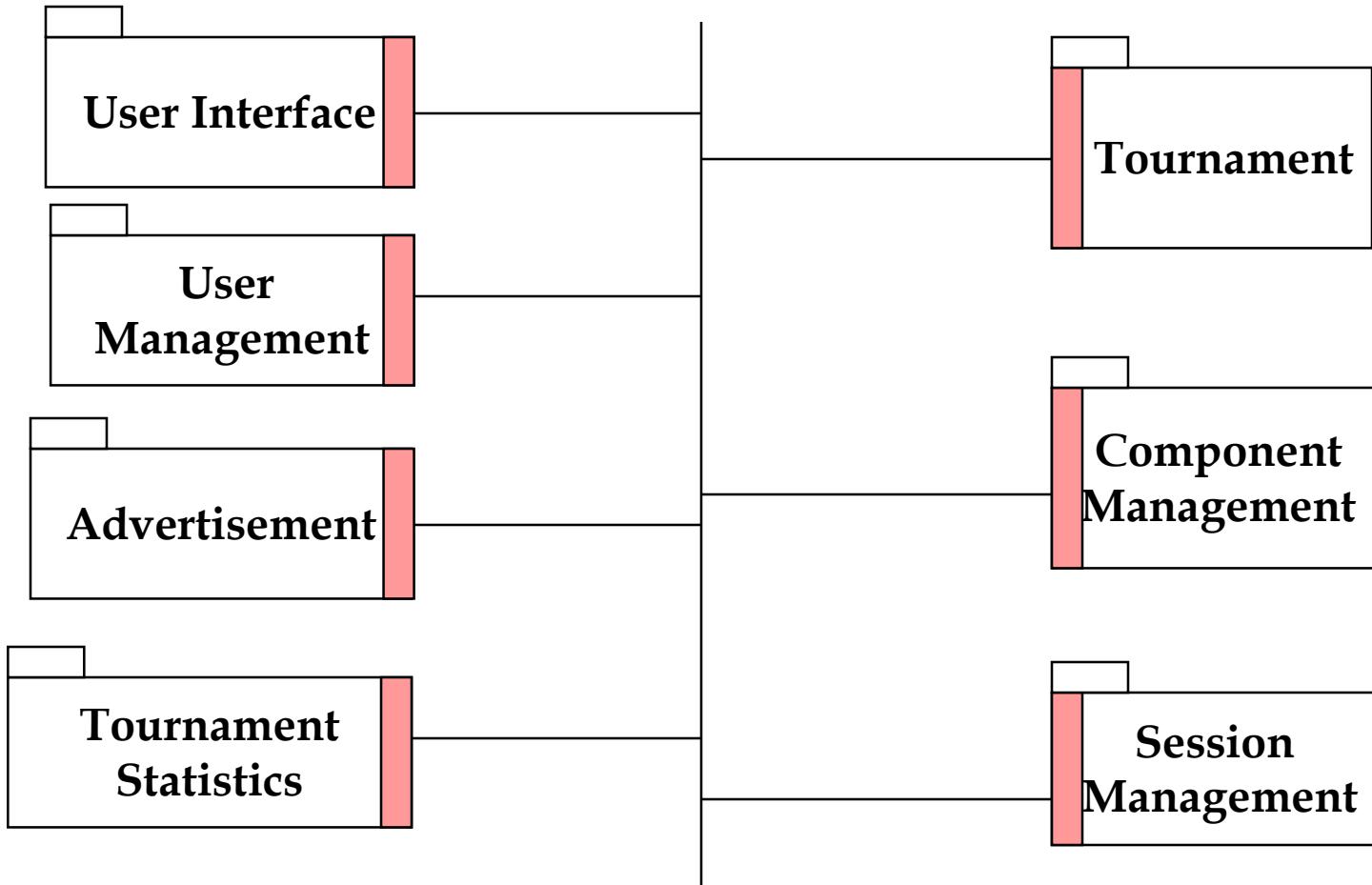
ARENA Subsystem Decomposition



Example of a Bad Subsystem Decomposition



Good Design: The System as set of Interface Objects



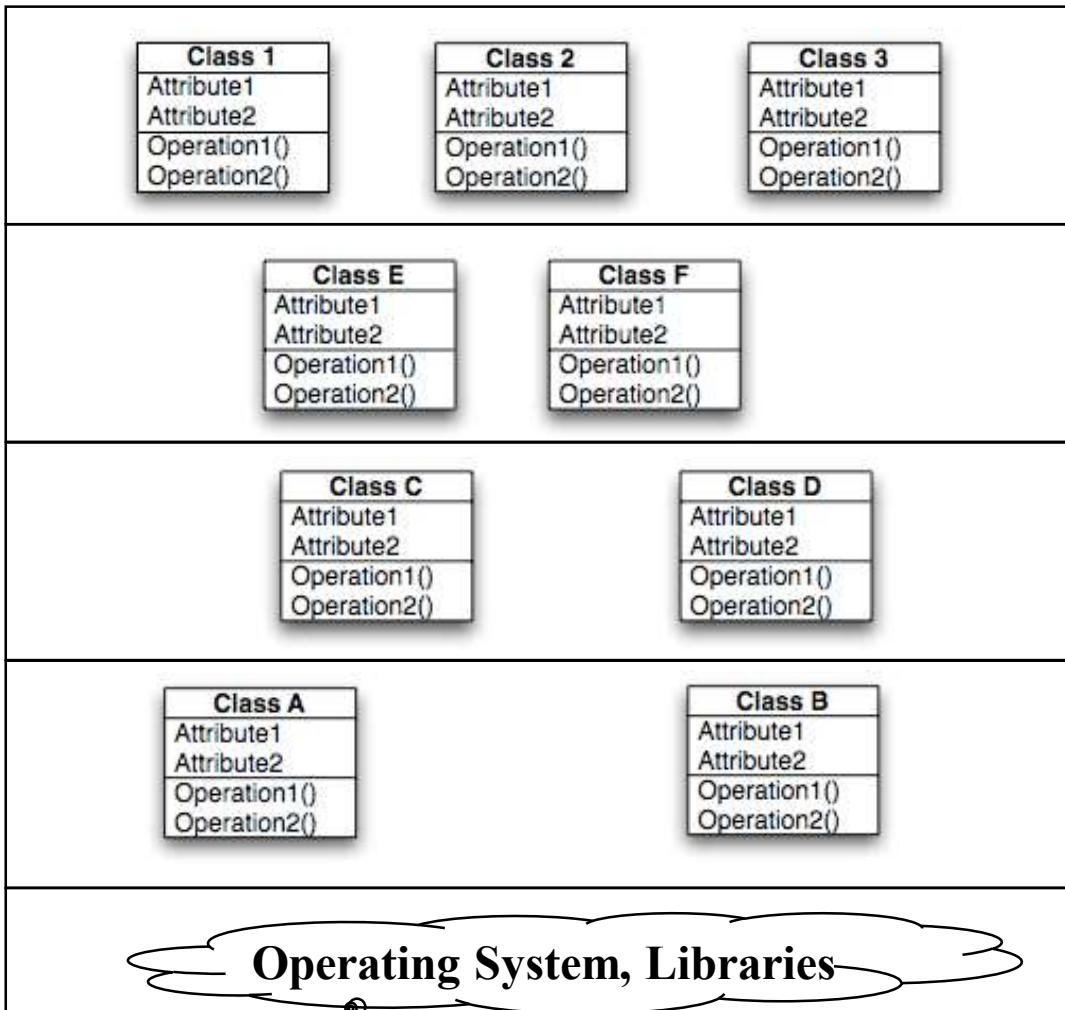
Subsystem Interface Objects

Virtual Machine

- A **virtual machine** is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A virtual machine is an abstraction that provides a set of attributes and operations
- The terms layer and virtual machine can be used interchangeably
 - Also sometimes called “level of abstraction”.

Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.



Virtual Machine 4 .

Virtual Machine 3

Virtual Machine 2

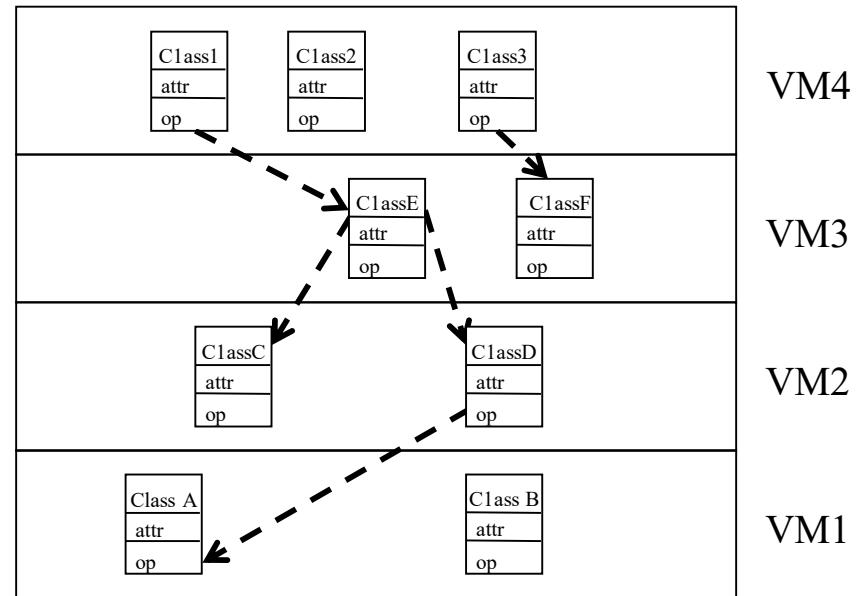
Virtual Machine 1

Existing System

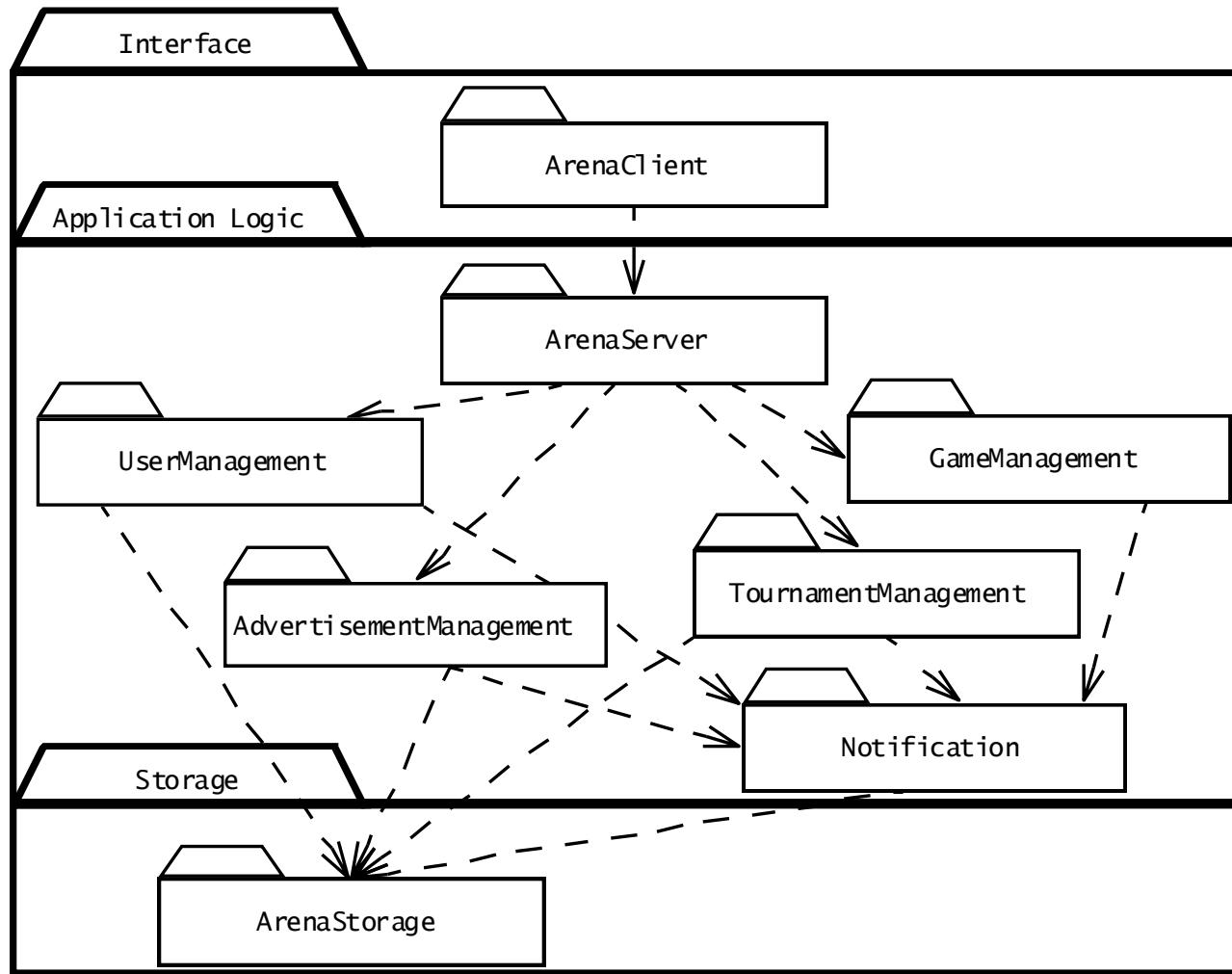
Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

Design goals:
Maintainability,
flexibility.



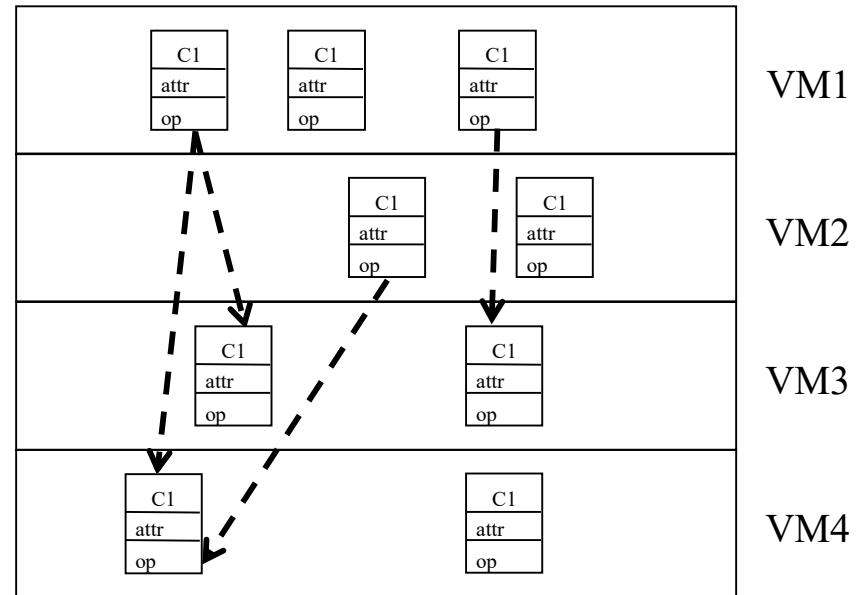
Opaque Layering in ARENA



Open Architecture (Transparent Layering)

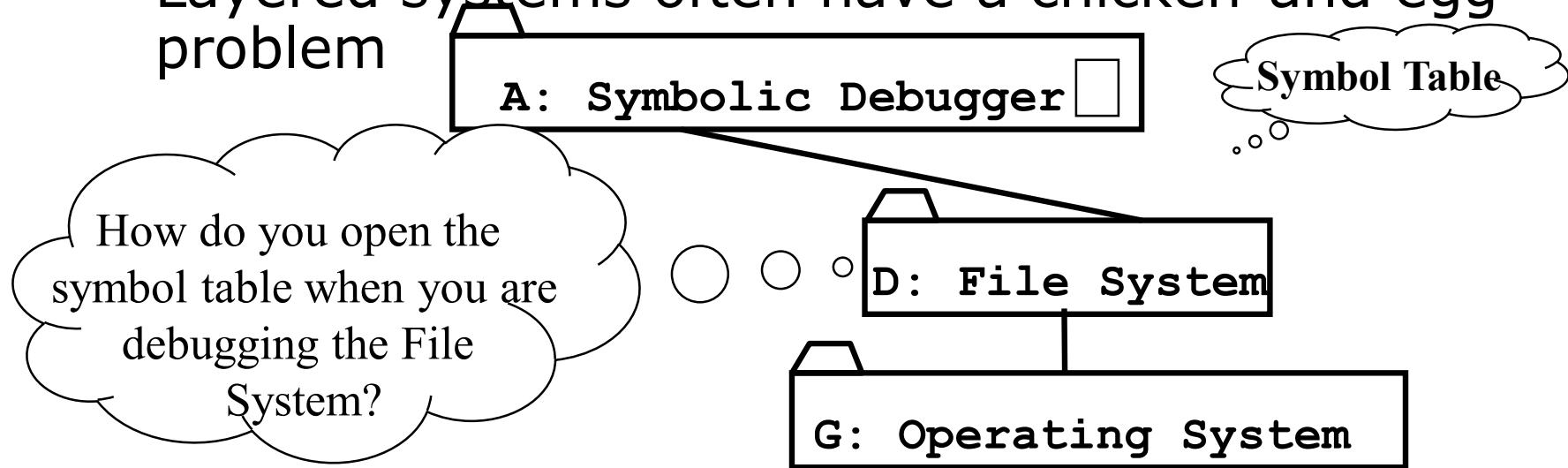
- Each virtual machine can call operations from any layer below

Design goal:
Runtime efficiency



Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design, because hierarchy reduces complexity
 - low coupling
- Closed architectures are more portable
- Open architectures are more efficient
- Layered systems often have a chicken-and egg problem



Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
 - **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via many associations
 - **Low coherence:** Lots of miscellaneous and auxiliary classes, almost no associations
- **Coupling** measures dependency among subsystems
 - **High coupling:** Changes to one subsystem will have high impact on the other subsystem
 - **Low coupling:** A change in one subsystem does not affect any other subsystem.

Coupling and Coherence of Subsystems

Good Design

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
 - **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via associations
 - **Low coherence:** Lots of miscellaneous and auxiliary classes, no associations
- **Coupling** measures dependency among subsystems
 - **High coupling:** Changes to one subsystem will have high impact on the other subsystem
 - **Low coupling:** A change in one subsystem does not affect any other subsystem

How to achieve high Coherence

- High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
 - Does one subsystem always call another one for a specific service?
 - Yes: Consider moving them together into the same subsystem.
 - Which of the subsystems call each other for services?
 - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
 - Can the subsystems even be hierarchically ordered (in layers)?

How to achieve Low Coupling

- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**, Parnas)
- Questions to ask:
 - Does the calling class really have to know any attributes of classes in the lower layers?
 - Is it possible that the calling class calls only operations of the lower level classes?

David Parnas, *1941,
Developed the concept of
modularity in design.



Architectural Style vs Architecture

- **Subsystem decomposition:** Identification of subsystems, services, and their association to each other (hierarchical, peer-to-peer, etc)
- **Architectural Style:** A pattern for a subsystem decomposition
- **Software Architecture:** Instance of an architectural style.

Examples of Architectural Styles

- Client/Server
- Peer-To-Peer
- Repository
- Model/View/Controller
- Three-tier, Four-tier Architecture
- Service-Oriented Architecture (SOA)
- Pipes and Filters

Client/Server Architectural Style

- One or many **servers** provide services to instances of subsystems, called **clients**
- Each client calls on the server, which performs some service and returns the result
 - The clients know the *interface* of the server
 - The server does not need to know the interface of the client
- The response in general is immediate
- End users interact only with the client.



Client/Server Architectures

- Often used in the design of database systems
 - Front-end: User application (client)
 - Back end: Database access and manipulation (server)
- Functions performed by client:
 - Input from the user (Customized user interface)
 - Front-end processing of input data
- Functions performed by the database server:
 - Centralized data management
 - Data integrity and database consistency
 - Database security

Design Goals for Client/Server Architectures

Service Portability

Server runs on many operating systems and many networking environments

Location-Transparency

Server might itself be distributed, but provides a single "logical" service to the user

High Performance

Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations

Scalability

Server can handle large # of clients

Flexibility

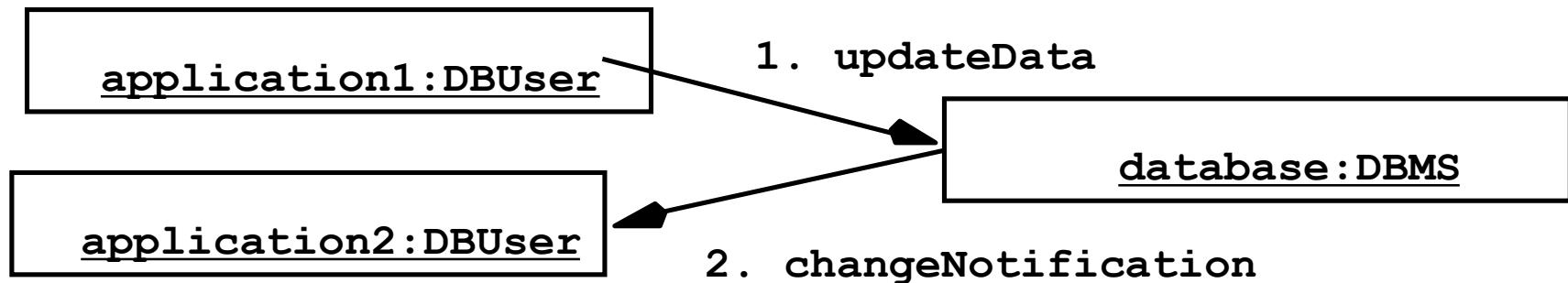
User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)

Reliability

A measure of success with which the observed behavior of a system confirms to the specification of its behavior (Chapter 11: Testing)

Problems with Client/Server Architectures

- Client/Server systems do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example:
 - Database must process queries from application and should be able to send notifications to the application when data have changed



Peer-to-Peer Architectural Style

Specification of Client/Server Architectural Style

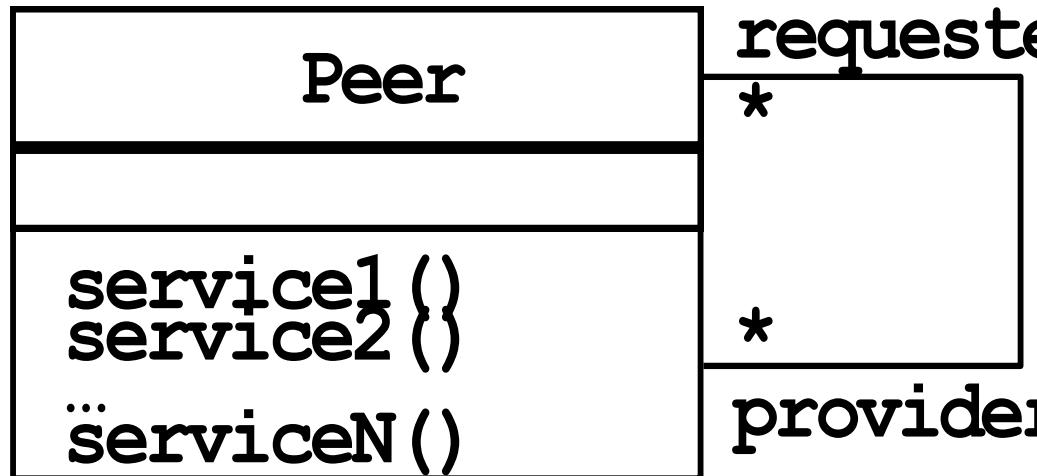
“Clients can be servers and servers can be clients”

Introduction a new abstraction: **Peer can be both
“Clients and servers”**

How do we model this statement? With Inheritance?

Proposal 1: “A peer can be either a client or a server”

Proposal 2: “A peer can be a client as well as a server”.



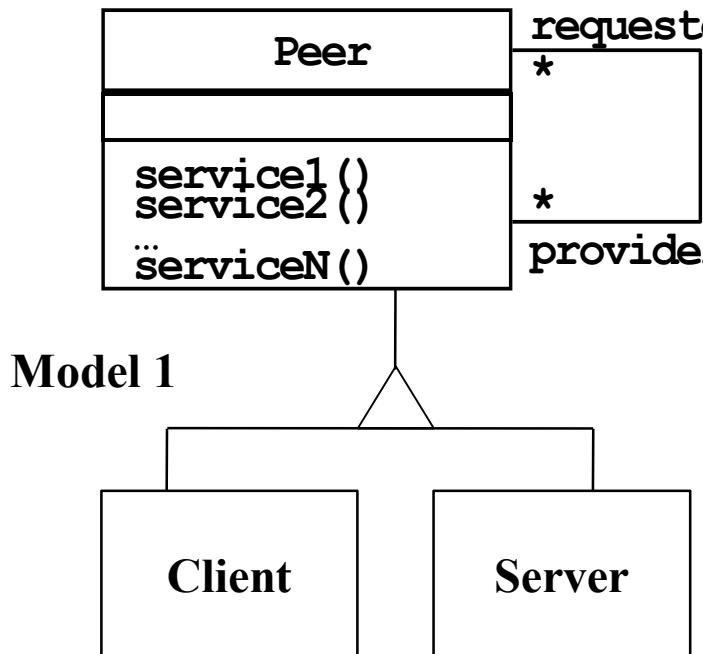
Relationship Client/Server & Peer-to-Peer

Problem statement “Clients can be servers and servers can be clients”

Which model is correct?

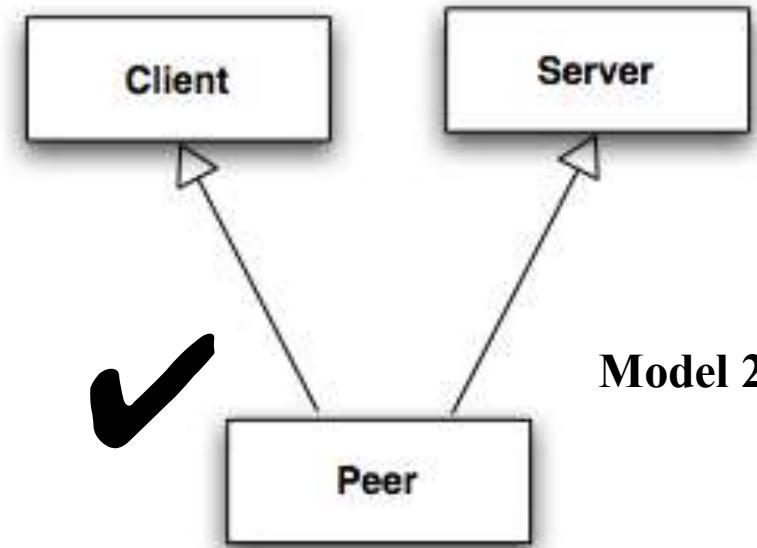
Model 1: “A peer can be either a client or a server”

Model 2: “A peer can be a client as well as a server”



Model 1

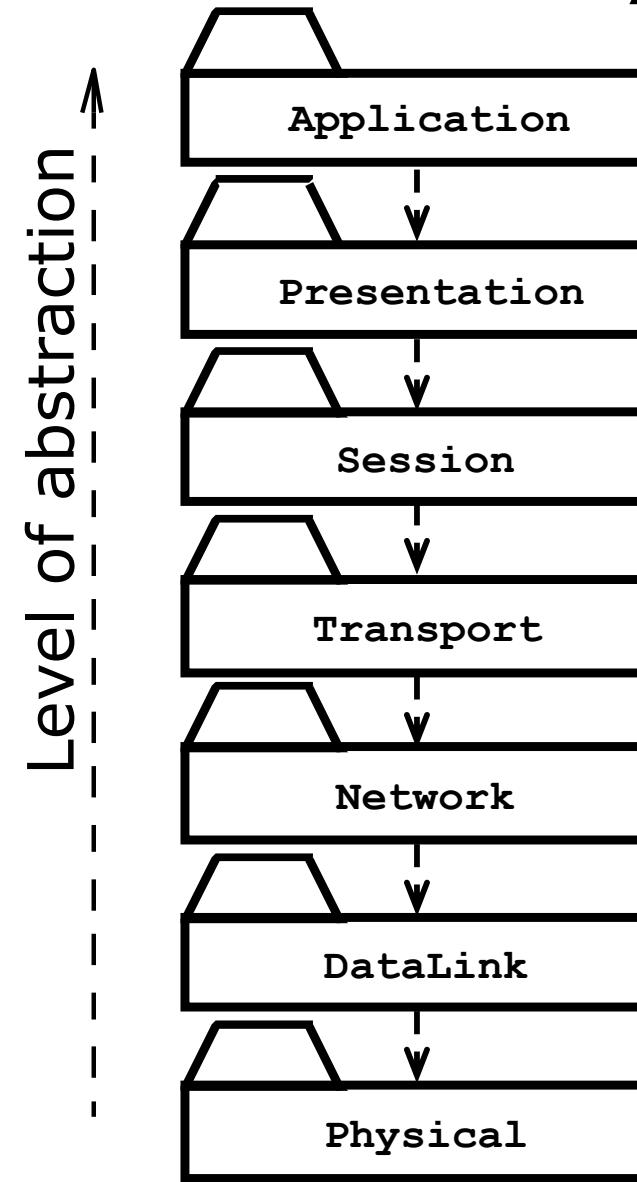
?



Model 2

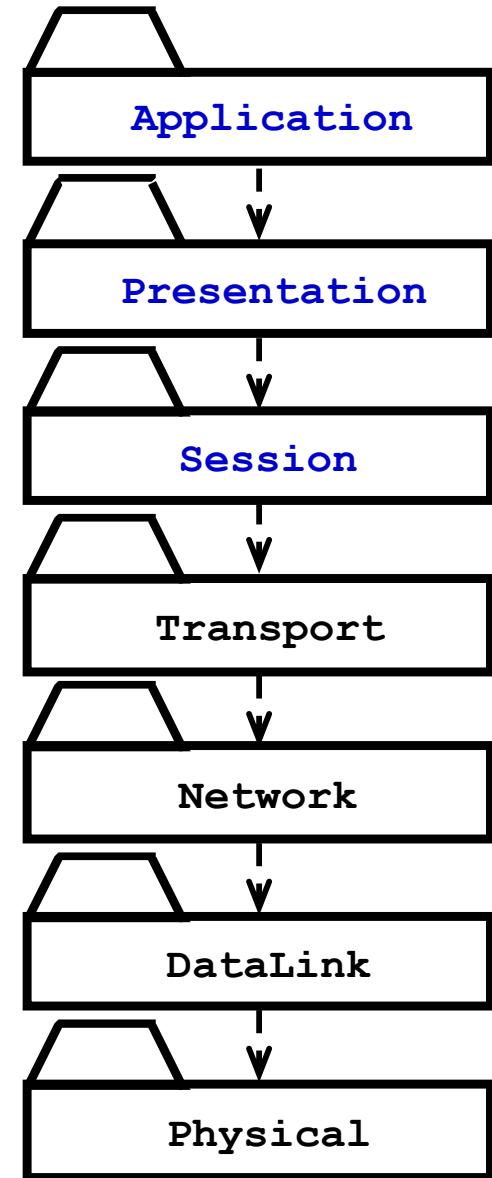
Example: Peer-to-Peer Architectural Style

- ISO's OSI Reference Model
 - ISO = International Standard Organization
 - OSI = Open System Interconnection
- Reference model which defines 7 layers and communication protocols between the layers



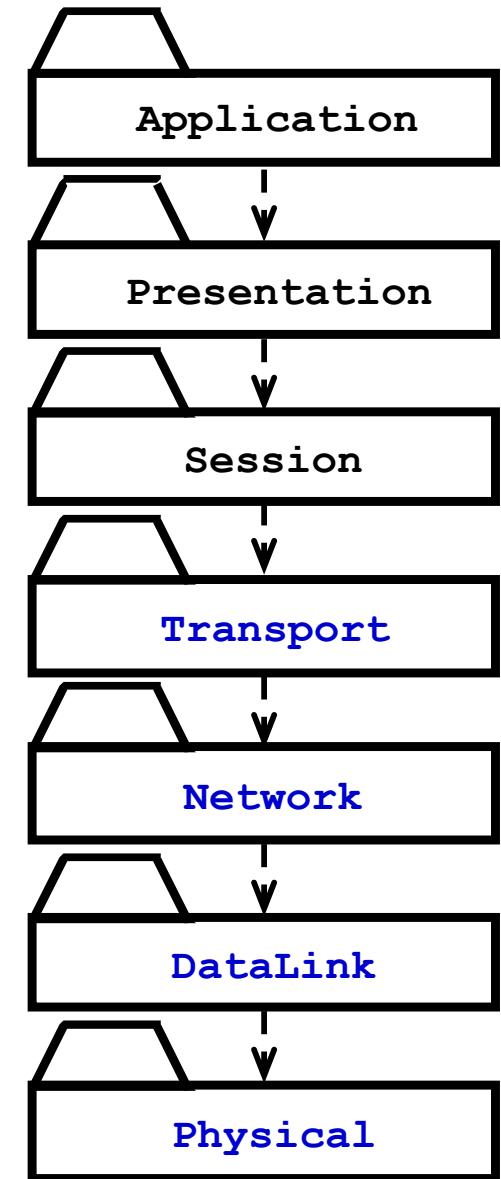
OSI Model Layers and Services

- The **Application layer** is the system you are building (unless you build a protocol stack)
 - ! • The application layer is usually layered itself
- The **Presentation layer** performs data transformation services, such as byte swapping and encryption
- The **Session layer** is responsible for initializing a connection, including authentication



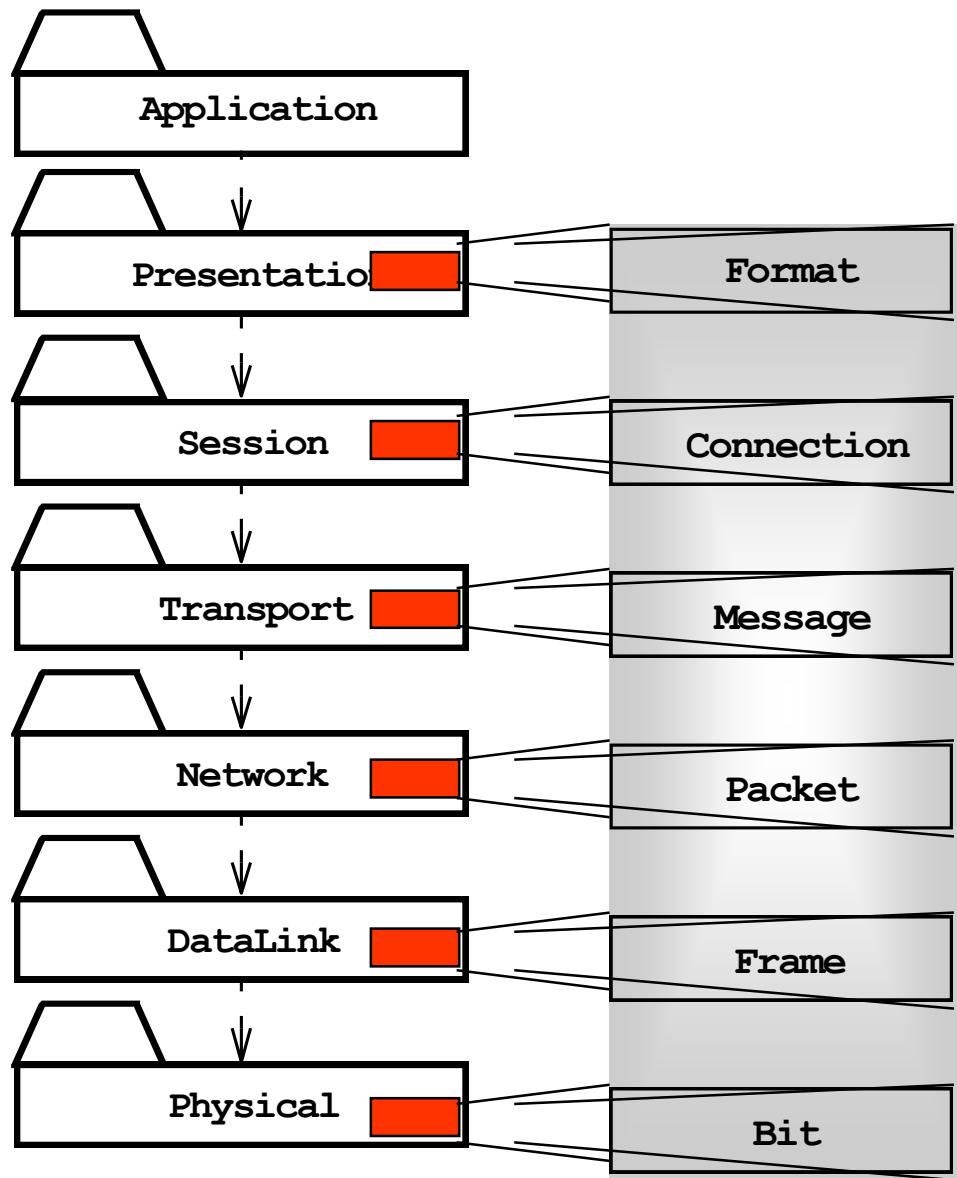
OSI Model Layers and their Services

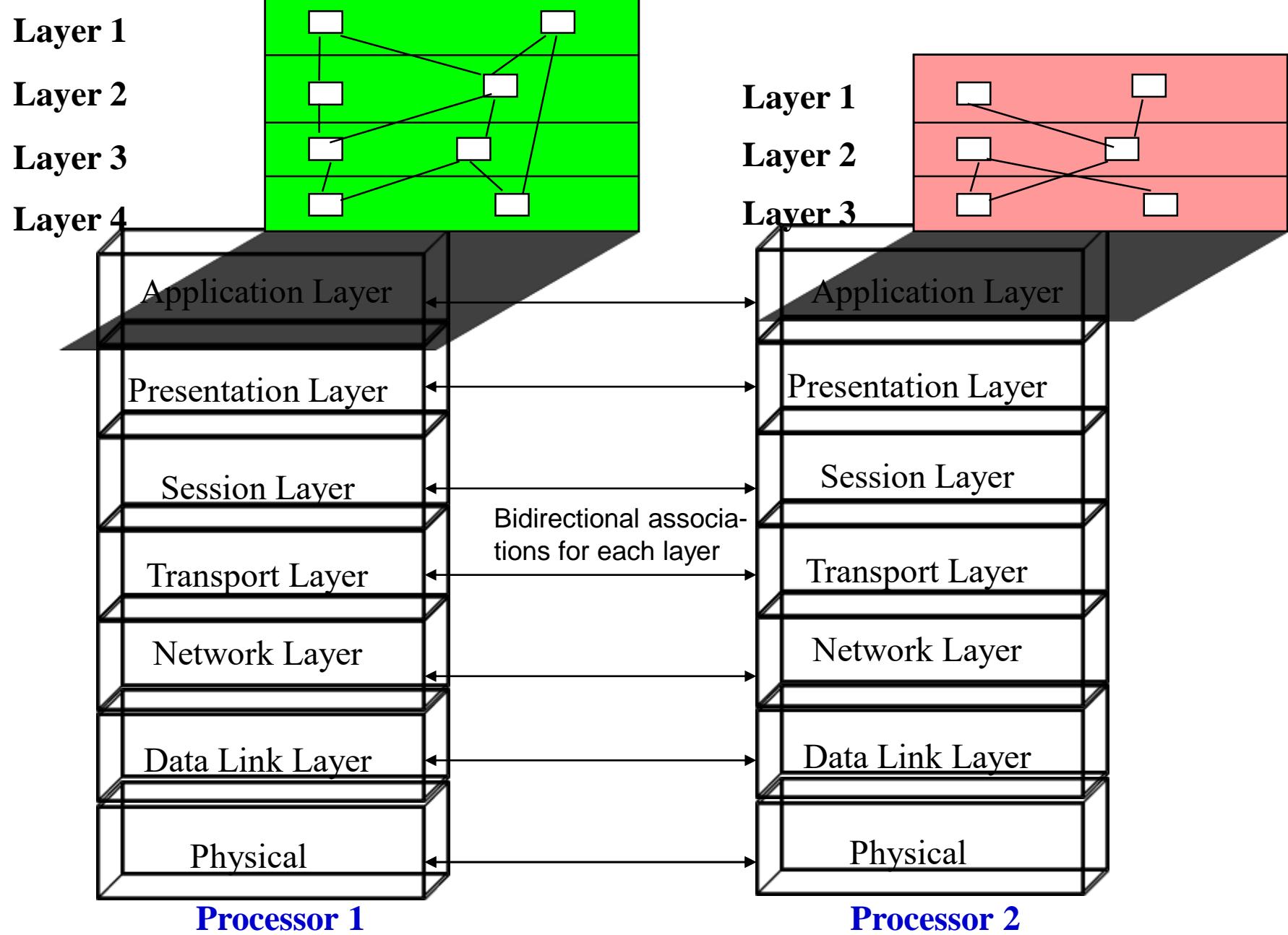
- The **Transport layer** is responsible for reliably transmitting messages
 - Used by Unix programmers who transmit messages over TCP/IP sockets
- The **Network layer** ensures transmission and routing
 - Services: Transmit and route data within the network
- The **Datalink layer** models frames
 - Services: Transmit frames without error
- The **Physical layer** represents the hardware interface to the network
 - Services: sendBit() and receiveBit()



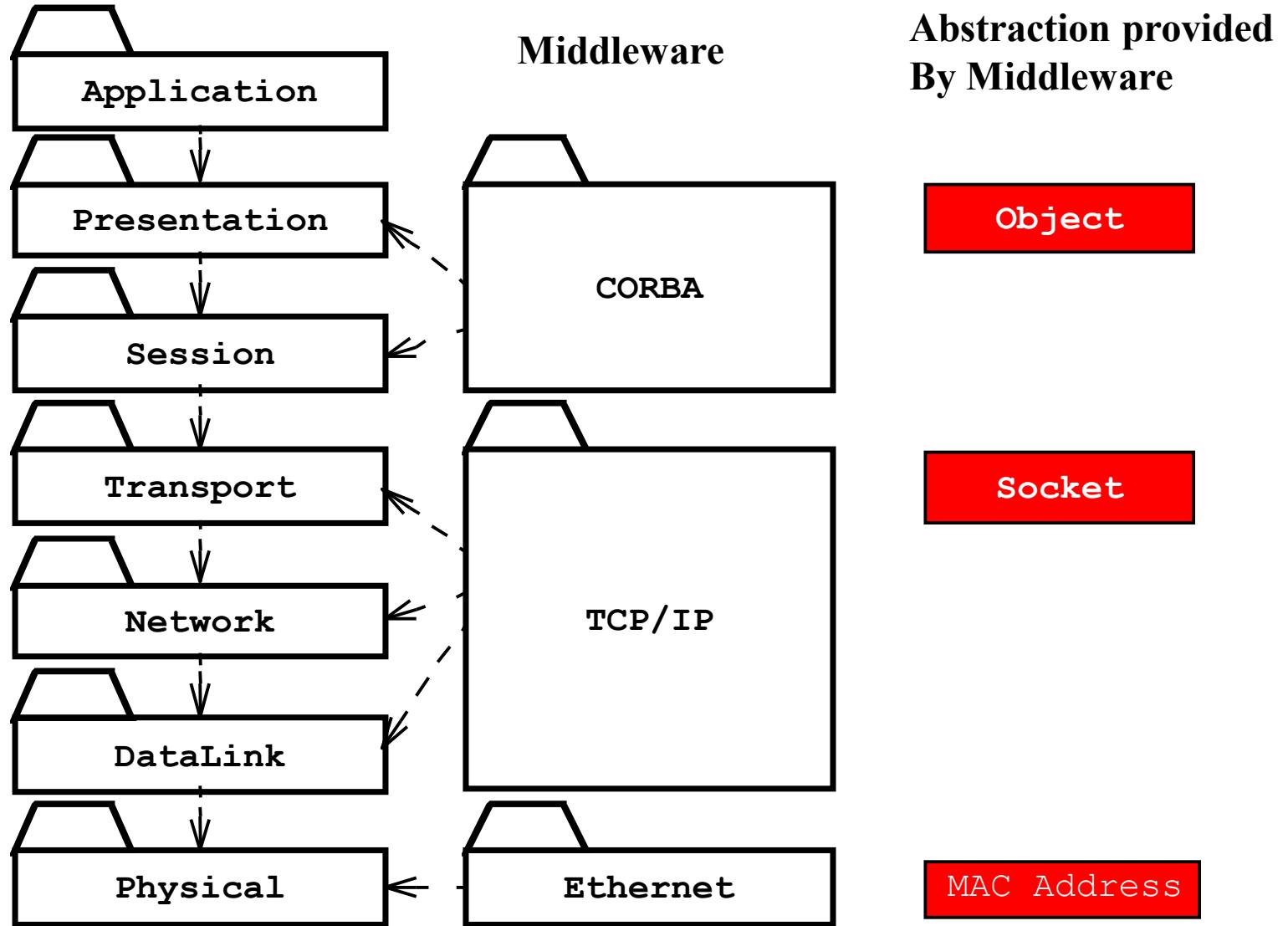
An Object-Oriented View of the OSI Model

- The OSI Model is a closed software architecture (i.e., it uses opaque layering)
- Each layer can be modeled as a UML package containing a set of classes available for the layer above



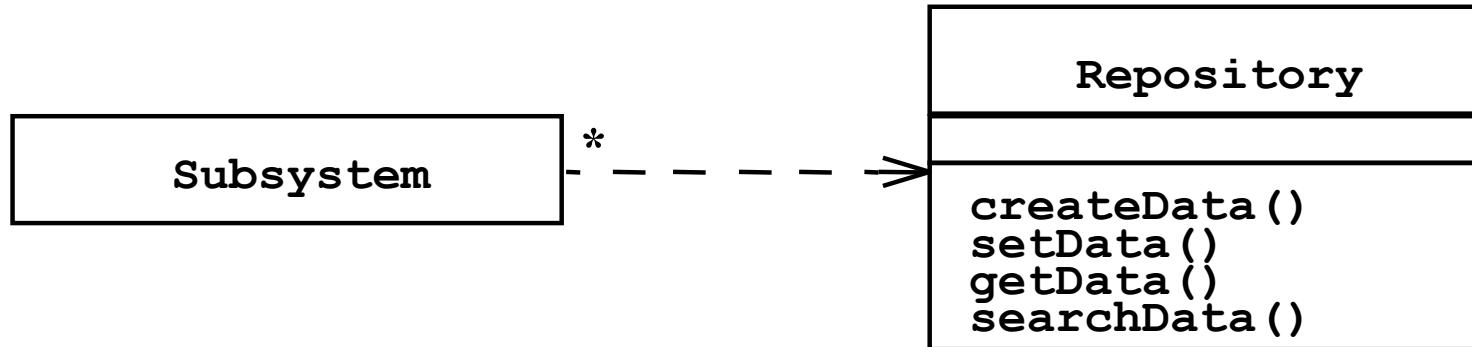


Middleware Allows Focus On Higher Layers



Repository Architectural Style

- Subsystems access and modify data from a single data structure called the **repository**
- Historically called **blackboard architecture** (Erman, Hayes-Roth and Reddy 1980)
- Subsystems are loosely coupled (interact only through the repository)
- Control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives



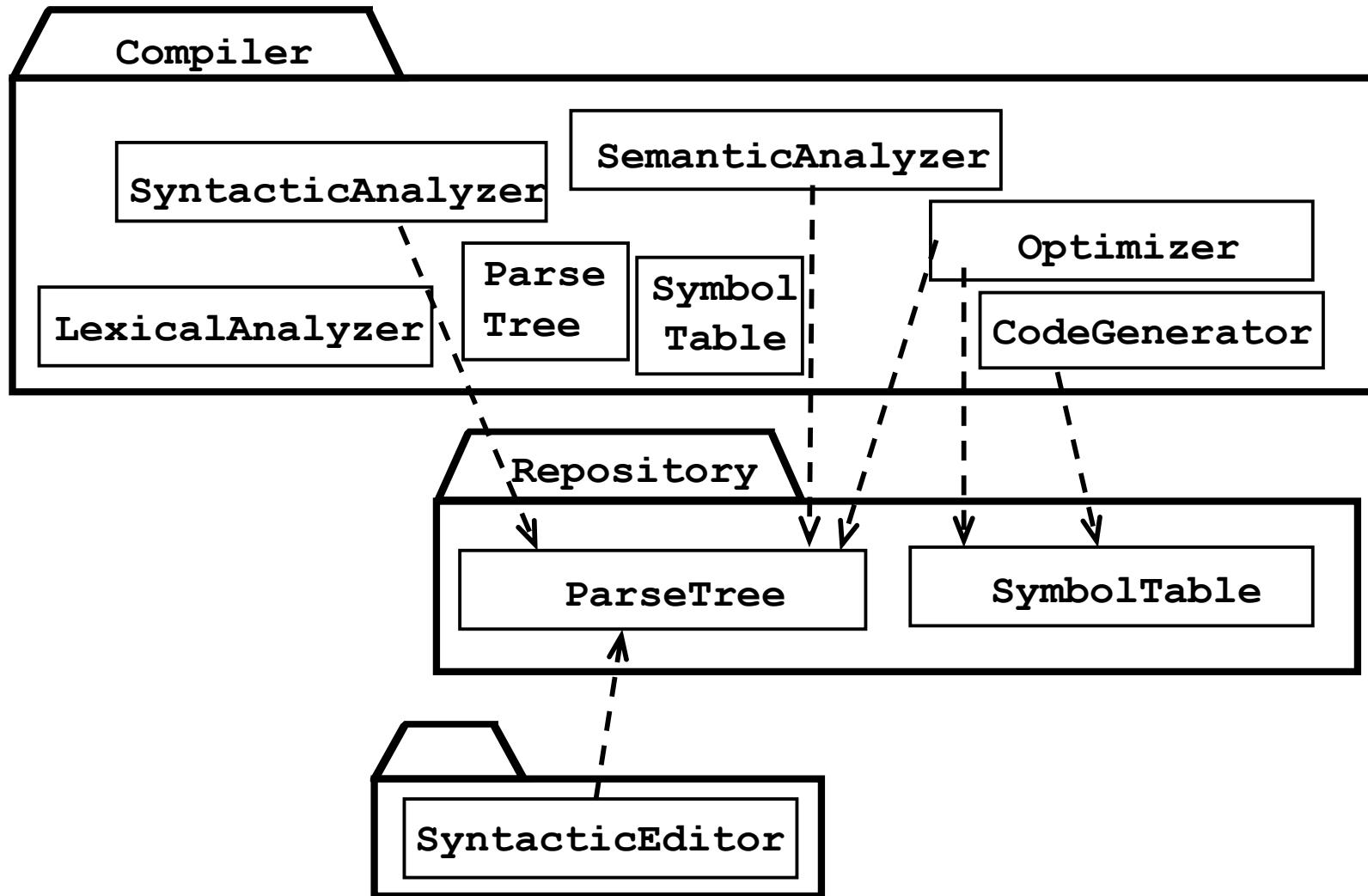
Blackboard Subsystem Decomposition

- A blackboard-system consists of three major components
 - The **blackboard**. A shared repository of problems, partial solutions and new information.
 - The **knowledge sources** (KSs). Each knowledge source embodies specific expertise. It reads the information placed on the blackboard and places new information on the blackboard.
 - The **control shell**. It controls the flow of problem-solving activity in the system, in particular how the knowledge sources get notified of new information put into the blackboard.



Raj Reddy, *1937, AI pioneer
- Major contributions to speech, vision, robotics, e.g. Hearsay and Harpy
- Founding Director of Robotics Institute, HCII, Center for Machine Learning, etc
1994: Turing Award (with Ed Feigenbaum).

Repository Architecture Example: Integrated Development Environment (IDE)



Repository Architecture Example: Integrated Development Environment (IDE)_2

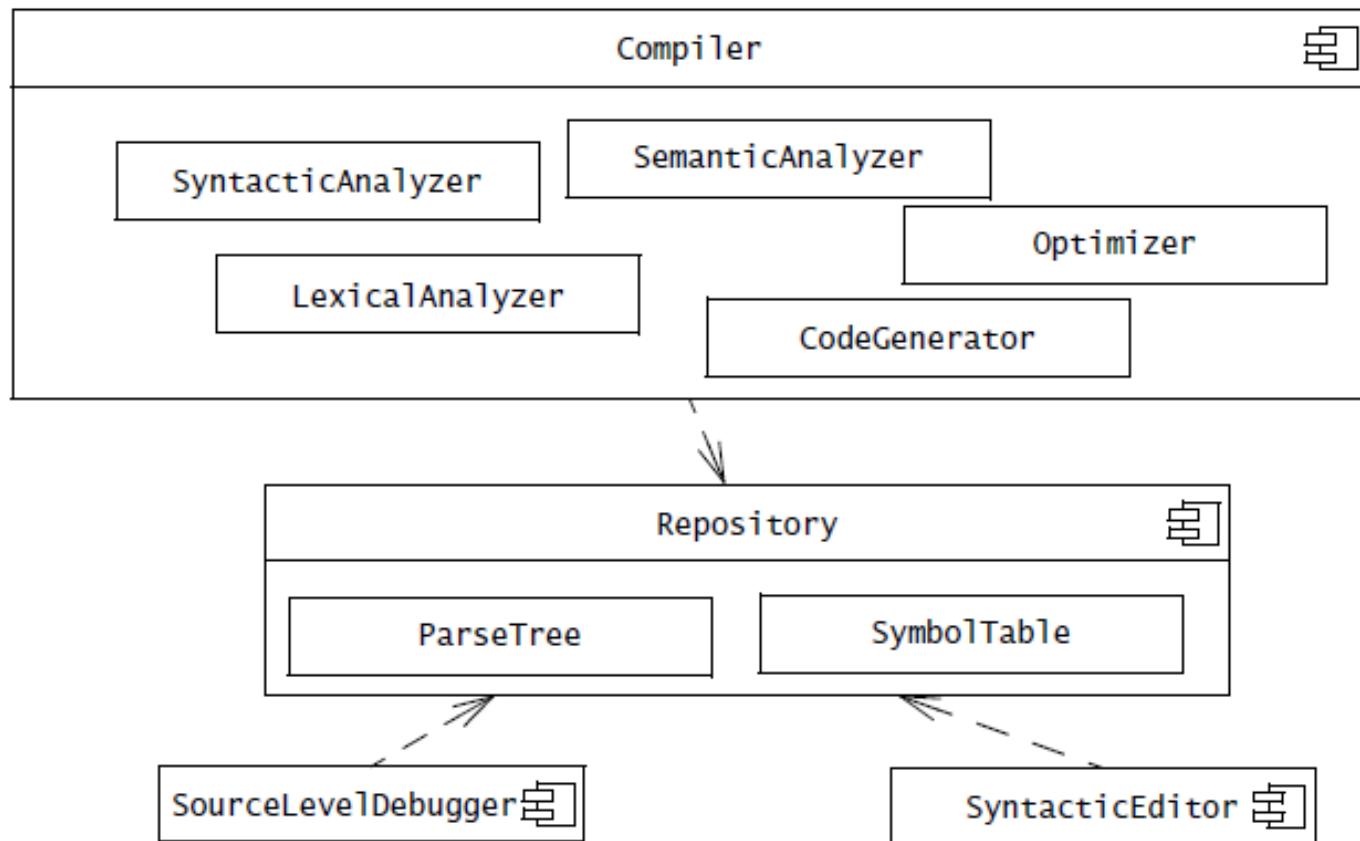


Figure 6-14 An instance of the repository architectural style (UML component diagram). A Compiler incrementally generates a ParseTree and a SymbolTable that can be used by SourceLevelDebuggers and SyntaxEditors.

Providing Consistent Views

- **Problem:** In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
 - The user interface cannot be reimplemented without changing the representation of the entity objects
 - The entity objects cannot be reorganized without changing the user interface
- **Solution: Decoupling!** The model-view-controller architectural style decouples data access (entity objects) and data presentation (boundary objects)
 - The Data Presentation subsystem is called the **View**
 - The Data Access subsystem is called the **Model**
 - The **Controller** subsystem mediates between View (data presentation) and Model (data access)
- Often called **MVC**.

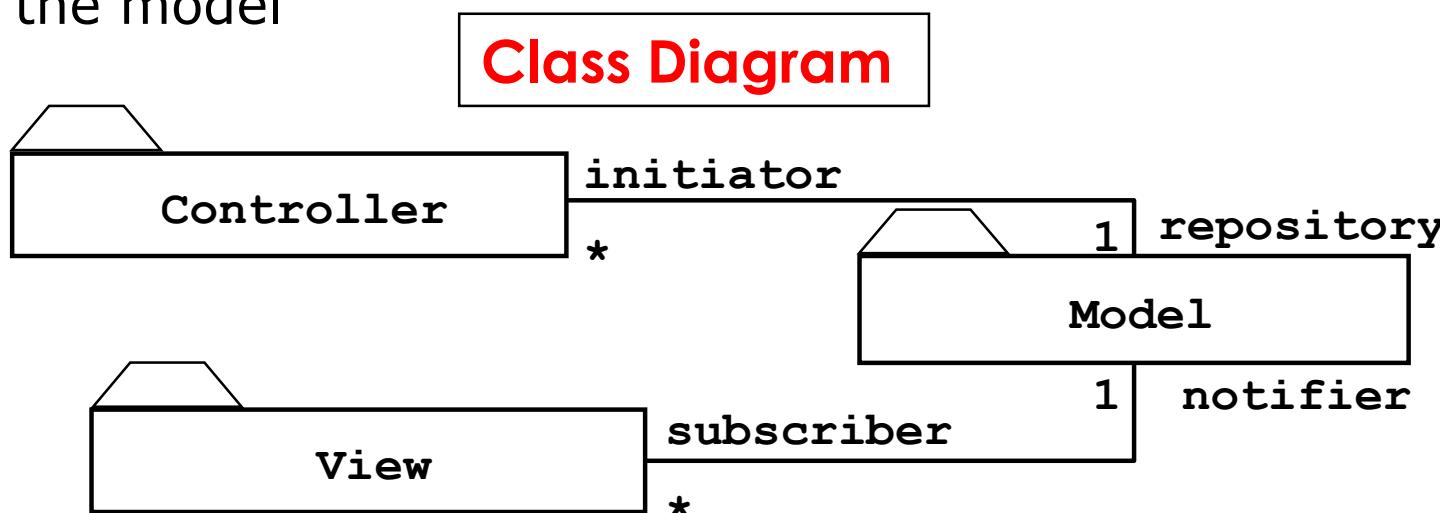
Model-View-Controller Architectural Style

- Subsystems are classified into 3 different types

Model subsystem: Responsible for application domain knowledge

View subsystem: Responsible for displaying application domain objects to the user

Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model

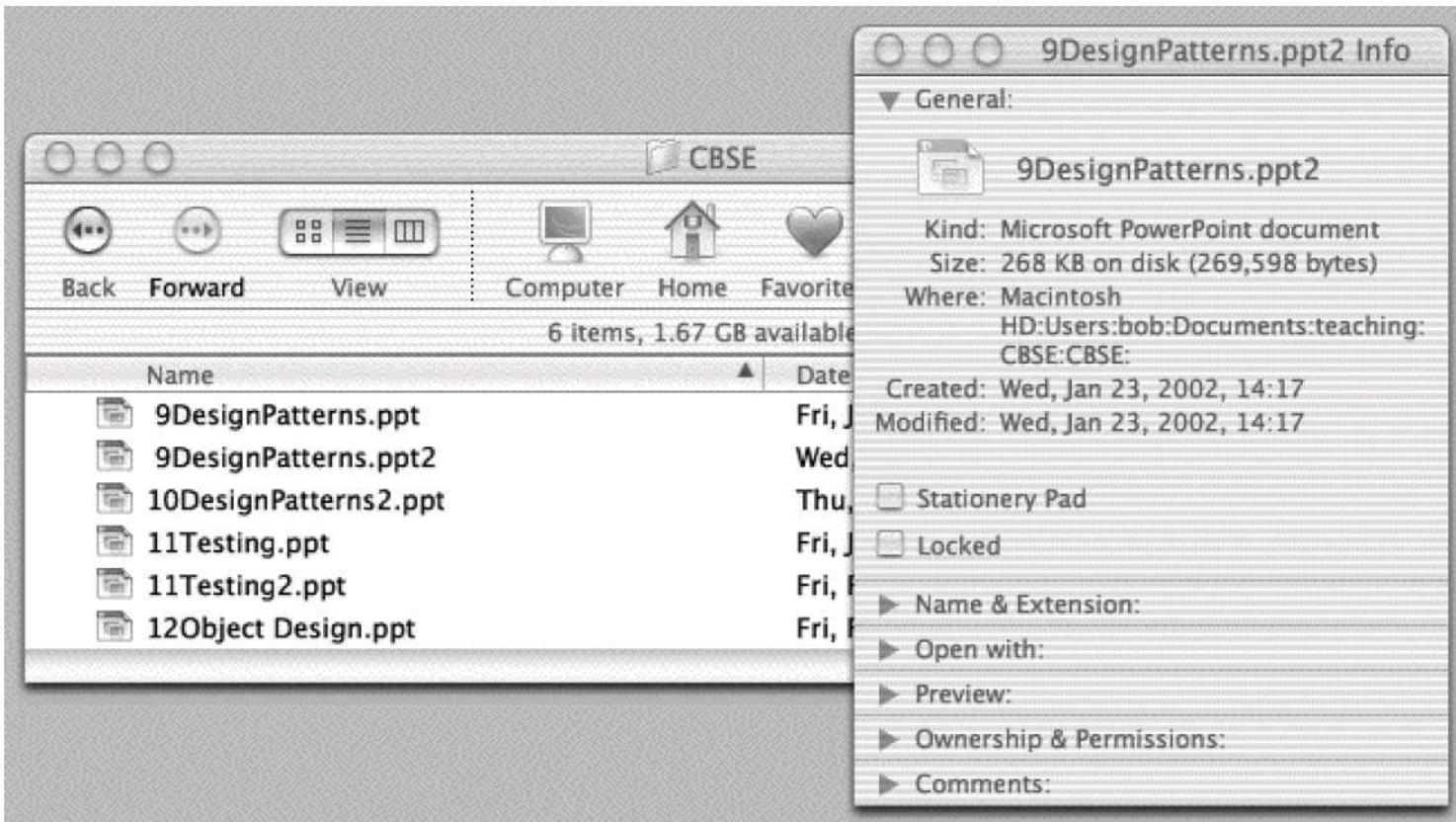


Better understanding with a Collaboration Diagram

UML Collaboration Diagram

- A **Collaboration Diagram** is an instance diagram that visualizes the interactions between objects as a flow of messages. Messages can be events or calls to operations
- Collaboration diagrams **describe the static structure** as well as the **dynamic behavior of a system**:
 - The static structure is obtained from the UML class diagram
 - Collaboration diagrams reuse the layout of classes and associations in the class diagram
 - The dynamic behavior is obtained from the dynamic model (UML sequence diagrams and UML statechart diagrams)
 - Messages between objects are labeled with a chronological number and placed near the link the message is sent over
- Reading a collaboration diagram involves starting at message 1.0, and following the messages from object to object.

Example: Modeling the Sequence of Events in MVC



An example of MVC architectural style. The “model” is the filename 9DesignPAtterns2.ppt. If the file name is changed, both views are updated by the “controller.”

Example: Modeling the Sequence of Events in MVC

UML Class Diagram

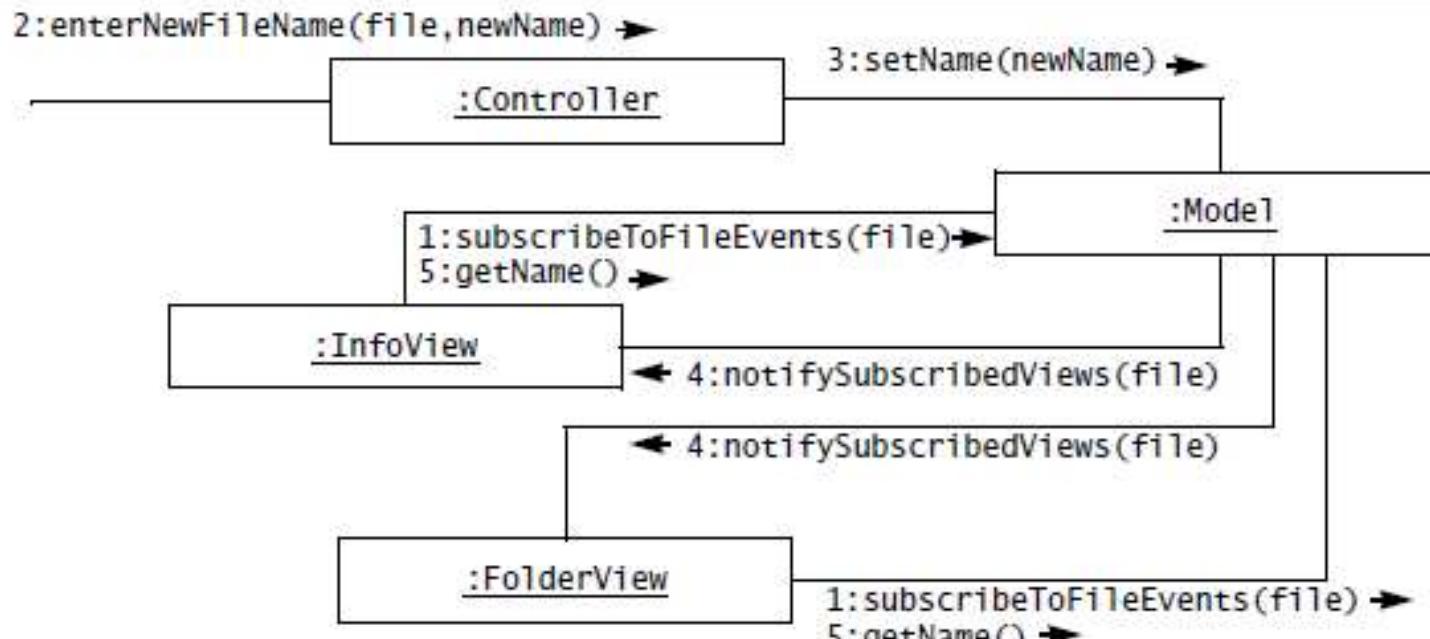
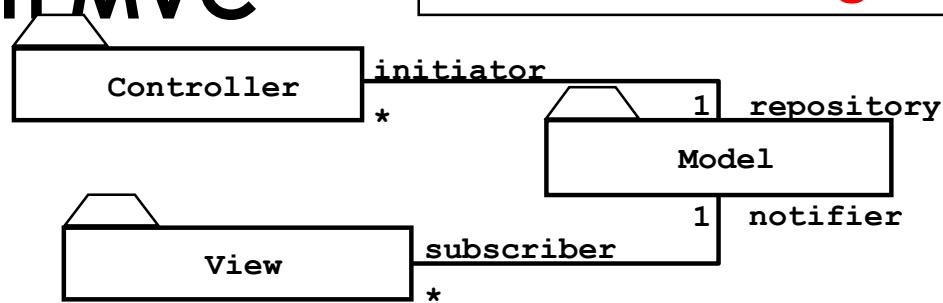


Figure 6-17 Sequence of events in the Model/View/Control architectural style (UML communication diagram).

3-Layer-Architectural Style

3-Tier Architecture

Definition: 3-Layer Architectural Style

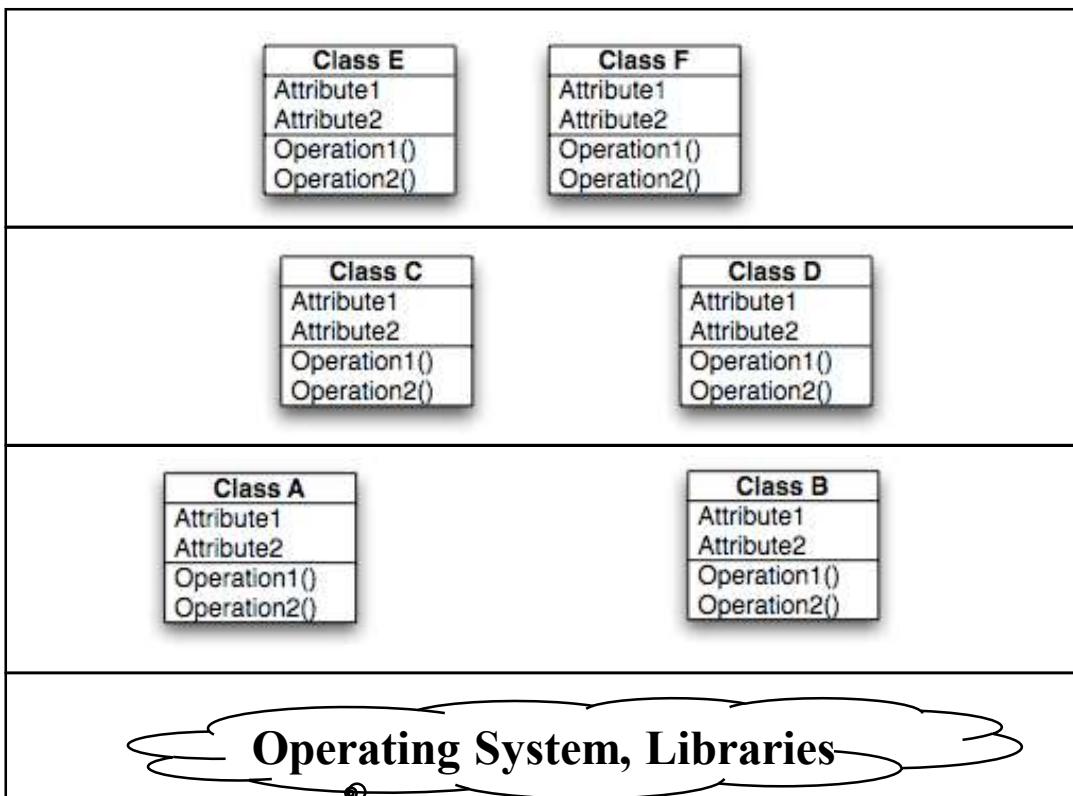
- An architectural style, where an application consists of 3 hierarchically ordered subsystems
 - A user interface, middleware and a database system
 - The middleware subsystem services data requests between the user interface and the database subsystem

Definition: 3-Tier Architecture

- A software architecture where the 3 layers are allocated on 3 separate hardware nodes
- Note: **Layer** is a type (e.g. class, subsystem) and **Tier** is an instance (e.g. object, hardware node)
- Layer and Tier are often used interchangeably.

Virtual Machines in 3-Layer Architectural Style

A 3-Layer Architectural Style is a hierarchy of 3 virtual machines usually called presentation, application and data layer



Presentation Layer
(Client Layer)

Application Layer
(Middleware,
Business Logic)

Data Layer

Existing System

Example of a 3-Layer Architectural Style

- Three-Layer architectural style are often used for the development of Websites:
 1. The **Web Browser** implements the user interface
 2. The **Web Server** serves requests from the web browser
 3. The **Database** manages and provides access to the persistent data.

A 4-Layer Architectural Style

4-Layer-architectural styles (4-Tier Architectures) are usually used for the development of electronic commerce sites. The layers are

1. The **Web Browser**, providing the user interface
2. A **Web Server**, serving static HTML requests
3. An **Application Server**, providing session management (for example the contents of an electronic shopping cart) and processing of dynamic HTML requests
4. A back end **Database**, that manages and provides access to the persistent data
 - In current 4-tier architectures, this is usually a relational Database management system (RDBMS).

Example of a 4-Layer Architectural Style

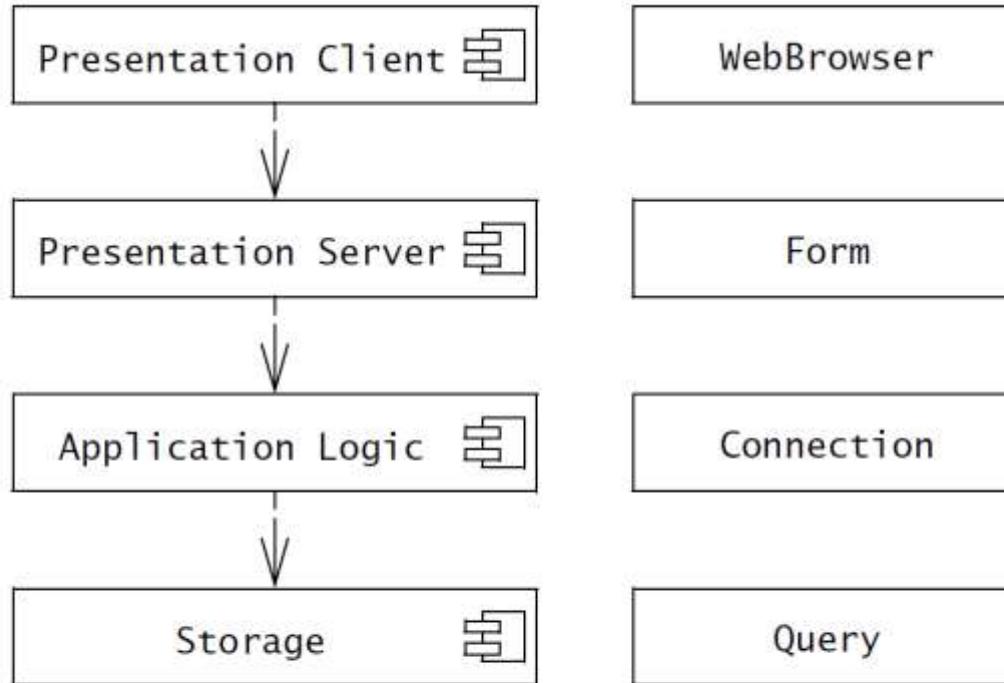


Figure 6-23 Four-tier architectural style (UML component diagram). The Interface layer of the three-tier style is split into two layers to enable more variability on the user interface style.

MVC vs. 3-Tier Architectural Style

- The **MVC** architectural style is **nonhierarchical** (triangular):
 - View subsystem sends updates to the Controller subsystem
 - Controller subsystem updates the Model subsystem
 - View subsystem is updated directly from the Model subsystem
- The **3-tier** architectural style is **hierarchical** (linear):
 - The presentation layer never communicates directly with the data layer (opaque architecture)
 - All communication must pass through the middleware layer
- **History:**
 - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc
 - 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms.

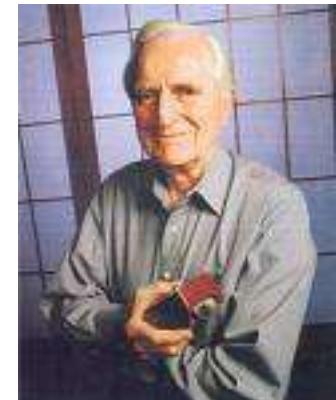
History at Xerox Parc



Xerox PARC (Palo Alto Research Center)

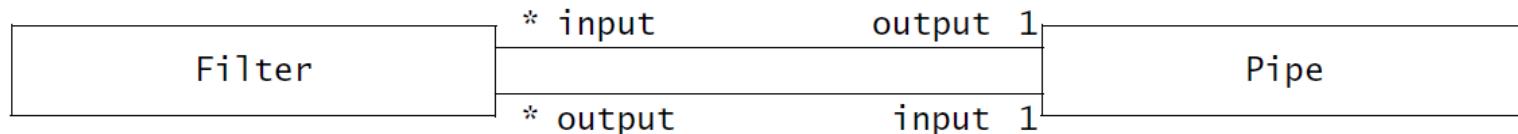
Founded in 1970 by Xerox, since 2002 a separate company PARC (wholly owned by Xerox). Best known for the invention of

- Laser printer (1973, Gary Starkweather)
- Ethernet (1973, Bob Metcalfe)
- Modern personal computer (1973, Alto, Bravo)
- Graphical user interface (GUI) based on WIMP
 - Windows, icons, menus and pointing device
 - Based on Doug Engelbart's invention of the mouse in 1965
- Object-oriented programming (Smalltalk, 1970s, Adele Goldberg)
- Ubiquitous computing (1990, Mark Weiser).



Pipes and Filters

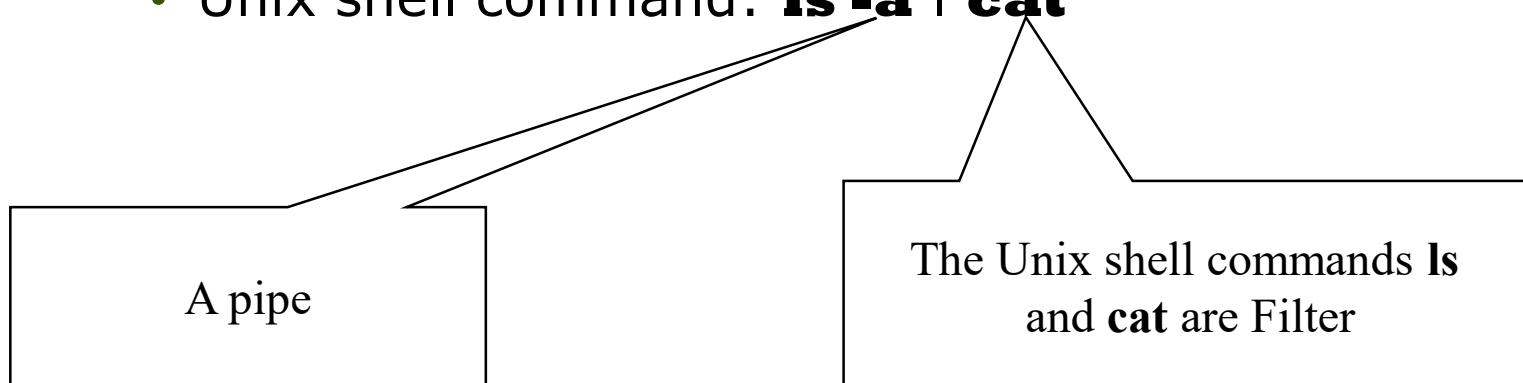
- A **pipeline** consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element
 - Usually some amount of buffering is provided between consecutive elements
 - The information that flows in these pipelines is often a stream of records, bytes or bits.



Pipe and filter architectural style. A Filter can have many inputs and outputs. A Pipe connects one of the outputs of a Filter to one of the inputs of another Filter.

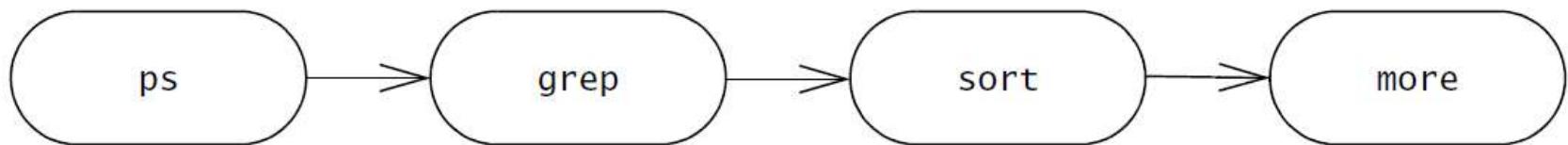
Pipes and Filters Architectural Style

- An architectural style that consists of two subsystems called pipes and filters
 - **Filter:** A subsystem that does a processing step
 - **Pipe:** A Pipe is a connection between two processing steps
- Each filter has an input pipe and an output pipe.
 - The data from the input pipe are processed by the filter and then moved to the output pipe
- Example of a Pipes-and-Filters architecture: Unix
 - Unix shell command: **ls -a | cat**



Pipes and Filters Architectural Style

```
% ps auxwww | grep dutoit | sort | more  
dutoit 19737 0.2 1.6 1908 1500 pts/6 0 15:24:36 0:00 -tcsh  
dutoit 19858 0.2 0.7 816 580 pts/6 S 15:38:46 0:00 grep dutoit  
dutoit 19859 0.2 0.6 812 540 pts/6 0 15:38:47 0:00 sort
```



Unix command line as an instance of the pipe and filter style (UML activity diagram).

Additional Readings

- E.W. Dijkstra (1968)
 - The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457
- D. Parnas (1972)
 - On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058
- L.D. Erman, F. Hayes-Roth (1980)
 - The Hearsay-II-Speech-Understanding System, ACM Computing Surveys, Vol 12. No. 2, pp 213-253
- J.D. Day and H. Zimmermann (1983)
 - The OSI Reference Model, Proc. IEEE, Vol.71, 1334-1340
- Jostein Gaarder (1991)
 - Sophie's World: A Novel about the History of Philosophy.

Summary

- System Design
 - An activity that reduces the gap between the problem and an existing (virtual) machine
- Design Goals Definition
 - Describes the important system qualities
 - Defines the values against which options are evaluated
- Subsystem Decomposition
 - Decomposes the overall system into manageable parts by using the principles of cohesion and coherence
- Architectural Style
 - A pattern of a typical subsystem decomposition
- Software architecture
 - An instance of an architectural style
 - Client Server, Peer-to-Peer, Model-View-Controller.

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 8, Object Design: Reusing Pattern Solutions



Object Design

- Purpose of object design:
 - Prepare for the implementation of the system model based on design decisions
 - Transform the system model (optimize it)
- Investigate alternative ways to implement the system model
 - Use design goals: minimize execution time, memory and other measures of cost.
- Object design serves as the basis of implementation.

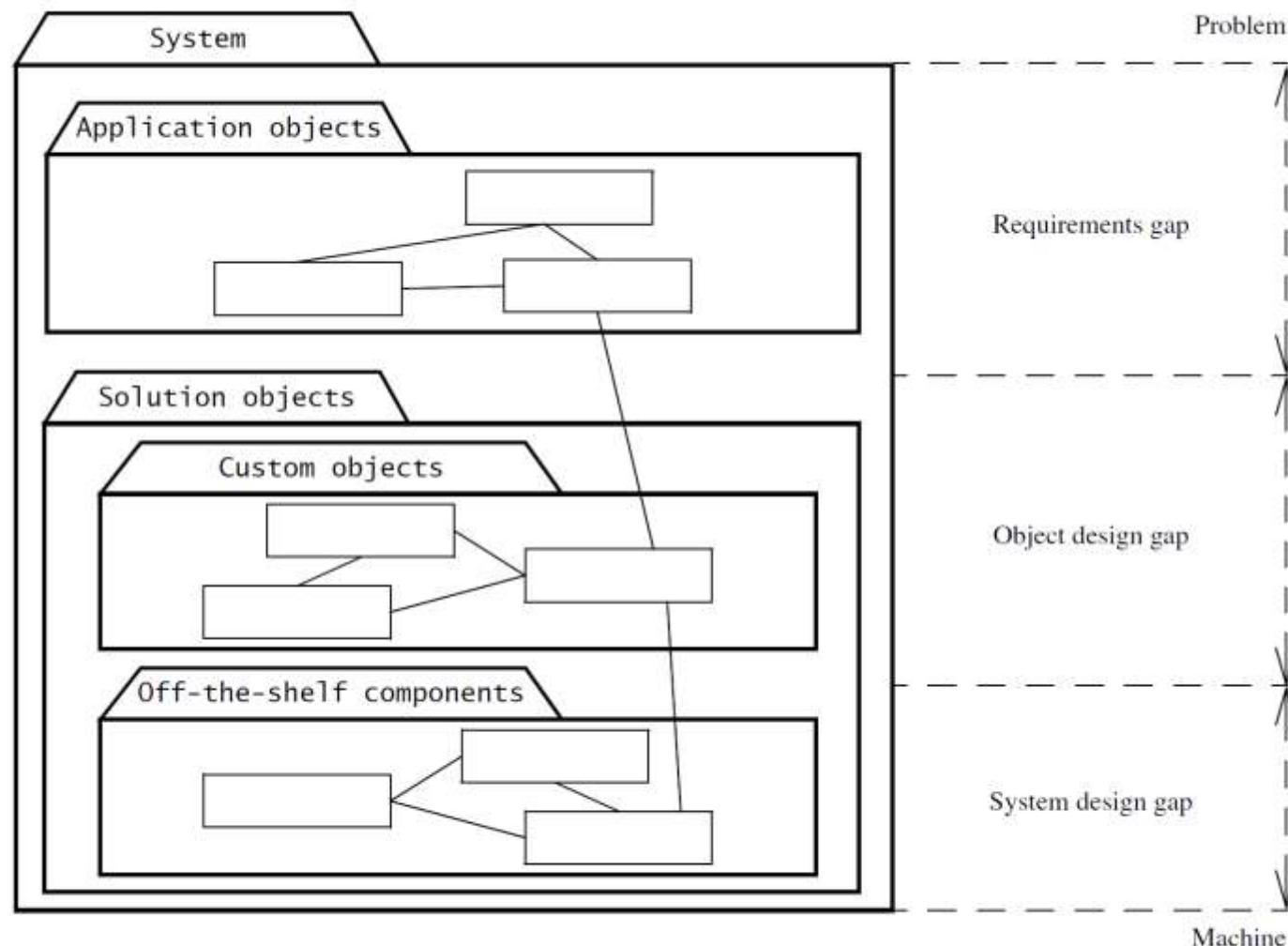
Object Design Activities

- During object design, we **close the gap between the application objects and the off-the-shelf components** by identifying additional solution objects and refining existing objects. Object design includes:
 - **reuse**, during which we identify off-the-shelf components and design patterns to make use of existing solutions
 - **service specification**, during which we precisely describe each class interface
 - **object model restructuring**, during which we transform the object design model to improve its understandability and extensibility
 - **model optimization**, during which we transform the object design model to address performance criteria such as response time or memory utilization.

An overview of the Object Design

- Conceptually, software system development fills the gap between a given problem and an existing machine.
 - Analysis reduces the gap between the problem and the machine by identifying objects representing problem-specific concepts.
 - System design reduces the gap between the problem and the machine in two ways. First, system design results in a virtual machine that provides a higher level of abstraction than the machine. This is done by selecting off-the-shelf components for standard services such as middleware, user interface toolkits, application frameworks, and class libraries. Second, system design identifies off-the-shelf components for application domain objects such as reusable class libraries of banking objects.

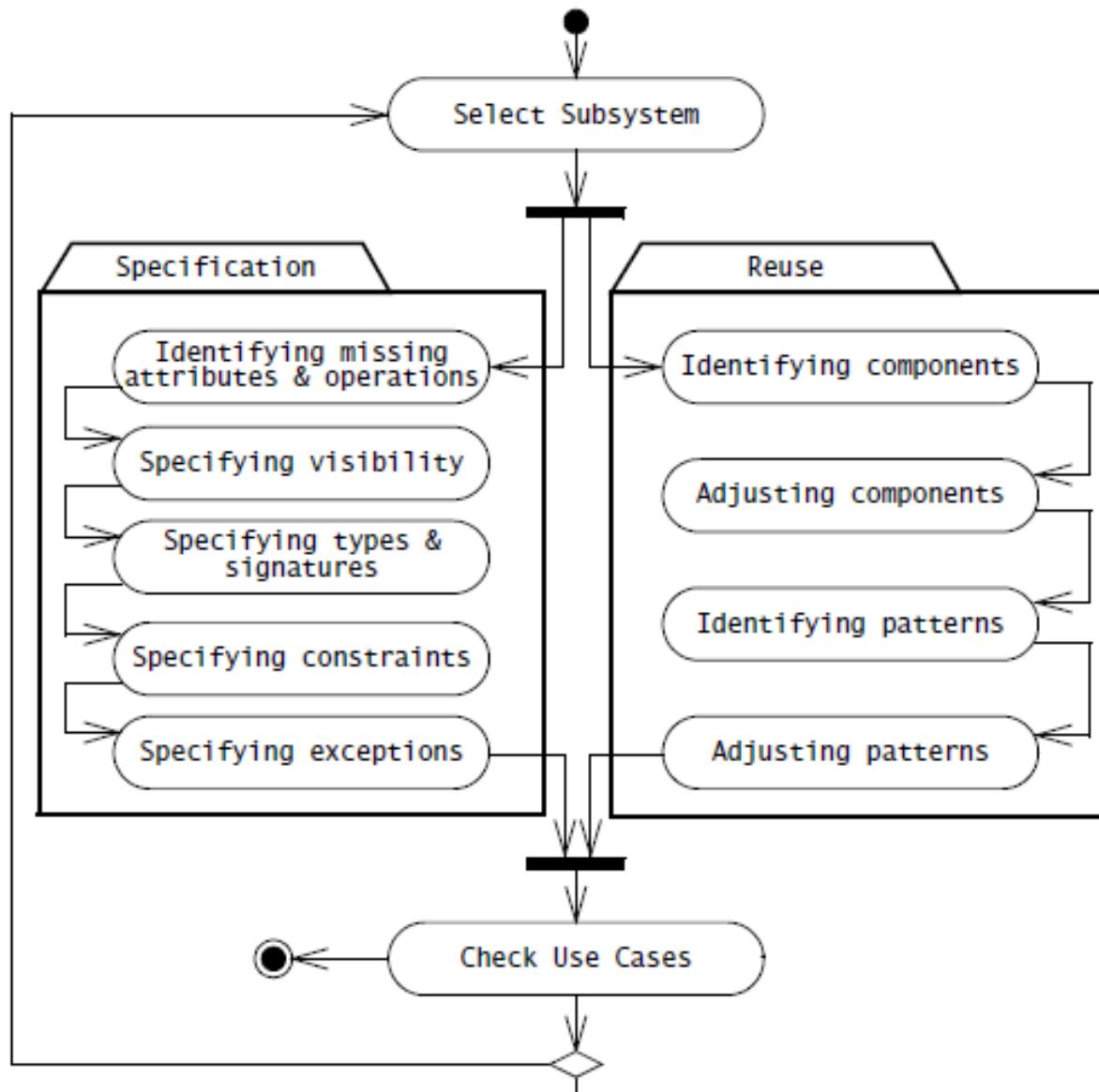
System Development



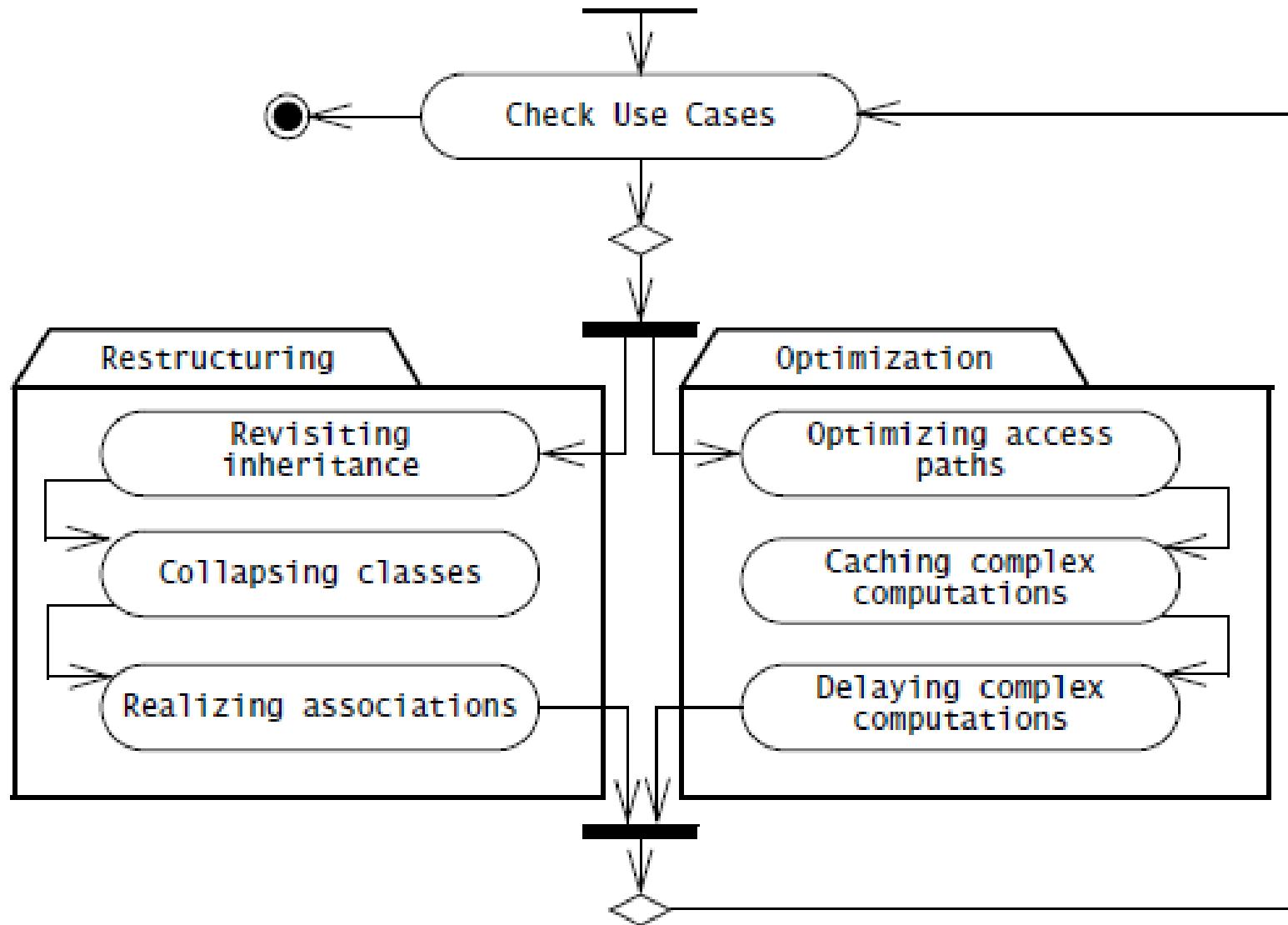
Customization: Build Custom Objects

- Problem: Close the object design gap
 - Develop new functionality
- Main goal:
 - Reuse knowledge from previous experience
 - Reuse functionality already available
- **Composition** (also called Black Box Reuse)
 - New functionality is obtained by aggregation
 - The new object with more functionality is an aggregation of existing objects
- **Inheritance** (also called White-box Reuse)
 - New functionality is obtained by inheritance

Activities of Object Design



Activities of Object Design_2

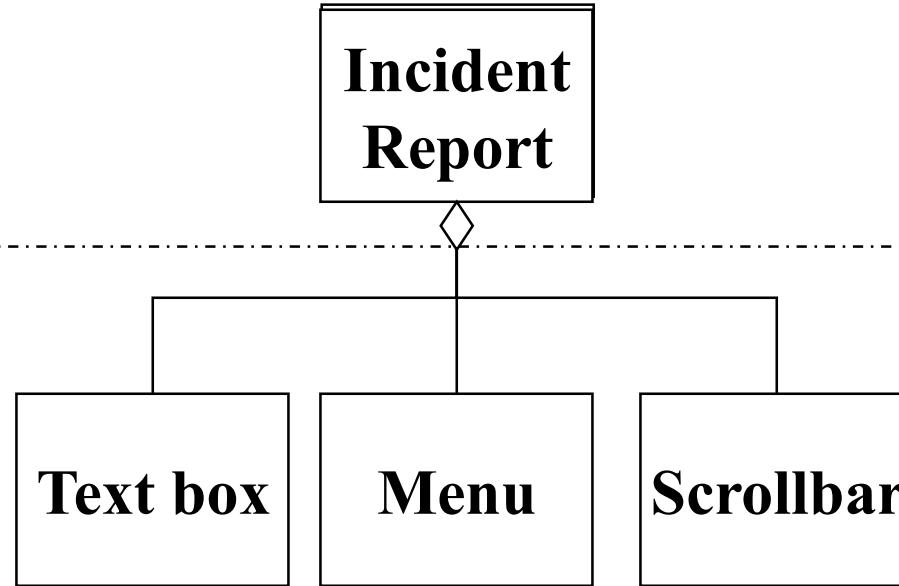


Application Objects and Design Objects

- **Application objects**, also called “domain objects,” represent concepts of the domain that are relevant to the system.
- **Solution objects** represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.
- **During system design**, we **identify solution objects in terms of software and hardware platforms**.
- **During object design**, we **refine and detail both application and solution objects and identify additional solution objects needed to bridge the object design gap**.

Identification of new Objects during Object Design

Requirements Analysis
(Language of Application Domain)



Object Design
(Language of Solution Domain)

Specification Inheritance and Implementation Inheritance

- During analysis, we use **inheritance to classify objects into taxonomies**. This allows us to **differentiate the common behavior** of the general **case, that is, the parent** (also called the “**base class**”), **from the behavior that is specific to specialized objects**, that are, the **children classes** (also called the “**derived classes**”).
- During object design, the focus of inheritance is to reduce redundancy and enhance extensibility. By factoring all redundant behavior into a single parent class, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we have to make changes only once for all children classes. By providing abstract classes and interfaces that are used by the application, we can write new specialized behavior.

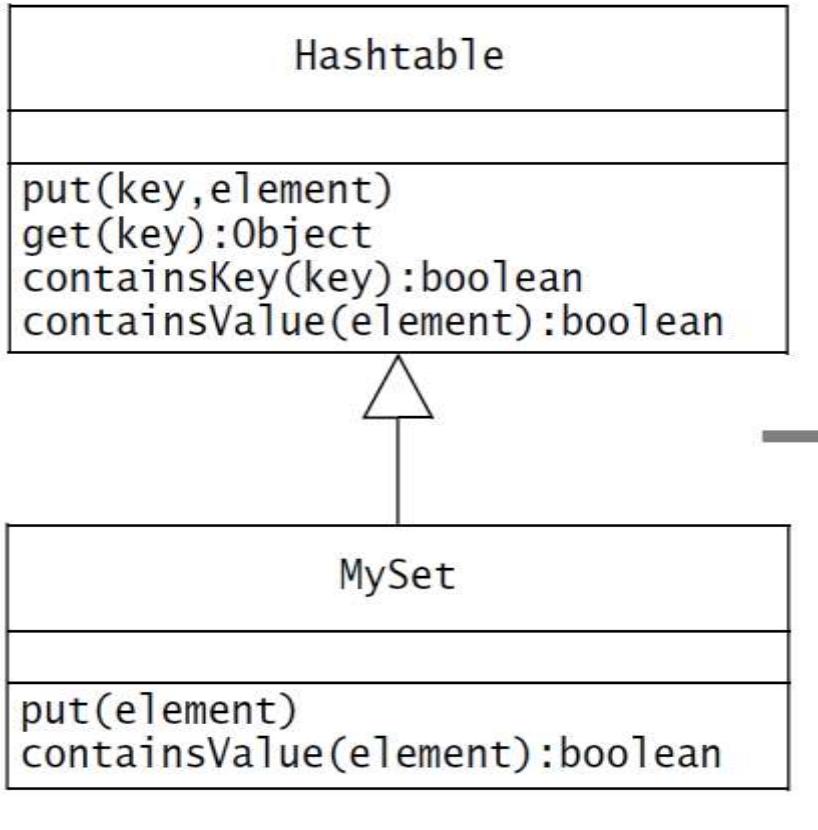
Specification Inheritance and Implementation Inheritance_2

- Although inheritance can make an analysis model more understandable and an object design model more modifiable or extensible, these benefits do not occur automatically.
- On the contrary, **inheritance is such a powerful mechanism that novice developers often produce code that is more obfuscated and more brittle than if they had not used inheritance in the first place.**

Specification Inheritance and Implementation

Inheritance_3

Object design model before transformation



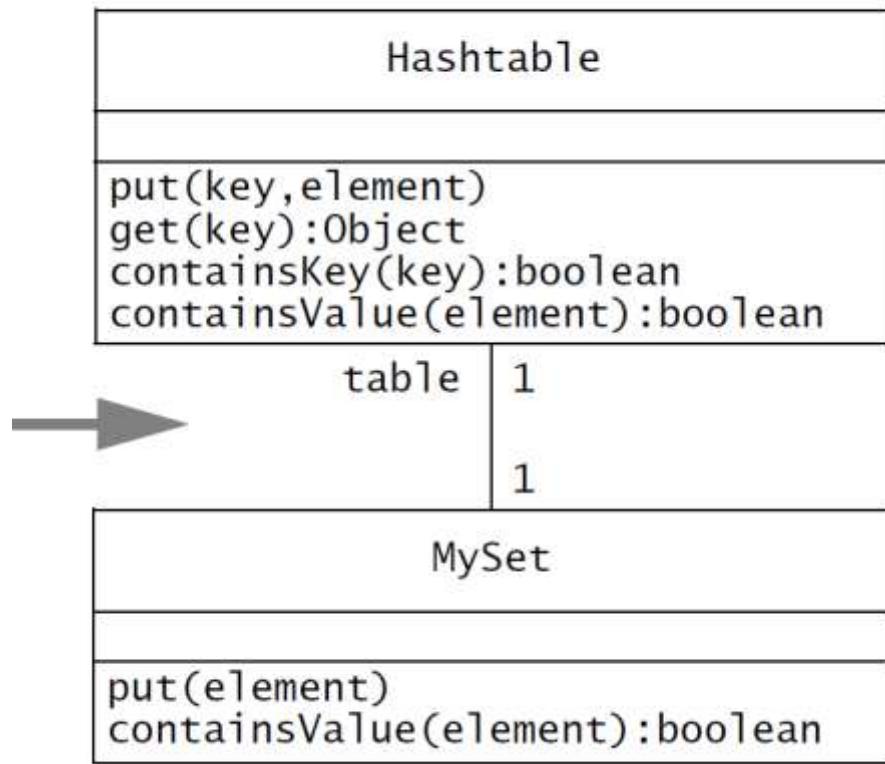
/ Implementation of MySet using inheritance */*

```
class MySet extends Hashtable {  
    /* Constructor omitted */  
    MySet() {  
    }  
    void put(Object element) {  
        if (!containsKey(element)){  
            put(element, this);  
        }  
    }  
    boolean containsValue(Object element){  
        return containsKey(element);  
    }  
    /* Other methods omitted */  
}
```

Specification Inheritance and Implementation

Inheritance_4

Object design model after transformation



```
/* Implementation of MySet using delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object element) {
        return
            (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

Inheritance metamodel

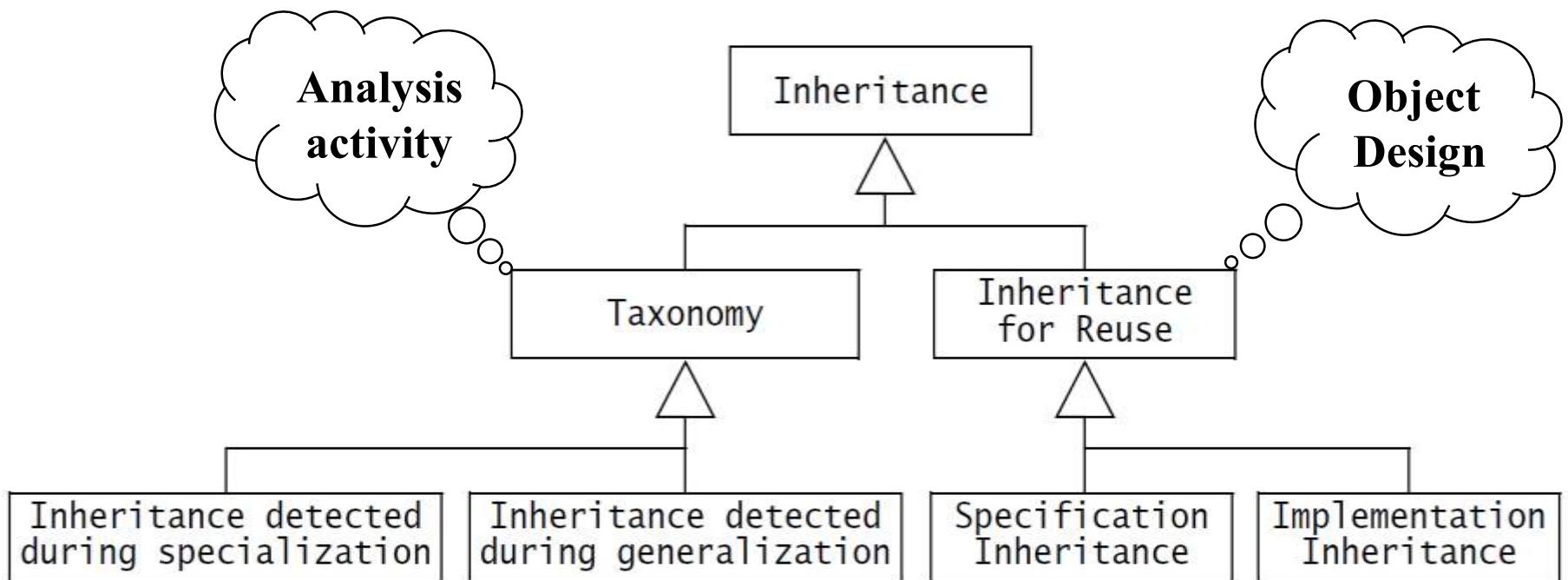


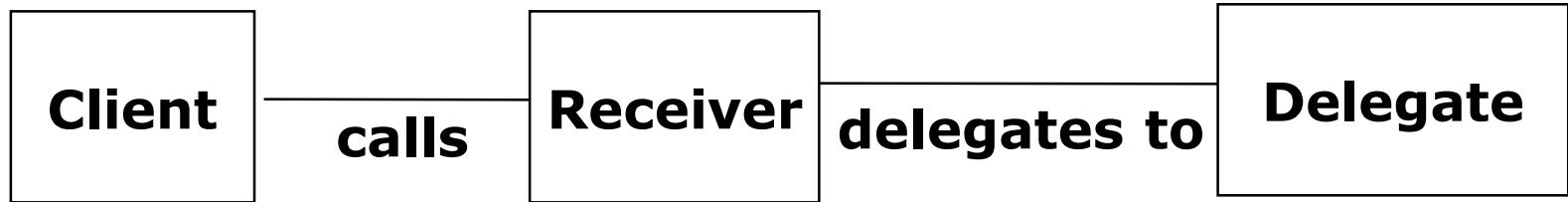
Figure 8-4 Inheritance meta-model (UML class diagram).

Delegation

- **Delegation is the alternative to implementation inheritance** that should be used when reuse is desired.
- A **class is said to delegate** to another class if it implements an operation by referring another operation of the other class.
- **Delegation** makes explicit the dependencies between the reused class and the new class. The right column of Figure 8-3 shows an implementation of MySet using delegation instead of implementation inheritance. The only significant change is the private field table and its initialization in the MySet() constructor.

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation two objects are involved in handling a request from a Client
 - The Receiver object delegates operations to the Delegate object
 - The Receiver object makes sure, that the Client does not misuse the Delegate object.



Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
 - Flexibility: Any object can be replaced at run time by another one
 - Inefficiency: Objects are encapsulated
- Inheritance
 - Straightforward to use
 - Supported by many programming languages
 - Easy to implement new functionality
 - Exposes a subclass to details of its super class
 - Change in the parent class requires recompilation of the subclass.

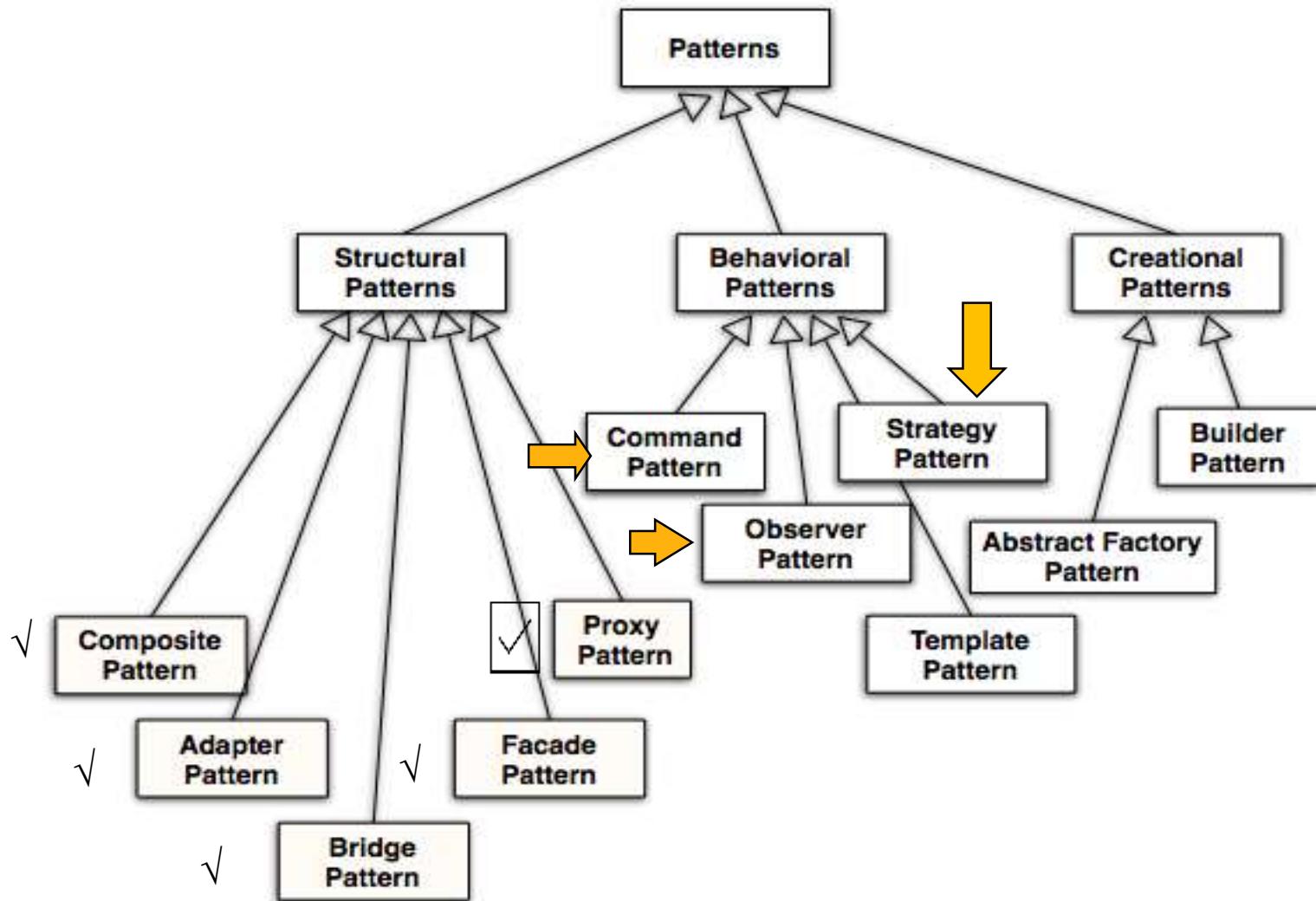
The Liskov Substitution Principle

- If **an object of type S can be substituted in all the places** where an object of type T is expected, then S is a subtype of T.
- An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.

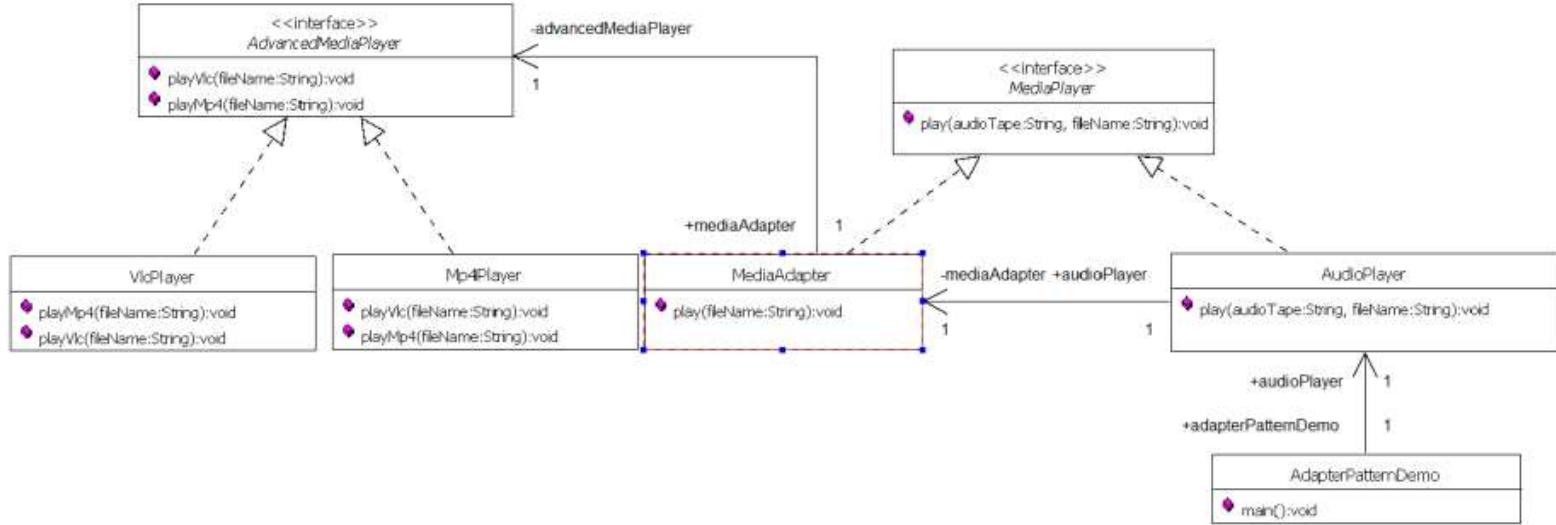
Delegation and Inheritance in Design Patterns

- In object-oriented development, **design patterns** are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]. A design pattern has four elements:
 1. A **name** that uniquely identifies the pattern from other patterns.
 2. A **problem description** that describes the situations in which the pattern can be used. Problems addressed by design patterns are usually the realization of modifiability and extensibility design goals and nonfunctional requirements.
 3. A **solution** stated as a set of collaborating classes and interfaces.
 4. A set of **consequences** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

A Taxonomy of Design Patterns

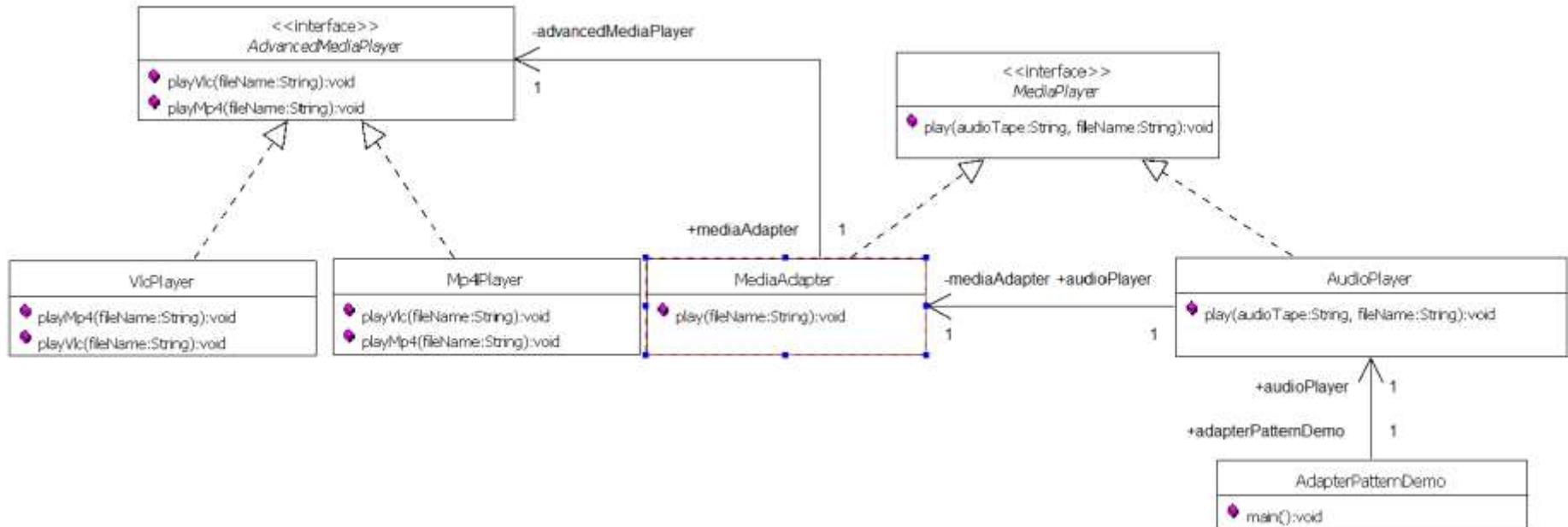


The Adapter Pattern



- **Adapter pattern** works as a **bridge between two incompatible** interfaces. This design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class **MediaAdapter** which is responsible to join functionalities of independent or incompatible interfaces.

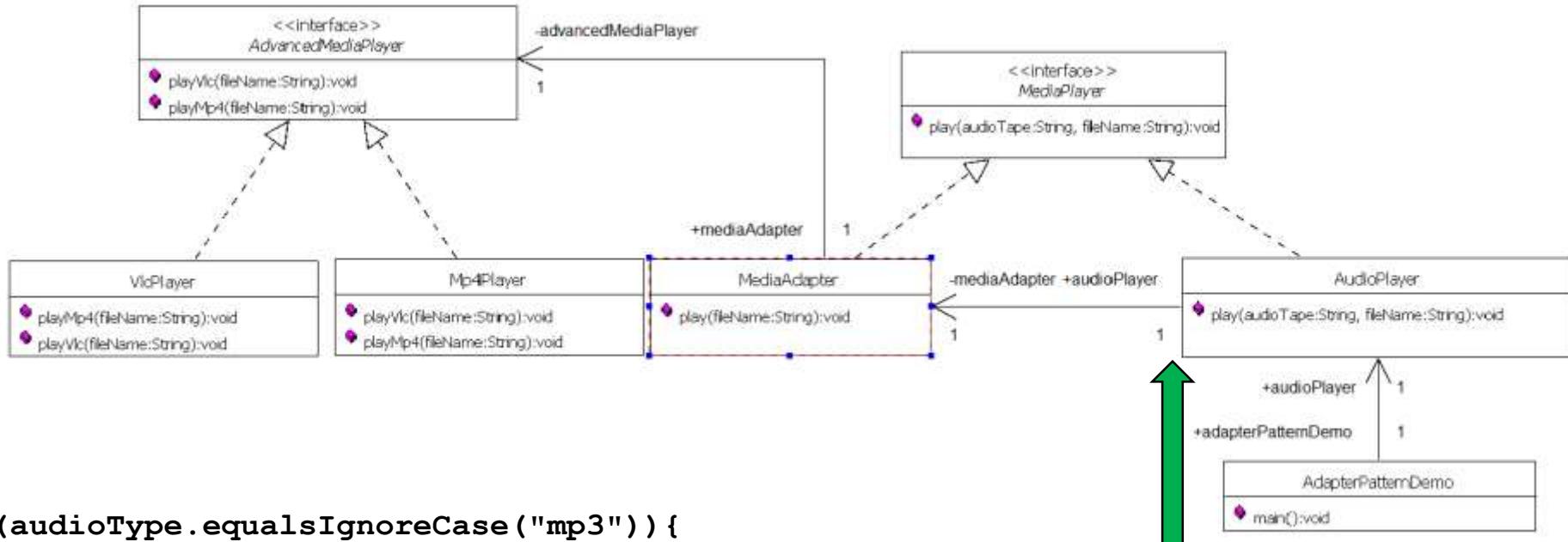
The Adapter Pattern_2



```
public static void main(String[] args) {
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
}
```

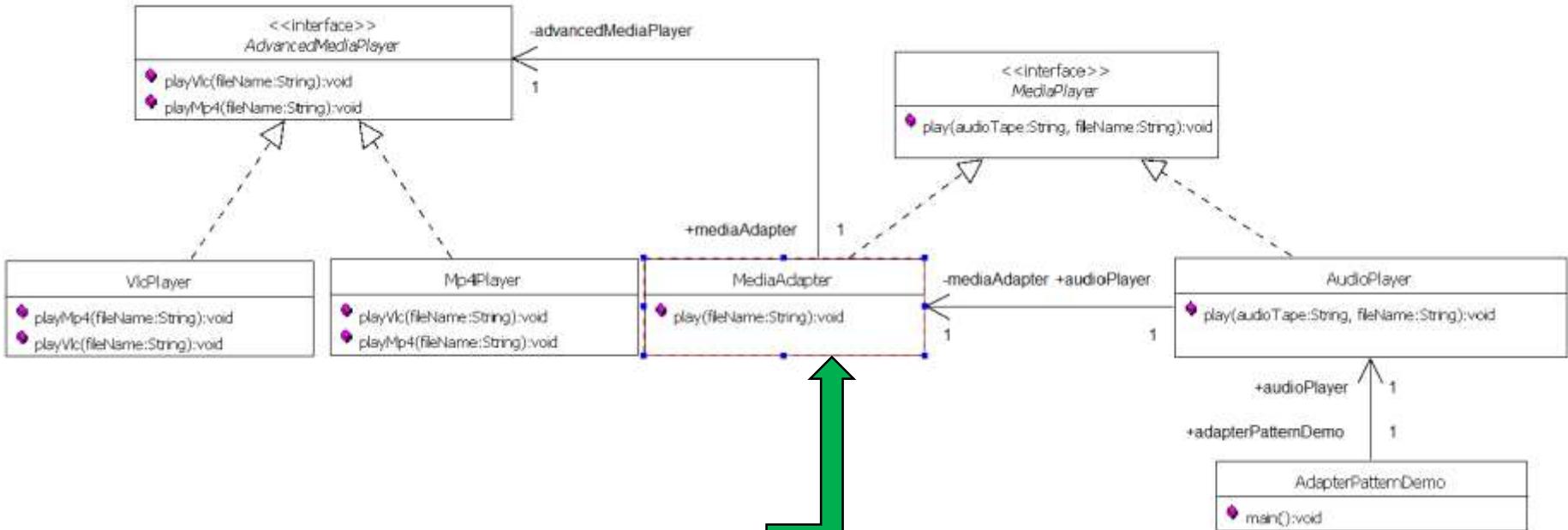
The Adapter Pattern_3



```

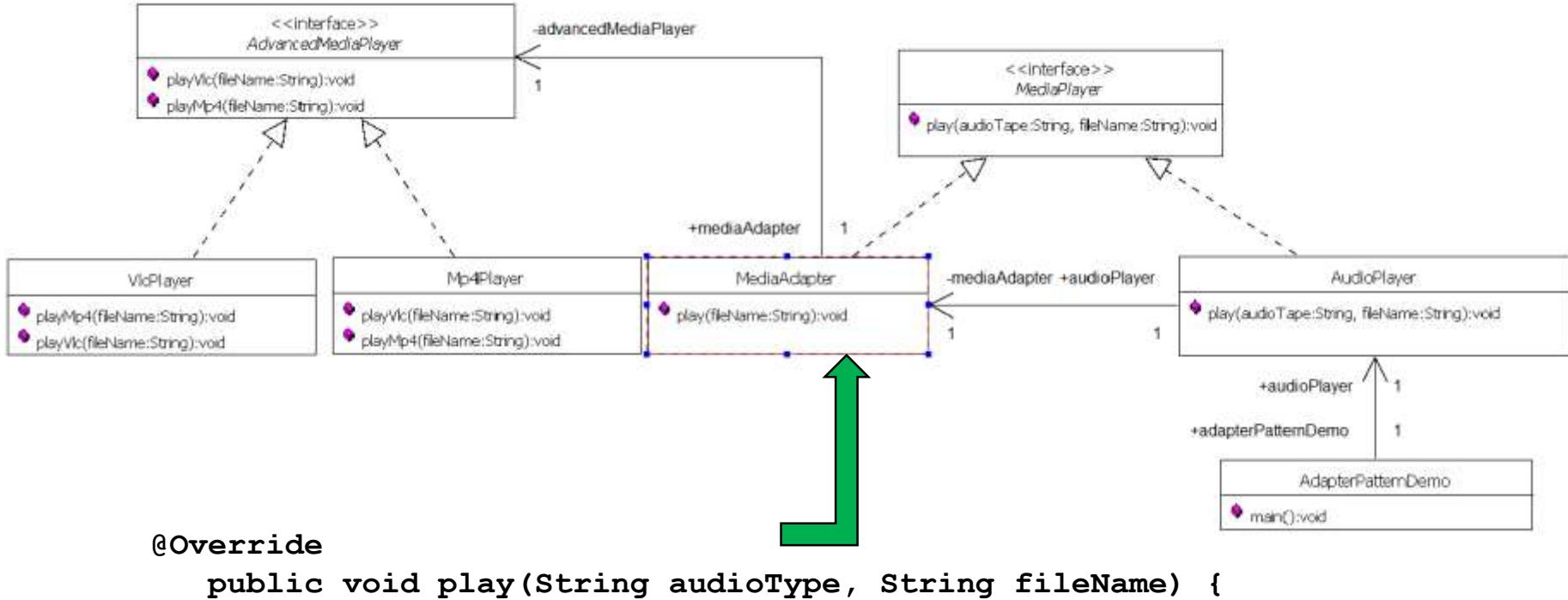
if(audioType.equalsIgnoreCase("mp3")){
    System.out.println("Playing mp3 file. Name: " + fileName);
}
else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
    mediaAdapter = new MediaAdapter(audioType);
    mediaAdapter.play(audioType, fileName);
}
else{
    System.out.println("Invalid media. " + audioType + " format not supported");
}
  
```

The Adapter Pattern_3



```
public MediaAdapter(String audioType){  
  
    if(audioType.equalsIgnoreCase("vlc")) {  
        advancedMusicPlayer = new VlcPlayer();  
  
    } else if (audioType.equalsIgnoreCase("mp4")) {  
        advancedMusicPlayer = new Mp4Player();  
    }  
}
```

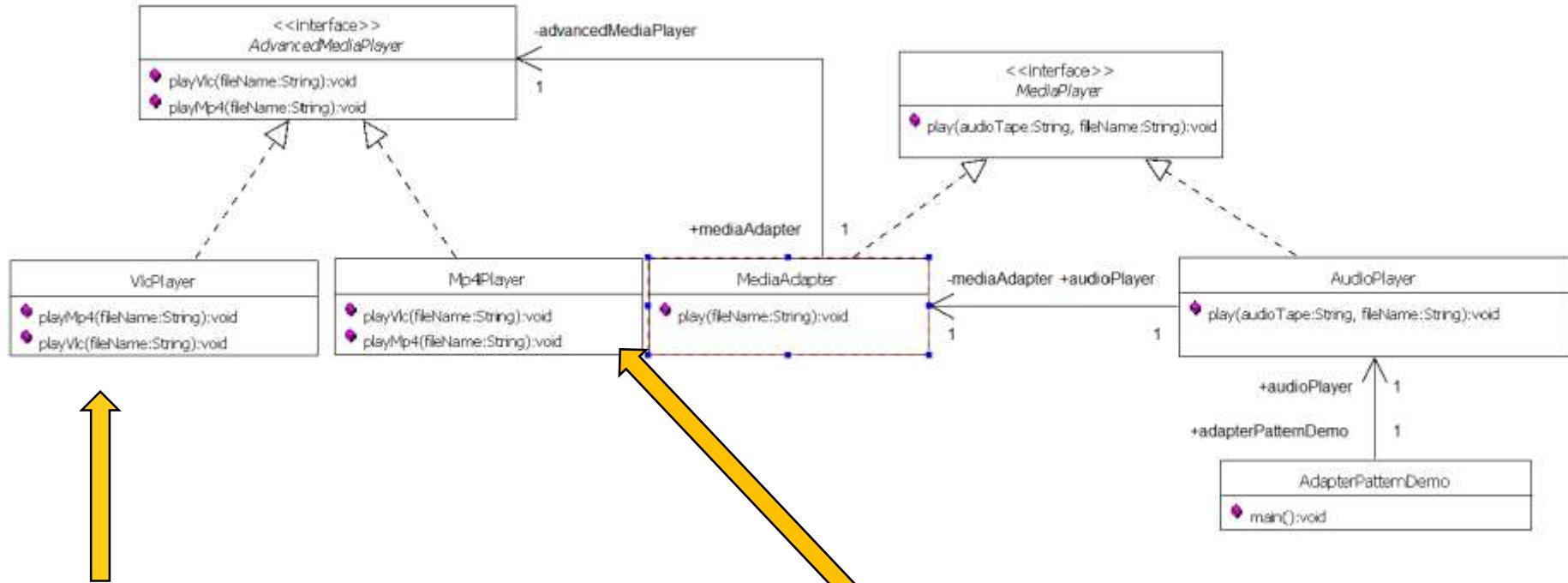
The Adapter Pattern_4



```
@Override
public void play(String audioType, String fileName) {

    if(audioType.equalsIgnoreCase("vlc")){
        advancedMusicPlayer.playVlc(fileName);
    }
    else if(audioType.equalsIgnoreCase("mp4")){
        advancedMusicPlayer.playMp4(fileName);
    }
}
```

The Adapter Pattern_5



@Override

```

public void playMp4(String fileName) {
}
@Override
public void playVlc(String fileName) {
    System.out.println("Playing vlc file. Name: "+ fileName);
}
  
```

@Override

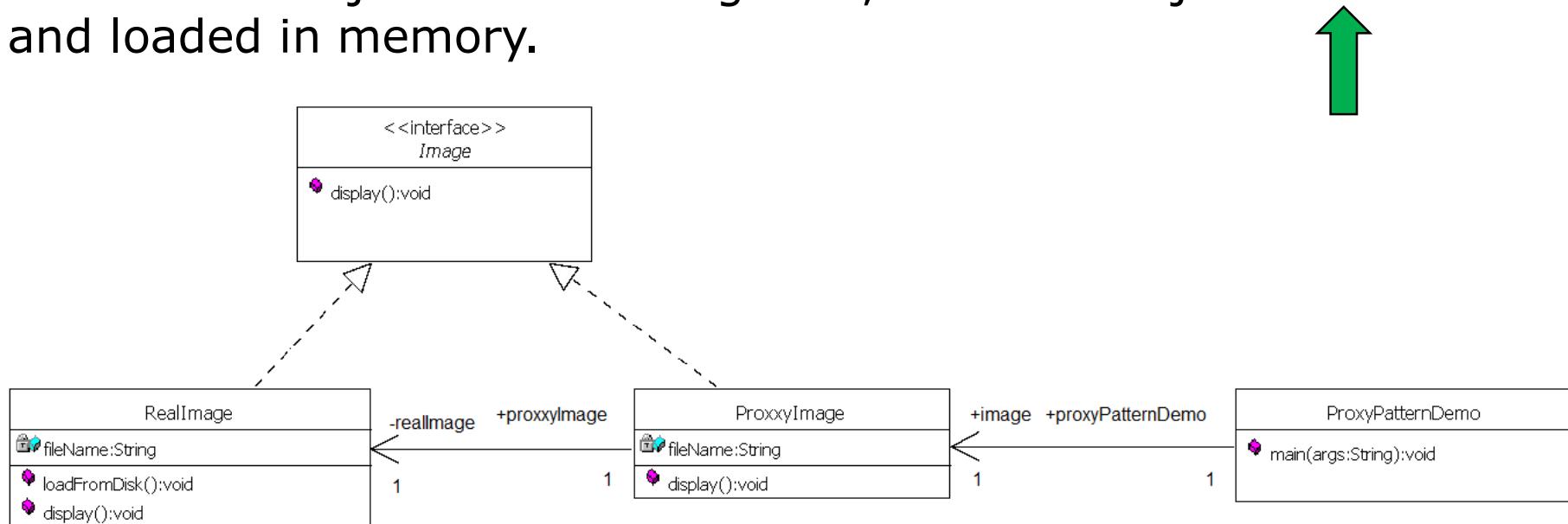
```

public void playVlc(String fileName) {
}
@Override
public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
}
  
```

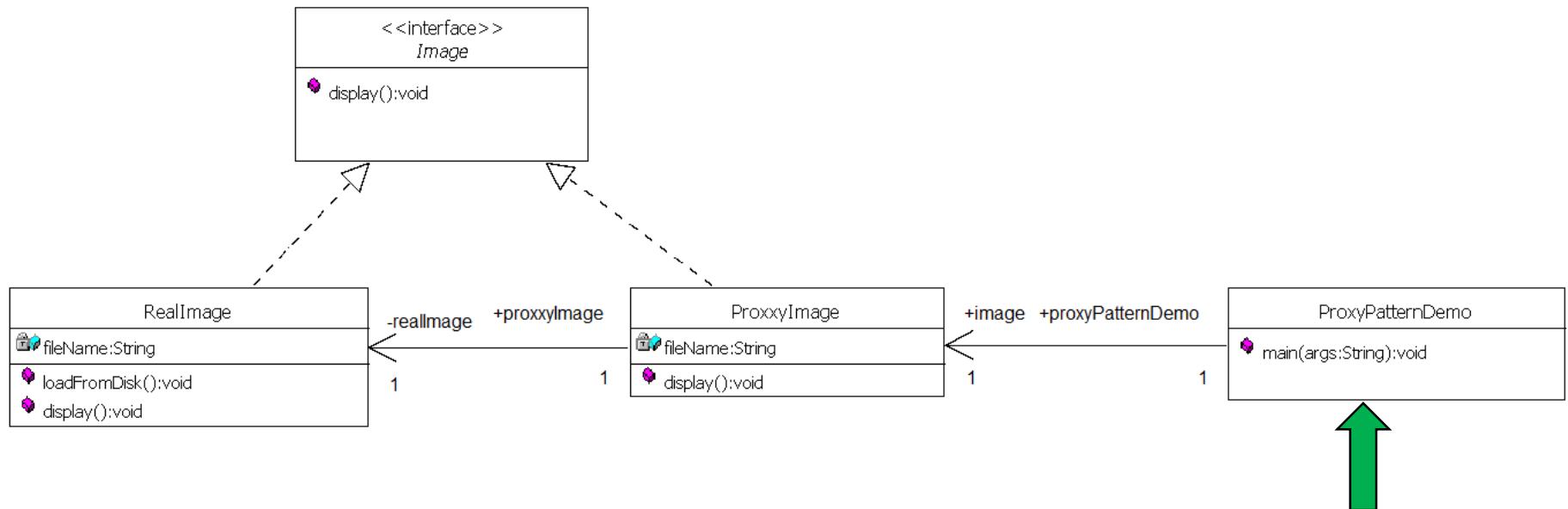
The Proxy Pattern

The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface. The ProxyObject stores a subset of the attributes of the RealObject.

The ProxyObject handles certain requests completely (e.g., determining the size of an image), whereas others are delegated to the RealObject. After delegation, the RealObject is created and loaded in memory.

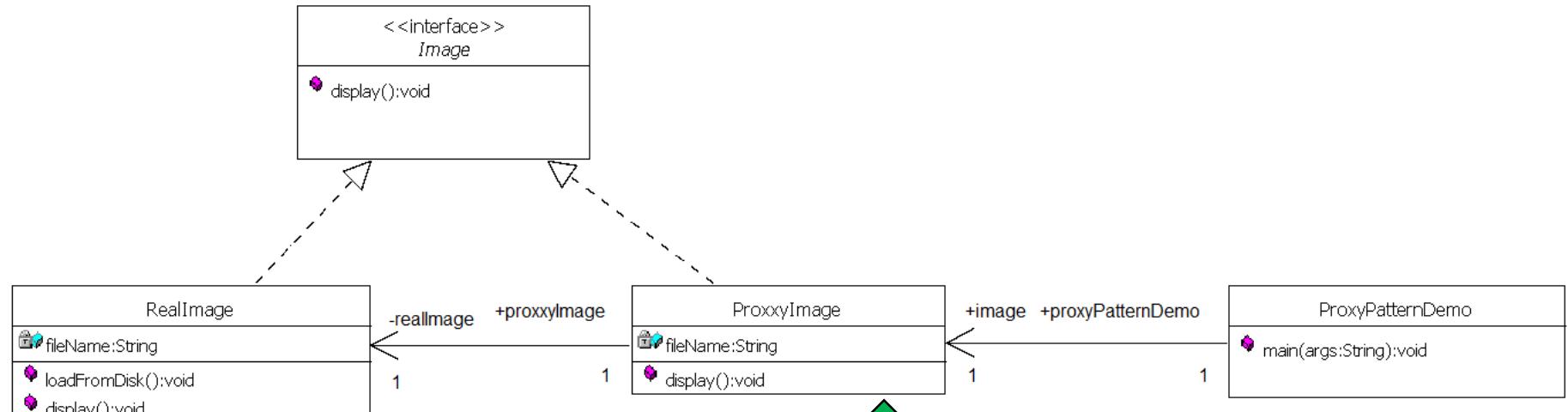


The Proxy Pattern_2



```
public static void main(String[] args) {  
    Image image = new  
    ProxyImage("test_10mb.jpg");  
  
    //image will be loaded from disk  
    image.display();  
    System.out.println("");  
  
    //image will not be loaded from disk  
    image.display();  
}
```

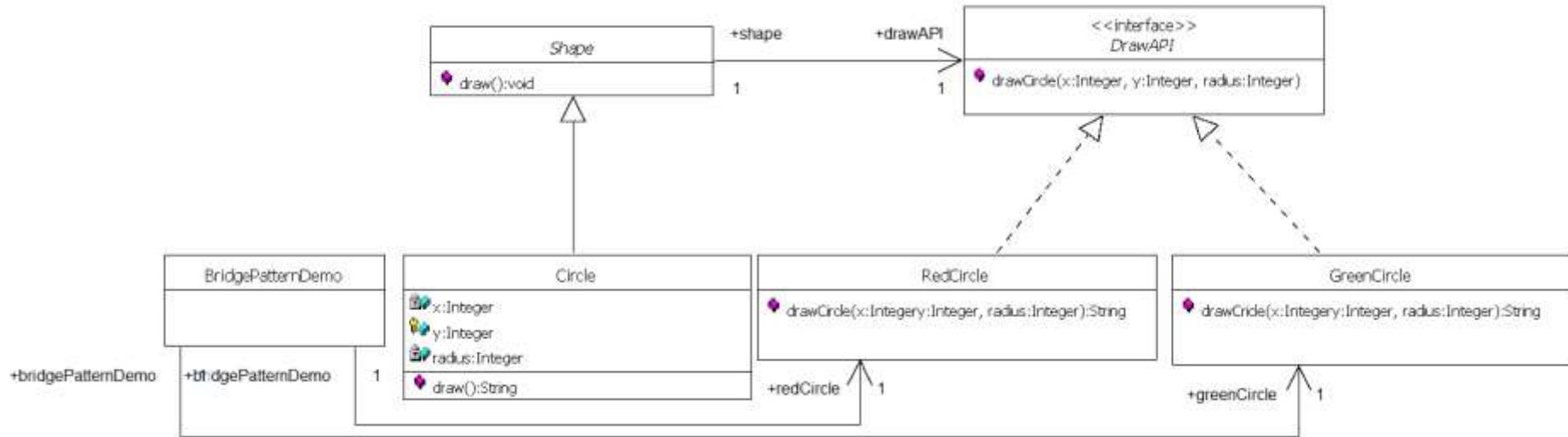
The Proxy Pattern_3



```
public RealImage(String fileName) {
    this.fileName = fileName;
    loadFromDisk(fileName);
}
@Override
public void display() {
    System.out.println("Displaying " +
fileName);
}
private void loadFromDisk(String
fileName) {
    System.out.println("Loading " +
fileName);
}
```

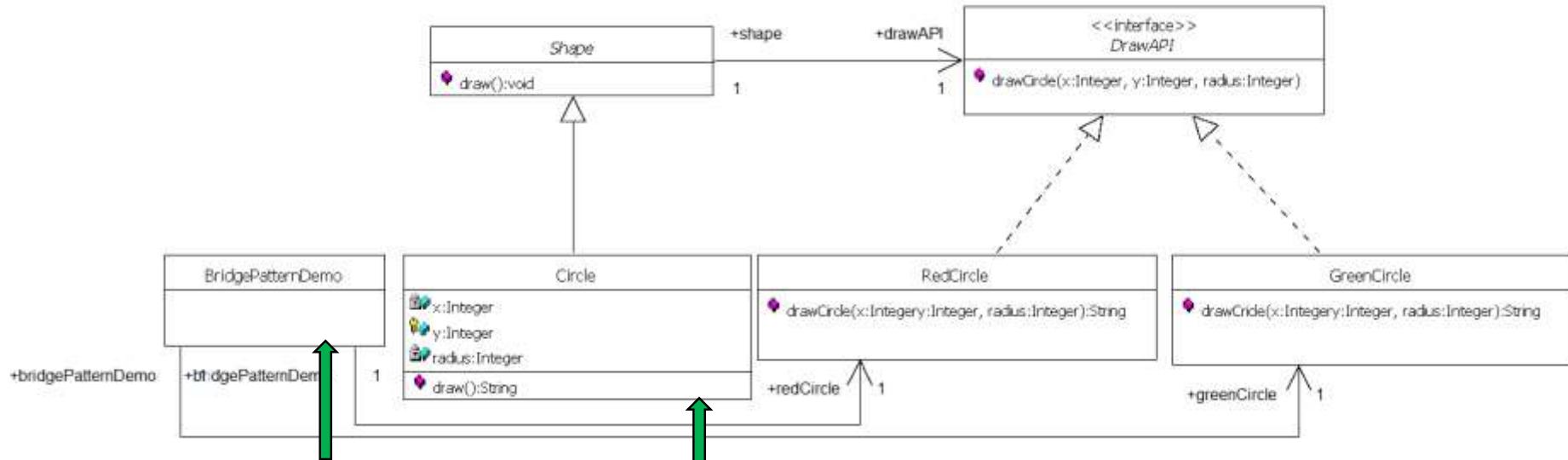
```
public ProxyImage(String fileName) {
    this.fileName = fileName;
}
@Override
public void display() {
    if(realImage == null){
        realImage = new RealImage(fileName);
    }
    realImage.display();
}
```

The Bridge Pattern



- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. The *DrawAPI* interface acts as a bridge implementer.
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

The Bridge Pattern



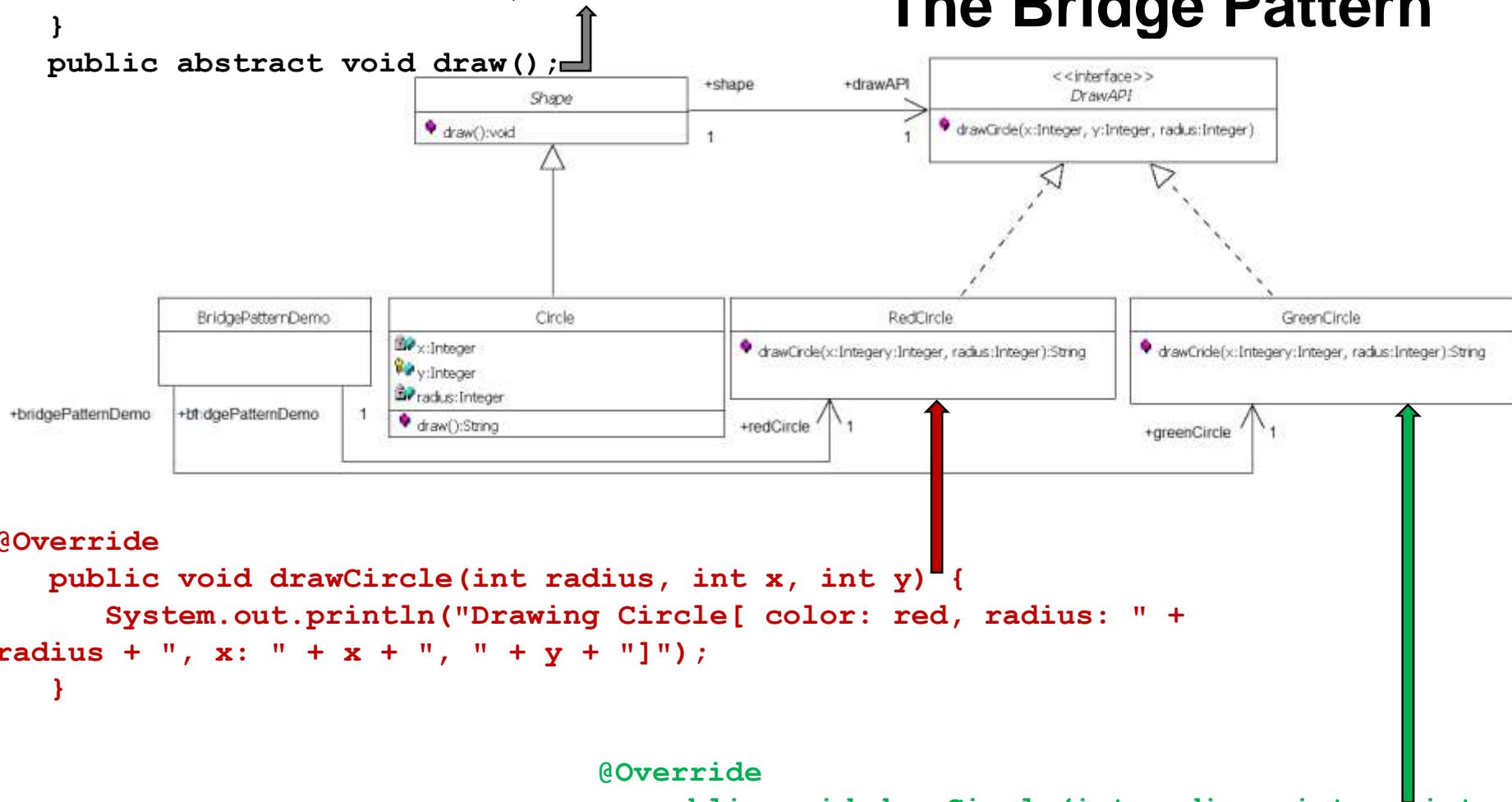
```
public static void main(String[] args) {
    Shape redCircle = new Circle(100,100, 10, new RedCircle());
    Shape greenCircle = new Circle(100,100, 10, new
GreenCircle());
    redCircle.draw();    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
    greenCircle.draw();        super(drawAPI);
}                                this.x = x;
                                this.y = y;
                                this.radius = radius;
}
public void draw() {
    drawAPI.drawCircle(radius,x,y);
}
}
```

```

protected Shape (DrawAPI drawAPI) {
    this.drawAPI = drawAPI;
}
public abstract void draw();

```

The Bridge Pattern



The Composite Pattern

- Modern toolkits enable developers to organize the user interface objects into hierarchies of aggregate nodes, called “panels,” that can be manipulated the same way as the concrete user interface objects.
- For example, our preferences dialog can include a top panel for the title of the dialog and instructions for the user, a center panel containing the checkboxes and their labels, and a bottom panel for the ‘ok’ and ‘cancel’ button.
- Each panel is responsible for the layout of its subpanels, called “children,” and the overall dialog only must deal with the three panels (Figures 8-14 and 8-15).

The Composite Pattern_2

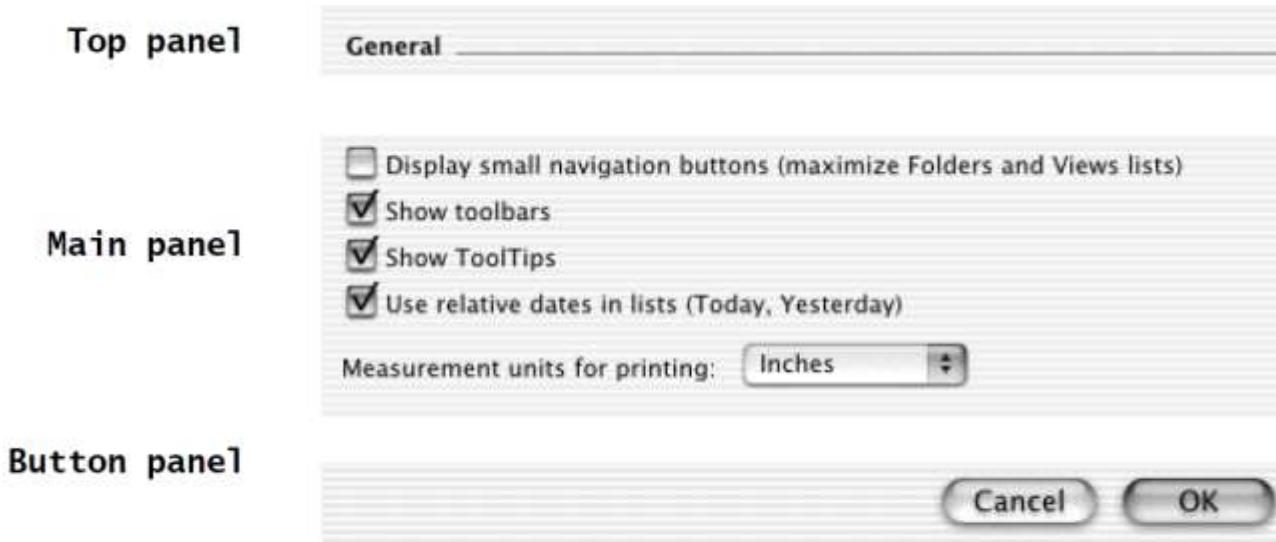


Figure 8-14 Anatomy of a preference dialog. Aggregates, called “panels,” are used for grouping user interface objects that need to be resized and moved together.

The Composite Pattern_3

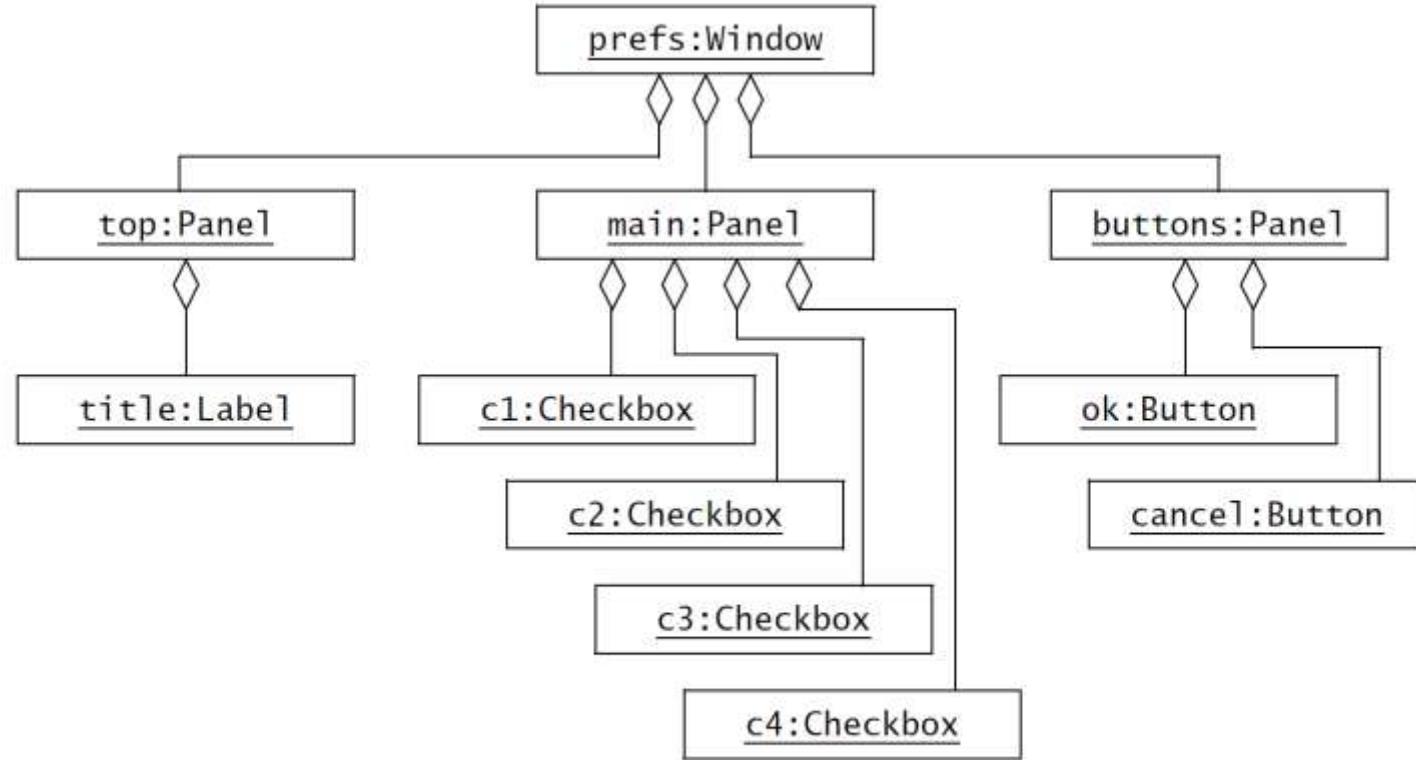


Figure 8-15 UML object diagram for the user interface objects of Figure 8-14.

The Composite Pattern_4

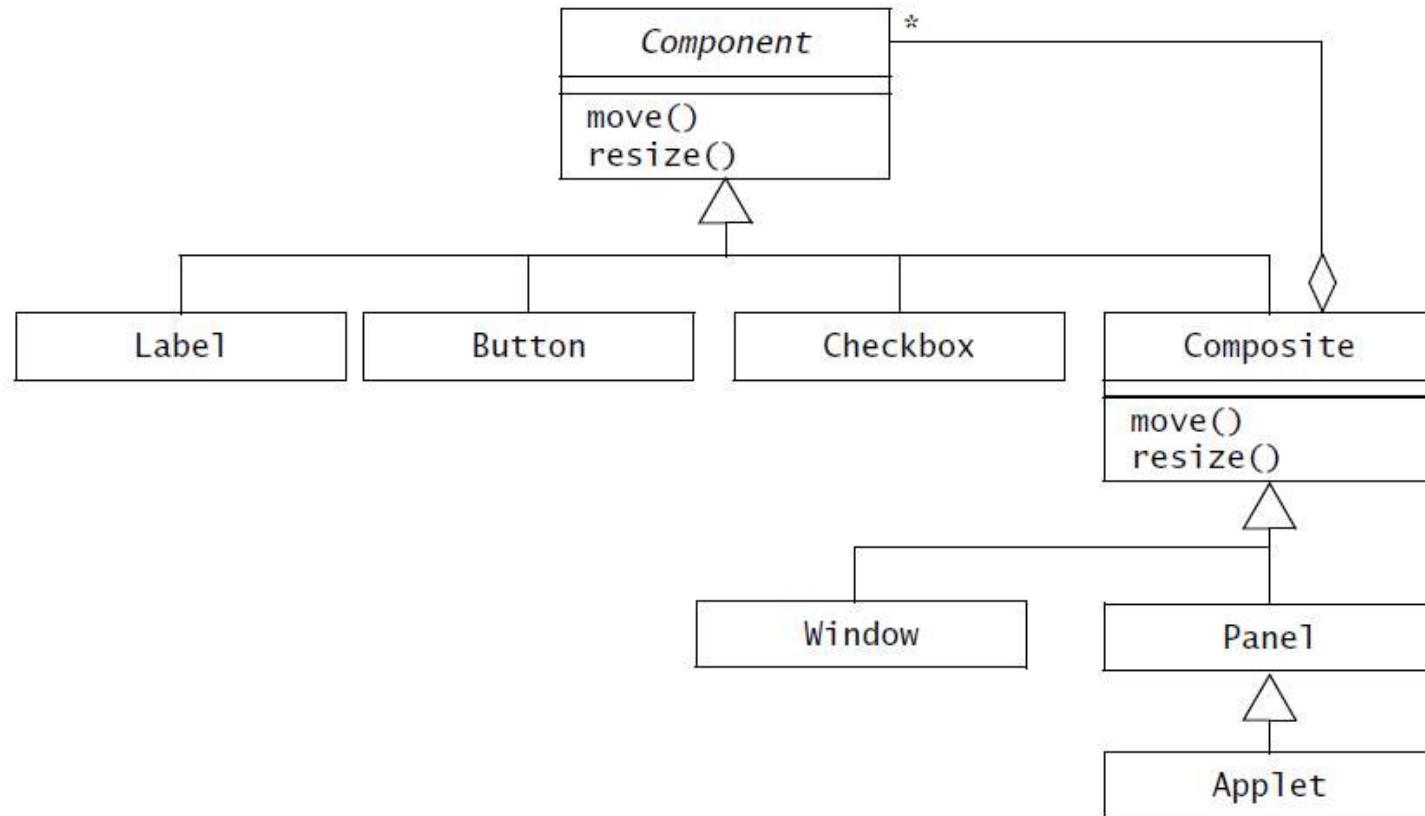
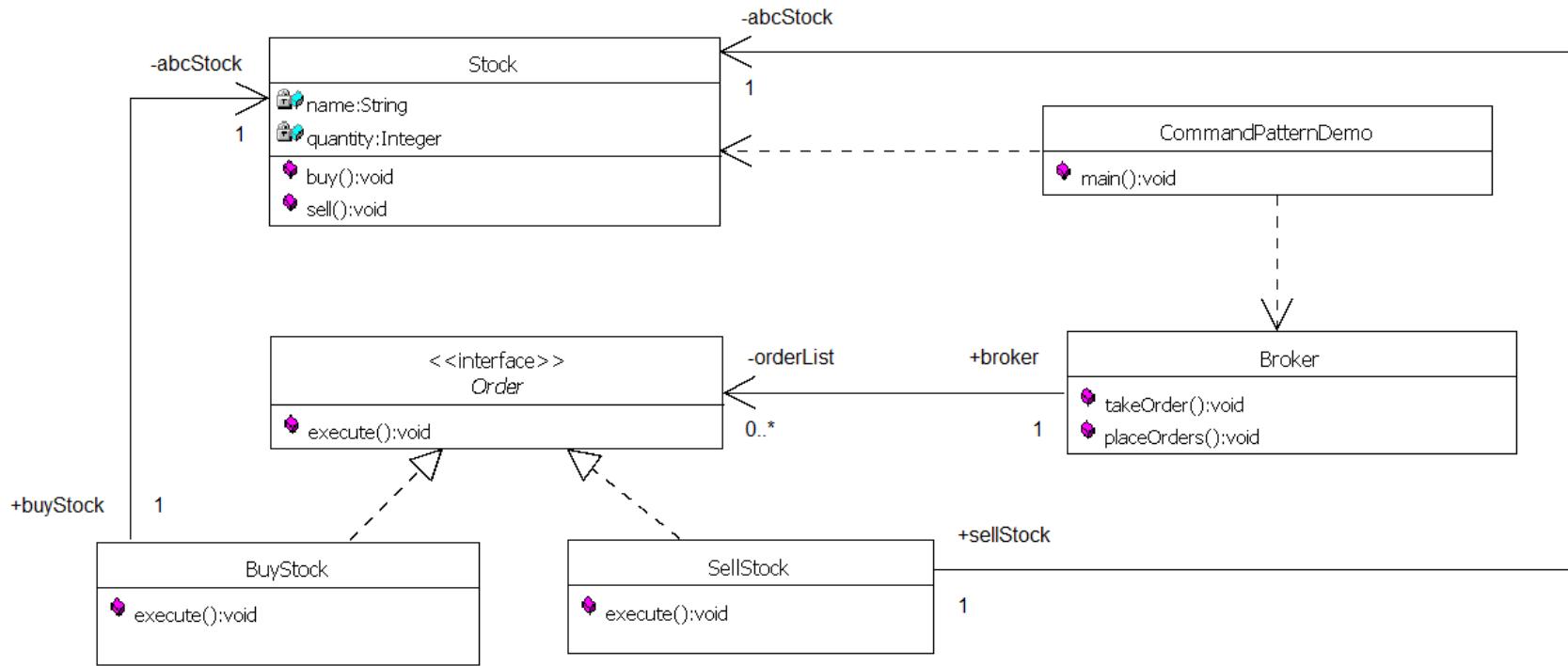


Figure 8-16 Applying the Composite design pattern to user interface widgets (UML class diagram). The Swing Component hierarchy is a Composite in which leaf widgets (e.g., Checkbox, Button, Label) specialize the Component interface, and aggregates (e.g., Panel, Window) specialize the Composite abstract class. Moving or resizing a Composite impacts all of its children.

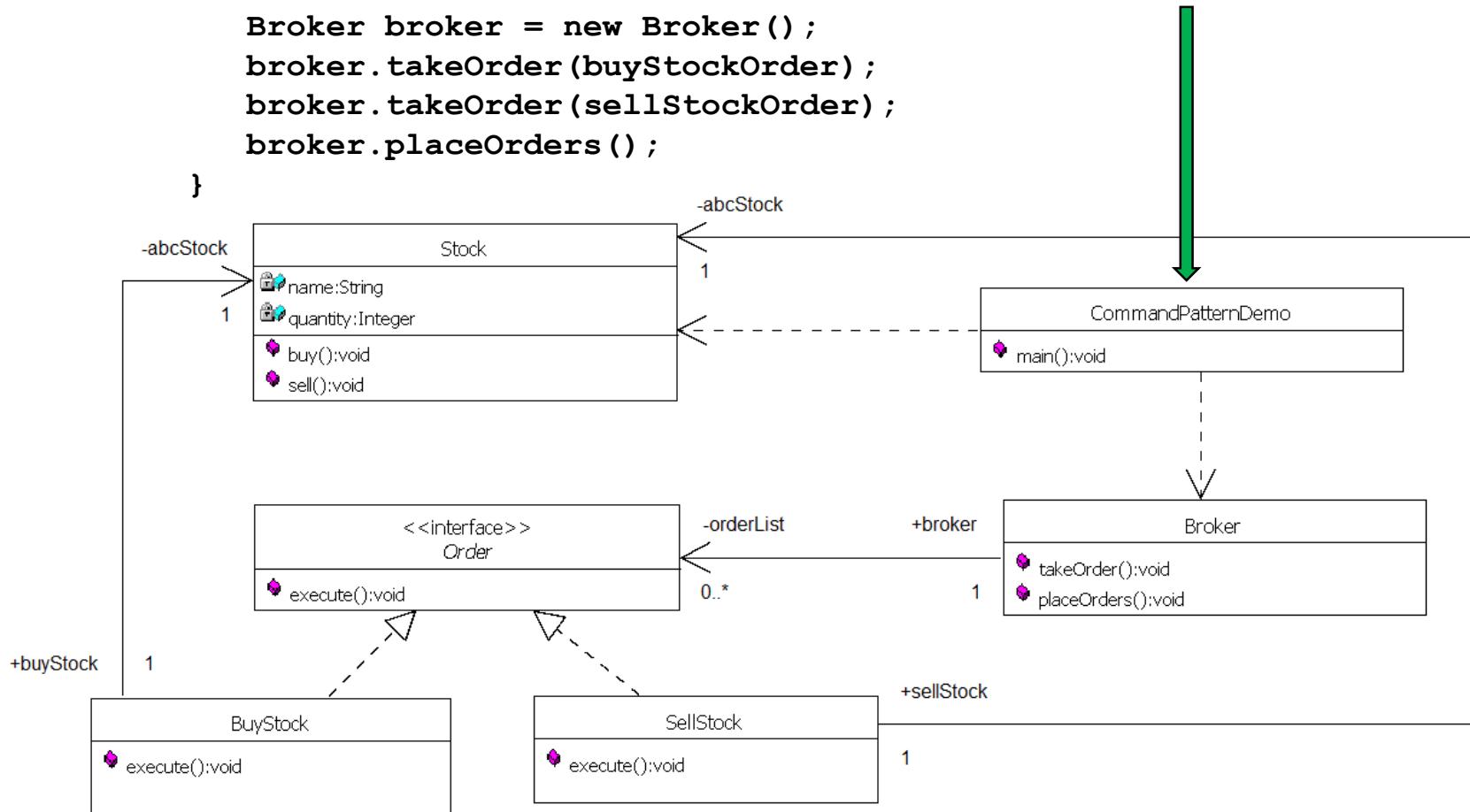
The Command Pattern



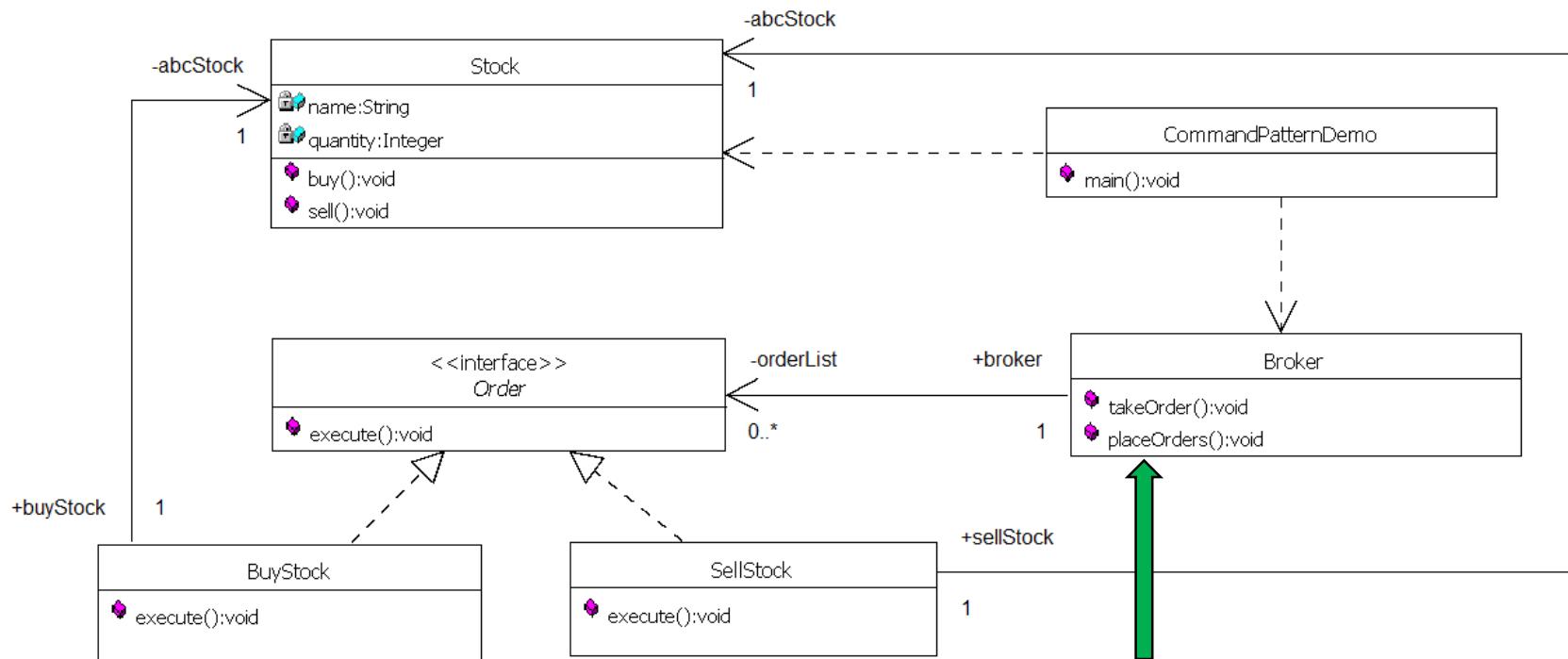
A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

The Command Pattern_2

```
public static void main(String[] args) {  
    Stock abcStock = new Stock();  
    BuyStock buyStockOrder = new BuyStock(abcStock);  
    SellStock sellStockOrder = new SellStock(abcStock);  
    Broker broker = new Broker();  
    broker.takeOrder(buyStockOrder);  
    broker.takeOrder(sellStockOrder);  
    broker.placeOrders();  
}
```



The Command Pattern_3



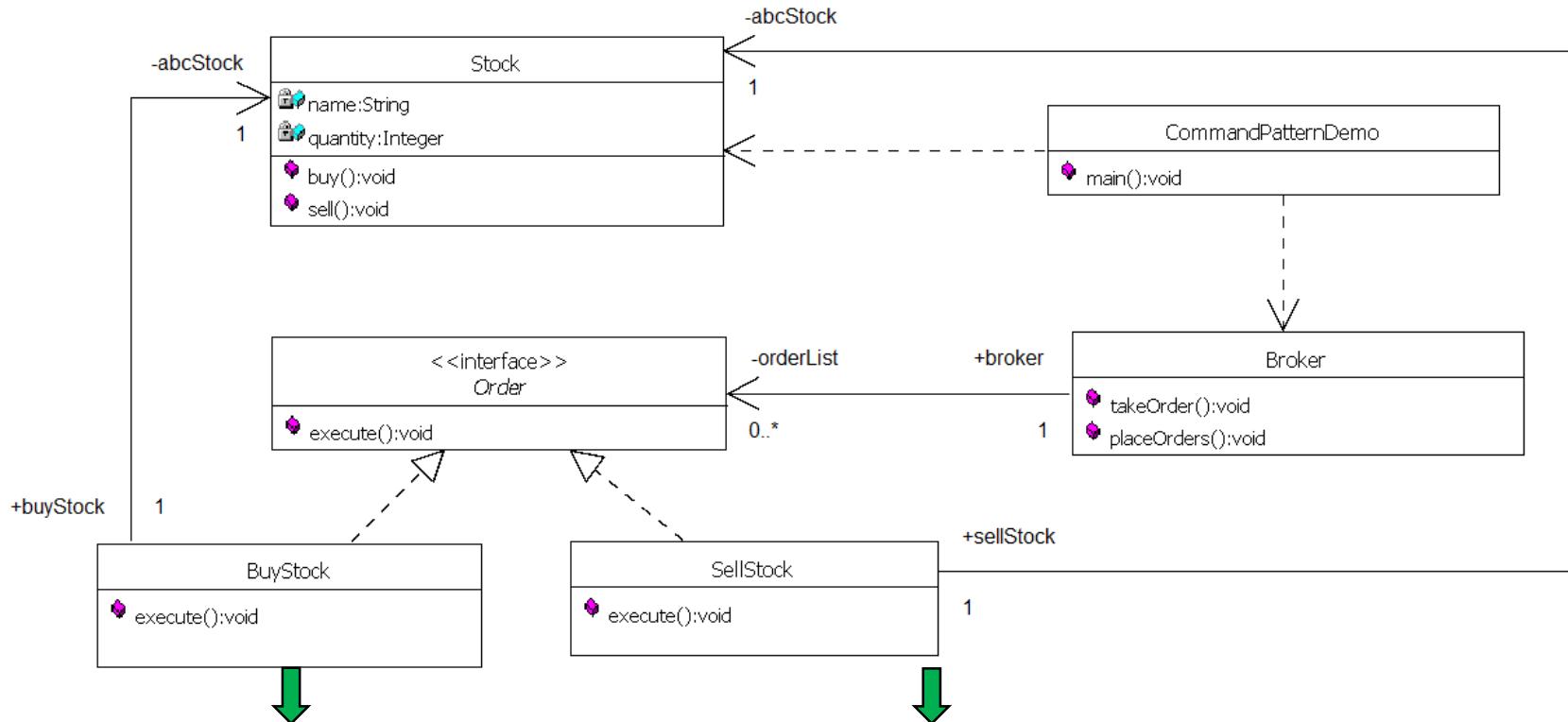
```

private List<Order> orderList = new ArrayList<Order>();

public void takeOrder(Order order) {
    orderList.add(order);
}

public void placeOrders() {
    for (Order order : orderList) {
        order.execute();
    }
    orderList.clear();
}
  
```

The Command Pattern_4



```

public BuyStock(Stock abcStock) {
    this.abcStock = abcStock;
}
  
```

```

public void execute() {
    abcStock.buy();
}
  
```

```

public SellStock(Stock abcStock) {
    this.abcStock = abcStock;
}
  
```

```

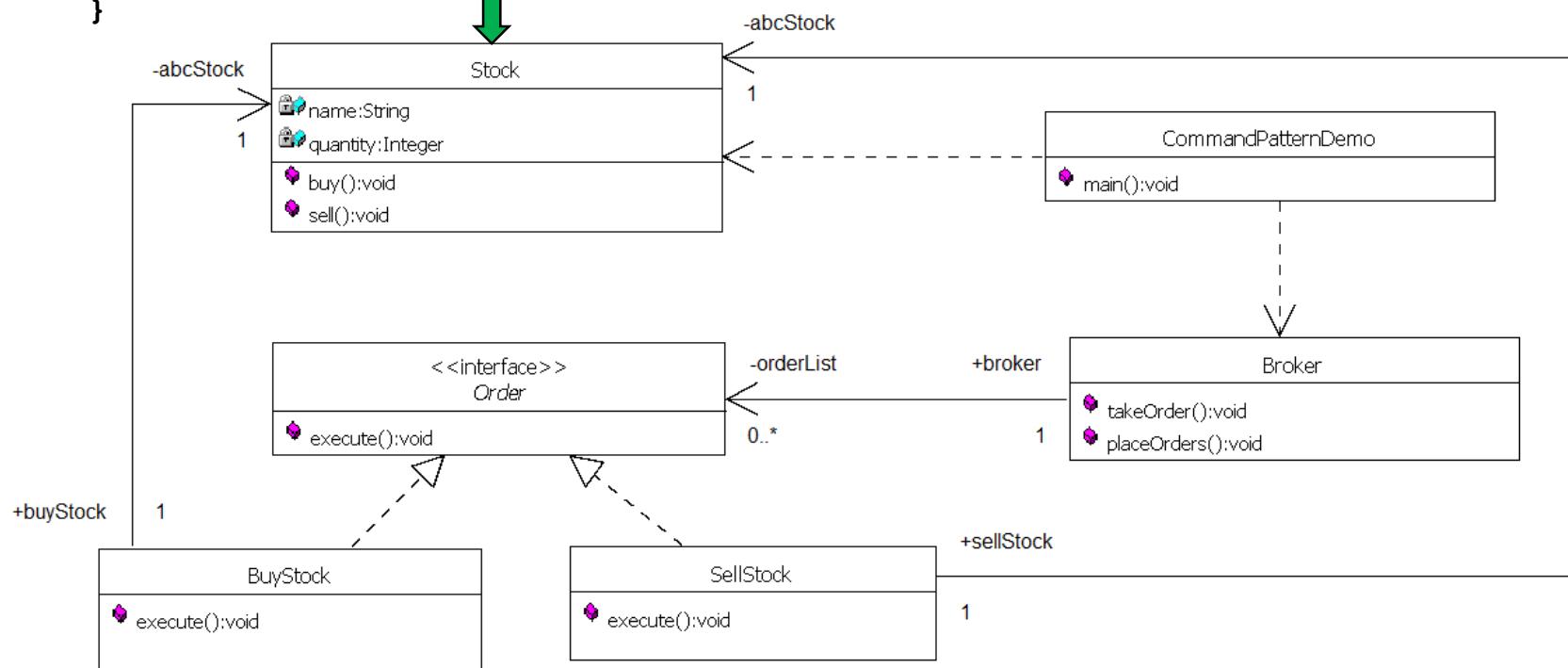
public void execute() {
    abcStock.sell();
}
  
```

The Command Pattern_5

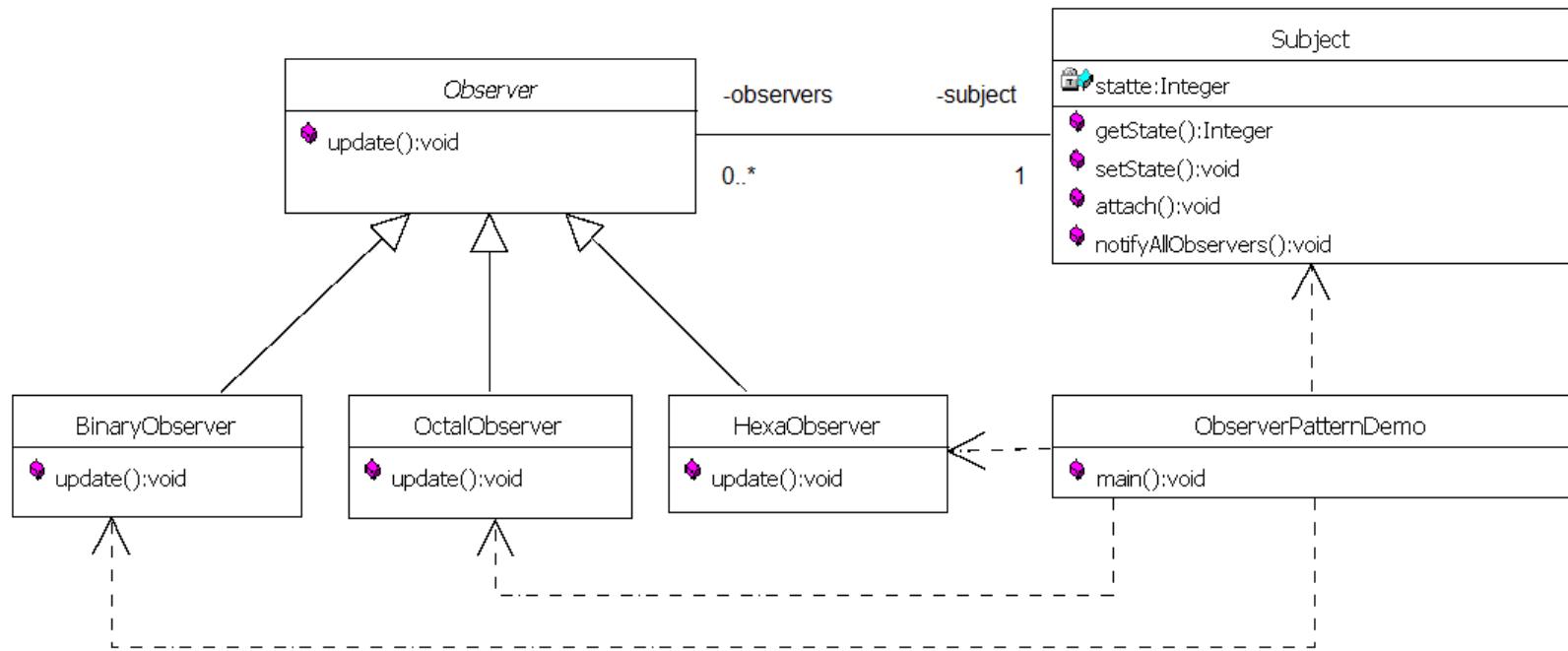
```

private String name = "ABC";
private int quantity = 10;
public void buy(){
    System.out.println("Stock [ Name: "+name+", "
        Quantity: " + quantity +" ] bought");
}
public void sell(){
    System.out.println("Stock [ Name: "+name+", "
        Quantity: " + quantity +" ] sold");
}

```

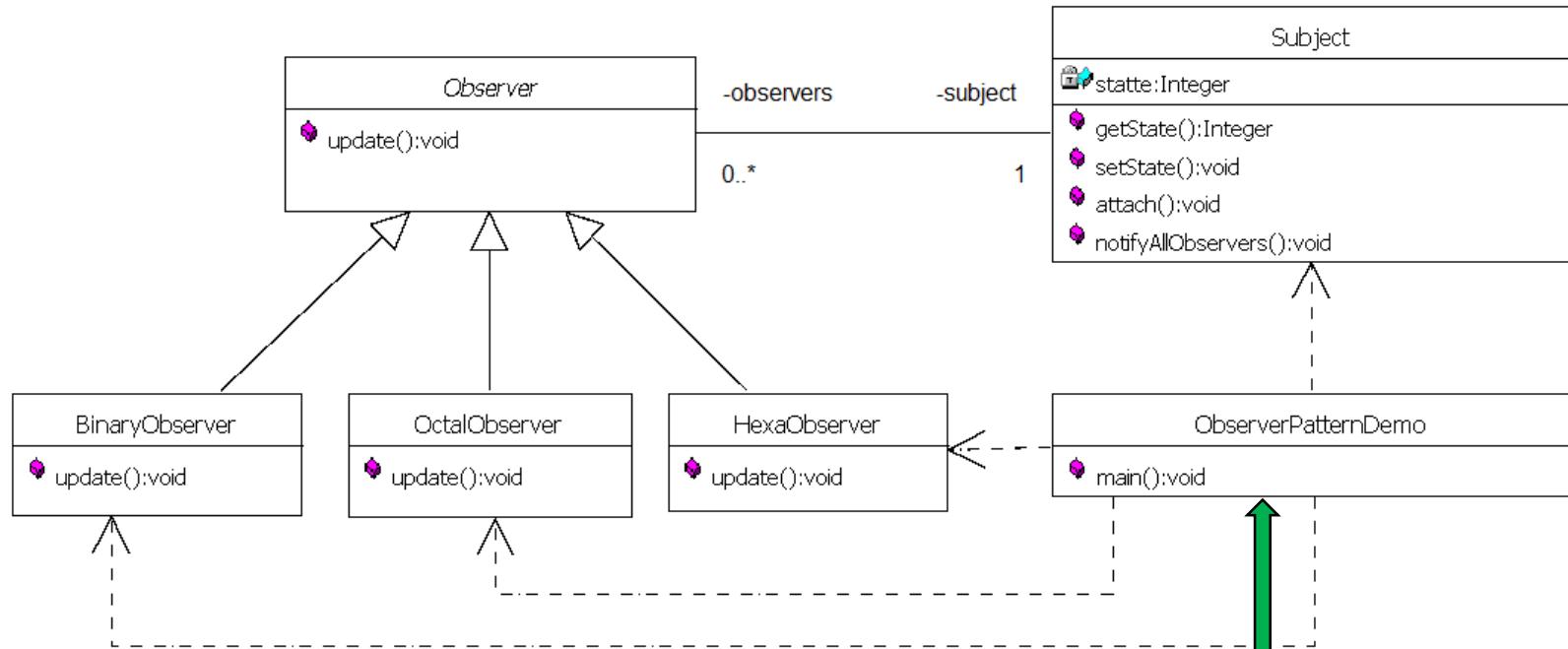


The Observer Pattern



Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

The Observer Pattern_2



```
public static void main(String[] args) {
    Subject subject = new Subject();
    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);
    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
}
```

The Observer Pattern_3

```

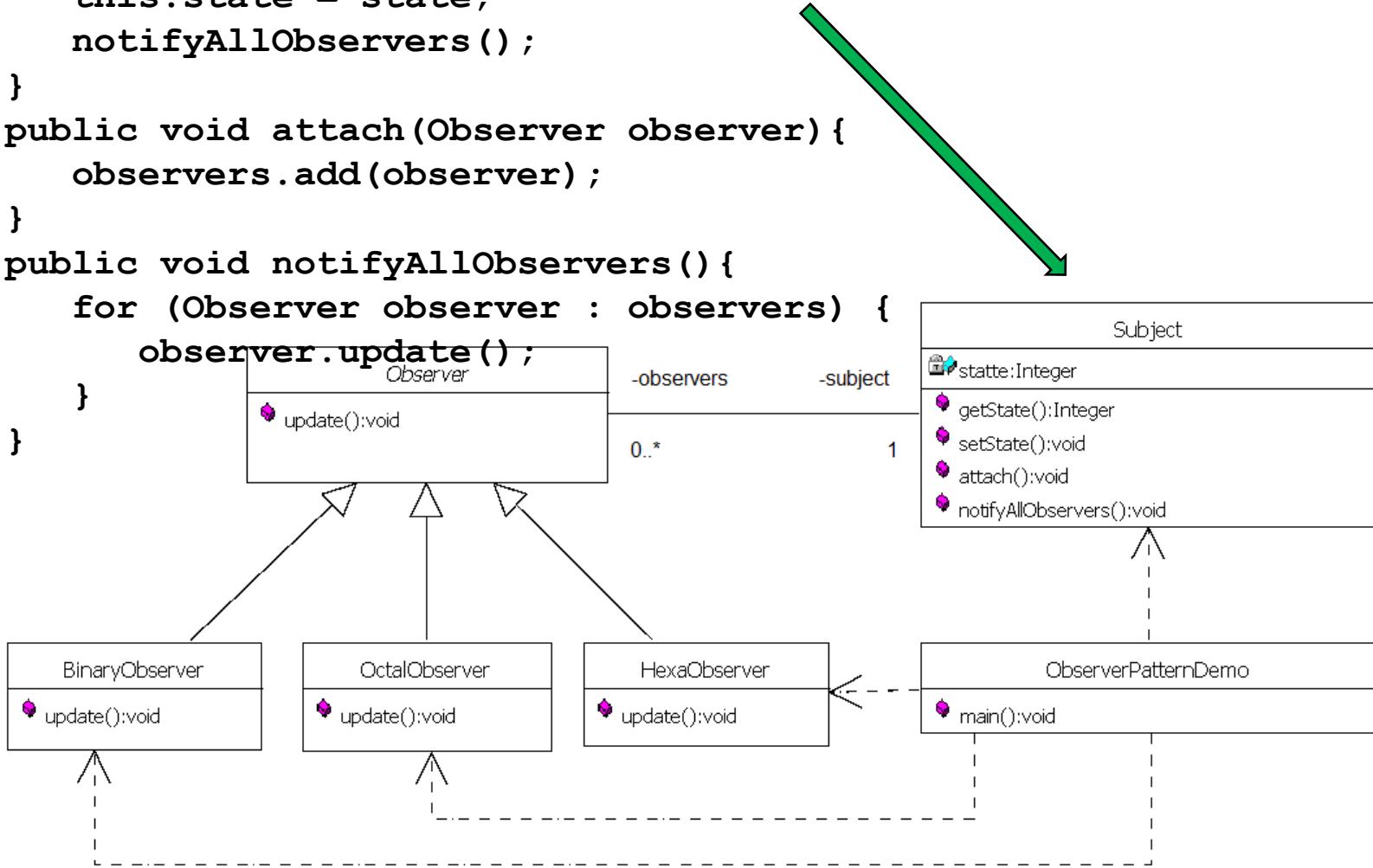
public int getState() {
    return state;
}

public void setState(int state) {
    this.state = state;
    notifyAllObservers();
}

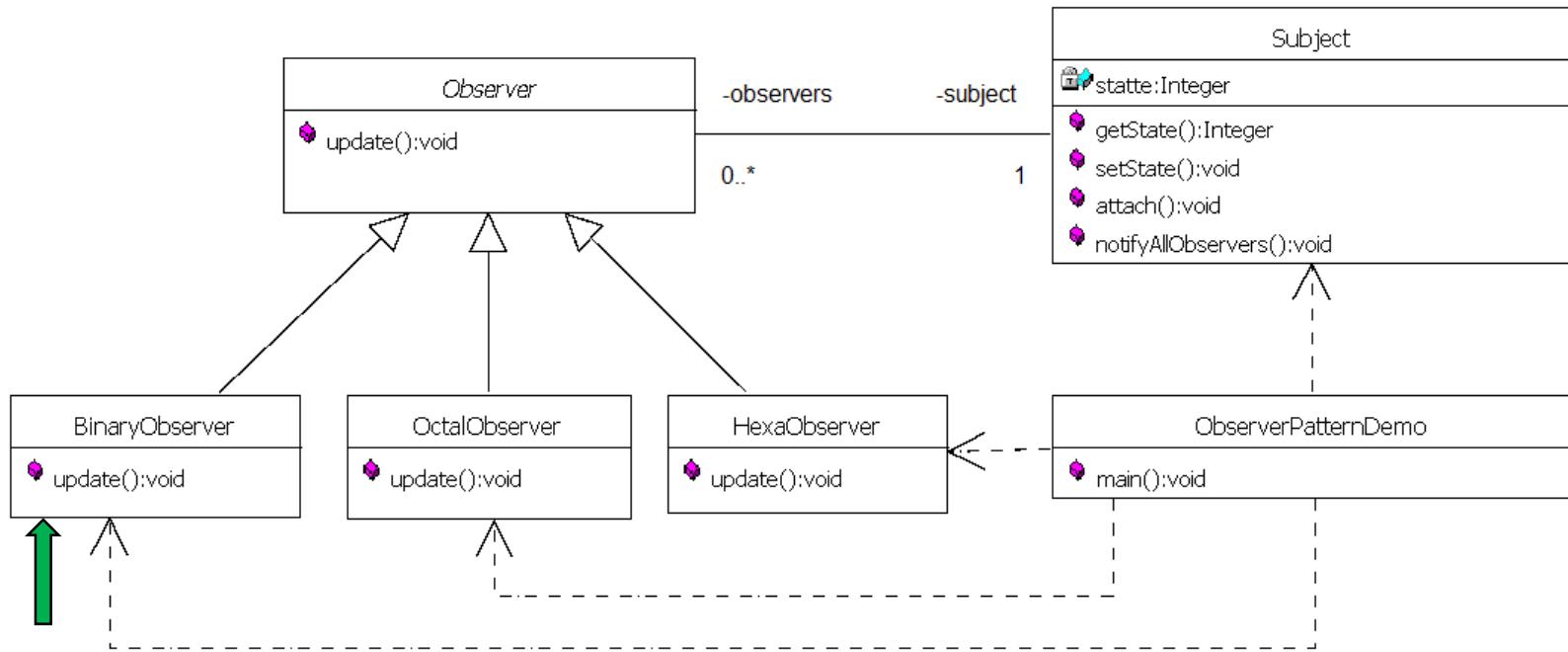
public void attach(Observer observer) {
    observers.add(observer);
}

public void notifyAllObservers() {
    for (Observer observer : observers) {
        observer.update();
    }
}

```

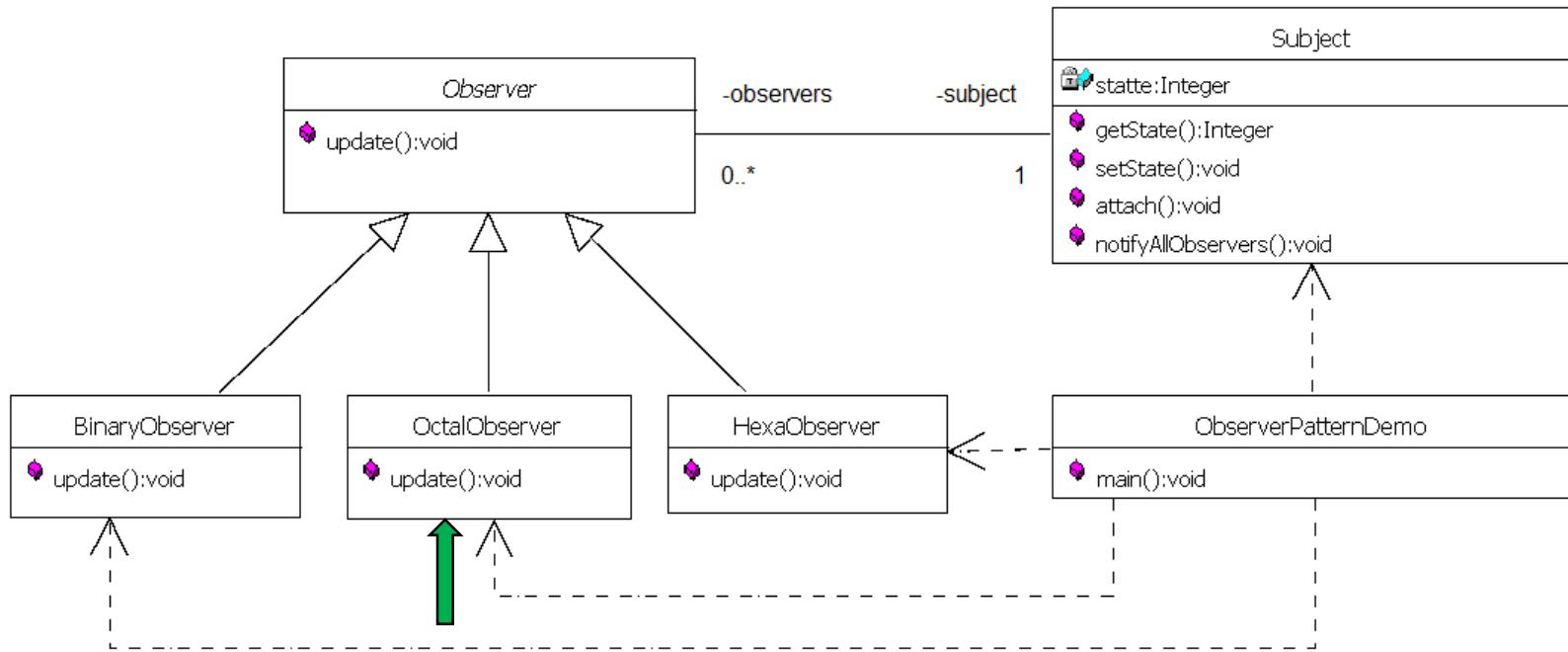


The Observer Pattern_4



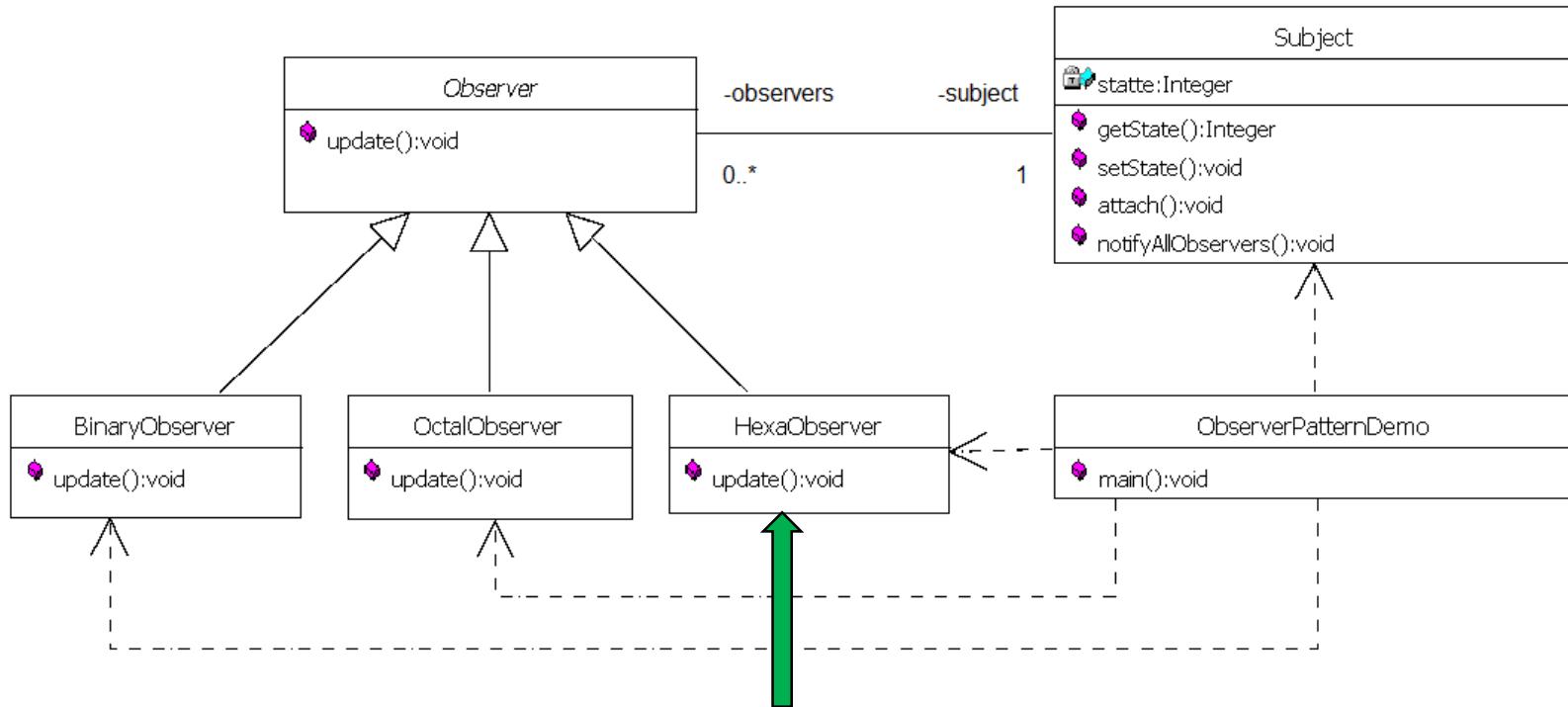
```
public BinaryObserver(Subject subject) {  
    this.subject = subject;  
    this.subject.attach(this);  
}  
@Override  
public void update() {  
    System.out.println("Binary String: " +  
Integer.toBinaryString(subject.getState()) );  
}
```

The Observer Pattern_5



```
public OctalObserver(Subject subject) {
    this.subject = subject;
    this.subject.attach(this);
}
@Override
public void update() {
    System.out.println("Octal String: " +
Integer.toOctalString(subject.getState()) );
}
```

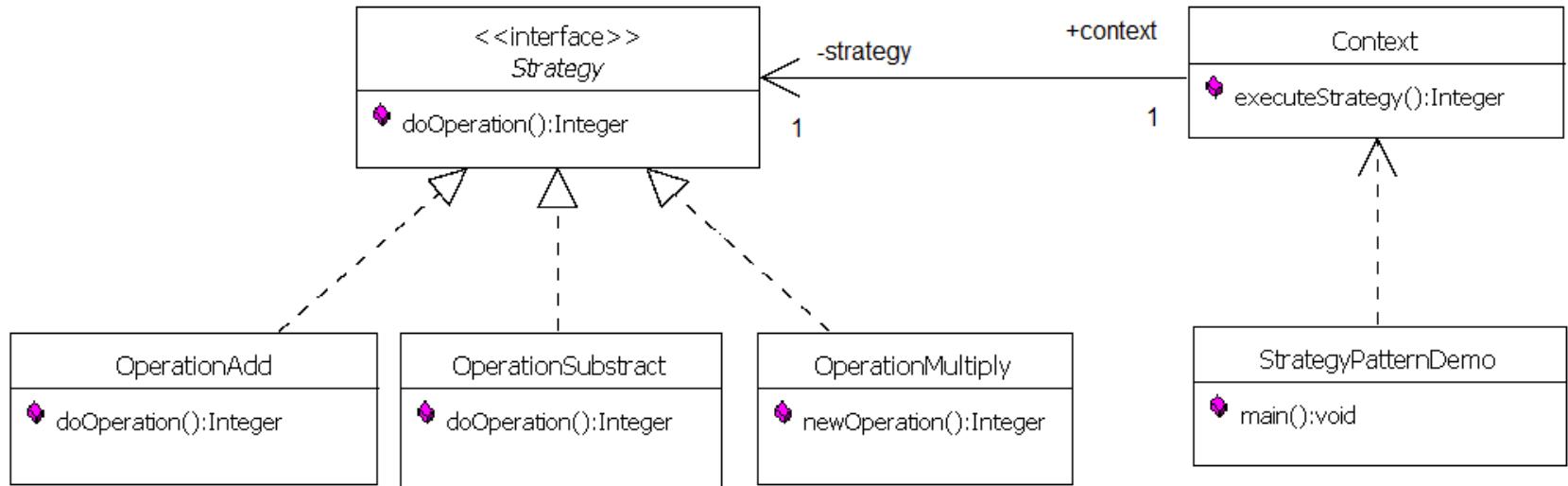
The Observer Pattern_6



```
public HexaObserver(Subject subject) {
    this.subject = subject;
    this.subject.attach(this);
}

@Override
public void update() {
    System.out.println("Hex String: " + Integer.toHexString(
subject.getState().toUpperCase()));
}
```

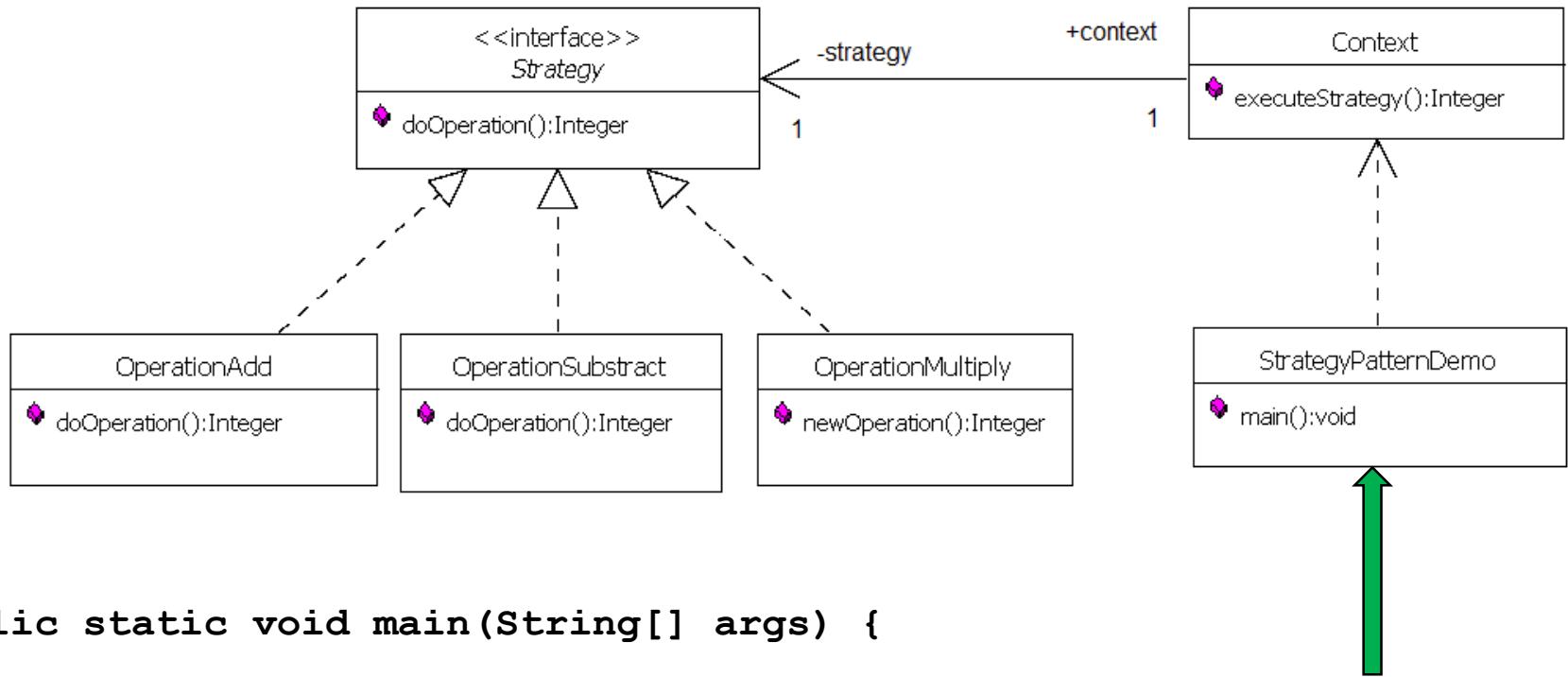
The Strategy Pattern



In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

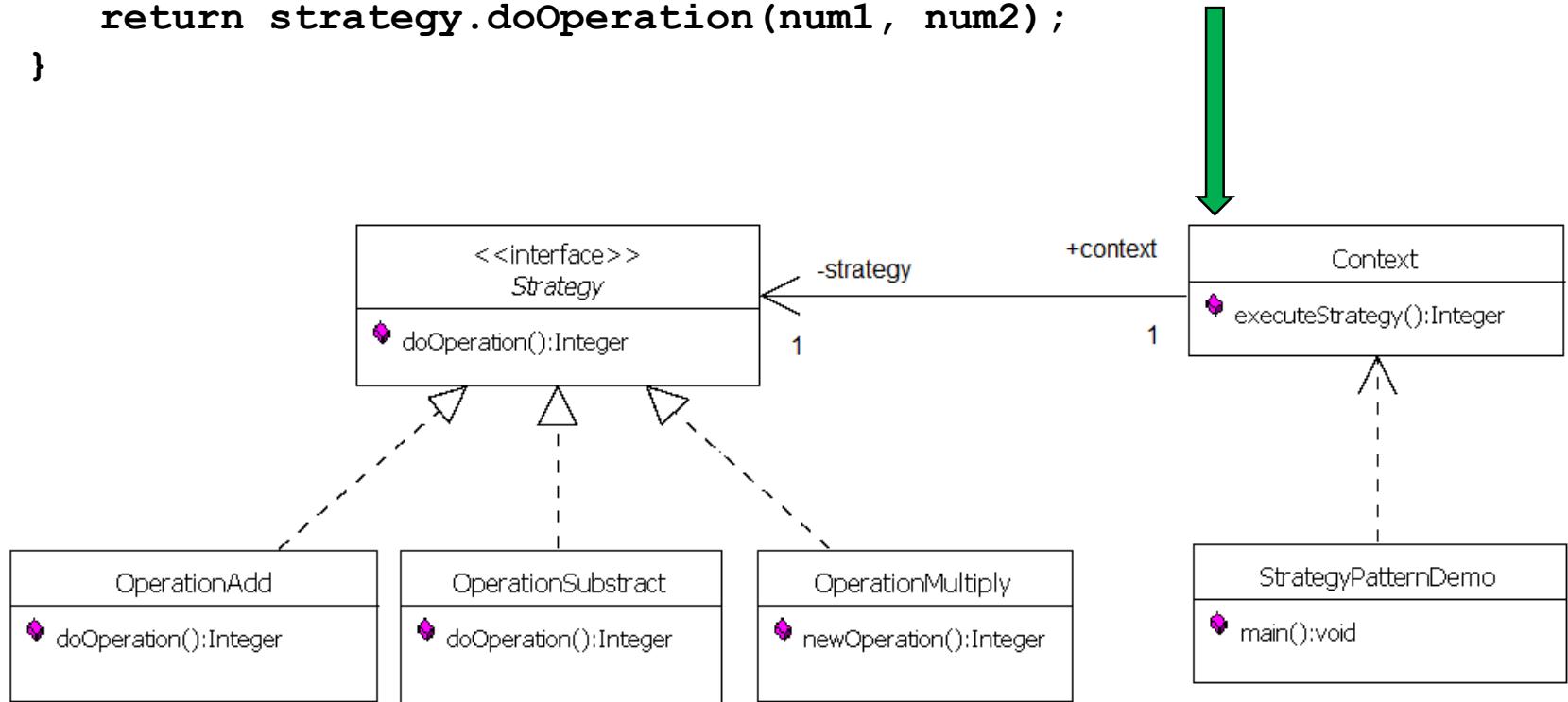
The Strategy Pattern_2



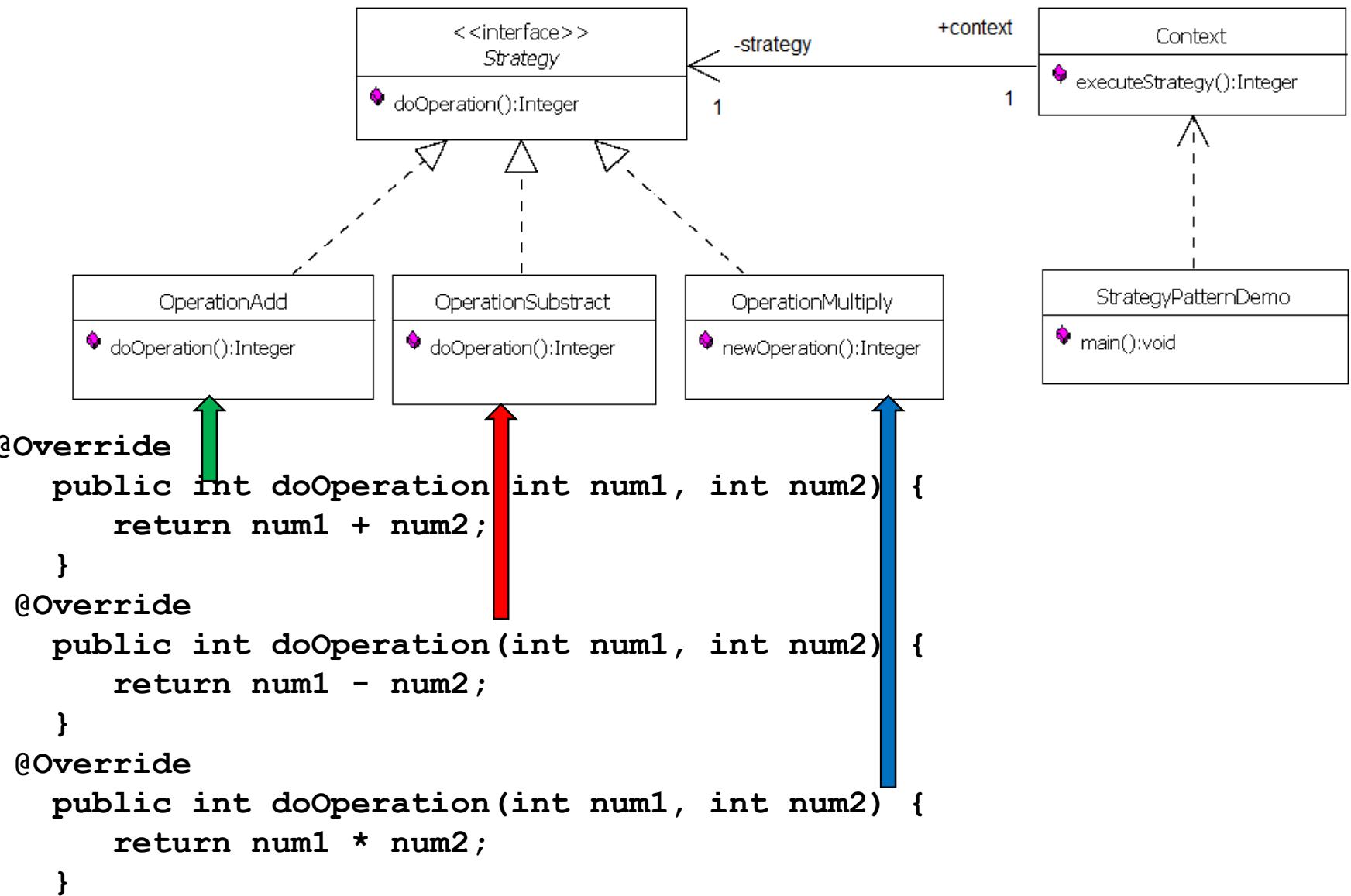
```
public static void main(String[] args) {  
  
    Context context = new Context(new OperationAdd());  
    System.out.println("10 + 5 = " + context.executeStrategy(10, 5));  
  
    context = new Context(new OperationSubtract());  
    System.out.println("10 - 5 = " + context.executeStrategy(10, 5));  
  
    context = new Context(new OperationMultiply());  
    System.out.println("10 * 5 = " + context.executeStrategy(10, 5));  
}  
.
```

The Strategy Pattern_3

```
public Context(Strategy strategy) {  
    this.strategy = strategy;  
}  
  
public int executeStrategy(int num1, int num2) {  
    return strategy.doOperation(num1, num2);  
}
```



The Strategy Pattern_4



Object Design Activities

1. Reuse: Identification of existing solutions

- Use of inheritance
- Off-the-shelf components and additional solution objects
- Design patterns

2. Interface specification

- Describes precisely each class interface

3. Object model restructuring

- Transforms the object design model to improve its understandability and extensibility

4. Object model optimization

- Transforms the object design model to address performance criteria such as response time or memory utilization.

**Object
Design**

**Mapping
Models to
Code**

Modeling of the Real World

- Modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.
- There is a need for *reusable* and flexible designs
- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge.

References for Patterns examples

- Design Patterns in Java Tutorial
- https://www.tutorialspoint.com/design_pattern/index.htm

The Object Constraint Language (OCL)

Dan CHIOREAN



“Babes-Bolyai” University CLUJ-NAPOCA

chiorean@cs.ubbcluj.ro

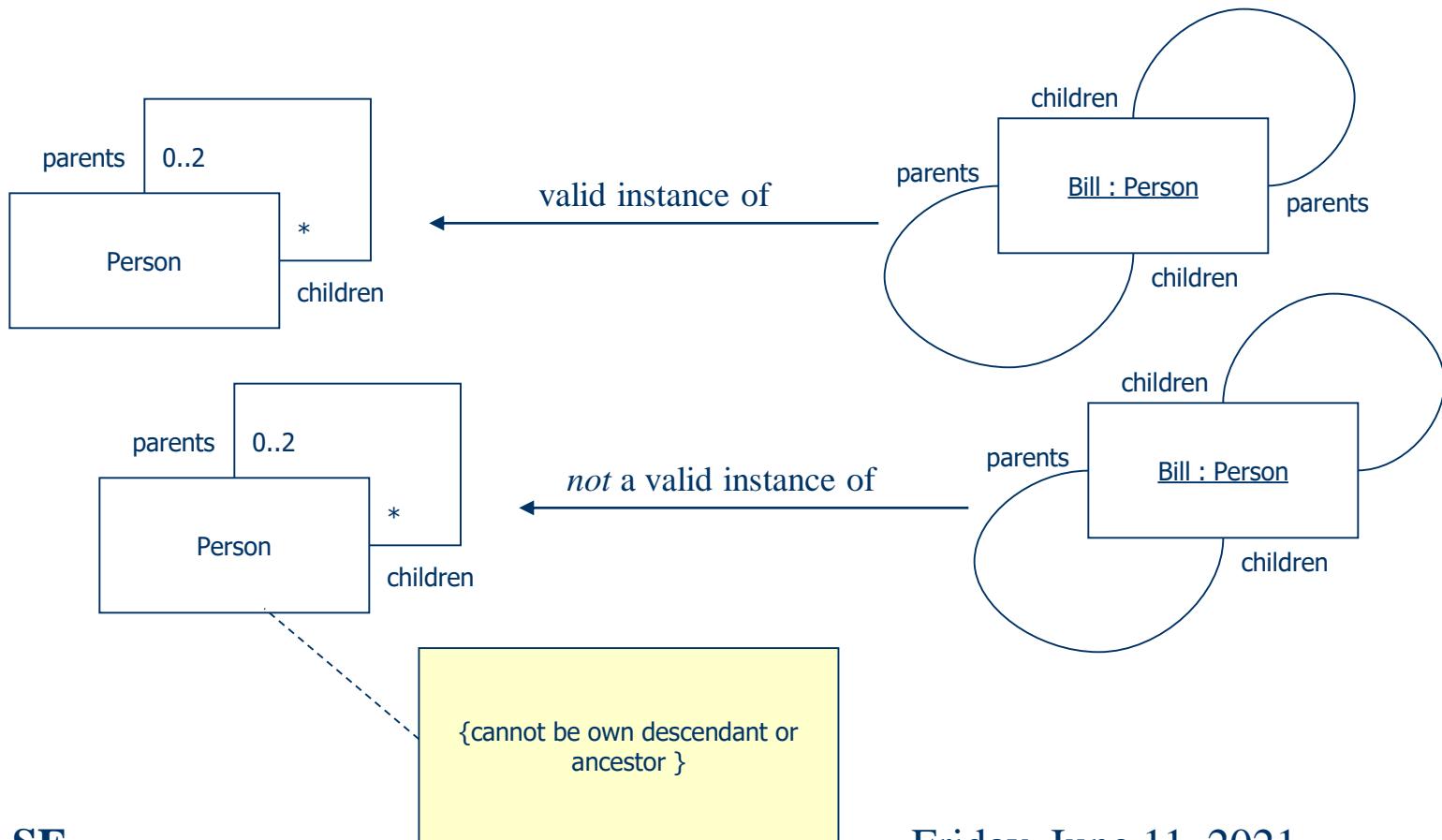
The Object Constraint Language UML & OCL

"In theory there is no difference between theory and practice; in practice there is."

Yogi Bera

The Object Constraint Language

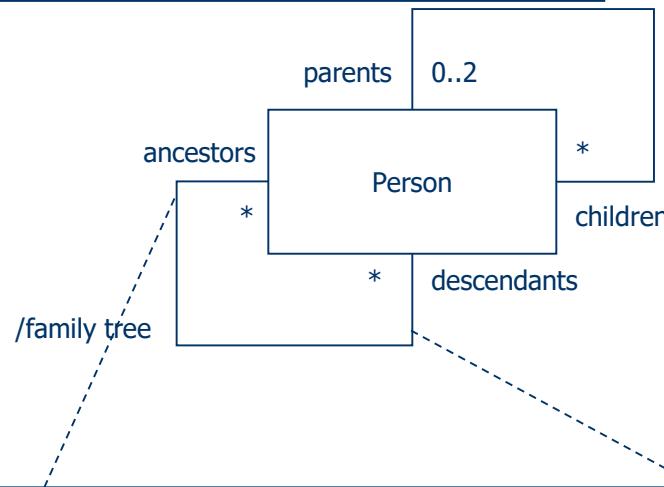
Only using UML diagrams can't tell the whole story



The Object Constraint Language

OCL supports the unambiguous constraints specifications

```
{ancestors = parents->union(parents.ancestors->asSet())}  
acs:Set(Person) = self.parents->union(parents->closure(p| p.parents))  
{descendants = children->union(children.descendants->asSet())}  
des:Set(Person) = self.children->union(children->closure(p | p.children))
```



```
{ancestors->excludes(self) and descendants->excludes(self) }
```

The Object Constraint Language OCLE 2.0 <http://lci.csubbclui.ro/ocle/>

ocle 2.0 - OCL Environment

File Model Project Edit Tools Options Help

S NewClassDiagram Person A_Person_Person Collaborations NewObjectDiagram p1_Person p10_Person p2_Person p3_Person p4_Person p5_Person p6_Person p7_Person p8_Person p9_Person A_p10_p6 A_p10_p9 A_p1_p1 A_p2_p2 A_p3_p3 A_p5_p4 A_p6_p3 A_p6_p4 A_p7_p5 A_p7_p8 String

NewClassDiagram

Person +parents 0..2 +children 0..

name:String

NewObjectDiagram

p1_Person name = Ion p2_Person name = Maria g1_Person father = Ion p4_Person name = Valde p5_Person name = Gheorghe p6_Person name = Ana p7_Person name = Valentina g10_Person name = Alexandru

C:\Users\Public\Examples_OCLE\FamilyModel.bor

```
model FamilyModel

context Person
    int parents() = self.parents->size();
    self.parents->excludes(self);

context Person
    def ancestors():Set[Person] = {self.parents->union(parents.ancestors())->asSet};

    let descendants:Set(Person) = {self.children->union(children.descendants())->asSet};

    int desc:Set(Person) = self.children->union|children->closure(p | p.children());
    let sons:Set(Person) = self.parents->union|parents->closure(p1 | p1.parents());
    int inv: int->excludes(self);

endmodel

// Set(i, 2, 5, Undefined)->size() /* returns 4 */
// Set(), 1+1, 5, Undefined)->size()
// Set(1, 1+1, 5, Undefined)->object(i | i=Undefined)
// Bag(1, 2, 1, Undefined)->count() /* returns 1 */
// Bag(1, 2, 1, Undefined, 5, Undefined)->object(i | i=Undefined)
// Bag(1, 2, 1, Undefined, 5, Undefined)->select(i | i <> Undefined)
// Bag(1, 2, 1, Undefined, 5, Undefined)->select(i | i > 1)
```

Java(TM) SE Development Kit 6 Update 24 (64-bit) 146 MB

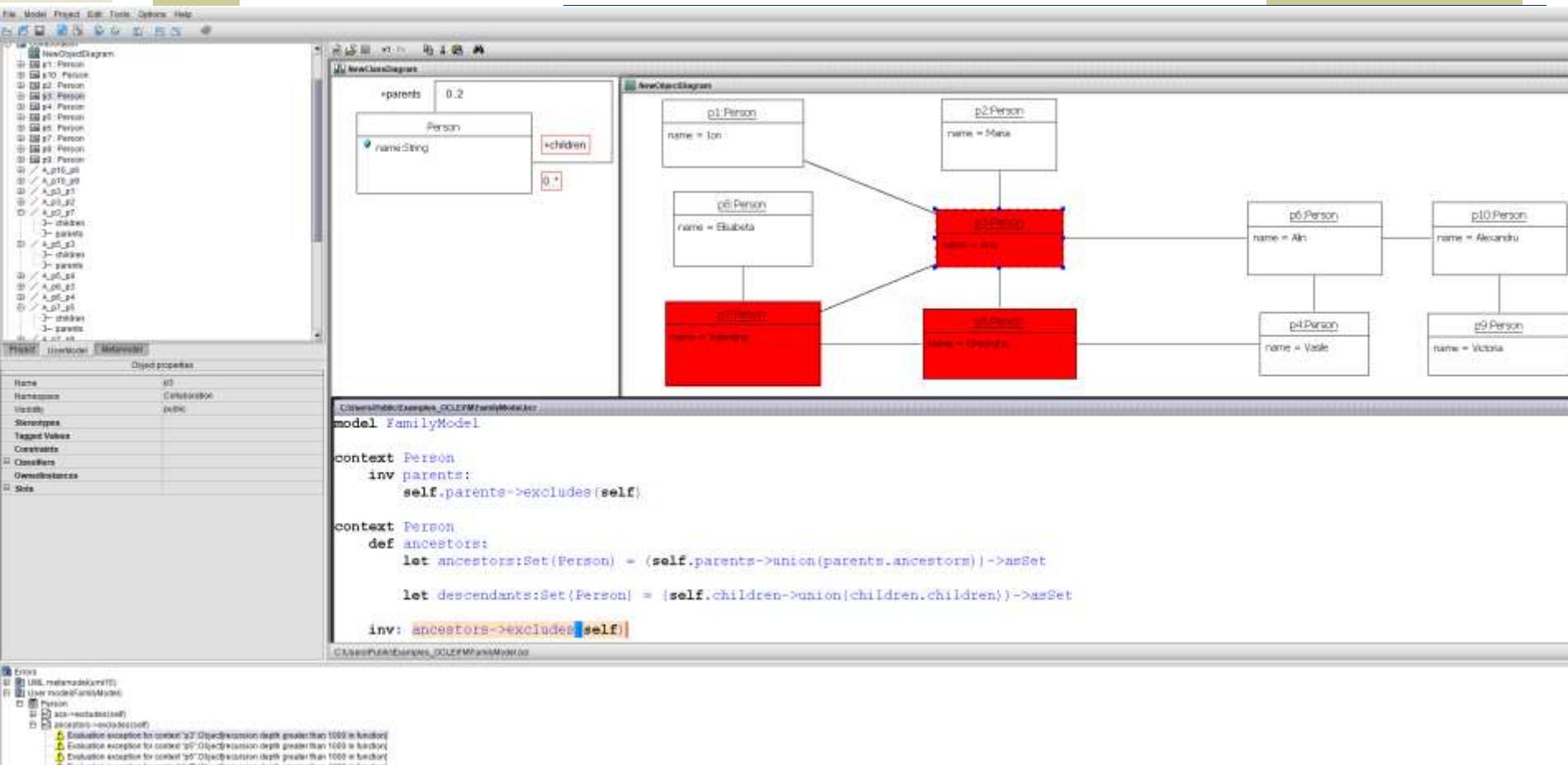
4/16/2019

1.6.0.240

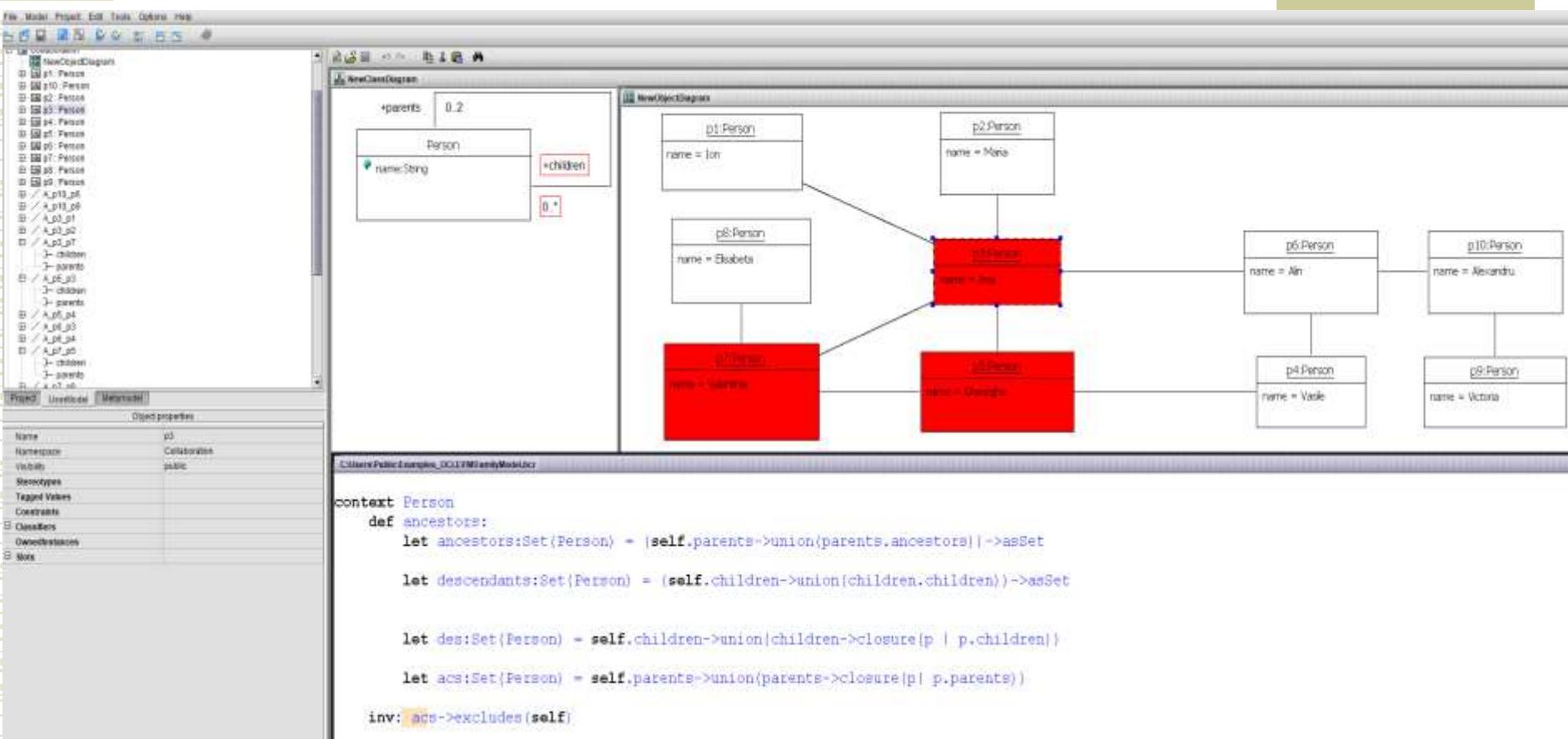
Activated UML model FamilyModel
Activated project FamilyModel
Opened file C:\Users\Public\Examples_OCLE\FamilyModel.bor
Compiling... successfully completed

LOG Messages OCL output Evaluation Search results

The Object Constraint Language Recursive Specification Drawback



The Object Constraint Language hidden specification errors



The Object Constraint Language

The context and main features

- ◆ OCL has arise from the necessity of bridging an expressivity gap of UML
 - ◆ diagrams are unable to capture most of the constraints imposed by nontrivial software systems
- ◆ => OCL is not a standalone language, but a language meant to complement UML. OCL expressions are meaningful in the context of a valid UML/(MOF based) model [except OCL literal expressions].
- ◆ the current OCL standard is a side free effects (declarative) language. So, the evaluation of an OCL specification does not change the UML model

The Object Constraint Language

The context and main features 2

- ◆ OCL is a typed language – all elements of OCL expressions are typed => the type of any OCL expression can be obtained by inference
- ◆ OCL supports first order logic
- ◆ **the language supports main features of OOP.** OCL specifications are inherited in descendants, where they may be overwritten. Constraint redefinition complies with the rules established in Design By Contract. The language supports type-casting, including upper type-casting

The Object Constraint Language

The purposes of using OCL

- ◆ **to support:**
- ◆ *model navigation* - querying the information in a model through (repeated) navigation of its association relationships using role names;
- ◆ *specification of assertions* - explicit definition of pre/post-conditions and invariants, as promoted by Design by Contract;
- ◆ *definitions of behavior* - body specifications for the query operations included in the model, derivation rules for existing attributes and references, as well as the definition of new operations and attributes for the model;
- ◆ *specification of guards, of type invariants for stereotypes*, etc.

The Object Constraint Language

The purposes of using OCL 2

- The objectives of using OCL go beyond the purpose of accomplishing a complete and unambiguous description of the problem solution by means of models.
- The final target is **to produce high quality software by using models**. Translating model-level OCL specifications into code must be done in a natural and unequivocal manner. Moreover, the results obtained when evaluating OCL specifications on model instantiations should be equivalent to the results obtained at run time, when evaluating their corresponding code on similar configurations of objects.

The Object Constraint Language

OCL Types

- Predefined types:
 - Basic Types: Boolean, Integer, Real, and String
 - Collections Types: Collection, Set, Bag, OrderedSet, and Sequence
- User-defined types:
 - Enumeration Type, Tuple Type
 - Types defined in the UML diagrams
 - OclAny, OclVoid, OclInvalid

The Object Constraint Language

OCL Types

- ◆ Precedence order for operations, starting with highest:
 - `@pre`
 - dot and arrow operations: `'.'` and `'->'`
 - unary 'not' and unary minus `'-'`
 - `'*'` and `'/'`
 - `'+'` and binary `'-'`
 - 'if-then-else-endif'
 - `'<'`, `'>'`, `'<='`, `'>='`
 - `'='`, `'<>'`
 - 'and', 'or' and 'xor'
 - 'implies'
- ◆ Parentheses `(''` and `)'` can be used to change precedence
 - Often useful simply to make it clearer

The Object Constraint Language

OCL specification

- OCL specifications are in a context like:
 - **context** ClassName/InterfaceName
inv [nameOfInv] : | **def** DefName:/DefSignature:=oclExpr
 - **let** varName:Type = oclExpr
 - **context** ClassName::OpSignature
 - **pre** [nameOfPre] : | **post** [nameOfPost] :
 - **package** PackageName
 - ...
 - **endpackage**
 - **context** StateMachineName
inv [nameOfInv] :

The Object Constraint Language

Boolean values

Operation	Notation	Result Type
or	$a \text{ or } b$	Boolean
and	$a \text{ and } b$	Boolean
exclusive or	$a \text{ xor } b$	Boolean
negation	$\text{not } a$	Boolean
equals	$a = b$	Boolean
not equals	$a \neq b$	Boolean
implies	$a \text{ implies } b$	Boolean

The Object Constraint Language

Boolean values 2

a	b	not a	a or b	a and b	a implies b	a xor b
false	false	true	false	false	true	false
false	true	true	true	false	true	true
false	ε	true	⊥	false	true	⊥
false	⊥	true	⊥	false	true	⊥
true	false	false	true	false	false	true
true	true	false	true	true	true	false
true	ε	false	true	⊥	⊥	⊥
true	⊥	false	true	⊥	⊥	⊥
ε	false	⊥	⊥	false	⊥	⊥
ε	true	⊥	true	⊥	⊥	⊥
ε	ε	⊥	⊥	⊥	⊥	⊥
ε	⊥	⊥	⊥	⊥	⊥	⊥
⊥	false	⊥	⊥	false	⊥	⊥
⊥	true	⊥	true	⊥	⊥	⊥
⊥	ε	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

Four-valued logic(ε invalid value, ⊥ undefined value)

The Object Constraint Language

Evaluating expressions containing undefined values

```
Set{1, 2, 5, Undefined}->size()  
  
Set{1, 1+1, 5, Undefined}->size()  
  
Set{1, 1+1, 5, Undefined}->reject(i | i=Undefined)  
  
Bag{1, 2, 1, Undefined}->count(1)  
  
Bag{1, 2, 1, Undefined, 5, Undefined}->reject(i | i=Undefined)  
  
Bag{1, 2, 1, Undefined, 5, Undefined}->select(i | i <> Undefined)  
  
Bag{1, 2, 1, Undefined, 5, Undefined}->select(i | i > 1)
```

The Object Constraint Language

Type system – important features

OclAny

- OclAny is the supertype of all types in UML models and is an instance of the metatype AnyType. Features of OclAny are available on each object. Each class of UML user models inherits all operations defined on OclAny. Most of them are basic operations in object-oriented languages, such as:
 - **operations for testing the equality of two objects**
 - `= (object2:OclAny) : Boolean`
 - `<> (object2:OclAny) : Boolean`
 - **operations inferring the objects type or state**
 - `oclIsTypeOf (type:Classifier) : Boolean`
 - `oclIsKindOf (type:Classifier) : Boolean`
 - `oclType () : Classifier`
 - `oclIsInState (statespec:OclState) : Boolean`

The Object Constraint Language

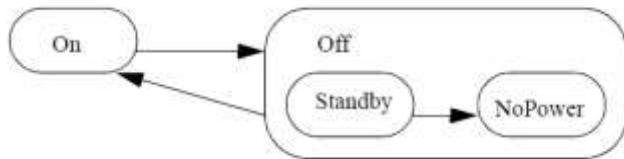
Type system – important features 2

OclAny

- operations specifying type casting
 - `oclAsType(type:Classifier) :T`, where T is a classifier
- other operations
 - `oclIsNew() :Boolean`, conceived to be used in postconditions, to support the identification of objects created after the method starts executing
 - `oclIsUndefined() :Boolean` and `oclIsInvalid() :Boolean` are two operations returning true when the receiver object or data value is undefined/unknown or when an exception has been triggered, respectively.

The Object Constraint Language

OclAny– oclIsInState



The operation **oclInState(s)** results in true if the object is in the state **s**. Values for **s** are the names of the states in the statemachine(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon ‘::’ . In the example statemachine above, values for **s** can be **On**, **Off**, **Off::Standby**, **Off::NoPower**.

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off::NoPower)
```

If there are multiple statemachines attached to the object’s classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double semicolon ::, as with nested states.

The Object Constraint Language

Accessing Overridden Properties

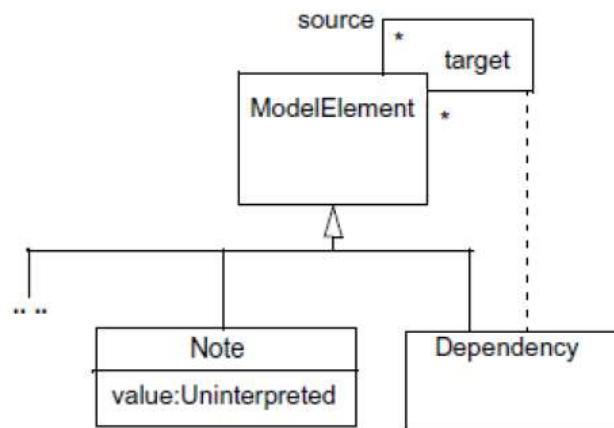


Figure 7.4 - Accessing Overridden Properties Example

```
context Dependency
  inv: self.source <> self
  inv: self.oclAsType(Dependency).source->isEmpty()
  inv: self.oclAsType(ModelElement).source->isEmpty()
```

The Object Constraint Language

pre & post(conditions)

- Constraint that specify the applicability and effect of an operation without stating an algorithm or implementation
- Are attached to an operation in a class diagram
- Allow a more complete specification of a system

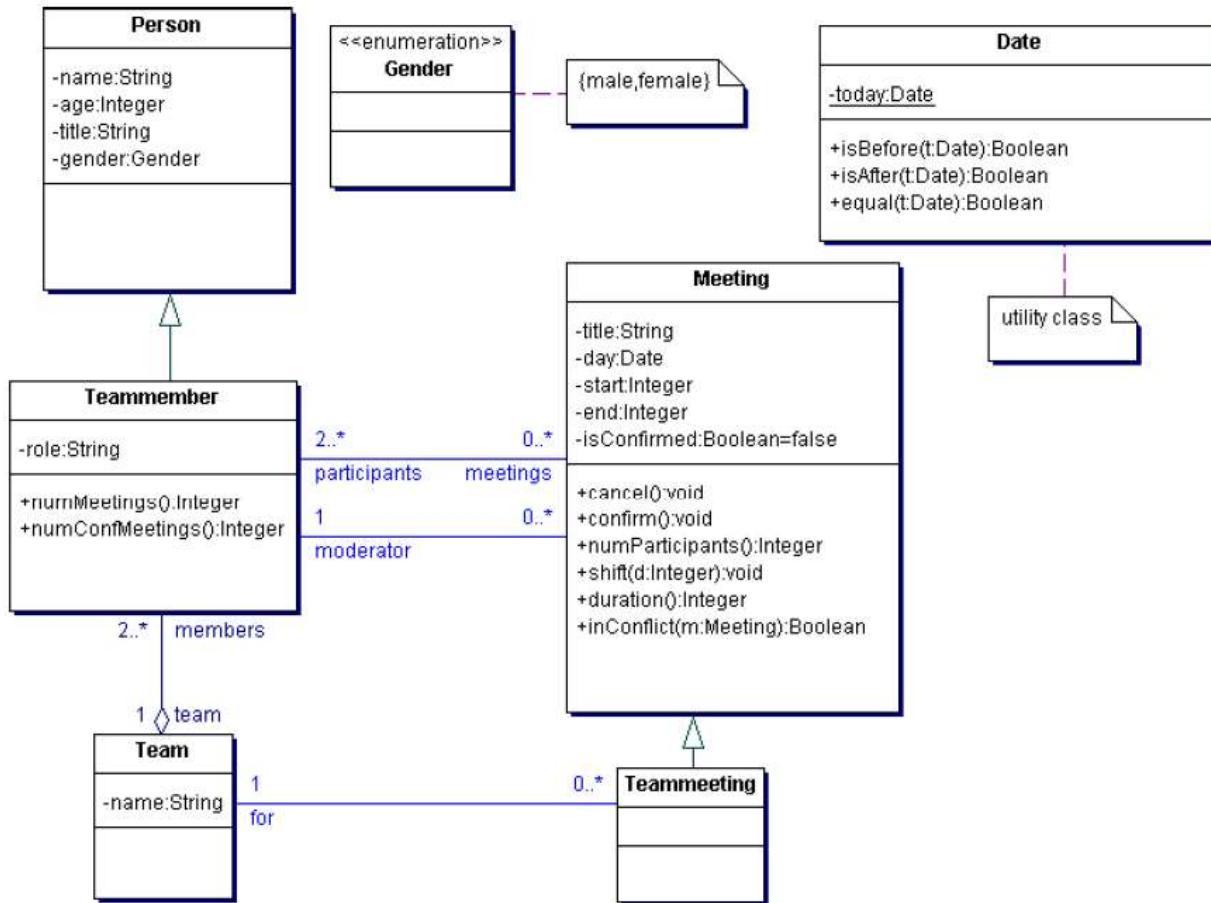
Preconditions must be true just prior to the execution of an operation

Precondition

Syntax

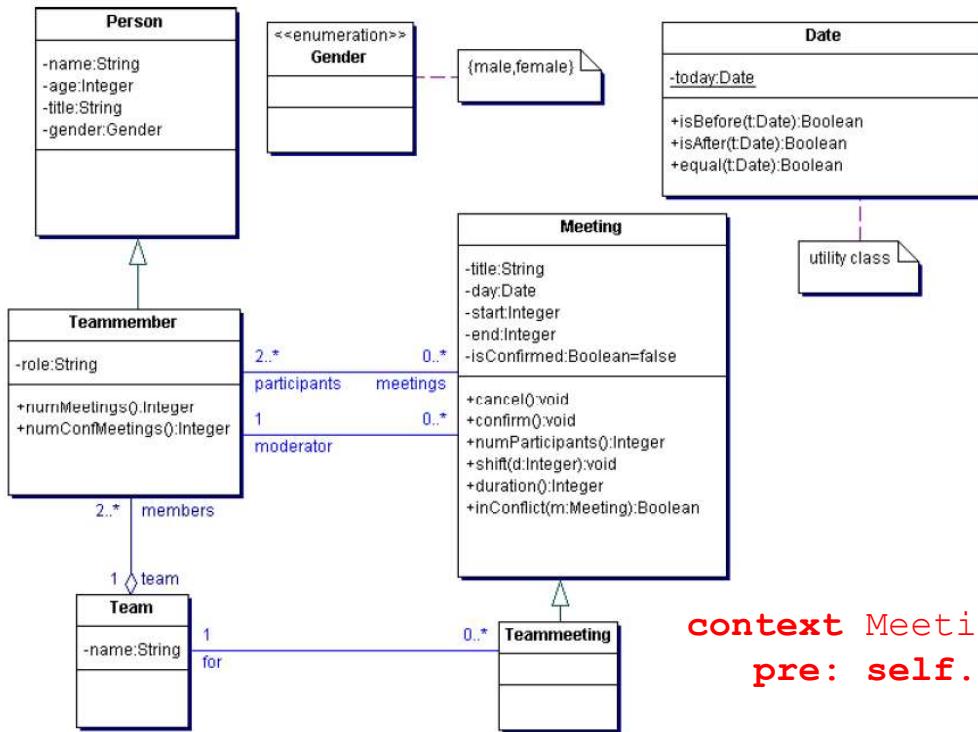
```
context <classifier>::<operation> (<parameters>)
  pre [<constraint name>] :
    <Boolean OCL expression>
```

The Object Constraint Language pre & post(conditions) 2



The Object Constraint Language

pre & post(conditions) 3



context `Meeting::shift(d: Integer)`
pre: `self.isConfirmed = false`

context `Meeting::shift(d: Integer)`
pre: `d>0`

context `Meeting::shift(d: Integer)`
pre: `self.isConfirmed = false and d>0`

The Object Constraint Language

pre & post(conditions) 4

Postcondition

Constraint that must be true just after to the execution of an operation. Postconditions are the way how the actual effect of an operation is described in OCL.

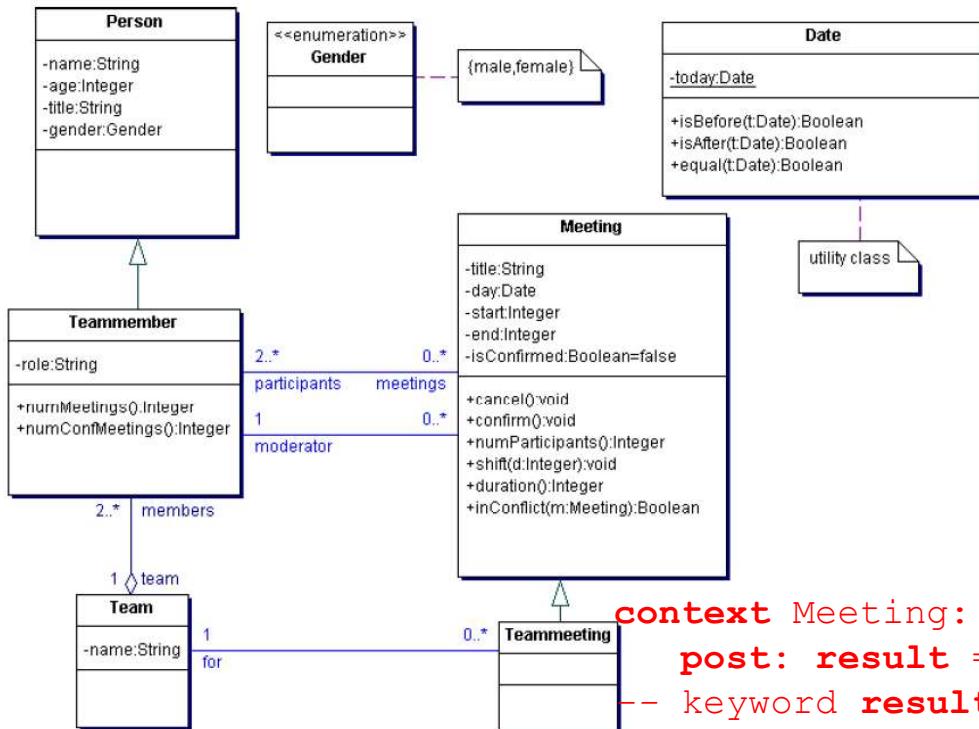
Syntax

```
context <classifier>::<operation> (<parameters>)
  post [<constraint name>]:
    <Boolean OCL expression>
```

- `vizElem@pre` indicates a part of an expression which is to be evaluated in the original state before execution of the operation
- `vizElem` refers to the value upon completion of the operation
- `@pre` is only allowed in postconditions

The Object Constraint Language

pre & post(conditions) 5



```
context Meeting::duration():Integer
post: result = self.end - self.start
-- keyword result refers to the result of the operation
```

```
context Meeting::confirm()
post: self.isConfirmed = true
```

```
context Meeting::shift(d:Integer)
post: start = start@pre + d and end = end@pre + d
```

The Object Constraint Language

Collection types

- Models are modeled by graphs => the need to navigate/query the model
- At the class level, this is done by accessing the **appropriate opposite association end**. When the multiplicity of the association end is greater than 1, navigation will result in a **Set**. When the association end is adorned with {ordered}, the result will be an **OrderedSet**. Therefore, the **collection types defined in the OCL Standard Library play an important role in OCL expressions.**
- Apart of the two above mentioned types, collection types also include **Bag**, **Sequence** and **Collection**, the latter being the common parent of the other four collection types.
- The hierarchy and behavior of these types has been inspired by Smalltalk Collection classes.

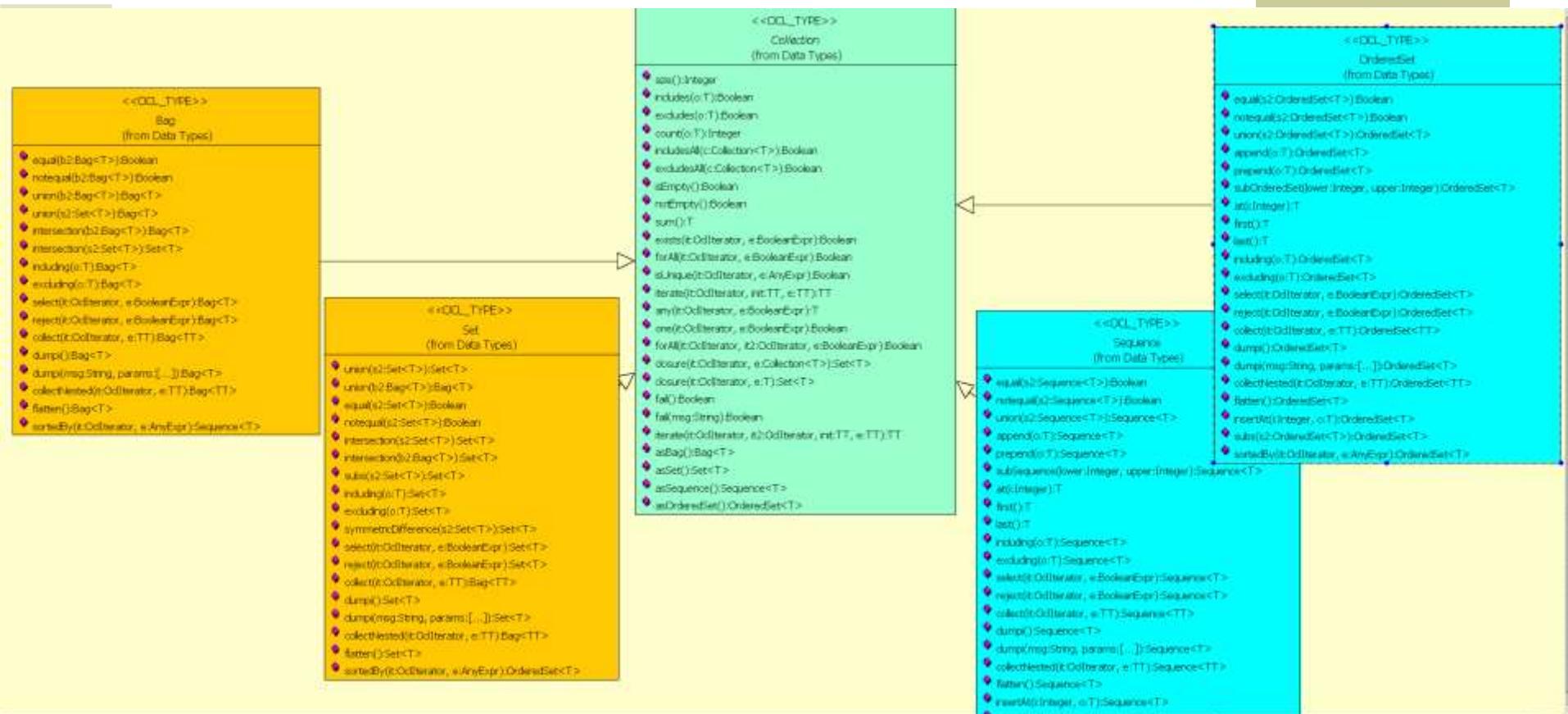
The Object Constraint Language

Collection types 2

- Usually, operations applied to collections are specified by using the arrow operator (->). This is meant to stress that, in case of collections, operations are potentially applied to many receivers (collection elements). However, in case of the **collect** operation, an exception (enabling to replace the -> operator with the dot operator (.)) is made.
- So, the expression **aCollection->collect(property)** is equivalent to **aCollection.property**. This exception, known as "shorthand collect", is very used in practice, since it leads to slightly shorter specifications.

The Object Constraint Language

Collection hierarchy in OCL



The Object Constraint Language

Collection – the **iterate** operation

- ◆ The **iterate** operation is very generic. All the operations: **reject**, **select**, **forAll**, **exists**, **collect** can all be described in terms of **iterate**.

```
collection->iterate( elem : Type; acc : Type = <expression> |  
expression-with-elem-and-acc )
```

- ◆ When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with elem-and-acc* is evaluated for each *elem*. After each evaluation of *expression-with elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection.

```
collection->collect(x : T | x.property)
```

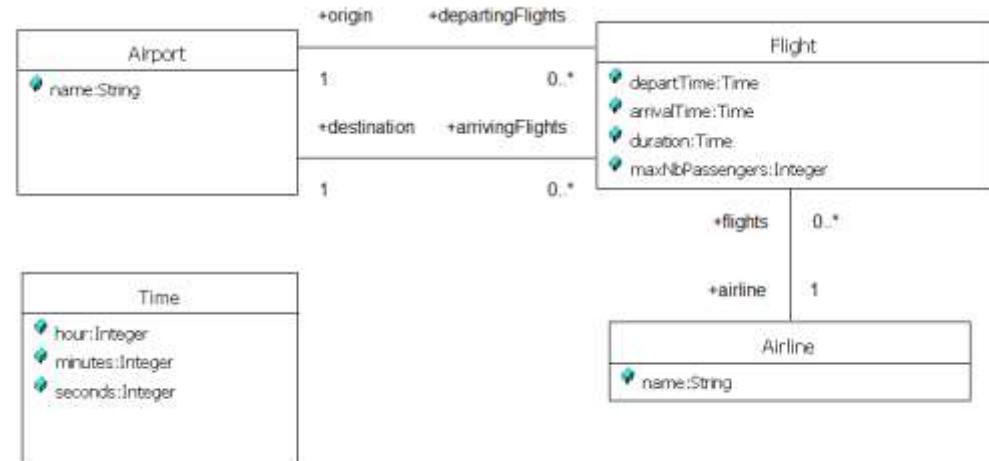
-- is identical to:

```
collection->iterate(x : T; acc : T2 = Bag{}|acc->including(x.property))  
SE
```

The Object Constraint Language

Collection types 3 - Navigating associations

- Every association in the model is a navigation path.
- The context of the expression is the starting point.
- Role names are used to identify the navigated association.

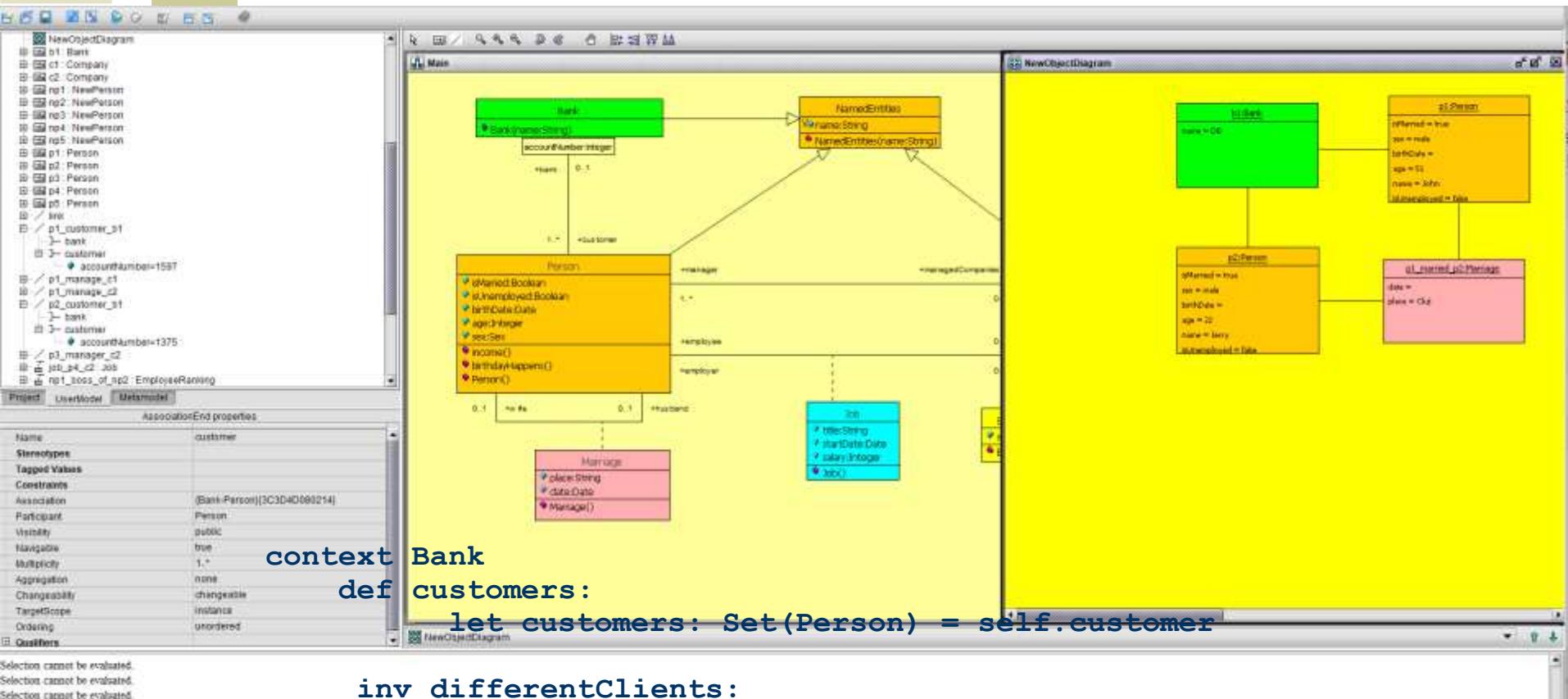


context Flight

inv: origin <> destination
inv: origin.name = "Amsterdam"

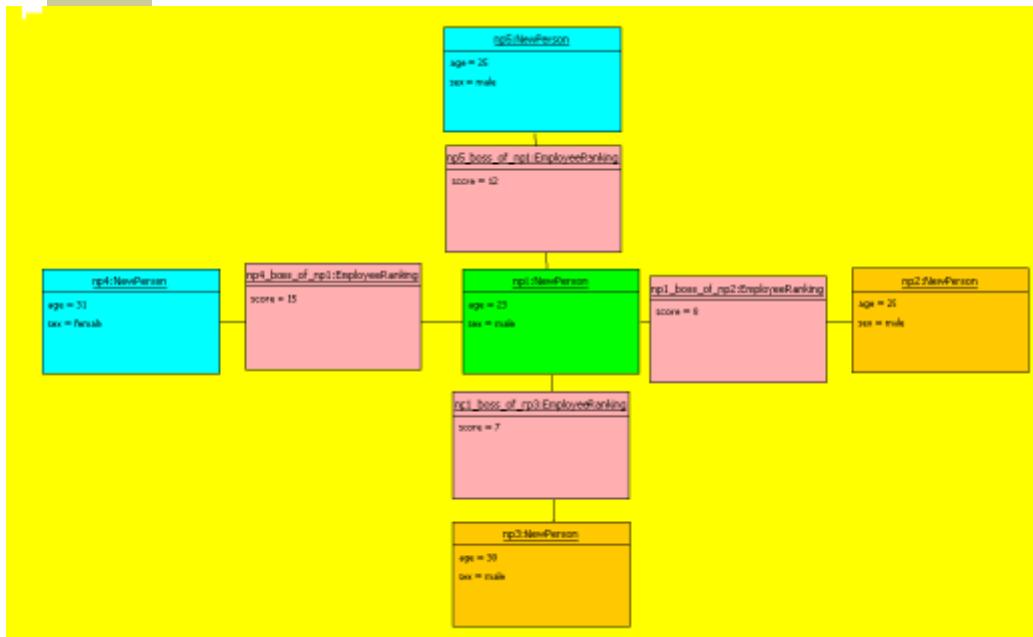
The Object Constraint Language

Qualified associations



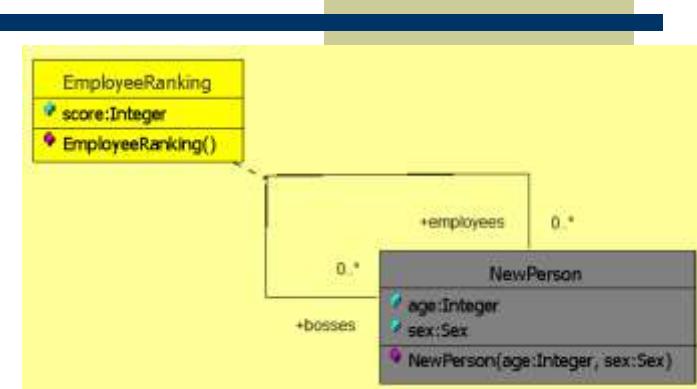
The Object Constraint Language

Association classes



```

context NewPerson
inv bosse_s_objectivy:
  let nB: Integer = self.bosses->size
  let nE: Integer = self.employees->size in
  if nB*nE = 0
    then true
    else
      (self.employeeRanking[bosses].score->sum / nB) <=
      (self.employeeRanking[employees].score->sum / nE)
  endif
  
```



The Object Constraint Language

Collection types 4 – the select & forAll operations

Syntax:

```
collection->select(elem : T | expression)
collection->select(elem | expression)
collection->select(expression)
```

The *select and reject* operations does not change the type of the collection
The *select* operation results in the subset of all elements for which *expression* is true

Syntax:

```
collection->forAll(elem : T | expr)
collection->forAll(elem | expr)
collection->forAll(expr)
```

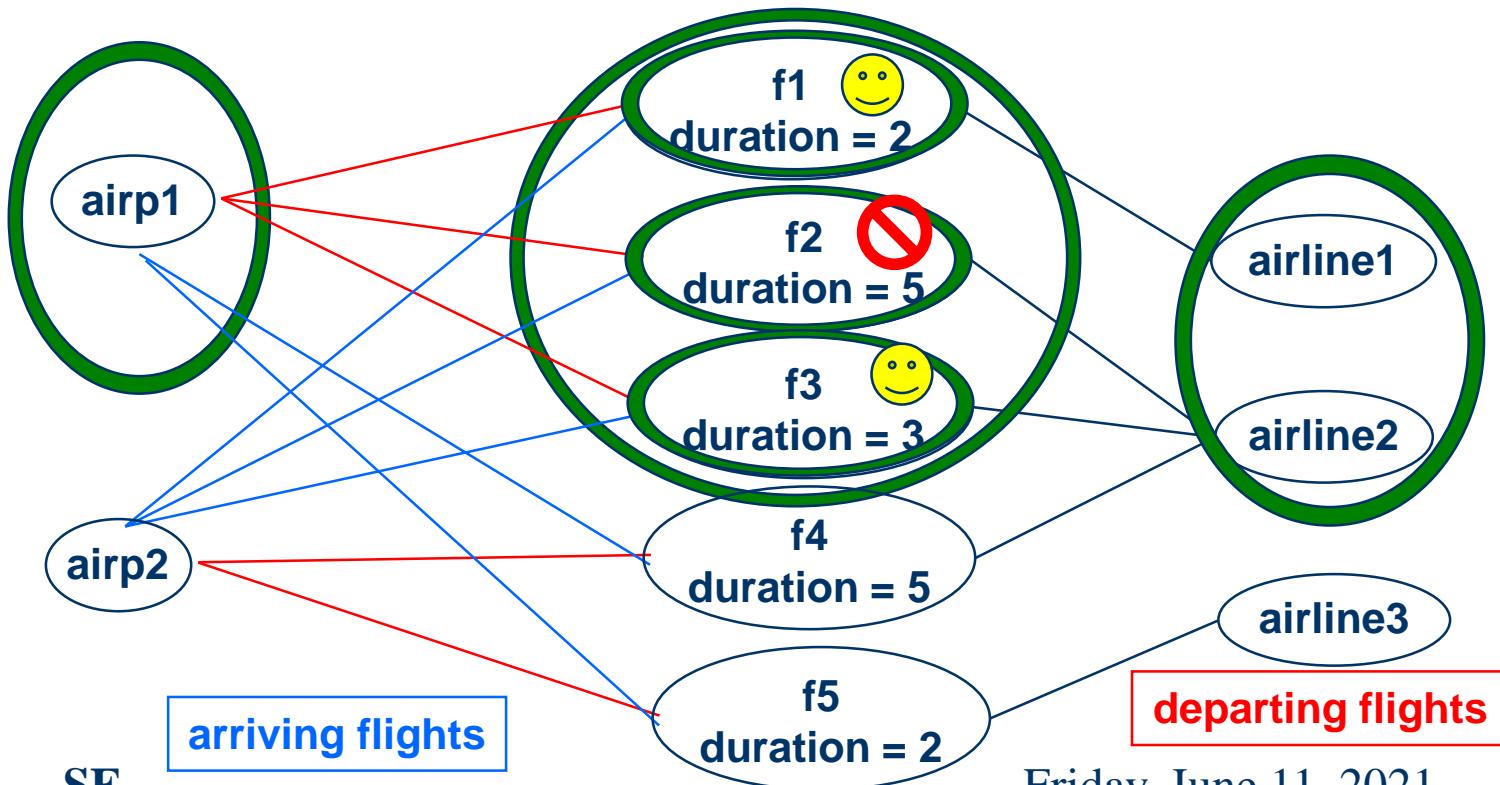
The *forAll* operation results in `true` if *expr* is `true` for all elements of the collection. In case of medium and large size collections it's difficult to find element(s) violating the invariant

The Object Constraint Language

Collection types 5 – the select operation

context Airport inv:

self.departingFlights->select (duration<4)->notEmpty

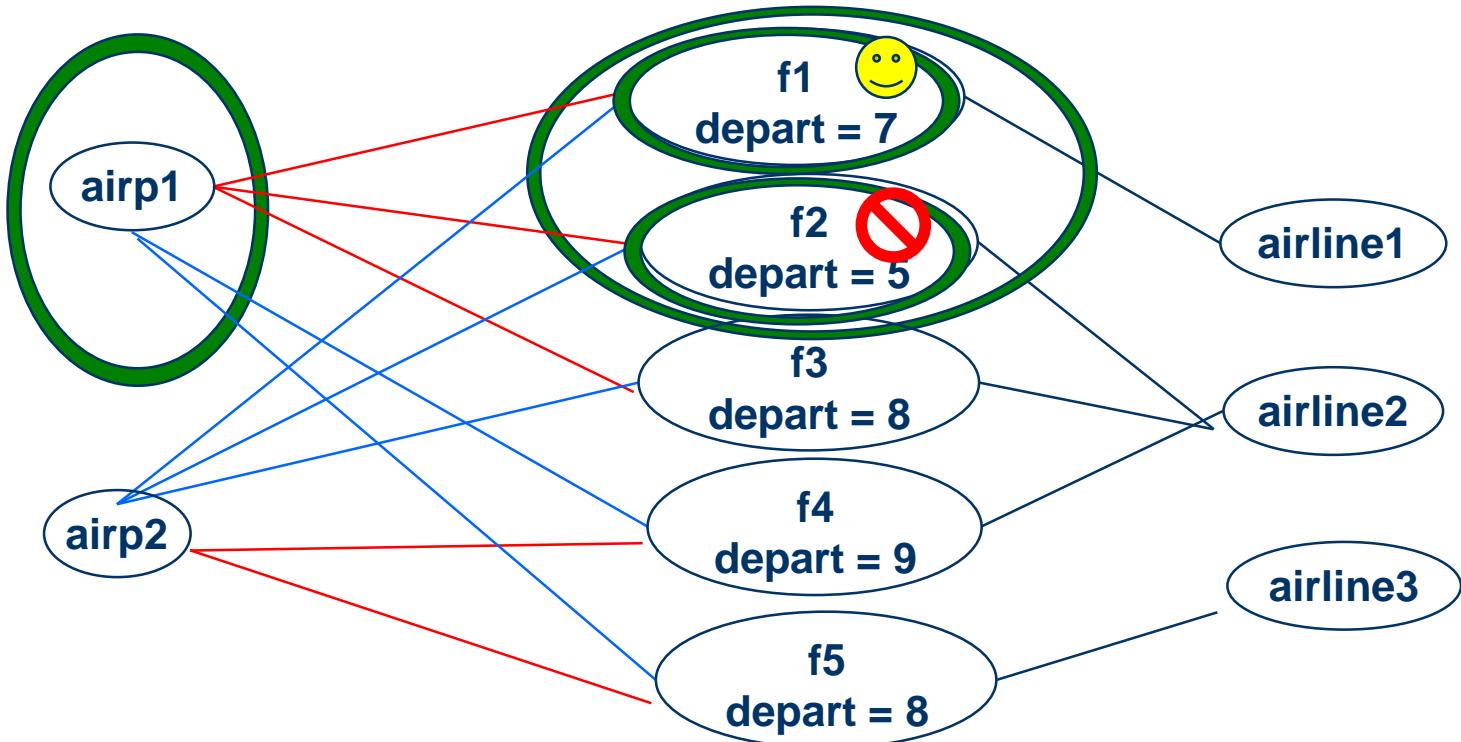


The Object Constraint Language

Collection types 6 – the forAll operation

context Airport **inv:**

self.departingFlights->forAll (departTime.hour>6)



departing flights

arriving flights

The Object Constraint Language

Collection types 7 – the exists operation

Syntax:

```
collection->exists(elem : T | expr)  
collection->exists(elem | expr)  
collection->exists(expr)
```

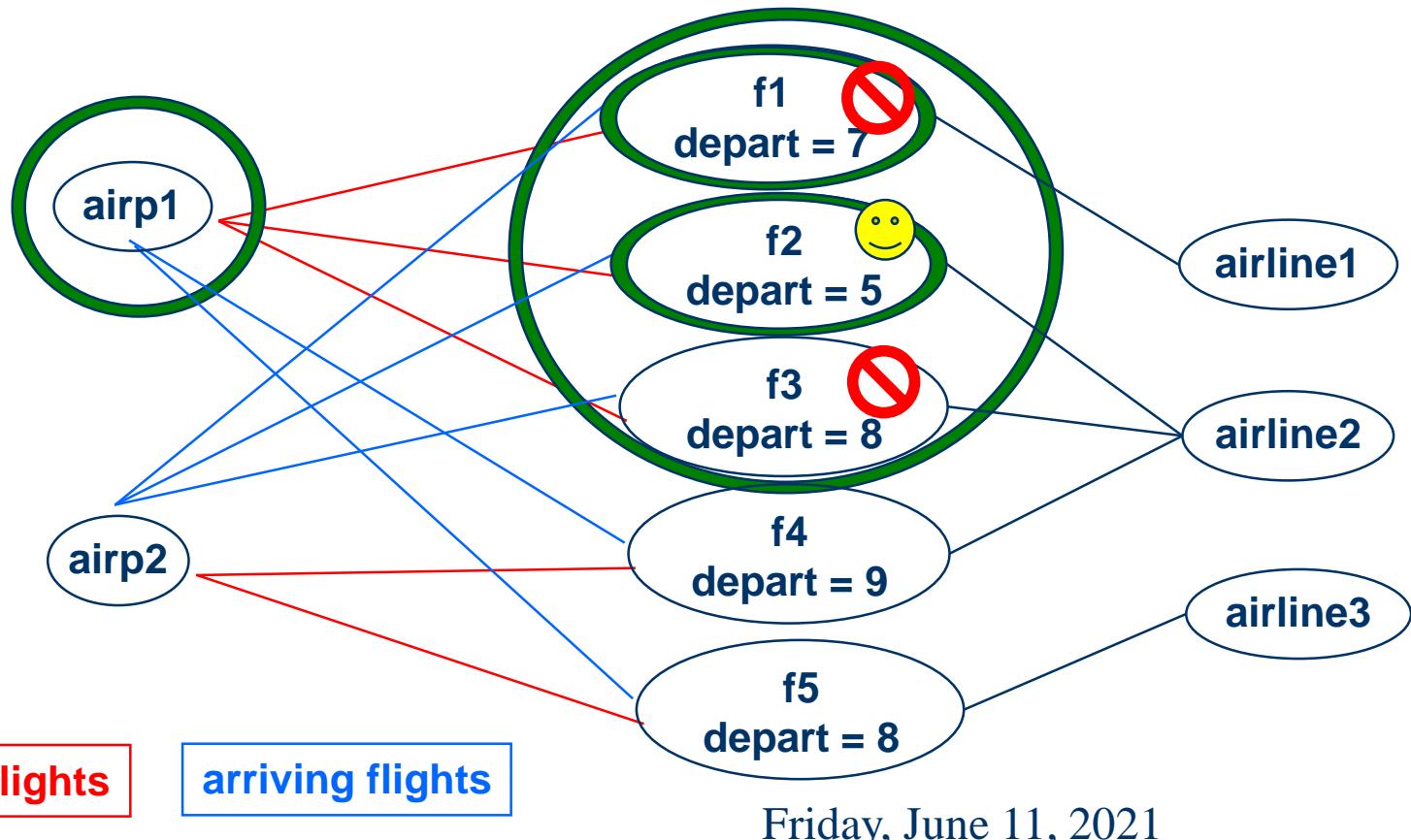
The exists operation results in true if there is at least one element in the collection for which the expression *expr* is true.

The Object Constraint Language

Collection types 8 – the exists operation

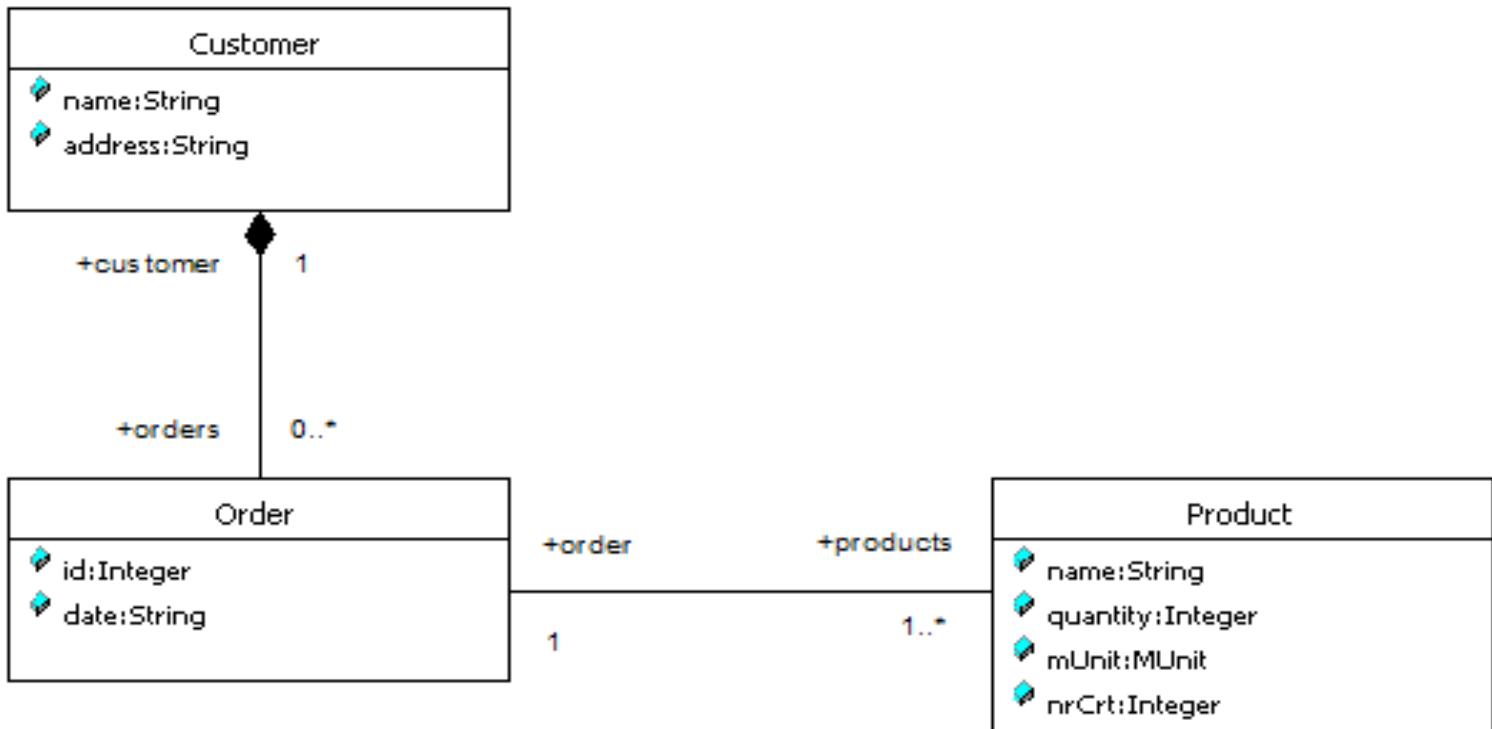
context Airport **inv:**

self.departingFlights->exists (departTime.hour<6)



The Object Constraint Language

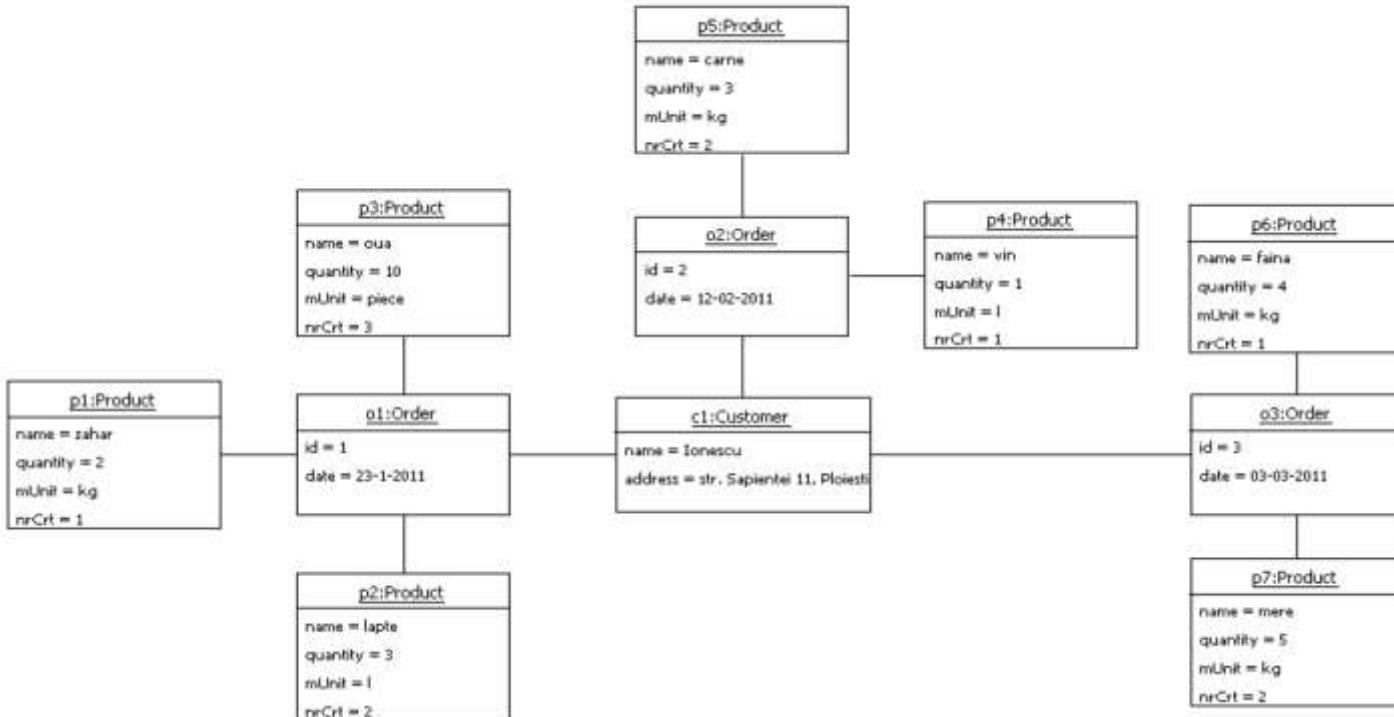
Collection types 9



Usually, users interested in knowing the products ordered by a customer evaluate the following OCL specification `self.orders->collect(products)`.

The Object Constraint Language

Collection types 10



context Customer

def productsOrderedByClient:**Bag**(Product)=**self**.orders.products

The result is: **Bag**{p1, p2, p3, p4, p5 , p6, p7}.

The Object Constraint Language

Collection types 11

- More information can be obtained by using the `collectNested` operation:

```
context Customer
```

```
def productsOrderedByClient_cn:Sequence(Sequence(Product)) =  
self.orders->collectNested(products)
```

- The result returned by the OCL evaluator will be:

```
Sequence{Sequence{p1, p2, p3}, Sequence{p4, p5}, Sequence{p6, p7}}.
```

- Compared with the result of the previous query, this time we know the products grouped by order, but we do not know the id or the reference of each order.

The Object Constraint Language

Collection types 12

- Compared with the result of the previous query, this time we know the products grouped by order, but we do not know the id or the reference of each order. Slightly detailing the previous **collectNested** operation will support us in obtaining the missing information.

context Customer

```
def productsOrderedByClient_cnd:Sequence(Sequence(OclAny)) =  
self.orders->collectNested(o| Sequence{o, o.products})
```

- The result returned will be:

Sequence{ Sequence{o1, Sequence{p1, p2, p3}}, Sequence{o2, Sequence{p4, p5}}, Sequence{o3, Sequence{p6, p7}} }.

The Object Constraint Language

Collection types 13

- Moreover, if one is interested in knowing the date of each order, the query can be refined as follows:

```
context Customer
def productsOrderedByClient_cndd:Sequence(Sequence(OclAny)) =
self.orders->collectNested(o| Sequence{o, o.date, o.products})
```

- This returns:

```
Sequence{Sequence{o1, '23-1-2011', Sequence{p1, p2, p3}},
Sequence{o2, '12-02-2011', Sequence{p4, p5}}, Sequence{o3,
'03-03-2011', Sequence{p6, p7}}}.  
}
```

The Object Constraint Language

Collection types 14

- Using a TupleType, the specification could be rewritten as:

```
context Customer
def clientProducts_nsct:Sequence(TupleType(date:String,
id:Integer, ordProds:Sequence(Product))) =
self.orders->collectNested(o| Tuple{date=o.date, id=o.id,
ordProds = o.products})
```

- Returning:

```
Sequence{Tuple{'23-1-2011', 1, Sequence{p1, p2, p3}},
Tuple{'12-02-2011', 2, Sequence{p4, p5}}, Tuple{'03-03-2011',
3, Sequence{p6, p7}}}.
```

The Object Constraint Language

Collection types 15

- Each operation on collections can be specified by using the iterate operation.
In our case, the last specification is equivalent to:

```
context Customer
def clientProducts_nsctI:Sequence(TupleType(date:String,
id:Integer, ordProds:Sequence(Product))) =
self.orders->iterate(o:Order; acc: Sequence(TupleType(date:String,
id:Integer, ordProds: Sequence(Product))) =
oclEmpty(Sequence(TupleType(date: String, id:Integer,
ordProds:Sequence(Product)))) | acc->including(Tuple{date=o.date,
id=o.id, ordProds = o.products }))
```

- This returns the same result. In case of using the iterate operation, the OCL specification is more complex than its equivalent using collectNested. This is because the iterate operation is the most generic operation on collections.

The Object Constraint Language

Collection types 15 - other operations on collections

- ◆ *isEmpty*: **true** if collection has no elements
- ◆ *notEmpty()*: **true** if collection has at least one element
- ◆ *size*: number of elements in collection
- ◆ *count(elem)*: number of occurrences of elem in collection
- ◆ *includes(elem)*: **true** if elem is in collection
- ◆ *including(elem)*: **returns** a collection a similar collection including elem
- ◆ *excludes(elem)*: **true** if elem is not in collection
- ◆ *includesAll(coll)*: **true** if all elements of coll are in collection

Testing-Oriented Improvements of OCL Specification Patterns

Dan CHIOREAN¹, Vladila PETRASCU¹, Ileana OBER²

¹**Babeş-Bolyai University, Cluj-Napoca**

²**Paul Sabatier University, Toulouse**

AQTR 2010 - Cluj-Napoca – 30 May

Presentation Overview

- Introduction
- Background & Related Work
- Our Approach:
 - Proposed OCL specification patterns
 - The case of invariants
 - The case of pre/post conditions
 - Tool support & Validation
- Conclusions and future work
- References

Introduction

- Detailed and unequivocal model specifications are a prerequisite for attaining the automated software development goal as promoted by the Model Driven Engineering paradigm.
- Obtaining as much information as possible, about causes of constraint failure, goes beyond testing requirements. It is intimately related to the reasons of using constraints in modeling and, more general, in software specification.
- The four main advantages of using the Design by Contract technique [1] are:
 1. help in writing correct software,
 2. documentation aid,
 3. support for testing, debugging and quality assurance,
 4. support for software fault tolerance.

Testing-Oriented OCL Specification Patterns

Background & Related Work

- In software modeling, *constraint patterns* [3] may be used to capture frequently occurring restrictions imposed on models. This paper is focused on describing and specifying such constraint patterns.
- We will clearly distinguish among the concepts of *constraint pattern* and that of *OCL specification pattern*, although the phrases appear to be used interchangeably in the literature ([6], [2]).
 - a *constraint pattern* denotes a logical constraint/restriction on a model
 - an *OCL specification pattern* refers to a proposed way of specifying constraints
- We describe solutions to the *constraint patterns* of interest as *OCL specification patterns*
- The OCL specifications found in the literature, including those for constraint patterns, are only focused on expressions' clearness. During testing and debugging however, just knowing that a system state is inconsistent or that a method pre/postcondition is not fulfilled is not enough

Testing-Oriented OCL Specification Patterns

Background & Related Work - 2

- Identifying the exact failure reasons is of utmost importance for error correction. This is the core-idea of the OCL specification approach that we promote. In this respect, in the following **we give improvements of existing OCL specification patterns, considering both the case of invariants and that of pre/post-conditions**

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants

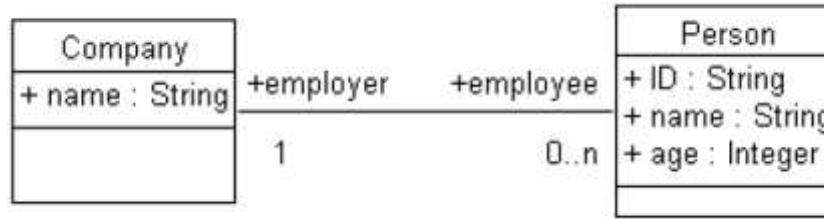


Figure 1 – A simple class model

- “All employees of a company should be aged at most 65”. We will refer to this particular constraint as *Complying with Retirement Age Limit (CRAL)*

- **Existent specifications:**

```
context Company
inv CRAL_E:
    self.employee->forAll( e | e.age <= 65)
```

- **Drawback:** we have no useful hint regarding the identity of those employees which are over the age limit!

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants - 2

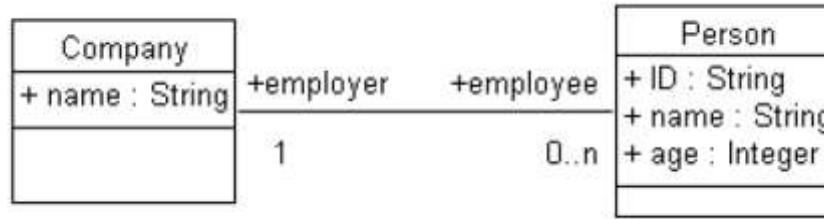


Figure 1 – A simple class model

- “All employees of a company should be aged at most 65”. We will refer to this particular constraint as *Complying with Retirement Age Limit (CRAL)*
- **Our proposal** is convenient not only when modeling, but also at runtime:

```
context Company
inv CRAL_P:
    self.employee->reject( e | e.age <= 65)->isEmpty()
```

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants - 3

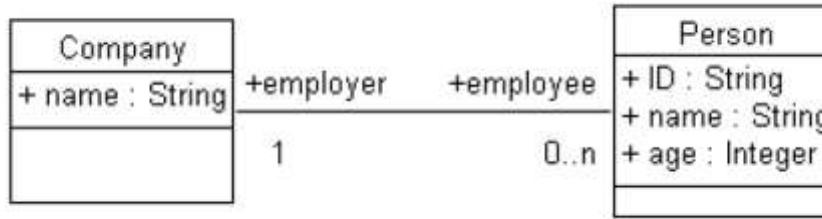


Figure 1 – A simple class model

- Two equivalent OCL specification patterns for the forAll modeling constraint:

```
pattern ForAll_Reject(objects: Collection(Object), properties: Set(Constraint))=
    objects->reject(y | oclAND(properties, y))->isEmpty()
```

- The CRAL_P specification is an instantiation of the ForAll_Reject specification pattern:

```
context Company
inv CRAL_P:
    ForAll_Reject(self.employee, Set{AttributeValueRestriction(age, <=, 65)})
```

```
pattern ForAll_Select(objects: Collection(Object), properties: Set(Constraint))=
    objects->select(y | not oclAND(properties, y))->isEmpty()
```

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants - 4

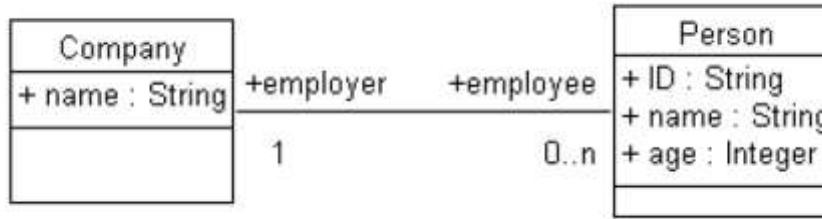


Figure 1 – A simple class model

- **“Global” uniqueness case (GUID):** Let us consider an application whose model contains a Person class having the same attributes as the one in Figure 1. Almost all the OCL specification proposals existent in the literature have one of the following shapes:

```
context Person
inv GUID_E1:
    Person.allInstances()->forAll(p, q| p<>q implies p.ID <> q.ID)
```

```
context Person
inv GUID_E2:
    Person.allInstances()->isUnique(ID)
```

- **Drawbacks:** both GUID_E1 and GUID_E2 break the semantics of invariants. Both specifications do not provide appropriate debugging support!

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants - 5

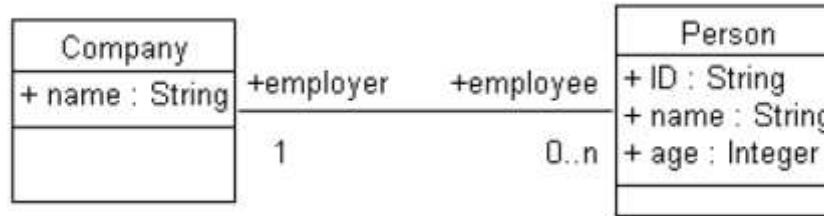


Figure 1 – A simple class model

```
context Person
```

```
inv GUID_P:  
    Person.allInstances() -> select(p | p.ID = self.ID) -> size() = 1
```

- The pattern corresponding to this specification is:

```
pattern GloballyUniqueIdentifier(class: Class, attribute: Property)=  
    class.allInstances() -> select(i | i.attribute = self.attribute) -> size() = 1
```

- **“Container-relative” uniqueness case (CUID):** in the context of Figure 1, supposes a constraint requiring that employees of a company should be uniquely identified by their IDs:

```
context Company
```

```
inv CUID_P1:  
    self.employee -> reject(e | self.employee.ID->count(e.ID)=1) -> isEmpty()
```

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of invariants - 6

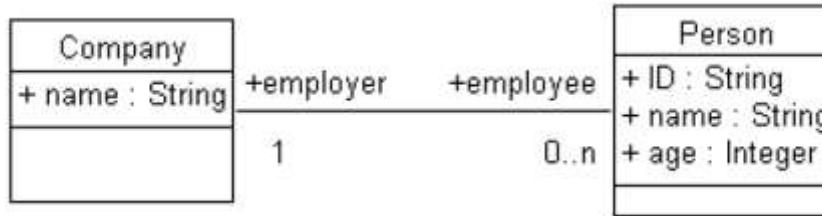


Figure 1 – A simple class model

- A more efficient specification:

```
context Company
inv CUID_P2:
let allIDs:Bag(String) = self.employee.ID in
self.employee->reject(e | allIDs->count(e.ID)=1)->isEmpty()
```

- with the corresponding pattern:

```
pattern ContainerRelativeUniqueIdentifier(class:Class, navigation:Property,
attribute:Property)=
let bag:Bag(OclAny) = self.navigation.attribute in
ForAll_Reject(navigation, UniqueOccurrenceInBag(bag, attribute))
```

- where, UniqueOccurrenceInBag, shortly UOB is a new atomic pattern requiring that “A given element has exactly one occurrence in a given bag of elements of the same type”

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of pre/post conditions

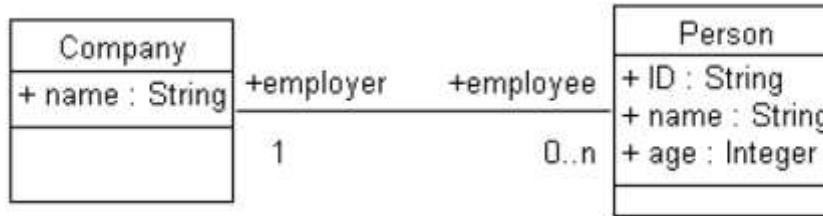


Figure 1 – A simple class model

- Using pre/post-conditions represents the correct attitude to adopt from a dynamic perspective "it is better to prevent than to cure". Breaking the uniqueness constraint imposed on IDs of employees can be prevented by means of appropriate pre/post-condition pairs for the modifiers that may violate this constraint. The modifiers concern adding an employee to a company and setting the name of an employee, respectively.

```
context Company::addEmployee(p:Person)

pre CUID_preAdd:
    self.employee->reject(e | e.ID <> p.ID)->isEmpty()

post CUID_postAdd:
    self.employee = self.employee@pre->including(p)
```

Testing-Oriented OCL Specification Patterns

Our Approach - Proposed OCL specification patterns - The case of pre/post conditions - 2

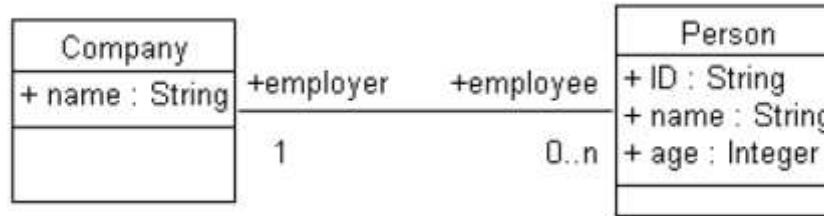


Figure 1 – A simple class model

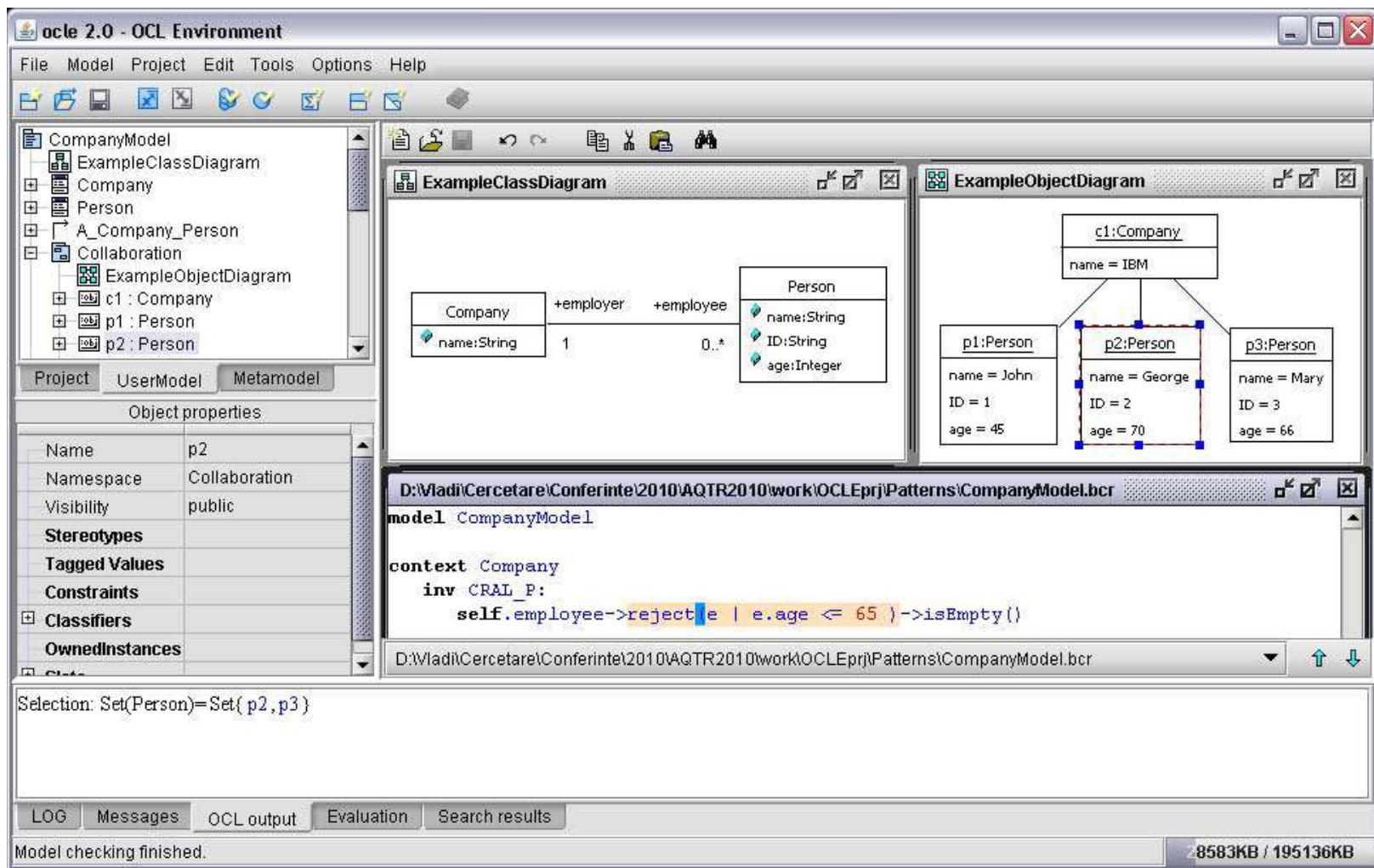
```
context Person::setId(id: String)

pre CUID_preSet:
    self.employer.employee->reject(e | e.ID <> id)->isEmpty()

post CUID_postSet:
    self.ID = id
```

Testing-Oriented OCL Specification Patterns

Our Approach - Tool support & Validation



Testing-Oriented OCL Specification Patterns

Conclusions and future work

- Related to the best known approaches in the field [6], [7], [3], [2], our contribution consists of:
 1. proposal of a pair of “efficient testing-oriented” OCL specification patterns for the *For All* constraint pattern;
 2. a deeper analysis of the *Unique Identifier* constraint pattern, with respect to the particular type of uniqueness imposed (global vs. container-relative);
 3. Proposal of correct/appropriate OCL specifications patterns for each uniqueness context;
 4. approaching the constraint patterns’ problem from both a static and a dynamic perspective, with appropriate specification patterns for each case;
 5. Validation of our proposed approach using appropriate tool-support.
- To our knowledge, this is the only approach to constraint patterns’ specification aimed at maximizing the amount of relevant testing/debugging related information

Testing-Oriented OCL Specification Patterns

Conclusions and future work - 2

- Future work targets at:
 1. identifying new constraint and OCL specification patterns, along with improving some of the existing ones;
 2. automating the instantiation of proposed patterns by means of an appropriate tool;
 3. developing an automated test-data generator;
 4. a detailed study on run-time exception handling.
- This work was supported by the research project ID 2049, sponsored by the Romanian National University Research Council (CNCSIS)

Testing-Oriented OCL Specification Patterns

References

- [1] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.
- [2] M. Wahler, “Using Patterns to Develop Consistent Design Constraints,” Ph.D. dissertation, ETH Zurich, Switzerland, 2008. [Online]. Available: <http://e-collection.ethbib.ethz.ch/eserv/eth:30499/eth-30499-02.pdf>
- [3] M. Wahler, J. Koehler, and A. D. Brucker, “Model-Driven Constraint Engineering,” *Electronic Communications of the EASST*, vol. 5, ISSN 1863-2122, 2006.
- [4] OMG, “Unified Modeling Language: Superstructure Version 2.2,” 2009. [Online]. Available: <http://www.omg.org/docs/formal/09-02-02.pdf>
- [5] ——, “OCL Specification v2.0,” 2006. [Online]. Available: <http://www.omg.org/spec/OCL/2.0/PDF>
- [6] J. Ackermann, “Frequently Occurring Patterns in Behavioral Specification of Software Components,” in *COEA*, 2005, pp. 41–56.
- [7] ——, “Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components,” in *Proceedings of the MoDELS’*

Testing-Oriented OCL Specification Patterns

References - 2

- [7] ——, “Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components,” in *Proceedings of the MoDELS’ 05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, ser. *Technical Report LGL-REPORT-2005-001*, T. Baar, Ed. EPFL, 2005, pp. 15–29.
- [8] E. Miliauskaite and L. Nemuraite, “Representation of Integrity Constraints in Conceptual Models,” *Information Technology and Control*, vol. 34, no. 4, pp. 355–365, 2005.
- [9] D. Costal, C. G’omez, A. Queralt, R. Ravent’os, and E. Teniente, “Facilitating the Definition of General Constraints in UML,” in *MoDELS*, 2006, pp. 260–274.
- [10] LCI, “Object Constraint Language Environment (OCLE),” 2005. [Online]. Available: <http://lci.cs.ubbcluj.ro/ocle/>

Avoiding OCL specification pitfalls

**Understanding Software Modeling by Using
Constraints**

Dan CHIOREAN

Understanding Software Modeling by Using Constraints

Motivation

- MDE technologies requires **a clear and complete model specification**
- Design by Contracts **supports** producing rigorous models
- Using this technique is not so widespread as expected
 - this state of facts was caused by different rationales
 - people are not yet convinced about the advantages that can be obtained
 - => they should be motivated about using constraints
 - this shape their mentality

Understanding Software Modeling by Using Constraints

- To prove:
 - the necessity of employing constraints, by means of relevant examples
- To presents:
 - best principles & valid practices for specifying constraints
- Our approach is focused on:
 - ✓ role of complete and unequivocal requirements,
 - ✓ importance of the rigor in specifying constraints,
 - ✓ specification process that supports the return in earlier phases, including requirements
- And highlights on the:
 - ✓ conformance between requirements & specifications,
 - ✓ importance of choosing a design model from different proposals,
 - ✓ need of testing specifications by using snapshot

Understanding Software Modeling by Using Constraints

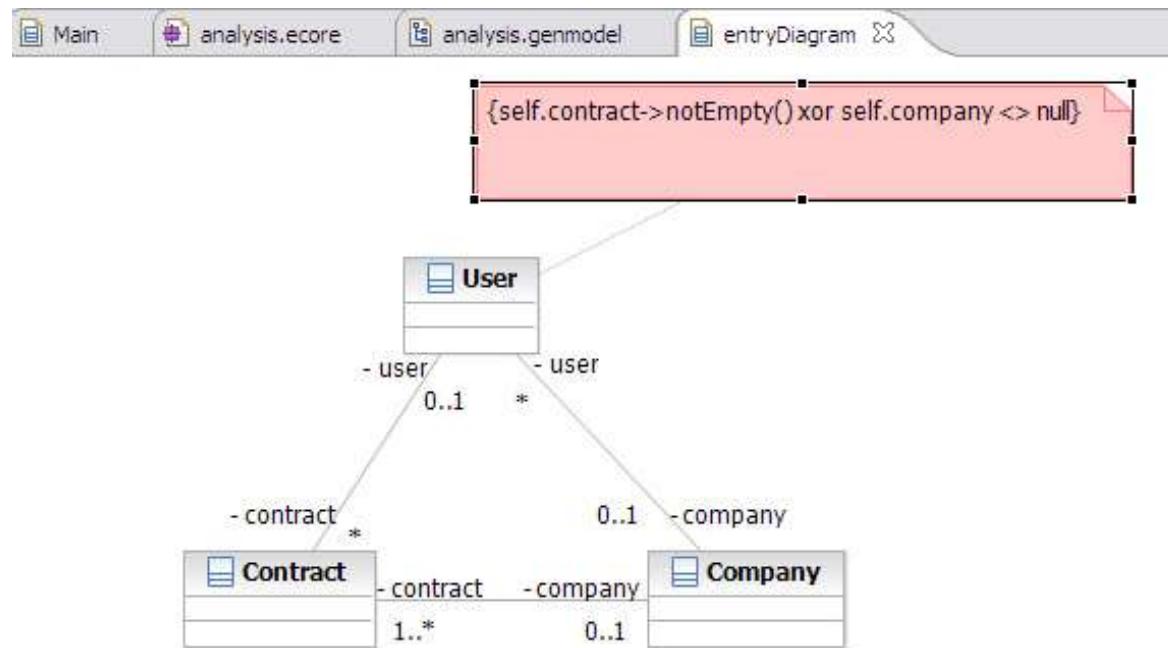
OCL specifications – state of facts

- many books, articles and slides about OCL: Warmer, Gogolla, Hussmann, Demuth, Atkinson, etc
 - many OCL examples (including the UML static semantics specification) are hasty,
 - avoiding hasty specification through some best practice & principles
- Aspects related to OCL specs:
- ✓ usefulness of information when constraints are broken
 - ✓ intelligibility and suggestiveness of specification
 - ✓ use of constraint specification patterns
 - ✓ conformity between evaluation of constraints specified in:
 - a) OCL and
 - b) programming languages (generated from specs)
 - ✓ independence of the results obtained from the OCL tools used

Testing Software Modeling by Using Constraints

Understanding the problem and the problem domain

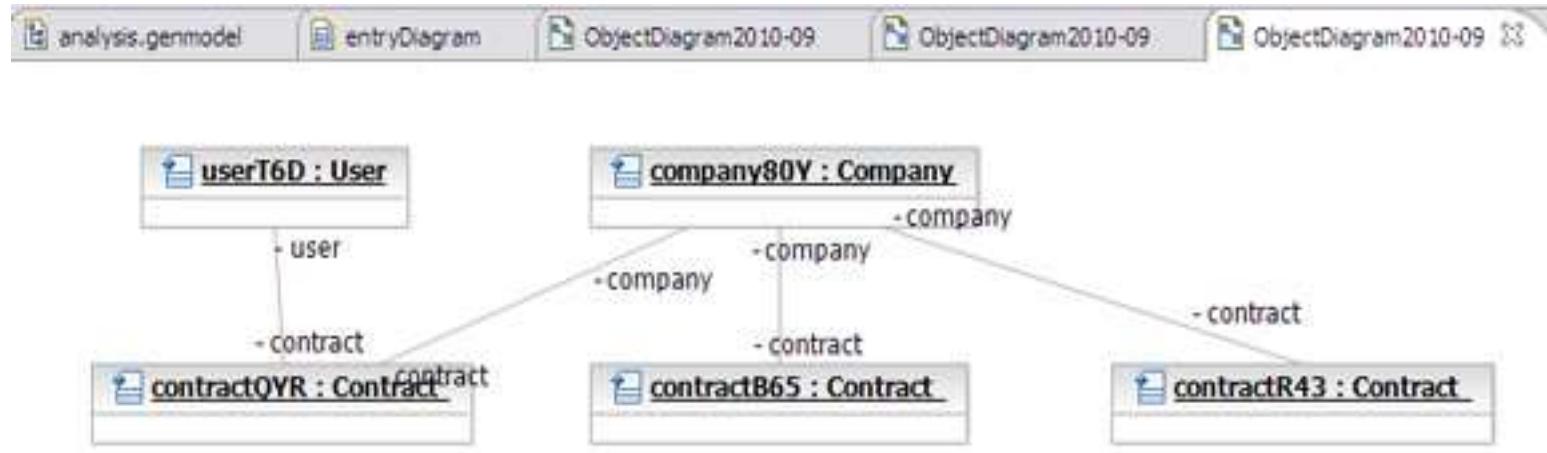
- “... the library offers a subscription to each person employed in an associated company. In this case, the employee does not have a contract with the library but with the society he works for, instead.”
- <https://www.ibm.com/developerworks/rational/library/accurate-domain-models-using-ocl-constraints-rational-software-architect/>



Understanding Software Modeling by Using Constraints

Understanding and improving the requirements

- snapshots must clarify requirements and the model, but ... "**one swallow does not make spring**"



```
context User
inv TodConstraint:
  self.contract->notEmpty() xor self.company <> null
```

- This model must does not conform to the informally described requirements, even before attaching constraints.

Understanding Software Modeling by Using Constraints

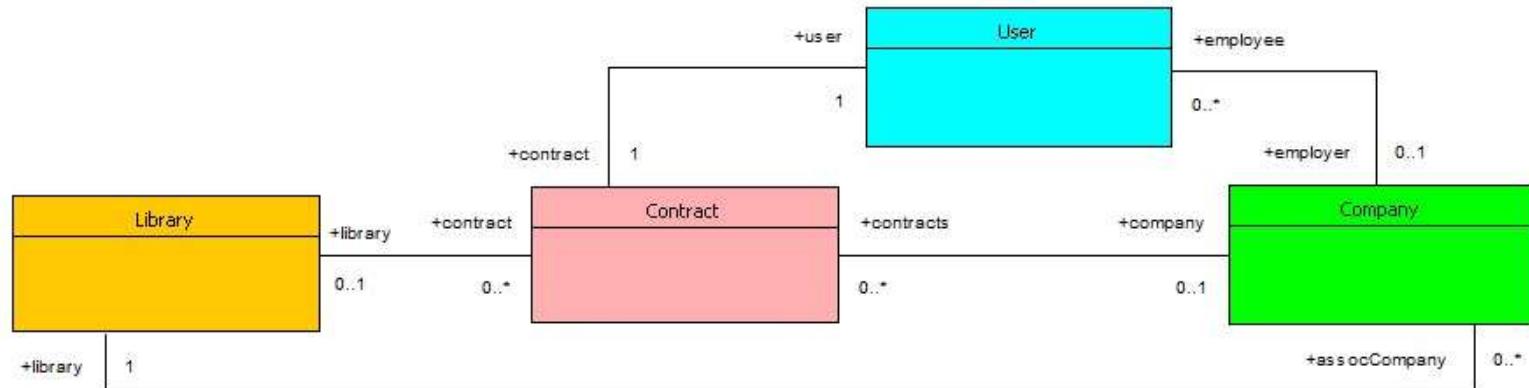
Understanding and improving the requirements_2

- The constraints specification process must ask for additional information, meant to support an improved description of requirements, a deeper understanding of the problem, and by consequence, a clear model specification.
- The definition of a contract, as taken from
<http://www.investorwords.com/>:
 - "A binding agreement between two or more parties for performing, or refraining from performing, some specified act(s) in exchange for lawful consideration. "
 - in our case, the parts in the contract are:
 - **the user** on the one hand,
 - and **the library or the company**, on the other hand.

Understanding Software Modeling by Using Constraints

Understanding and improving the requirements_3

- Improving the solution proposed by choosing an appropriate context & updating constraints,



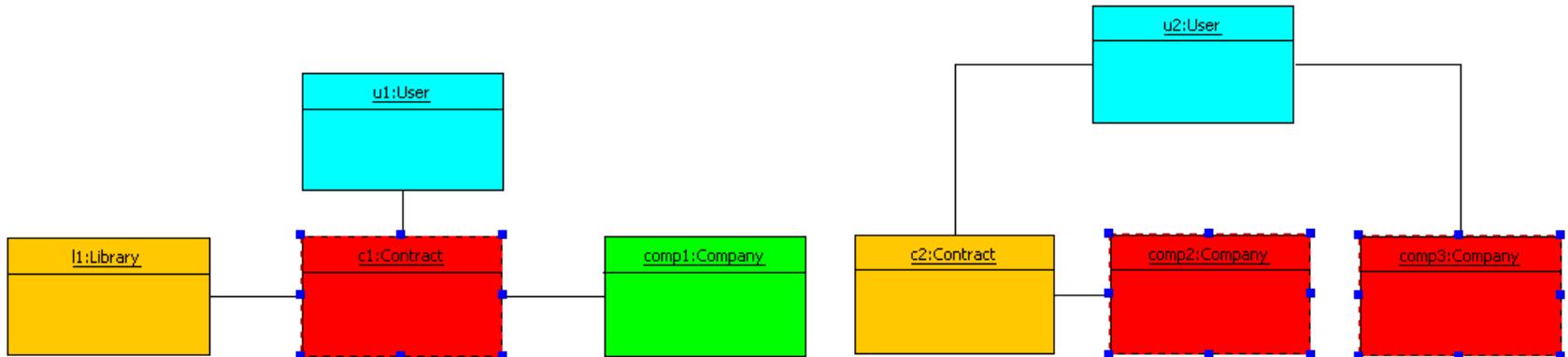
context User

```
inv contracts:  
    if self.contract.library->notEmpty  
        then self.employer->isEmpty and  
            self.contract.company->isEmpty  
        else self.employer = self.contract.company and  
            self.employer.library->notEmpty  
    endif
```

Understanding Software Modeling by Using Constraints

Using suggestive snapshots to test specifications

- Unwanted model instantiations that are not cached by the invariant proposed in [4]

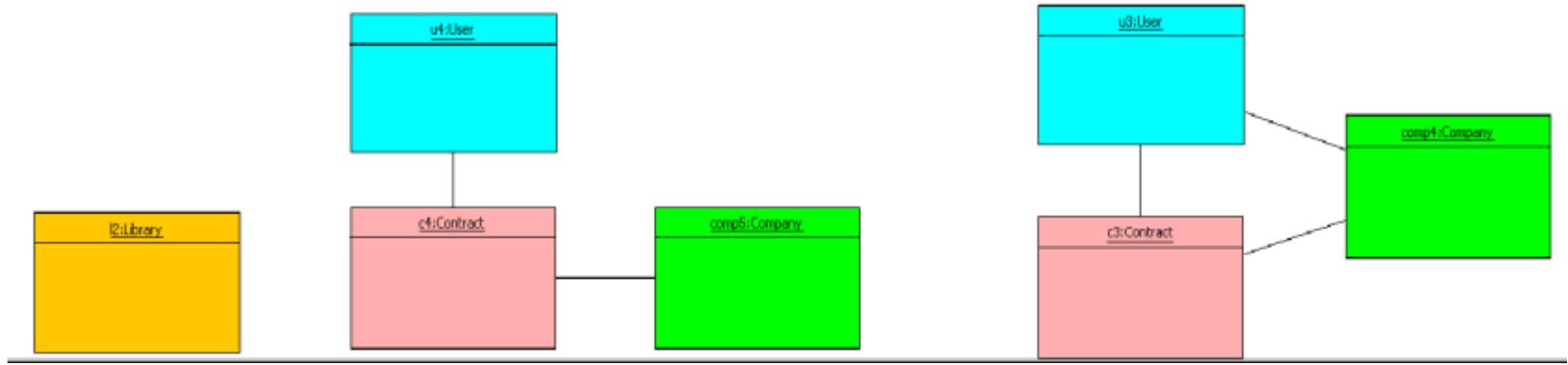


```
context User
inv contracts:
  if self.contract.library->notEmpty
    then self.employer->isEmpty and
        self.contract.company->isEmpty
    else self.employer = self.contract.company and
        self.employer.library->notEmpty
  endif
```

Understanding Software Modeling by Using Constraints

Using suggestive snapshots to test specifications

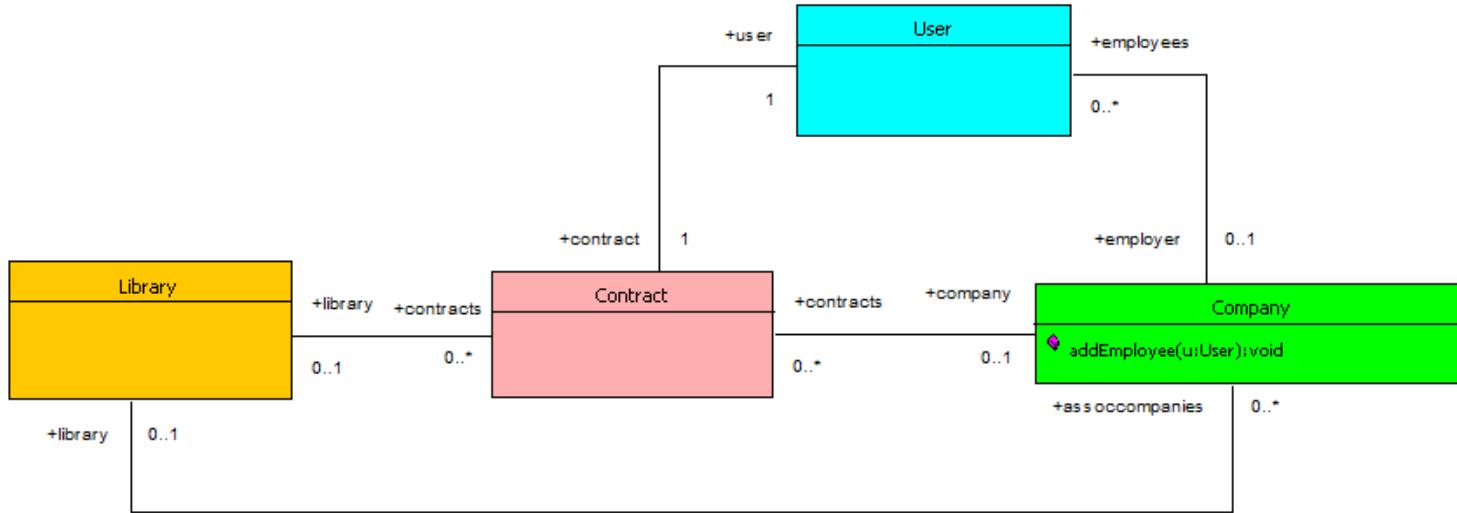
- Unwanted model instantiations



```
context User
  inv contracts:
    if self.contract.library->notEmpty
      then self.employer->isEmpty and
          self.contract.company->isEmpty
    else self.employer = self.contract.company and
          self.employer.library->notEmpty
    endif
```

Understanding Software Modeling by Using Constraints

It's better to prevent than cure



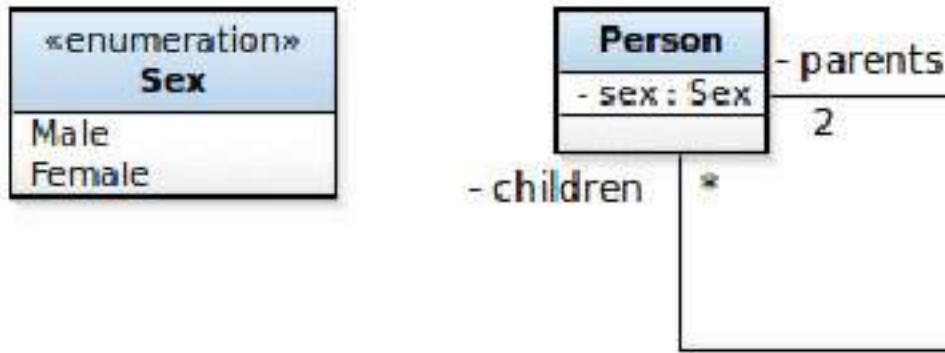
```
context Company::addEmployee(u:User):void
pre:
    self.employees->excludes(u)

post:
    self.employees@pre->including(u) = self.employees and
    u.contract.company = self and
    self.contracts.library->includes(u.contract.library@pre)
```

Understanding Software Modeling by Using Constraints

Understanding model semantics

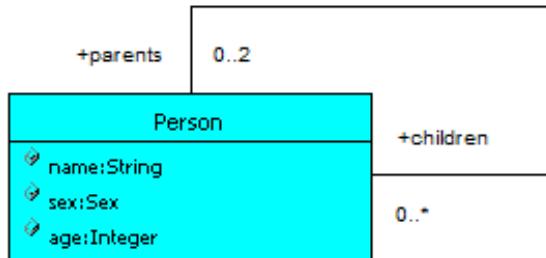
- in MDD technologies models are used to produce software



```
self.parents->asSequence () ->at (1) .sex <>  
self.parents->asSequence () ->at (2) .sex  
context Person  
inv parentsSex:  
    self.parents->size = 2 implies  
    self.parents->first.sex = Sex::female and  
    self.parents->last.sex = Sex::male
```

Understanding Software Modeling by Using Constraints

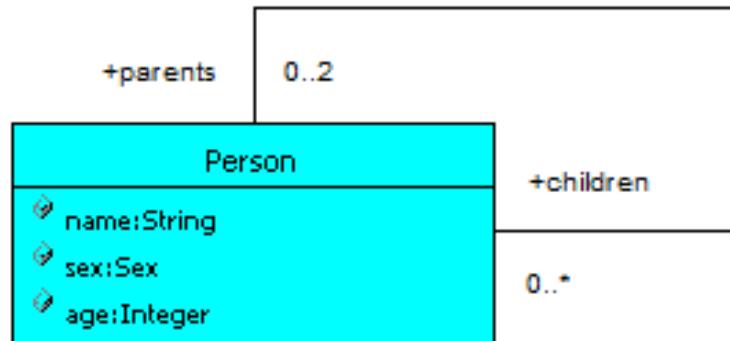
Understanding model semantics_2



```
context Person
    inv notSelfParent:
        self.parents->select(p | p = self)->isEmpty
context Person
    inv parentsAge:
        self.parents->reject(p|p.age - self.age >= 16)->isEmpty
context Person::addChildren(p:Person)
    pre childrenAge:
        self.children->excludes(p) and self.age - p.age >= 16
    post chidrenAge:
        self.children->includes(p)
```

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

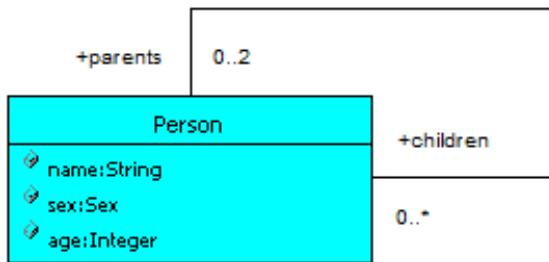


```
context Person
    def allAncestors() :Sequence(Person) =
        self.parents->union(self.parents.allAncestors())
```

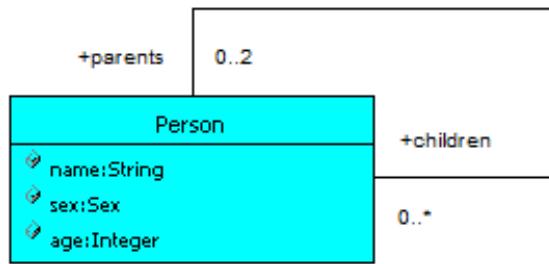
```
context Person
    def allAncestors() :Sequence(Person) =
        (Sequence{self}->closure(p | p.parents))->asSequence
```

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications



- The above specifications, although apparently correct, encloses a few pitfalls:
 1. If one of the parents is left **unspecified**, the model will comply with the WFRs, but the constraint evaluation will end up in an exception when trying to access the missing parent (due to the at(2) call).
 2. In case there are valid references to both parents, but the sex of one of them is not specified, the value of the corresponding subexpression is undefined and the whole expression reduces to either **Sex::Male <> undefined** or **Sex::Female <> undefined**. This later expressions provide tool-dependent evaluation results (true in case of USE [1] or Dresden OCL [2] and undefined in case of OCLE [3]).



Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

- The above specifications, although apparently correct, encloses a few pitfalls:
 3. In case one of the parents' sex is undefined, the code generated for the above constraint will provide evaluation results which depend on the position of the undefined value with respect to the comparison operator. According to the tests we have performed, the Java code generated by OCLE and Dresden OCL throws a `NullPointerException` when the undefined value is located at the left of the `<>` operator, while evaluating to true in case the undefined value is at the right of the operator and the reference at the left is a valid one. As we are in the context of the MDE paradigm, which relies extensively on automatic code generation, the results provided by the execution of the code corresponding to constraints is an aspect that must be considered when judging the quality of the formal constraints in question.
 4. The OCL expression would have been simpler (not needing an `asSequence()` call), in case an ordering relation on parents had been

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

- When any of the parents' sex is **undefined**, the invariant below evaluates to **false** in Dresden OCL and to **undefined** in OCLE. In similar circumstances, both Java code snippets generated for this invariant by the two tools return false when executed. Therefore, this invariant shape overcomes the drawbacks of the one from [4] previously pointed at items 1, 3 and 4. The triggering of a NullPointerException by the generated code in case of absence of one of the parents' sex has been avoided by placing the defined values (the **Sex::Female** and **Sex::Male** literals) on the left-hand side of equalities.

```
context Person
inv parentsSexP1:
  self.parents->size() = 2 implies
    Sex::Female = self.parents->first().sex and
    Sex::Male = self.parents->last().sex
```

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

- The solution to the problem mentioned at item 2 above comes from obeying to the separation of concerns principle. In order to avoid comparisons involving **undefined** values, whose results may vary with the OCL-supporting tool used, the equality tests of the parents' sex with the corresponding enumeration literals should be conditioned by both of them being specified. Such a solution is illustrated by means of the invariant proposal below.

```
context Person
inv parentsSexP2:
  self.parents->size() = 2 implies
    ( let mother = self.parents->first() in
      let father = self.parents->last() in
        if (not mother.sex.oclIsUndefined() and not father.sex.oclIsUndefined())
        then mother.sex = Sex::Female and father.sex = Sex::Male
        else false
      endif
    )
  )
```

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

- The lack of OCL specifications prohibiting undesired model instances (such as parents having the same sex, self-parentship or the lack of a minimum age difference among parents and children) seriously compromises model's integrity. The first prerequisite for models to reach their purpose is to have a complete and correct specification of requirements, and to deeply understand them.
- An incomplete specification reveals its limits when trying to answer questions on various situations that may arise.
- Specifying and evaluating OCL constraints should enable us to identify and eliminate bugs, by correcting the requirements and the OCL specifications themselves. Moreover, in the context of MDE, care should be taken to the shape of constraint specifications, ensuring that their evaluation using OCL-supporting tools provides identical results to those obtained by executing their equivalent code generated by those tools. Another conclusion, as important, is that the model proposed in the analyzed paper does not fully meet the needs of the addressed problem⁴ and we are therefore invited to seek for a better solution.

Understanding Software Modeling by Using Constraints

Possible pitfalls hidden in OCL specifications

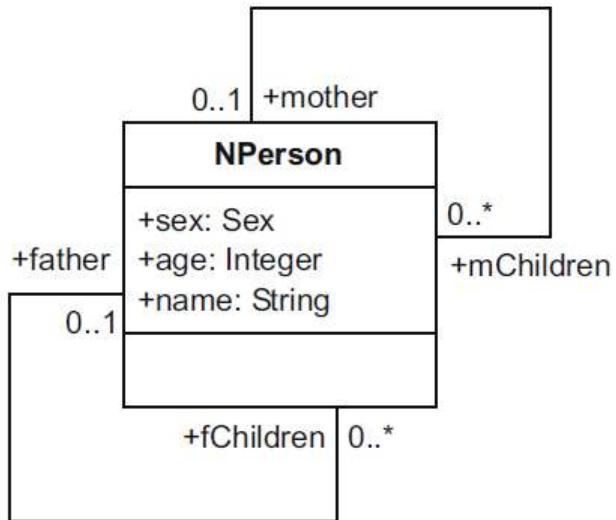
```
context Person
inv notSelfParent:
  self.parents->select(p | p = self)->isEmpty()
```

```
context Person
inv parentsAge:
  self.parents->reject(p | p.age - self.age >= 16)->isEmpty()
```

```
context Person::addChild(p:Person)
pre childrenAge:
  self.children->excludes(p) and self.age - p.age >= 16
post chidrenAge:
  self.children->includes(p)
```

Understanding Software Modeling by Using Constraints

Modeling alternatives

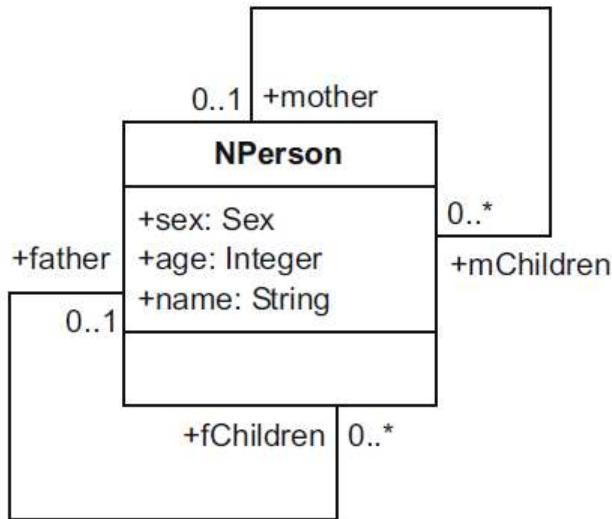


An equivalent model, but more adequate to the specification of the required constraints. the parents' roles are explicitly specified, with no extra memory required. When at least one parent's sex is **undefined**, the evaluation of this invariant returns **undefined** in OCLE and **false** in Dresden OCL, while the execution of the corresponding Java code outputs false in both cases. Given the invariant shape, the identification of the person breaking it is quite straightforward, therefore the problem can be rapidly fixed.

```
context NPerson
inv parentsSexP1:
  (self.mother->size() = 1 implies Sex::Female = self.mother.sex) and
  (self.father->size() = 1 implies Sex::Male = self.father.sex)
```

Understanding Software Modeling by Using Constraints

Modeling alternatives

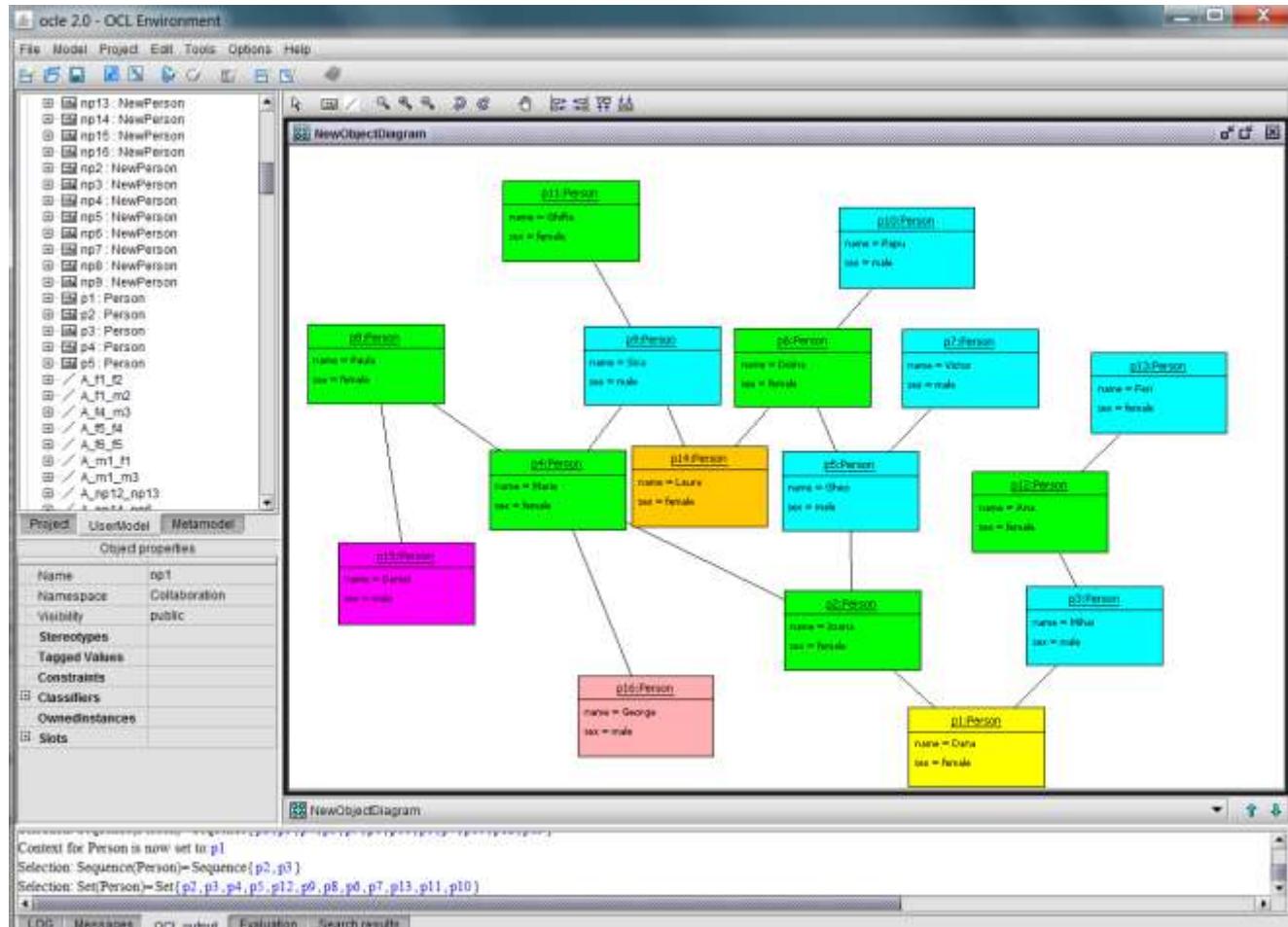


An alternative invariant shape, providing the same evaluation result in both OCLE and Dresden OCL, for both OCL and Java code is the one below.

```
context NPerson
inv parentsSexP2:
  (self.mother->size() = 1 implies
    (let ms:Sex = self.mother.sex in
      if not ms.oclisUndefined() then ms = Sex::Female
      else false endif )
  ) and
  (self.father->size() = 1 implies
    (let fs:Sex = self.father.sex in
      if not fs.oclisUndefined() then fs = Sex::Male
      else false endif)
  )
```

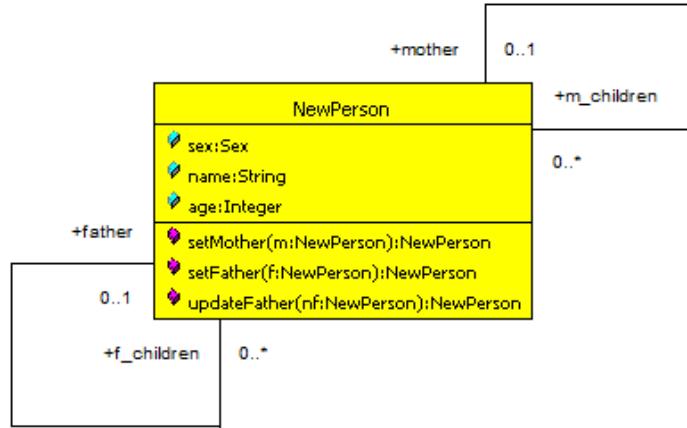
Understanding Software Modeling by Using Constraints

Querying the initial model



Understanding Software Modeling by Using Constraints

Explaining the Intended Model Uses_2

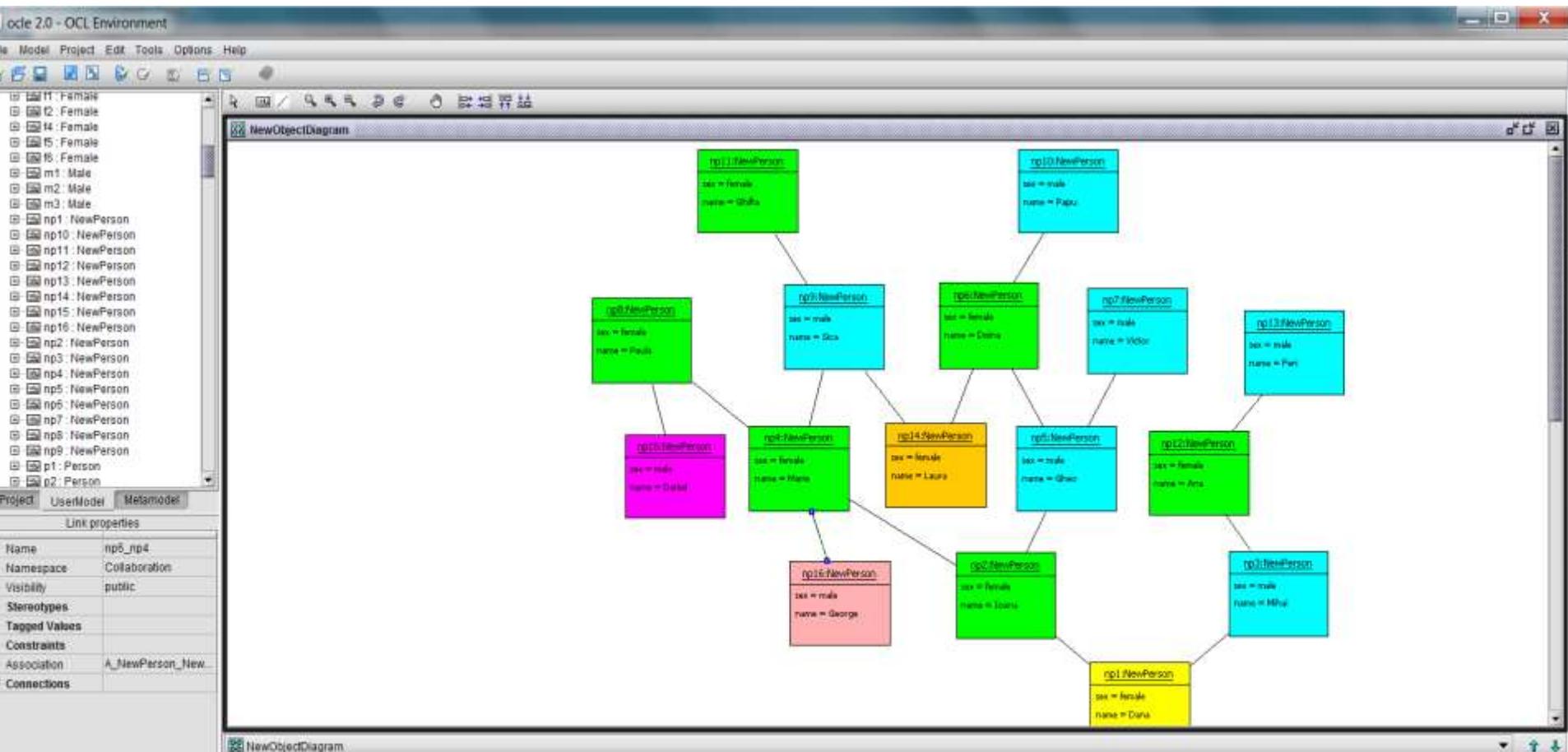


context NewPerson

```
def parents:TupleType(mother:Nperson, father:NPerson) =  
    Tuple{mother = self.mother, father = self.father}  
  
def allAncestors:Sequence(TupleType(mother:Nperson,  
    father:NPerson)) = Sequence{self.parents}->closure(i |  
    i.mother.parents, i.father.parents))->asSequence->  
    prepend(self.parents)
```

Understanding Software Modeling by Using Constraints

The result obtained when querying the NewPerson tree



```
context for NewPerson is now set to np1
selection: Tuple(mother:NewPerson,father:NewPerson)=Tuple { sp2,np3 }
selection: Sequence(Tuple(mother:NewPerson,father:NewPerson))=Sequence(Tuple { np2,np3 }, Tuple { np4,np5 }, Tuple { np12,Undefined }, Tuple { np8,np9 }, Tuple { np6,np7 }, Tuple { Undefined,np13 }, Undefined, Tuple { Undefined,Undefined }, Tuple { np11,Undefined }, Tuple { Undefined,np10 })
context for NewPerson is now set to np16
```

Understanding Software Modeling by Using Constraints

Conclusion

- Complementing model specification with constraints is crucial. This supports:
 - a rigorous model description and
 - detection of unwanted model instantiation
- Conformance with requirements is the first criterion quality criterion in constraint specification
 - This requires a thorough understanding of the problem and problem domain
- Abstraction is the first principle that have to be applied when modeling because identify the main concepts to use
- Identifying and specifying constraints follow abstraction because restricts on relationships between concepts and on their values

Understanding Software Modeling by Using Constraints

Conclusion_2

- The main attitude in specifying constraints is the rigor
- In our opinion, hastiness is the main enemy in the constraint specification process
- The quality of constraint specification includes also different other criteria
- Requirements must include the description of the intended usage of the system
- The constraints' role is to support a better quality of the model
- When judging the quality of the model, all descriptions including constraints must be considered
- Choosing the most appropriate model, supposes analyzing many proposals

Understanding Software Modeling by Using Constraints

References

- [1] A UML-based Specification Environment,
<http://www.db.informatik.uni-bremen.de/projects/USE>
- [2] Software Technology Group at Technische Universität Dresden:
Dresden OCL, <http://www.dresden-ocl.org/index.php/DresdenOCL>
- [3] LCI (Laboratorul de Cercetare în Informatică): Object Constraint
Language Environment (OCLE), <http://lci.cs.ubbcluj.ro/ocle/>
- [4] Todorova, A.: Produce more accurate domain models by using OCL
constraints (2011),
<https://www.ibm.com/developerworks/rational/library/accuratedomain-models-using-ocl-constraints-rational-software-architect/>

Object Design – Specifying Interfaces



Dan CHIOREAN
mainly based on chapter 9 of
Bruegge's book

An Overview of Interface Specification

At this point, we have made many decisions about the system and produced a wealth of models:

- The *analysis object model* describes the entity, boundary, and control objects that are visible to the user. AOM includes attributes and operations for each object.
- *Subsystem decomposition* describes how these objects are partitioned into cohesive pieces realized by different teams of developers. Each subsystem includes high-level service descriptions that indicate which functionality it provides to the others.
- *Hardware/software mapping* identifies the components that make up the virtual machine on which we build solution objects. This may include classes and APIs defined by existing components.

An Overview of Interface Specification_2

At this point, we have made many decisions about the system and produced a wealth of models:

- *Boundary use cases* describe, from the user's point of view, administrative and exceptional cases that the system handles.
- *Design patterns* selected during object design reuse describe partial object design models addressing specific design issues.

An Overview of Interface Specification_3

All these models, however, reflect only a partial view of the system. Many puzzle pieces are still missing, and many others are yet to be refined. The goal of object design is to produce an object design model that integrates all of the above information into a coherent and precise whole. Interface specification includes the following activities:

- *Identify missing attributes and operations.* During this activity, we examine each subsystem service and each analysis object. We identify missing operations and attributes that are needed to realize the subsystem service. We refine the current object design model and augment it with these operations.

An Overview of Interface Specification_4

- *Specify visibility and signatures.* During this activity, we decide which operations are available to other objects and subsystems, and which are used only within a subsystem. We also specify the return type of each operation as well as the number and type of its parameters. The goal of this activity is to reduce coupling among subsystems and provide a small and simple interface that can be understood easily by a single developer.
- *Specify contracts.* During this activity, we describe in terms of constraints the behavior of the operations provided by each object. For each operation, we describe the conditions that must be met before the operation is invoked and a specification of the result after the operation returns.

An Overview of Interface Specification_5

The large number of objects and developers, the high rate of change, and the concurrent number of decisions made during object design make object design much more complex than analysis or system design. This represents a management challenge, as many important decisions tend to be resolved independently and are not communicated to the rest of the project.

Object design requires much information to be made available among the developers so that decisions can be made consistent with decisions made by other developers and consistent with design goals.

Interface Specification Concepts

So far, we have treated all developers as equal. Now that we are delving into the details of object design and implementation, we need to differentiate developers based on their point of view. While all use the interface specification to communicate about the class of interest, they view the specifications from radically different point of views:

- The **class implementor** is responsible for realizing the class under consideration. Class implementors design the internal data structures and implement the code for each public operation. For them, the interface specification is a work assignment.

Interface Specification Concepts_2

- The **class user** invokes the operations provided by the class under consideration during the realization of another class, called the **client class**. For class users, the interface specification discloses the boundary of the class in terms of the services it provides and the assumptions it makes about the client class.
- The **class extender** develops specializations of the class under consideration. Like class implementors, class extenders may invoke operations provided by the class of interest, the class extenders focus on specialized versions of the same services.

Interface Specification Concepts_3

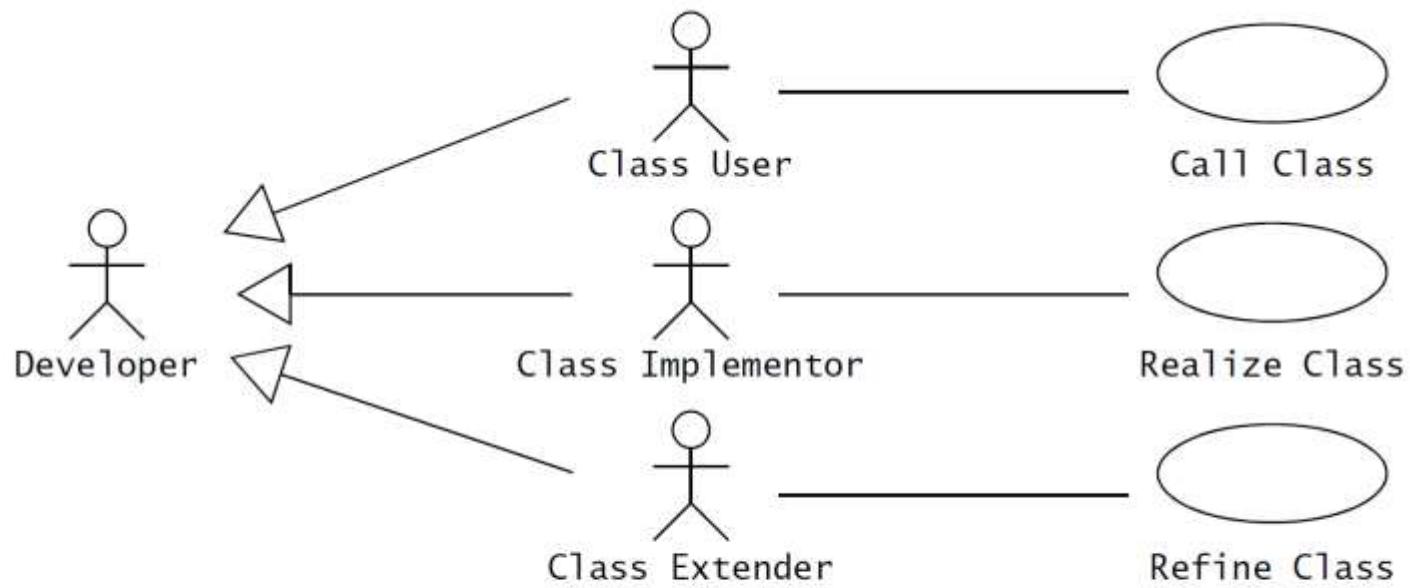


Figure 9-1 The Class Implementor, the Class Extender, and the Class User role (UML use case diagram).

Interface Specification Concepts_4

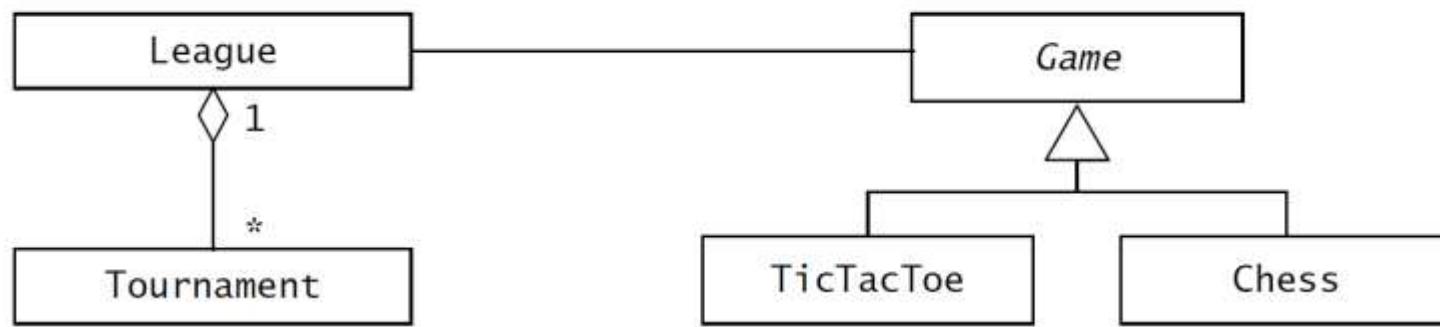


Figure 9-2 ARENA *Game* abstract class with user classes and extender classes.

Contracts: Invariants, Preconditions, and Postconditions

Contracts are constraints on a class that enable class users, implementors, and extenders to share the same assumptions about the class [Meyer, 1997]. A contract specifies constraints that the class user must meet before using the class as well as constraints that are ensured by the class implementor and the class extender when used:

- An **invariant** is a predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces. Invariants are used to specify consistency constraints among class attributes.

Contracts: Invariants, Preconditions, and Postconditions_2

- A **precondition** is a predicate that must be true before an operation is invoked. Preconditions are associated with a specific operation. Preconditions are used to specify constraints that a class user must meet before calling the operation.
- A **postcondition** is a predicate that must be true after an operation is invoked. Postconditions are associated with a specific operation. Postconditions are used to specify constraints that the class implementor and the class extender must ensure after the invocation of the operation.

Contracts: Invariants, Preconditions, and Postconditions_3

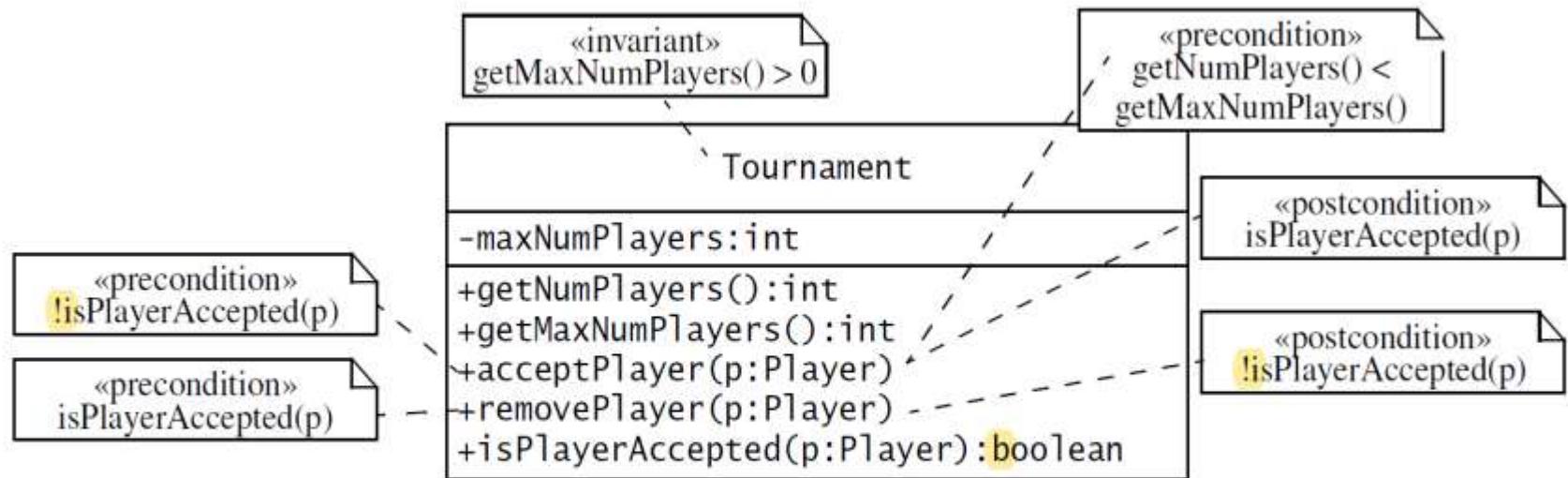


Figure 9-4 Examples of invariants, preconditions, and postconditions in OCL attached as notes to the UML model (UML class diagram).

Contracts: Invariants, Preconditions, and Postconditions_4

```
context Tournament::getNumPlayers():Integer  
    body: self.players->size()
```

```
context Tournament::getMaxNumPlayers():Integer  
    body: self.maxNumPlayers
```

```
context Tournament::removePlayer(p:Player) pre:  
    isPlayerAccepted(p)
```

```
context Tournament::removePlayer(p:Player) post:  
    !isPlayerAccepted(p)
```

```
context Tournament::removePlayer(p:Player) post:  
    getNumPlayers() = self@pre.getNumPlayers() - 1
```



Figure 9-6 Associations among League, Tournament, and Player classes in ARENA.

```
context Tournament::acceptPlayer(p:Player) pre:  
    getNumPlayers() < getMaxNumPlayers()
```

Contracts: Invariants, Preconditions, and Postconditions_5

```
context Tournament inv:  
    self.end - self.start <= 7
```

```
context League::getActivePlayers:Set post:  
    result = tournaments.players->asSet()
```

```
context Tournament inv:  
    matches->exists(m:Match | m.start.equals(start))
```

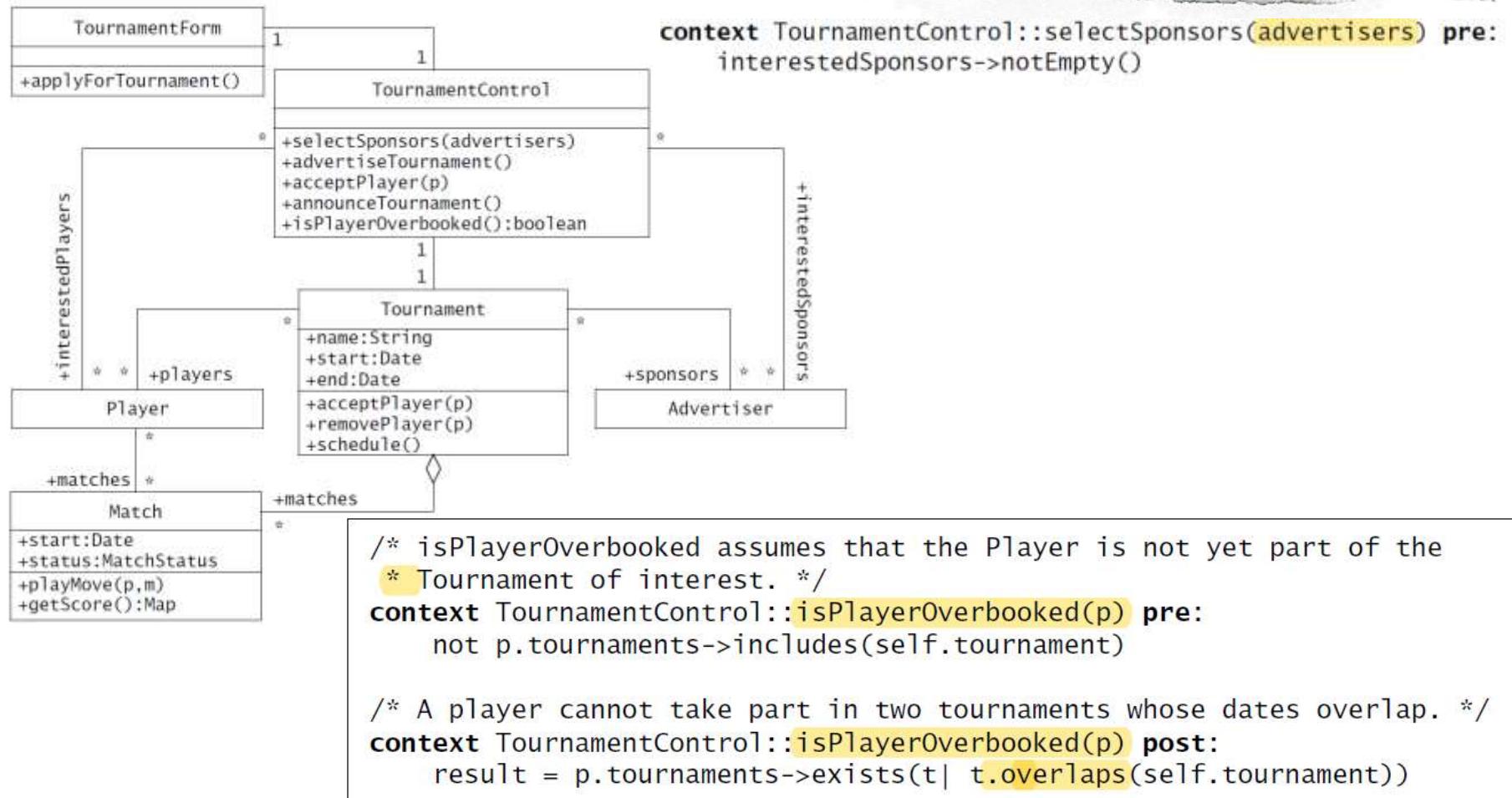


```
context Tournament inv:  
    matches->forAll(m:Match | m.start.after(start) and m.end.before(end))
```

layer classes in ARENA.

In OCL, **Date** is not a defined type – therefore **Date** needs to be defined at the model level. **Date** has 3 **Integer** attributes: **day**, **month** and **year** and a set of **associated constraints** and **operations**.

Contracts: Invariants, Preconditions, and Postconditions_6



Contracts: Invariants, Preconditions, and Postconditions_7

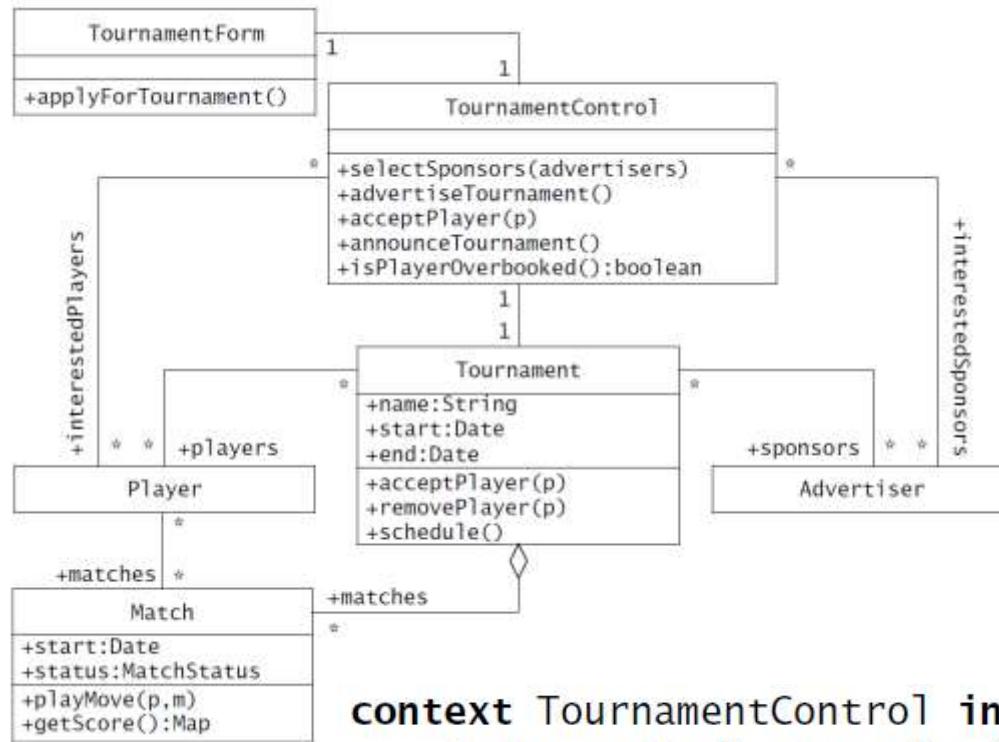
TournamentControl
+selectSponsors(advertisers) +advertiseTournament() +acceptPlayer(p) +announceTournament() +isPlayerOverbooked():boolean

```
/* Pre-and postconditions for ordering operations on TournamentControl */
context TournamentControl::selectSponsors(advertisers) pre:
    interestedSponsors->notEmpty() and
    tournament.sponsors->isEmpty()
context TournamentControl::selectSponsors(advertisers) post:
    tournament.sponsors.equals(advertisers)

context TournamentControl::advertiseTournament() pre:
    tournament.sponsors->isEmpty() and
    not tournament.advertised
context TournamentControl::advertiseTournament() post:
    tournament.advertised

context TournamentControl::acceptPlayer(p) pre:
    tournament.advertised and
    interestedPlayers->includes(p) and
    not isPlayerOverbooked(p)
context TournamentControl::acceptPlayer(p) post:
    tournament.players->includes(p)
```

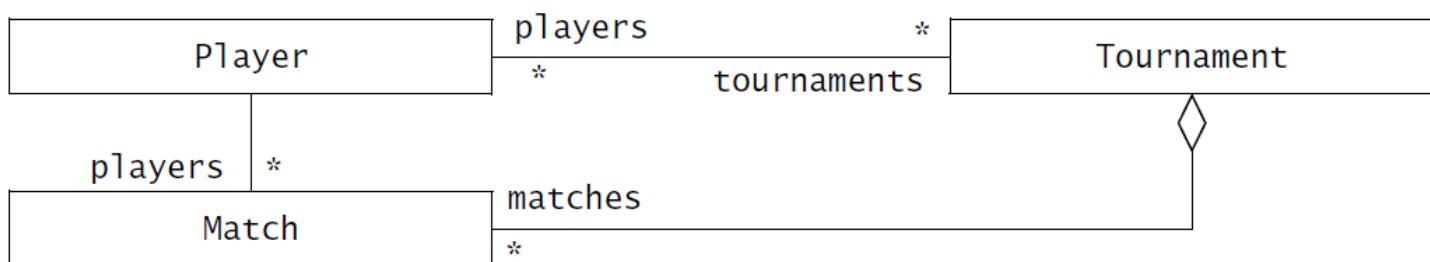
Contracts: Invariants, Preconditions, and Postconditions_8



context TournamentControl inv:

`tournament.players->forAll(p|
p.tournaments->forAll(t|
t <> tournament implies not t.overlap(tournament)))`

Contracts: Invariants, Preconditions, and Postconditions_9



```
/* A match can only involve players who are accepted in the tournament */
context Match inv:
    players->forAll(p|
        p.tournaments->exists(t|
            t.matches->includes(self)))
```

```
context Match inv:
    players.tournaments.matches.includes(self)
```

Inheriting Contracts

In a polymorphic language, a class can be substituted by any of its descendants. That is, a class user invoking operations on a class could be invoking instead a subclass. Hence, the class user expects that a contract that holds for the superclass still holds for the subclass. We call this **contract inheritance**.

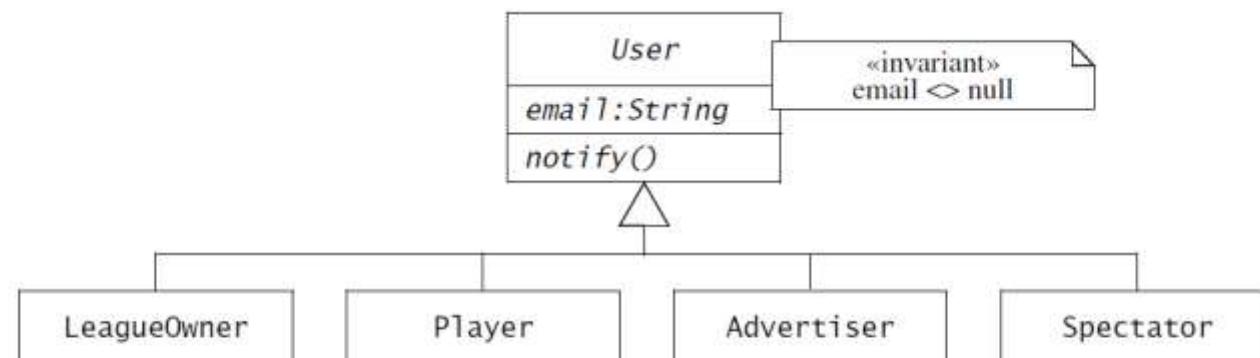
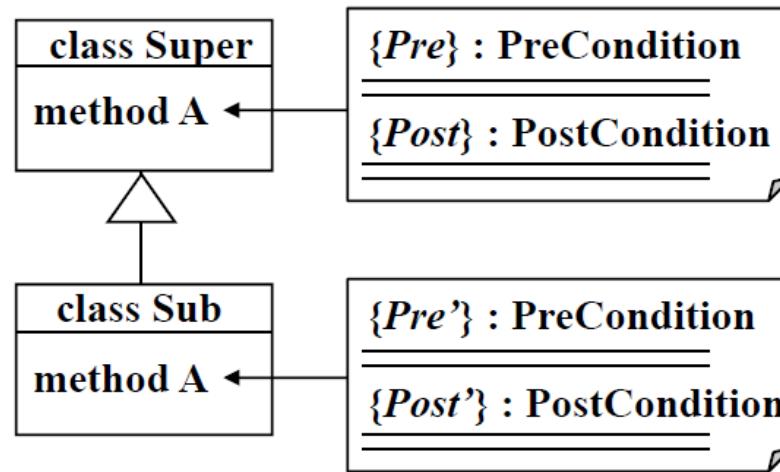


Figure 9-13 A simple example of contract inheritance: An invariant specified in a superclass must hold for all of its subclasses (UML class diagram with OCL constraint).

Inheriting Contracts_2

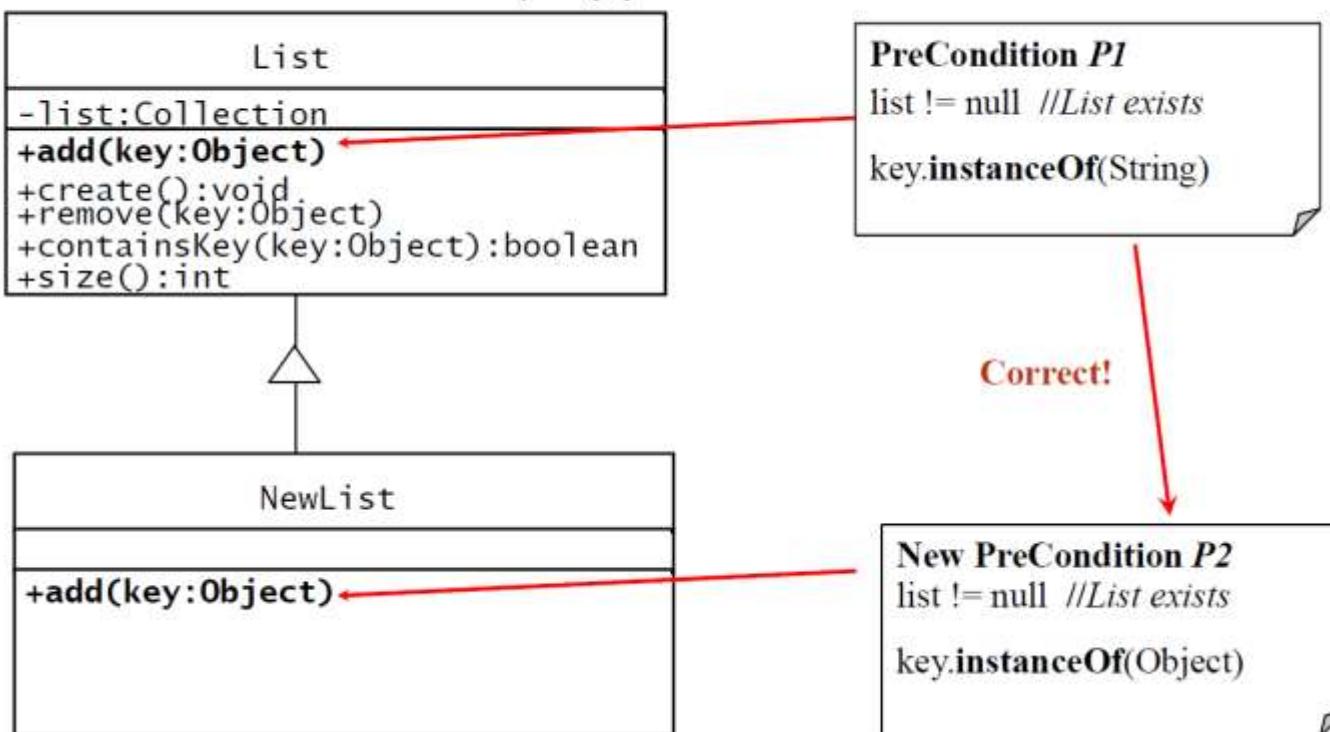


$\{A.Pre'\}$ is weaker than $\{A.Pre\} \iff A.Pre \Rightarrow A.Pre'$

$\{A.Post'\}$ is stronger than $\{A.Post\} \iff A.Post' \Rightarrow A.Post$

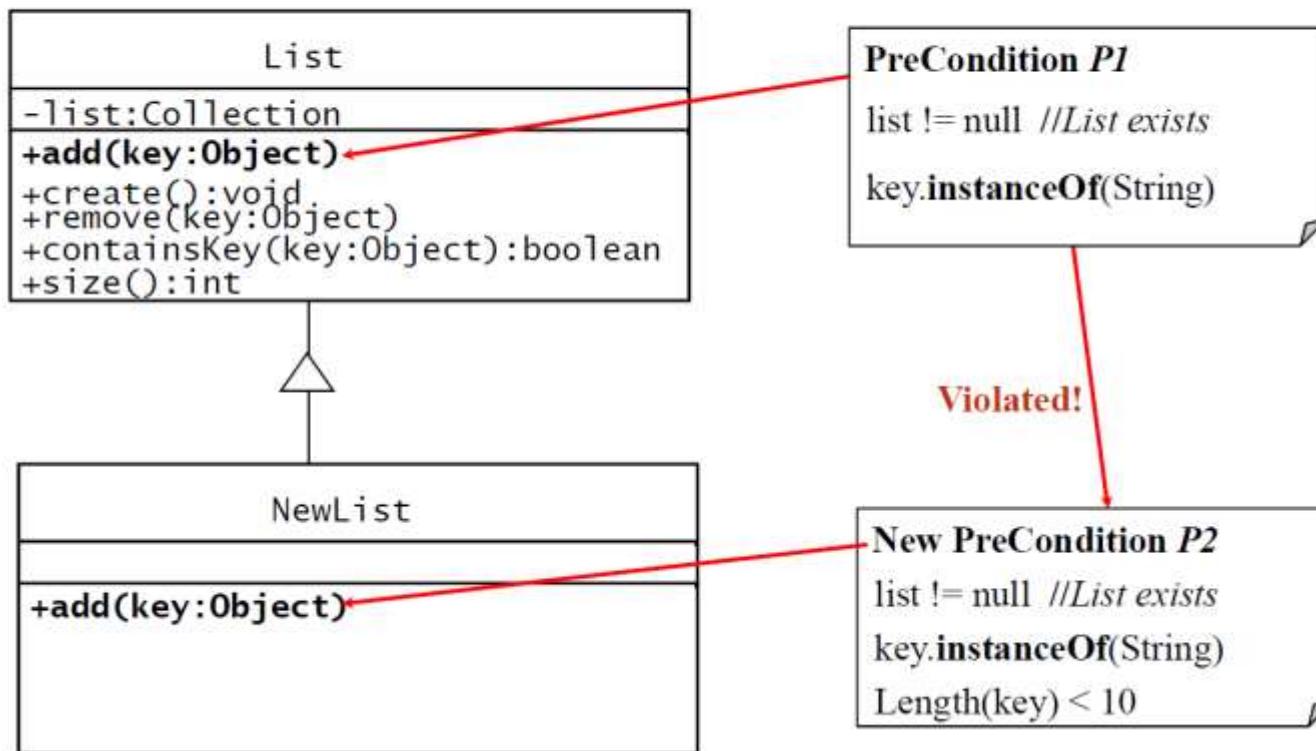
Inheriting Contracts_3

- ◆ A method of subclass is allowed to weaken the **preconditions** of the method it overrides. Example (1)



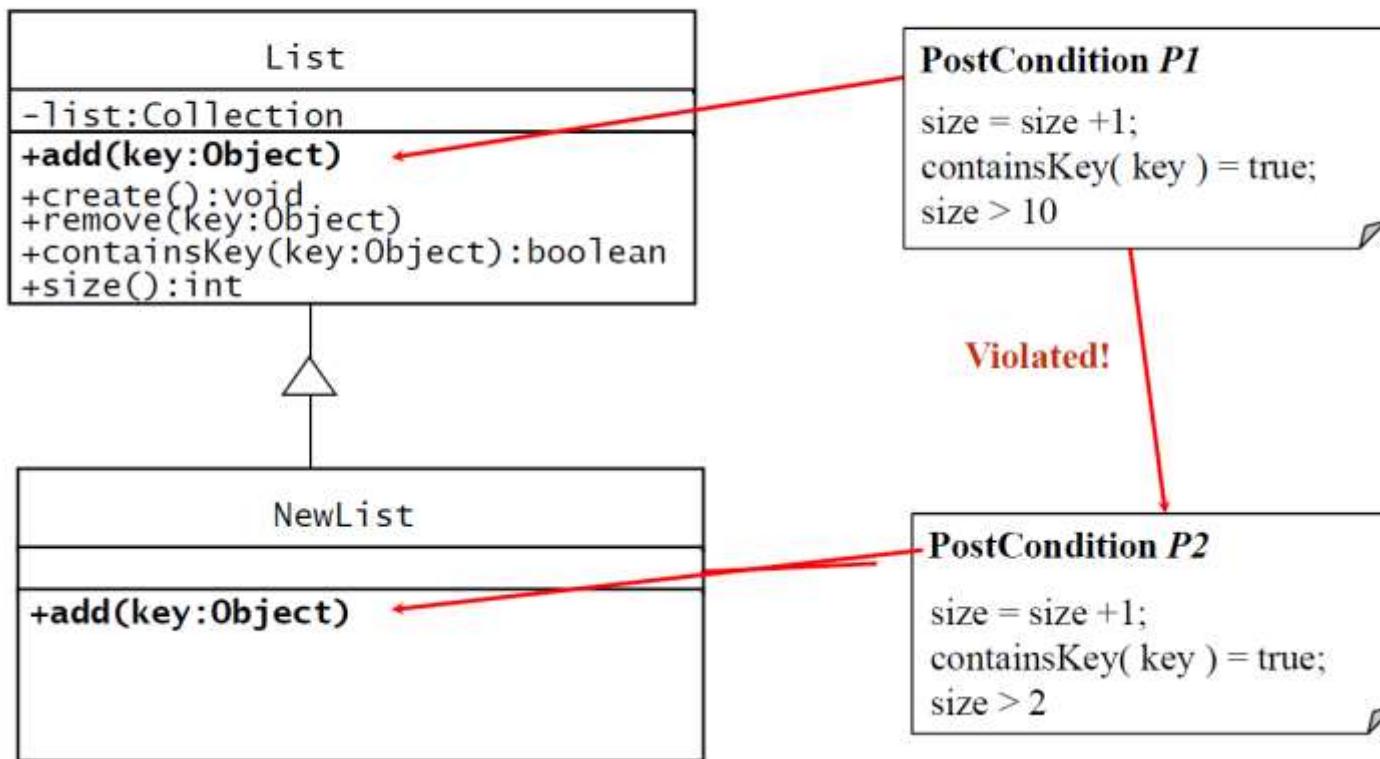
Inheriting Contracts_4

- ◆ A method of subclass is allowed to weaken the **preconditions** of the method it overrides. Example (2)



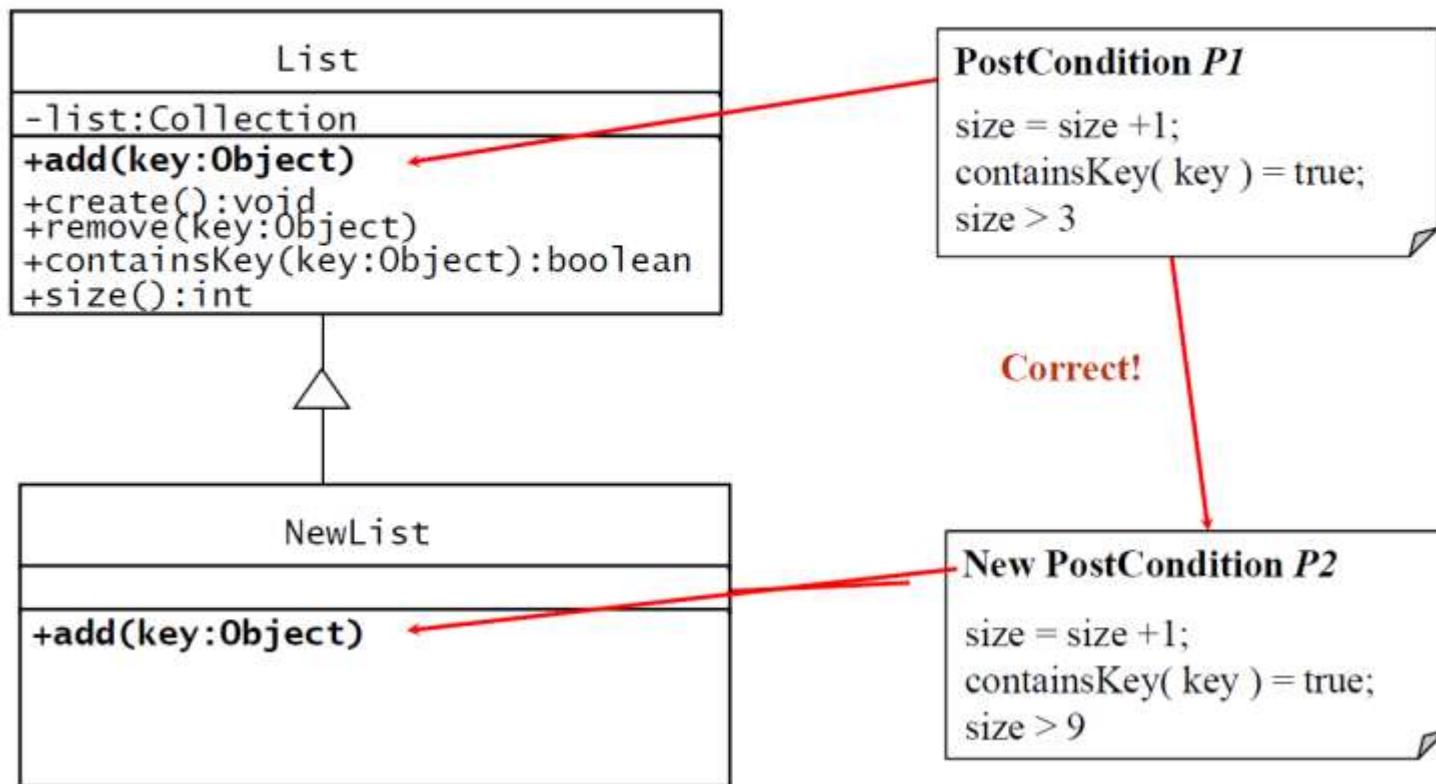
Inheriting Contracts_5

- ◆ A method must ensure a stronger **postcondition** as the method it overrides: Example:



Inheriting Contracts_6

- ◆ A method must ensure a stronger **postcondition** as the method it overrides: Example:



Managing Object Design

There are two primary management challenges during object design:

- *Increased communication complexity.* The number of participants involved during this phase of development increases dramatically. The object design models, and code are the result of the collaboration of many people. Management needs to ensure that decisions among these developers are made consistently with project goals.
- *Consistency with prior decisions and documents.* Developers often do not appreciate completely the consequences of analysis and system design decisions before object design. When detailing and refining the object design model, developers may question some of these decisions and reevaluate them. The management challenge is to maintain a record of these revised decisions and to make sure all documents reflect the current state of development.

Documenting Object Design

Object design is documented in the **Object Design Document (ODD)**. It describes object design trade-offs made by developers, guidelines they followed for subsystem interfaces, the decomposition of subsystems into packages and classes, and the class interfaces.

The ODD is used to exchange interface information among teams and as a reference during testing. The audience for the ODD includes system architects (i.e., the developers who participate in the system design), developers who implement each subsystem, and testers.

Documenting Object Design_2

There are three main approaches to documenting object design:

1. *Self-contained ODD generated from model.* The first approach is to document the object design model the same way we documented the analysis model or the system design model: we write and maintain a UML model and generate the document automatically. This document would duplicate any application objects identified during analysis. The disadvantages of this solution include redundancy with the Requirements Analysis Document (RAD) and a high level of effort for maintaining consistency with the RAD. Moreover, the ODD duplicates information in the source code and requires a high level of effort whenever the code changes. This often leads to an RAD and an ODD that are inaccurate or out of date.

Documenting Object Design_3

There are three main approaches to documenting object design:

2. *ODD as extension of the RAD.* The second approach is to treat the object design model as an extension of the analysis model. In other terms, the object design is considered as the set of application objects augmented with solution objects. The advantage of this solution is that maintaining consistency between the RAD and the ODD becomes much easier as a result of the reduction in redundancy. The disadvantages of this solution include polluting the RAD with information that is irrelevant to the client and the user. Moreover, object design is rarely as simple as identifying additional solution objects. Often, application objects are changed or transformed to accommodate design goals or efficiency concerns.

Documenting Object Design_4

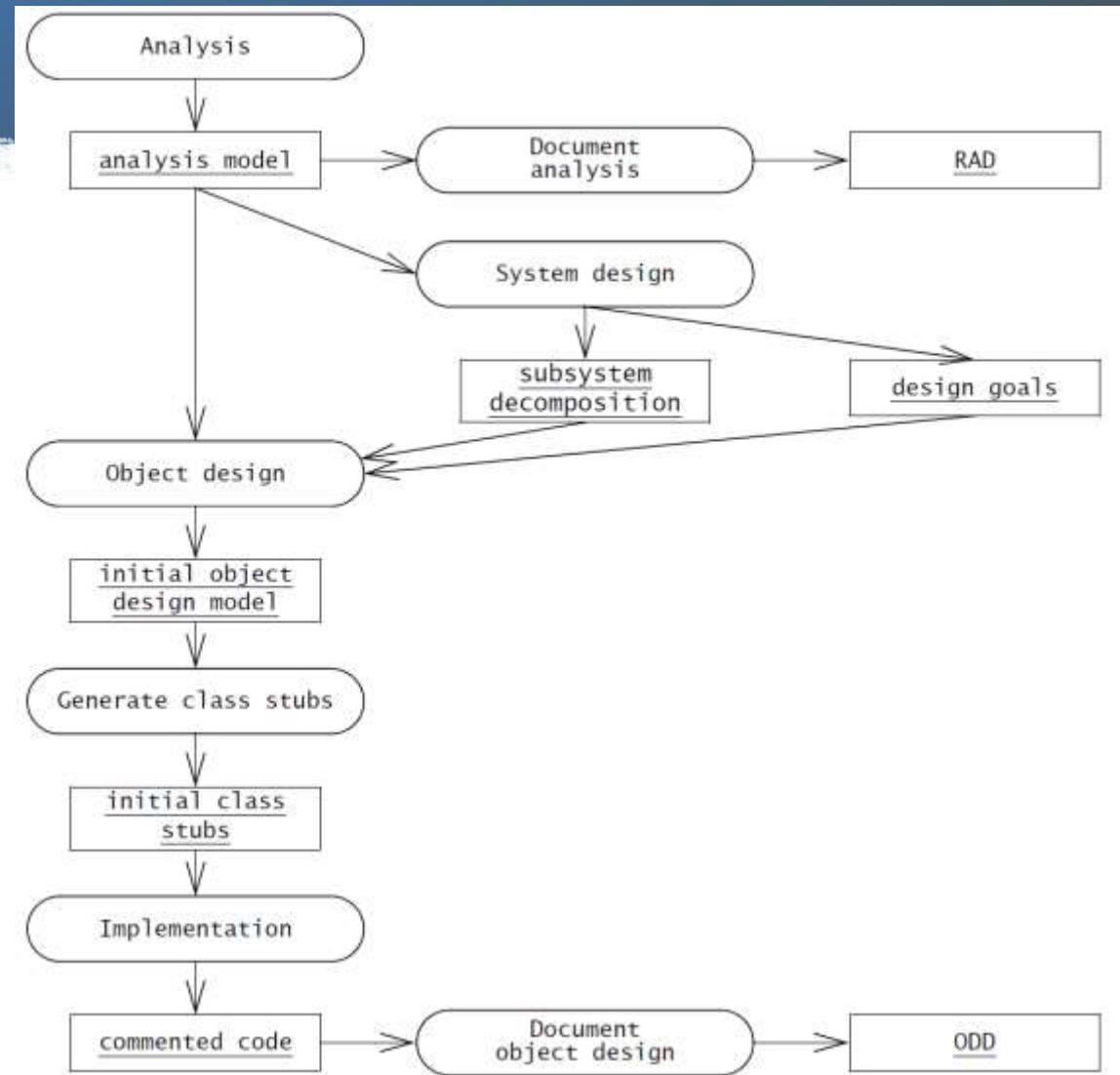
There are three main approaches to documenting object design:

3. *ODD embedded into source code.* The third approach is to embed the ODD into the source code. As in the first approach, we represent the ODD using a modeling tool (see Figure 9-14). Once the ODD becomes stable, we use the modeling tool to generate class stubs. We describe each class interface using tagged comments that distinguish source code comments from object design descriptions. We can then generate the ODD using a tool that parses the source code and extracts the relevant information (e.g., Javadoc [Javadoc, 2009a]). Once the object design model is documented in the code, we abandon the initial object design model. The advantage of this approach is that the consistency between the object design model and the source code is much easier to maintain: when changes are made to the source code, the tagged comments are updated and the ODD regenerated.

Documenting Object Design_5

The fundamental issue is one of maintaining consistency among two models and the source code. Ideally, we want to maintain the analysis model, the object design model, and the source code using a single tool. Objects would then be described once, and consistency among documentation, stubs, and code would be maintained automatically.

Documenting Object Design_6



Documenting Object Design_7

- The first section of the ODD is an ***Introduction*** to the document.
- The second section of the ODD, ***Packages***, describes the decomposition of subsystems into packages and the file organization of the code. This includes an overview of each package, its dependencies with other packages, and its expected usage.
- The third section, ***Class interfaces***, describes the classes and their public interfaces. This includes an overview of each class, its dependencies with other classes and packages, its public attributes, operations, and the exceptions they can raise.

Documenting Object Design_8

Object Design Document

1. Introduction
 - 1.1 Object design trade-offs
 - 1.2 Interface documentation guidelines
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 References
 2. Packages
 3. Class interfaces
- Glossary
-

Figure 9-15 Outline of the Object Design Document.

Using Contracts During Requirements Analysis

Some requirements analysis approaches advocate the use of constraints much earlier, for example, during the definition of the entity objects. In principle, OCL can be used in requirements analysis as well as in object design.

In general, developers consider specific project needs before deciding on a specific approach or level of formalism to be used when documenting operations. Examine the following trade-offs before deciding when to use constraints for which purpose:

Using Contracts During Requirements Analysis_2

- ***Communication among stakeholders.*** During software development, models support communication among stakeholders. Different models are used for different types of stakeholders. On the one hand, a use case or a user interface mock-up is much easier for a client to understand than an OCL constraint. On the other hand, an OCL constraint is much more precise statement for the class user.
- ***Level of detail and rate of change.*** Attaching constraints to an analysis model requires a much deeper understanding of the requirements. When this information is available, either from the user, the client, or general domain knowledge, this results in a more complete analysis model. When this information is not available, however, clients and developers may be forced to make decisions too early in the process, increasing the rate of change (and consequently, development cost) later in the development.

Using Contracts During Requirements Analysis_3

- ***Level of detail and elicitation effort.*** Similarly, eliciting detailed information from a user during analysis may require much more effort than eliciting this information later in the process, when early versions of the user interface are available and specific issues can be demonstrated. However, this approach assumes that modifying the components under consideration is relatively cheap and does not have a serious impact on the rest of the system. This is the case for user interface layout issues and dialog considerations.
- ***Testing requirements.*** During testing, we compare the actual behavior of the system or a class with the specified behavior. For automated tests or for stringent testing requirements (found, for example, in application domains such as traffic control, medicine, or pharmaceuticals), this requires a precise specification to test against. In this case, constraints help a lot if they are specified as early as possible.

Conclusions related to Interface Specification

- ***Identifying missing operations and writing contracts are overlapping activities.*** Writing contracts forces us to look carefully at each object and might result in finding new behaviors and boundary cases.
- ***Writing contracts leads to additional helper operations for inspecting objects.*** Although the relevant object state is accessible by examining attributes and navigating relevant associations, writing simple contracts encourages us to add helper methods for examining state. This results in simpler constraints and classes that are more easily Testable.
- ***Writing contracts leads to better abstractions.*** When inheriting contracts, only preconditions can be weakened. Postconditions and invariants can only be strengthened. Consequently, when writing contracts for abstract classes, we add abstract operations for describing properties of the concrete classes.

OCL – few operations on collections

/* All these are literal expressions – context independent */

$\text{Set}\{1,5,7,9, 8\} - \text{Set}\{2, 4, 6, 8, 5\} = \text{Set}\{1, 7, 9\}$

$\text{Set}\{1,5,7,9, 8\} \rightarrow \text{symmetricDifference}(\text{Set}\{2, 4, 6, 8, 5\}) = \text{Set}\{1, 7, 9, 2, 4, 6\}$

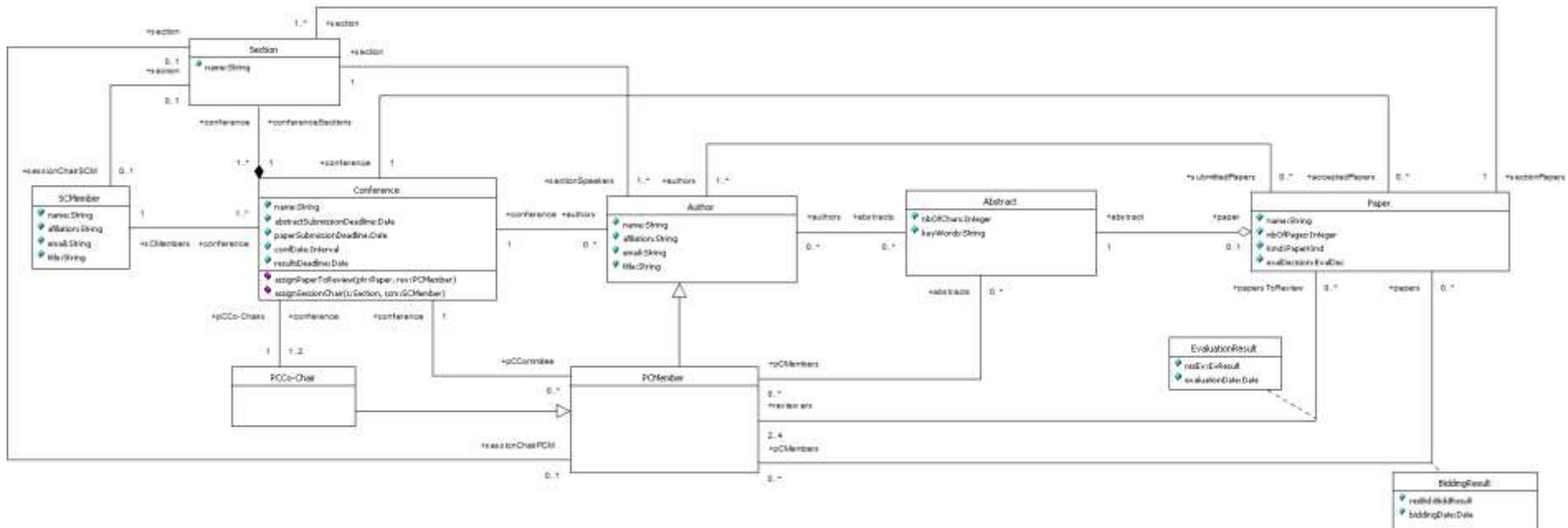
$\text{Set}\{1,5,7,9, 8\} \rightarrow \text{any(true)} = 1$

$\text{Set}\{1,5,7,9, 8\} \rightarrow \text{any}(i \mid i > 8) = 9$

$\text{Set}\{1,5,7,9, 8\} \rightarrow \text{any}(i \mid i \geq 7) = 7$

$\text{Set}\{1,5,7,9, 8\} \rightarrow \text{any}(i \mid i \geq 8) = 9$

Conference Management – Specifying Interfaces

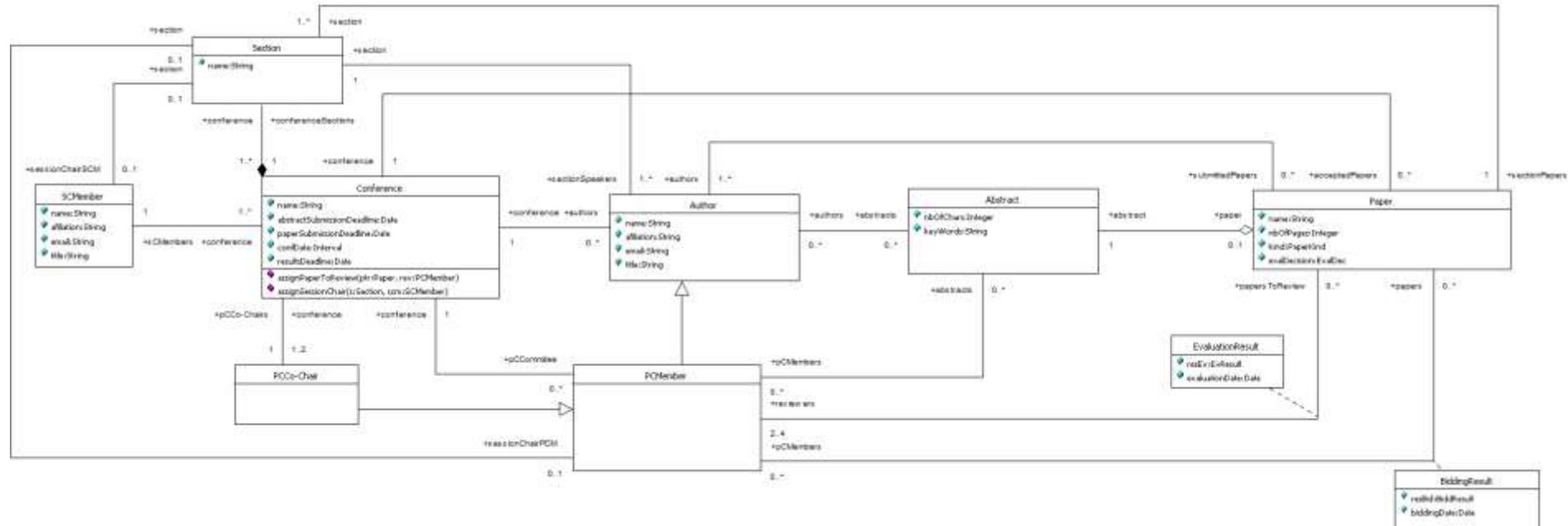


context PCMember

inv approprPapToReview:

```
self.papersToReview->select(p:Paper | Set{BiddResult::conflict, BiddResult::refuseToEv}->
  includes(p.biddingResult->any(br| br.pCMembers->includes(self).resBid))->isEmpty and
  self.papersToReview.authors->excludes(self.oclAsType(Author))
```

Conference Management – Specifying Interfaces

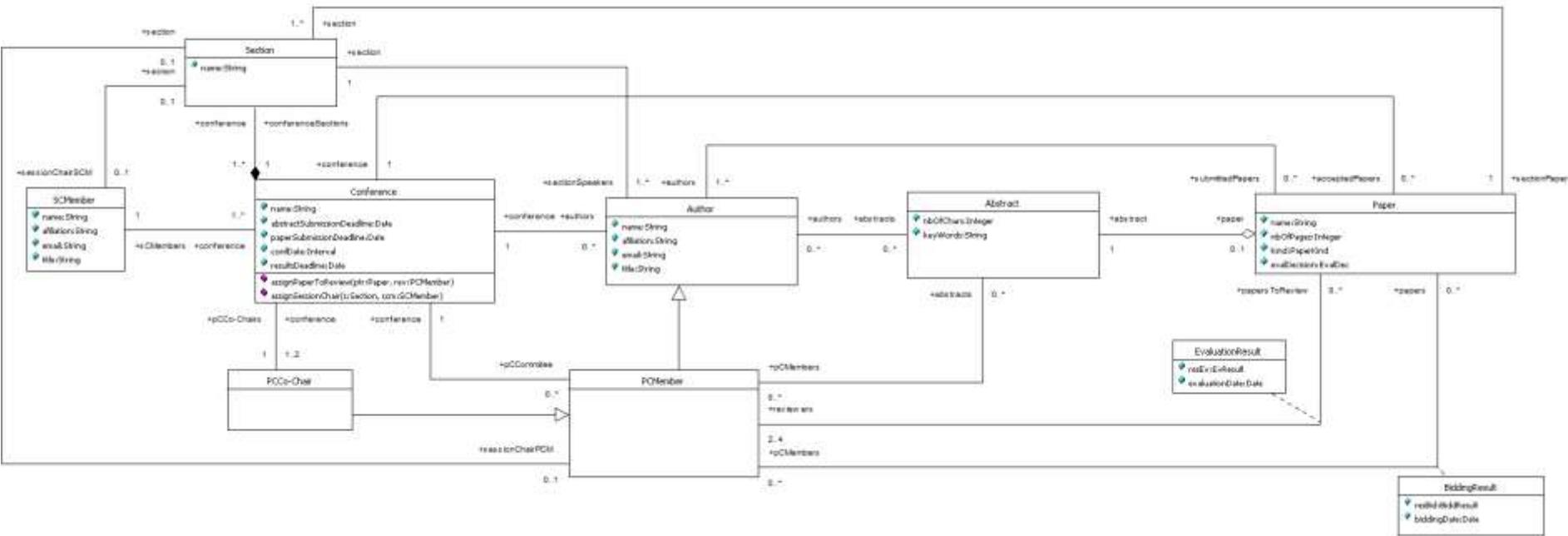


context Conference::assignPaperToReview(ptr:Paper, rev:PCM)

pre: `ptr.reviewers->size < 4 and ptr.reviewers->excludes(rev) and self.submittedPapers->includes(ptr) and self.pCCommitee->includes(rev) and Set{BiddResult::conflict, BiddResult::refuseToEv}->excludes(ptr.biddingResult->select(br | br.pCMembers = rev)->any(true).resBid)`

post: `ptr.reviewers->includes(rev) and ptr.reviewers->size = ptr.reviewers@pre->size + 1`

Conference Management – Specifying Interfaces

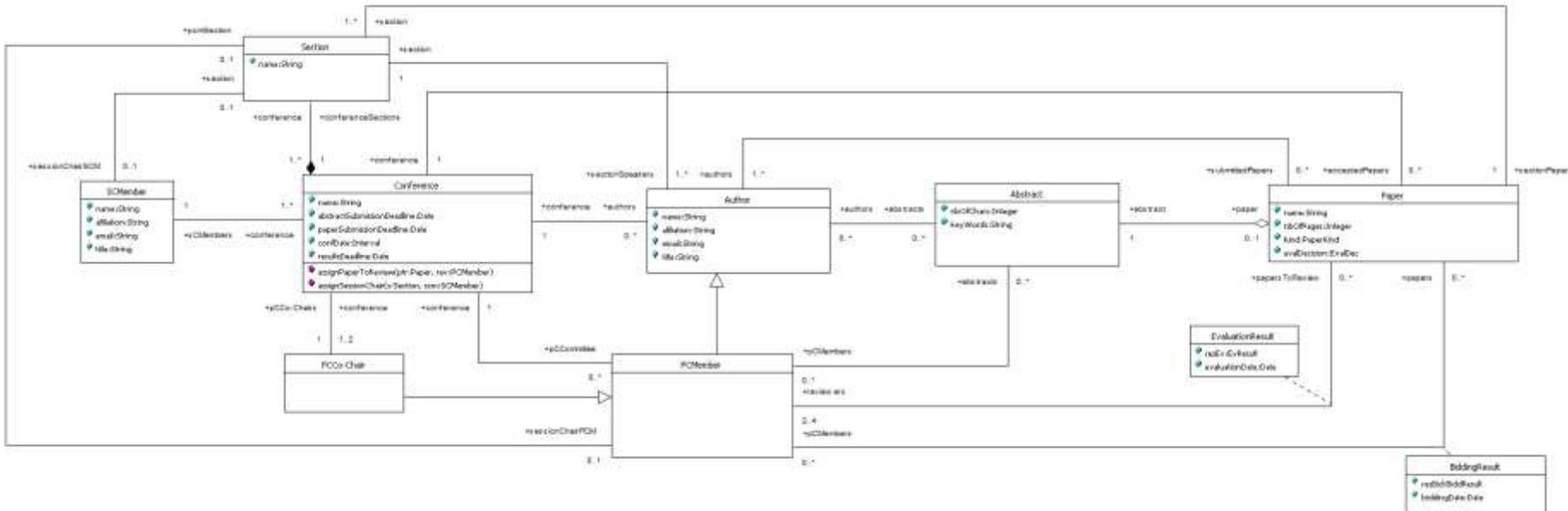


context PCMember

inv sessionChair:

not self.oclAsType(PCMember).section.isDefined implies
self.oclAsType(PCMember).section.sectionSpeakers->excludes(self.oclAsType(Author))

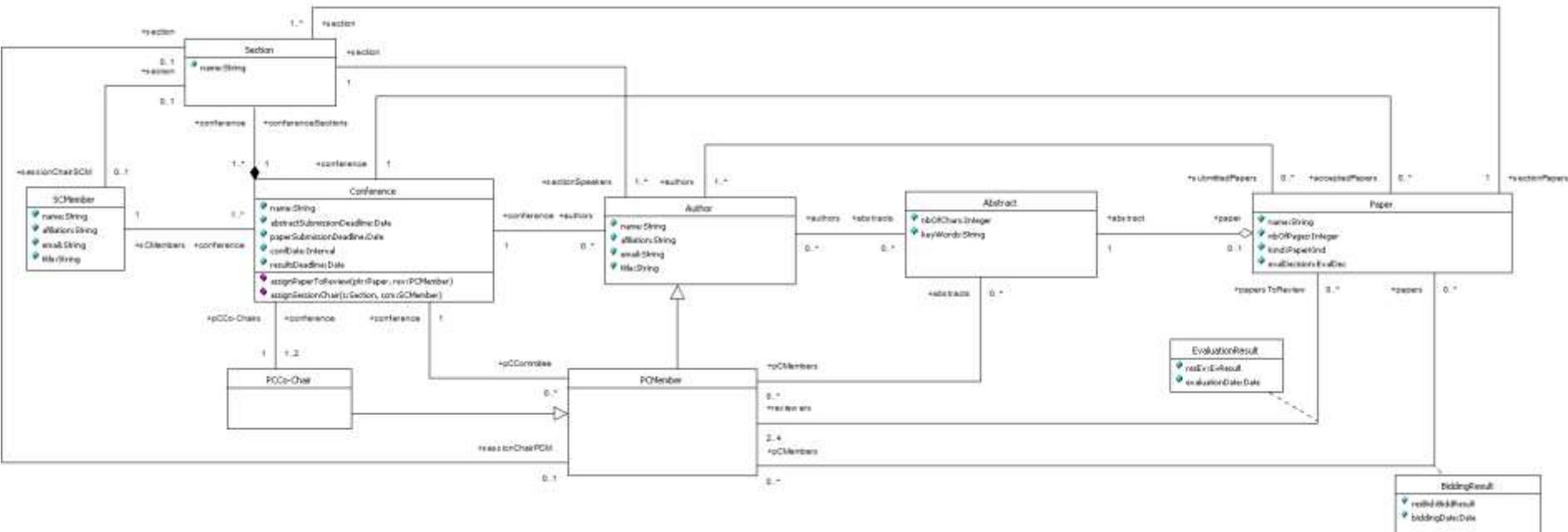
Conference Management – Specifying Interfaces



The previous OCL specification, using upcast for the section property visible in the PCMember context was made for an un compilable model because in the namespace of PCMember there is a conflict name. The correct solution is this.

```
context PCMember
inv sessionChair:
    not self.pcmSection.isDefined implies
        self.pcmSection.sectionSpeakers->excludes(self.oclAsType(Author))
```

Conference Management – Specifying Interfaces



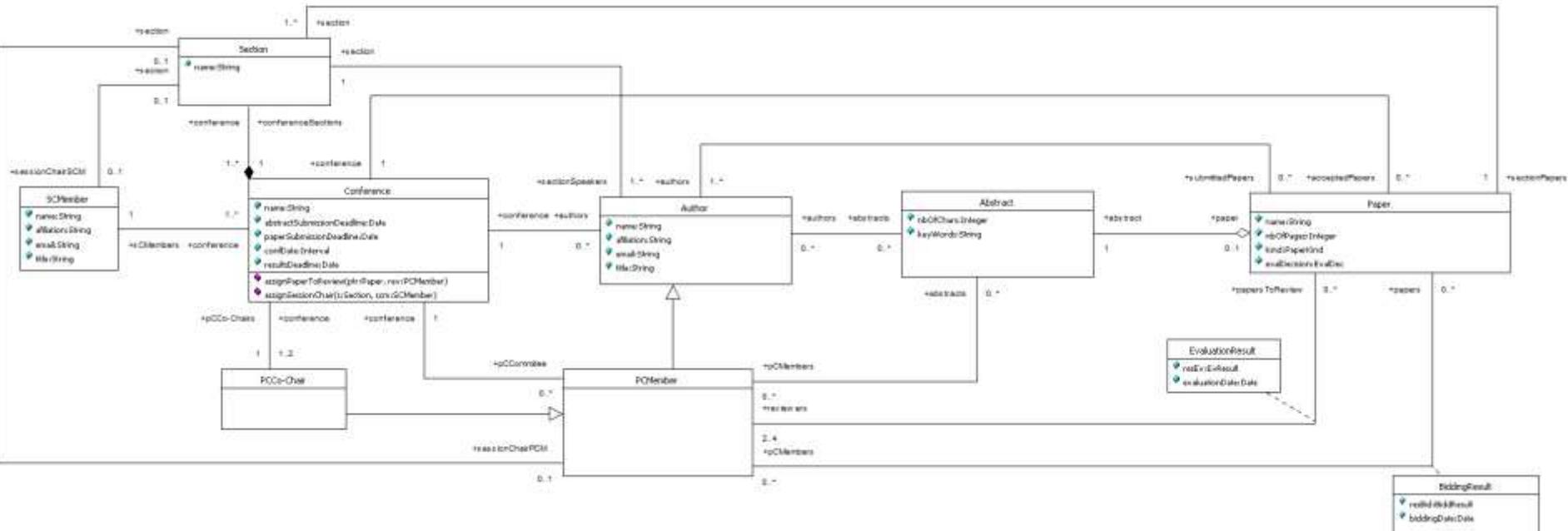
context Conference::assignSessionChair(s:Section, scm:SCMember)

pre:

s.sessionChairSCM.**isUndefined** and s.sessionChairPCM.**isUndefined** and
self.sCMembers->includes(scm) and **self.conferenceSections->includes(s)**

post: **not** s.sessionChairSCM.**isUndefined** /* **isUndefined** is oclIsUndefined() */

Conference Management – Specifying Interfaces



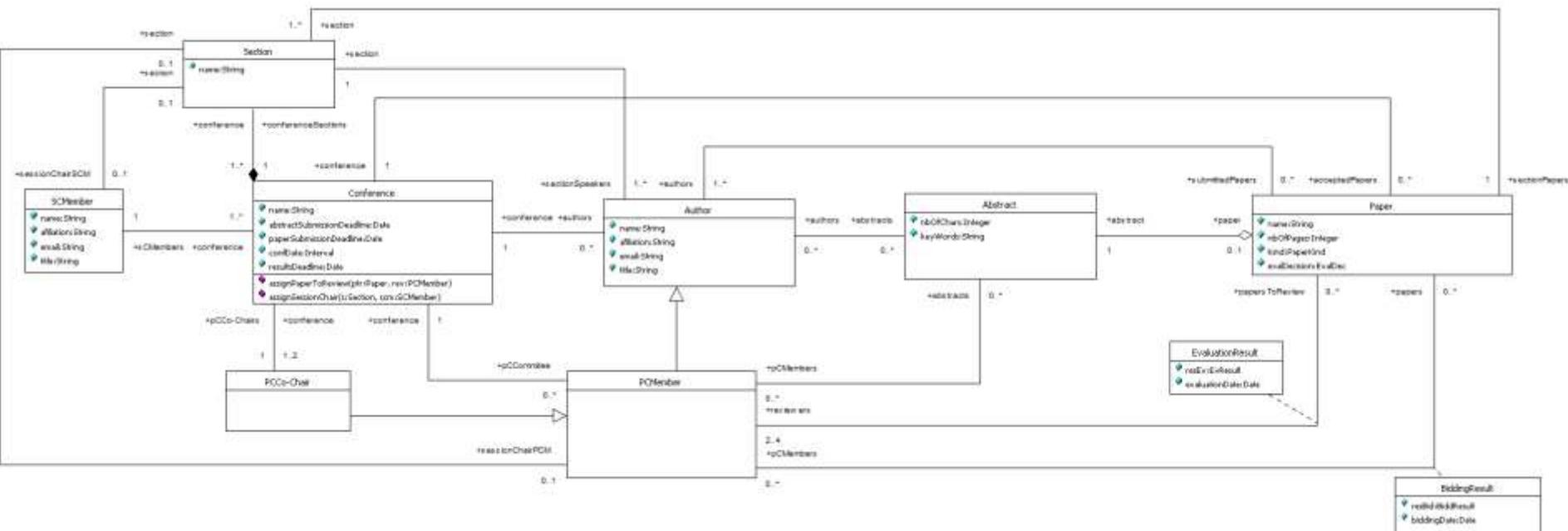
context Conference

def acceptRejectUndecided:

let allEvalResBorderline: Set(Paper)=**self**.authors.submittedPapers->asSet->select(p:Paper | p.evaluationResult.rezEv->forAll(rE | rE=EvResult::borderlinePaper))

let acceptedPapersC: Set(Paper) = **self**.authors.submittedPapers->asSet->select(p:Paper | Set{EvResult::strongAccept, EvResult::accept, EvResult::weakAccept, EvResult::borderlinePaper}->includesAll(p.evaluationResult.rezEv))-allEvalResBorderline

Conference Management – Specifying Interfaces



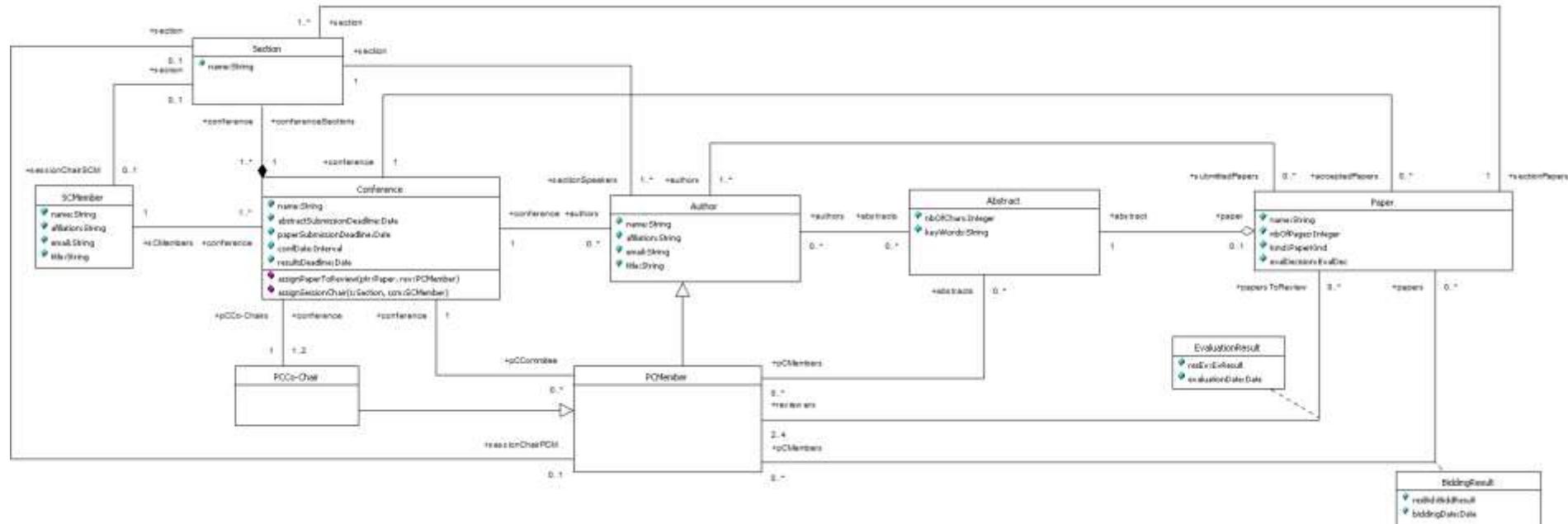
context Conference

def acceptRejectUndecided:

```
let rejectedPapersC: Set(Paper) = self.authors.submittedPapers->asSet->select(p:Paper |
Set{EvResult::strongReject, EvResult::reject, EvResult::weakReject, EvResult::borderlinePaper}->
includesAll(p.evaluationResult.rezEv))-allEvalResBorderline
```

```
let undecidedPapersC: Set(Paper) = self.authors.submittedPapers->asSet -acceptedPapersC
- rejectedPapersC
```

Conference Management – Specifying Interfaces



context Section

inv sessionChair:

self.sessionChairSCM->isEmpty xor self.sessionChairPCM->isEmpty

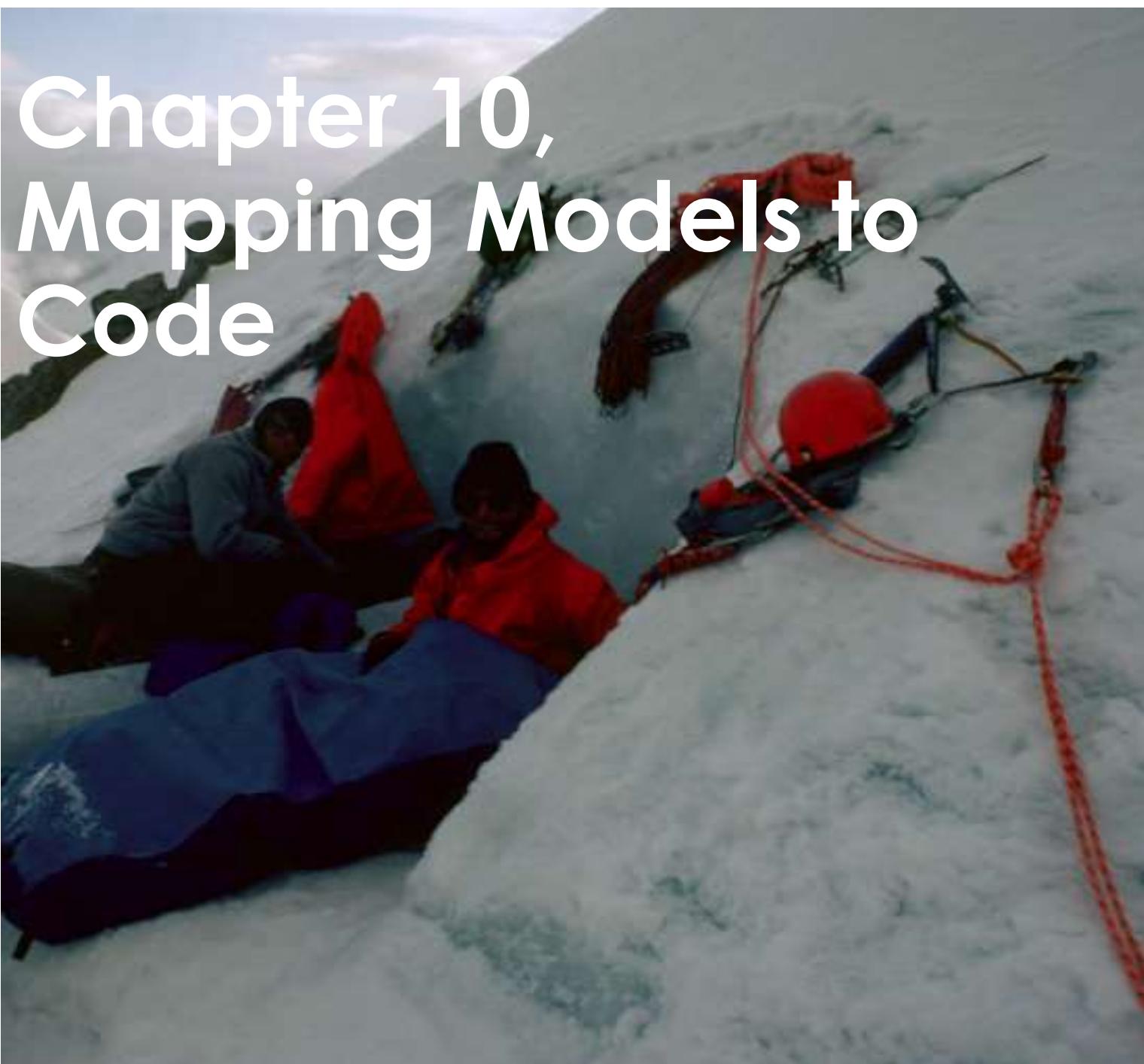
Advices for understanding/learning OCL

- Use an OCL tool!
- Specify the model using UML
- Specify OCL looking at the model!
- Start with the context (the visibility domain) and with the kind of specification: inv:, pre:, post:, def:, def: let:, body:
- In case of pre & post mention the full signature of the method
- Use **self** even if this seems to be redundant
- Compile de specifications
- Instantiate the model by means of snapshots
- Evaluate the model
- In the Foundation package of the UML 1.5 metamodel, OCLE show you the signature of all: collections, OclAny, Boolean, Integer, Real, String

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 10, Mapping Models to Code



State of the Art: Model-based Software Engineering

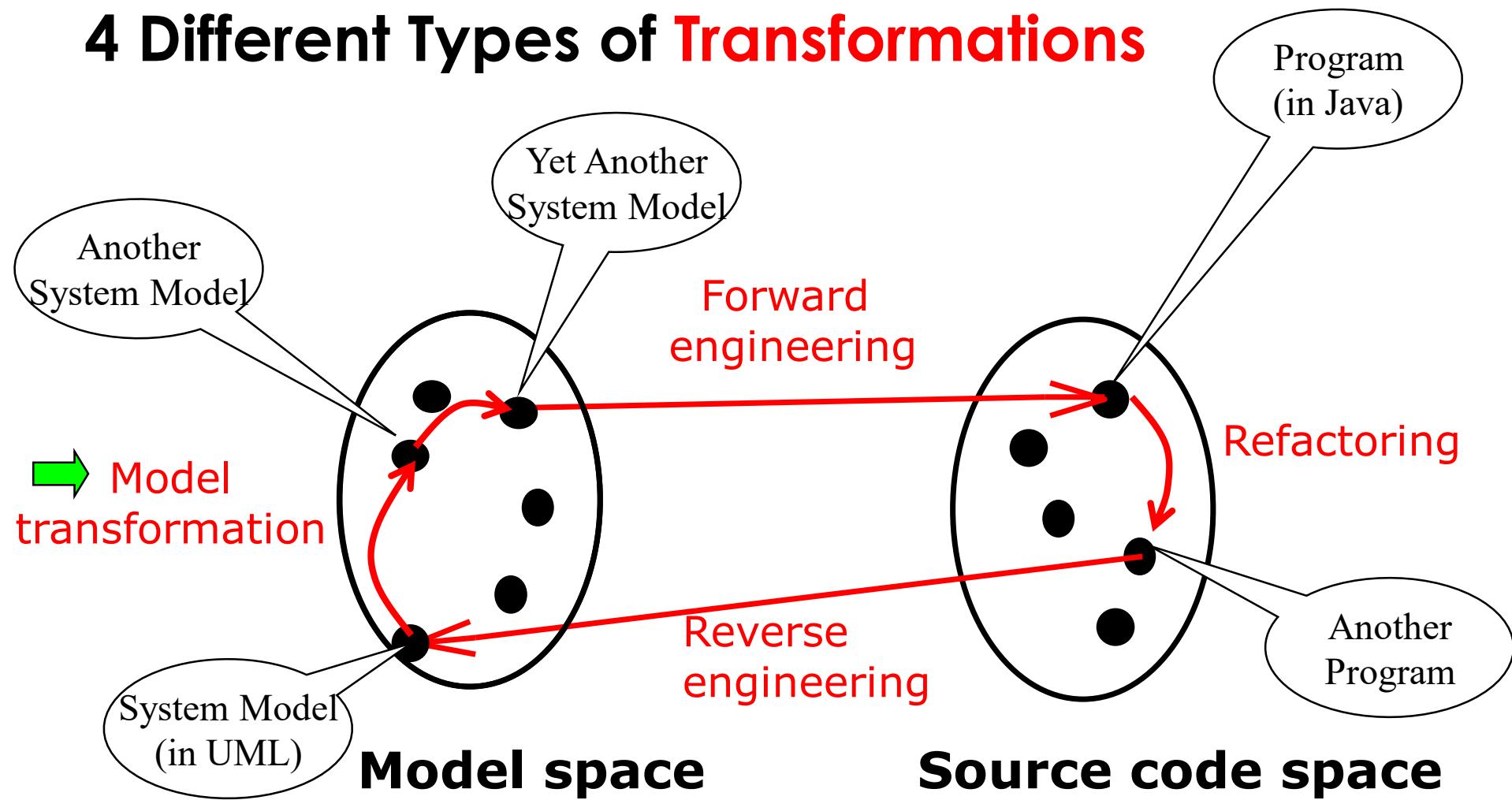
- The Vision
 - During object design we build an object design model that realizes the use case model, and which is the basis for implementation (model-driven design)
- The Reality
 - Working on the object design model involves many activities that are error prone
 - Examples:
 - A new parameter must be added to an operation. Because of time pressure it is added to the source code, but not to the object model
 - Additional attributes are added to an entity object, but the data base table is not updated (as a result, the new attributes are not persistent).

Other Object Design Activities

- Programming languages do not support the concept of a UML association
 - The associations of the object model must be transformed into collections of object references
- Many programming languages do not support contracts (invariants, pre and post conditions)
 - Developers must therefore manually transform contract specification into source code for detecting and handling contract violations
- The client changes the requirements during object design
 - The developer must change the interface specification of the involved classes
- All these object design activities cause **problems**, because they need to be done manually.

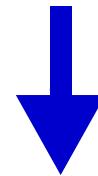
- Let us get a handle on these problems
- To do this we distinguish two kinds of spaces
 - the model space and the source code space
- and 4 different types of transformations
 - Model transformation
 - Forward engineering
 - Reverse engineering
 - Refactoring.

4 Different Types of Transformations

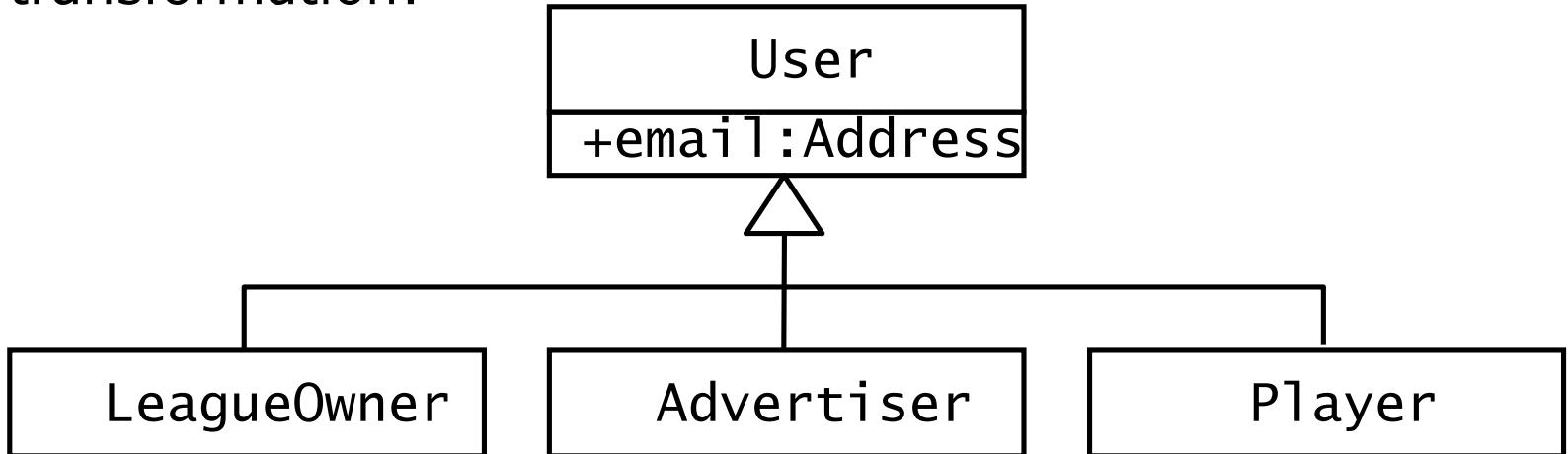


Model Transformation Example

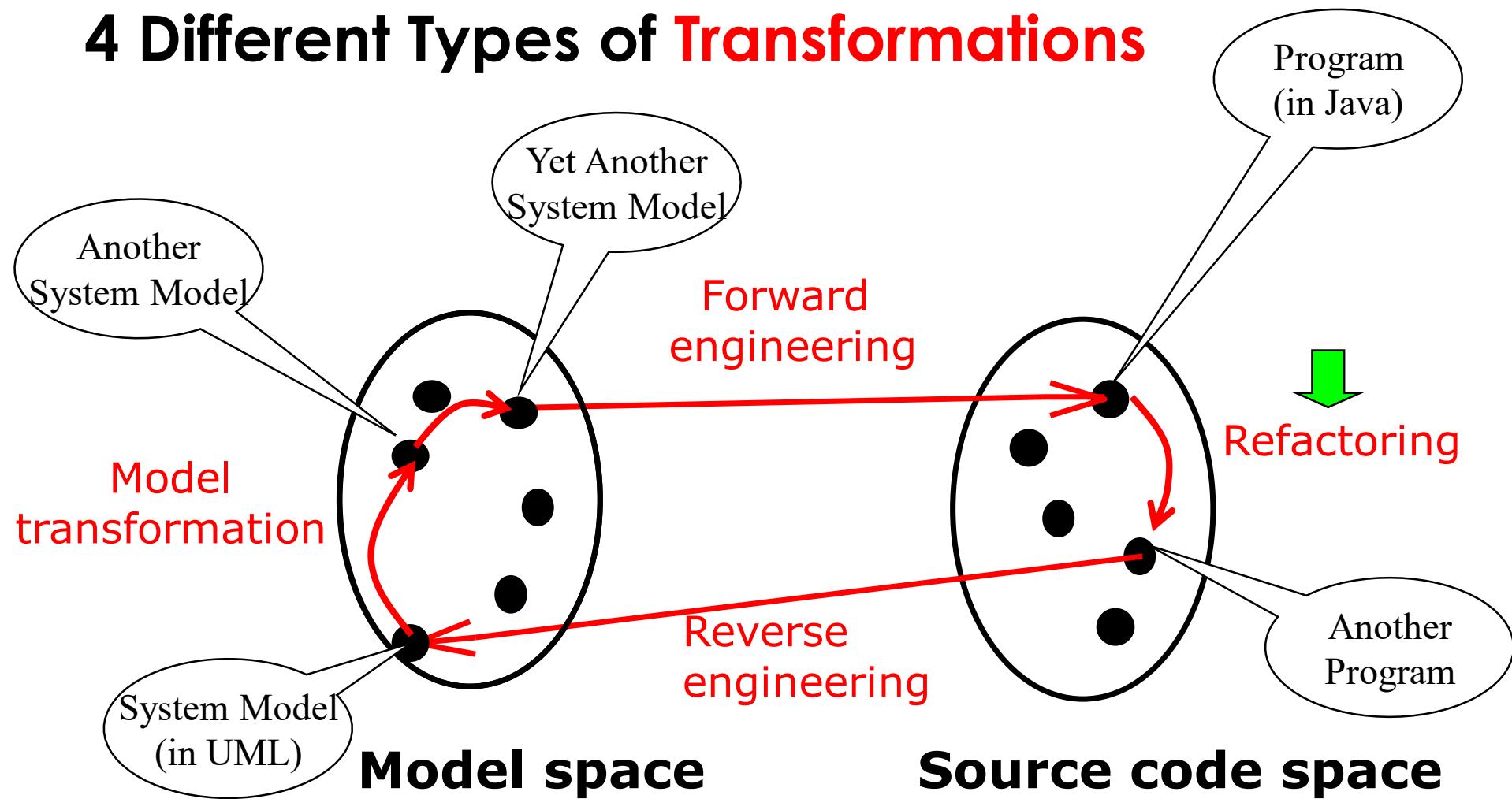
Object design model before transformation:



Object design model
after transformation:



4 Different Types of Transformations



Refactoring Example: Pull Up Field

```
public class Player {  
    private String email;  
    //...  
}  
  
public class LeagueOwner {  
    private String eMail;  
    //...  
}  
  
public class Advertiser {  
    private String  
        email_address;  
    //...  
}
```

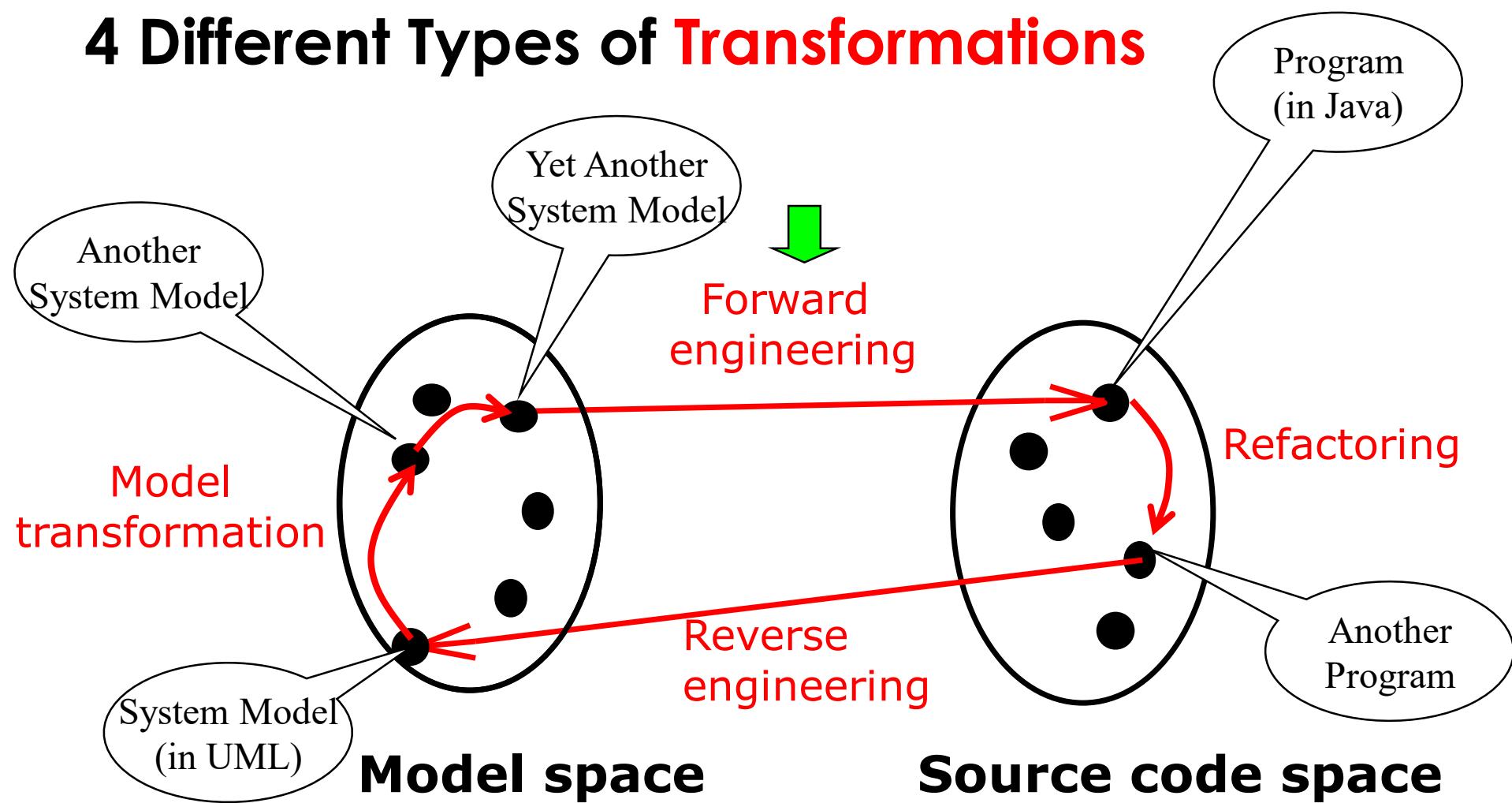
```
public class User {  
    private String email;  
}  
  
public class Player extends User {  
    //...  
}  
  
public class LeagueOwner extends  
User {  
    //...  
}  
  
public class Advertiser extends  
User {  
    //...  
}
```

Refactoring Example: Pull Up Constructor Body

```
public class User {  
    private String email;  
}  
  
public class Player extends User {  
    public Player(String email) {  
        this.email = email;  
    }  
}  
  
public class LeagueOwner extends User{  
    public LeagueOwner(String email) {  
        this.email = email;  
    }  
}  
  
public class Advertiser extendsUser{  
    public Advertiser(String email) {  
        this.email = email;  
    }  
}
```

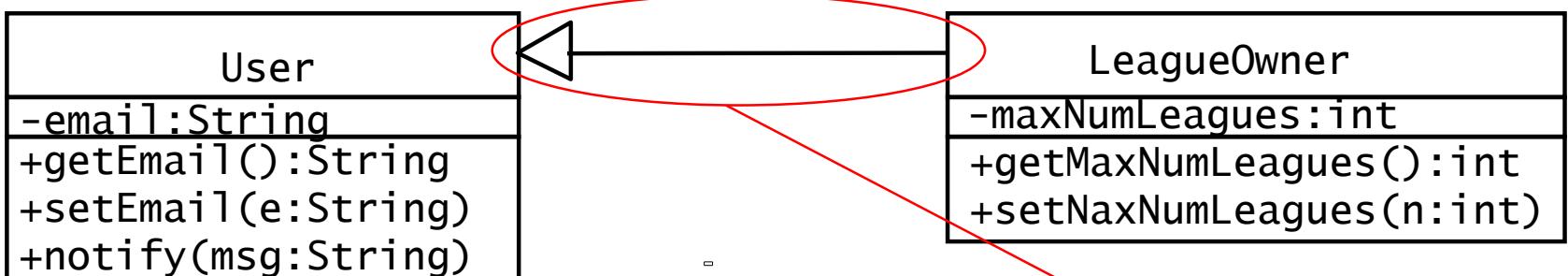
```
public class User {  
    public User(String email) {  
        this.email = email;  
    }  
}  
  
public class Player extends User {  
    public Player(String email)  
    {  
        super(email);  
    }  
}  
  
public class LeagueOwner extends User {  
    public LeagueOwner(String email) {  
        super(email);  
    }  
}  
  
public class Advertiser extends User {  
    public Advertiser(String email) {  
        super(email);  
    }  
}
```

4 Different Types of Transformations



Forward Engineering Example

Object design model before transformation:



Source code after transformation:

```
public class User {  
    private String email;  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String value){  
        email = value;  
    }  
    public void notify(String msg) {  
        // ....  
    }  
}
```

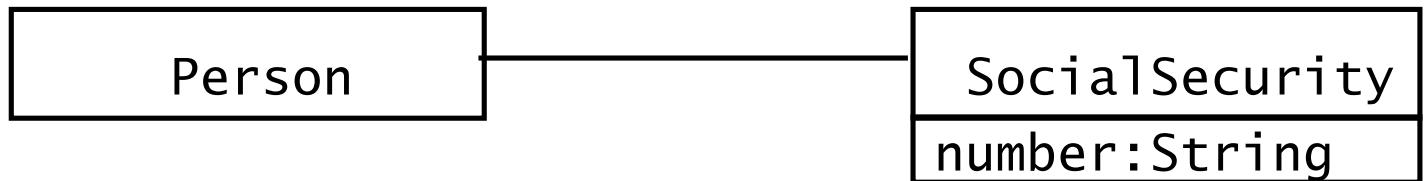
```
public class LeagueOwner extends User {  
    private int maxNumLeagues;  
    public int getMaxNumLeagues() {  
        return maxNumLeagues;  
    }  
    public void setMaxNumLeagues  
        (int value) {  
        maxNumLeagues = value;  
    }  
}
```

More Examples of Model Transformations and Forward Engineering

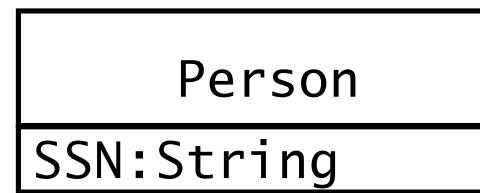
- Model Transformations
 - Goal: Optimizing the object design model
 - Collapsing objects
 - Delaying expensive computations
- Forward Engineering
 - Goal: Implementing the object design model in a programming language
 - Mapping inheritance
 - Mapping associations
 - Mapping contracts to exceptions
 - Mapping object models to tables

Collapsing Objects

Object design model before transformation:



Object design model after transformation:



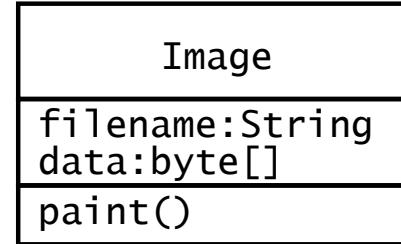
Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

Examples of Model Transformations and Forward Engineering

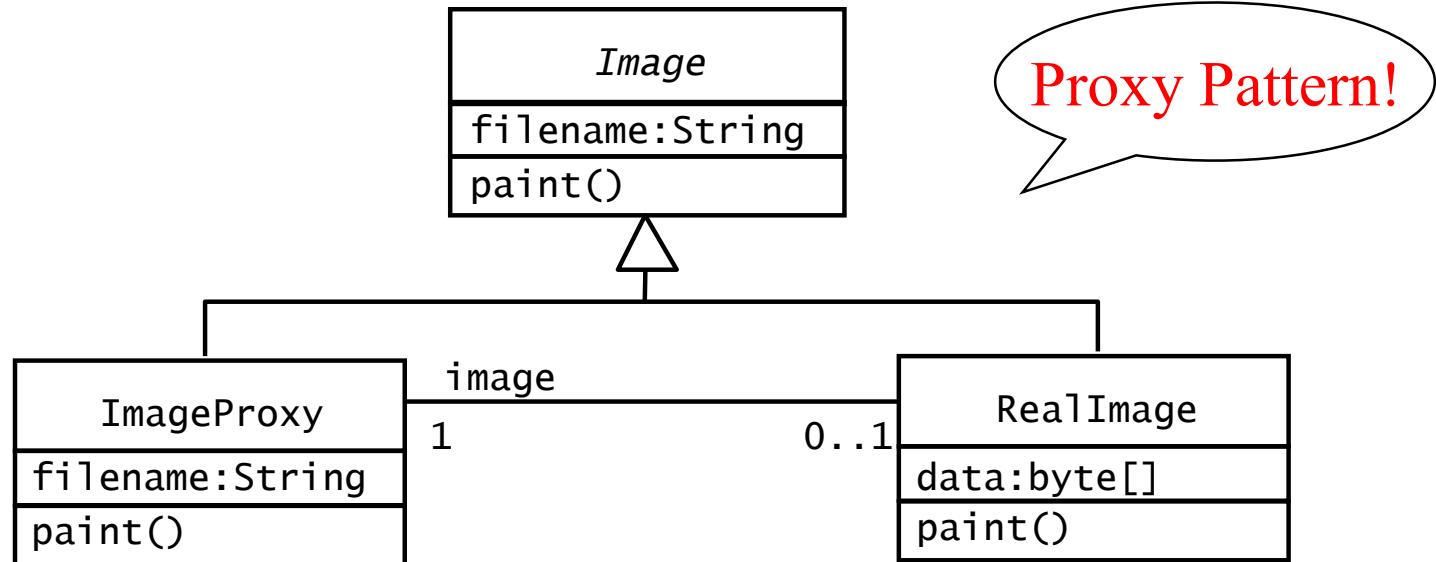
- Model Transformations
 - Goal: Optimizing the object design model
 - Collapsing objects
 - Delaying expensive computations
- Forward Engineering
 - Goal: Implementing the object design model in a programming language
 - Mapping inheritance
 - Mapping associations
 - Mapping contracts to exceptions
 - Mapping object models to tables

Delaying expensive computations

Object design model before transformation:



Object design model after transformation:



Using Different Views in Forward Engineering

- Architectural/Statical Views supports a complete automatic code generation 100%
 - Goal: Declaration of concepts: classes, interfaces, associations, implementation relationships (between classes and interfaces), import relationships (which correspond to dependencies in UML)
 - Full management of associations irrespective of their nature and properties, import declarations
- Dynamical Views
 - Goal: Generating the code corresponding to:
 - state transition diagrams and activities diagrams/flowcharts,
 - Sequence/Collaboration(Communication) diagram; method composition

Using Assertions and Observers in Forward Engineering

- OCL specifications can be transformed in OO programming languages. The management of assertion violation depends first on the target programming language and on the generated code, also.
 - In case of assertion violation, OCLE print a message.
- OCLE restrictions:
 - The current version does not consider code generation in case of Dynamical Views. However, due to support of Observers' code generation, the percent of code generated by OCLE is 70-75%. The missing part must be manually written.

Examples of Model Transformations and Forward Engineering

- Model Transformations
 - Goal: Optimizing the object design model
 - Collapsing objects
 - Delaying expensive computations
- Forward Engineering
 - Goal: Implementing the object design model in a programming language
 - Mapping inheritance
 - Mapping associations
 - Mapping contracts to exceptions
 - Mapping object models to tables

Forward Engineering: Mapping a UML Model into Source Code

- **Goal:** We have a UML-Model with inheritance.
We want to translate it into source code
- **Question:** Which mechanisms in the programming language can be used?
 - Let's focus on Java
- Java provides the following mechanisms:
 - Overwriting of methods (default in Java)
 - Final classes
 - Final methods
 - Abstract methods
 - Abstract classes
 - Interfaces.

Realizing Inheritance in Java

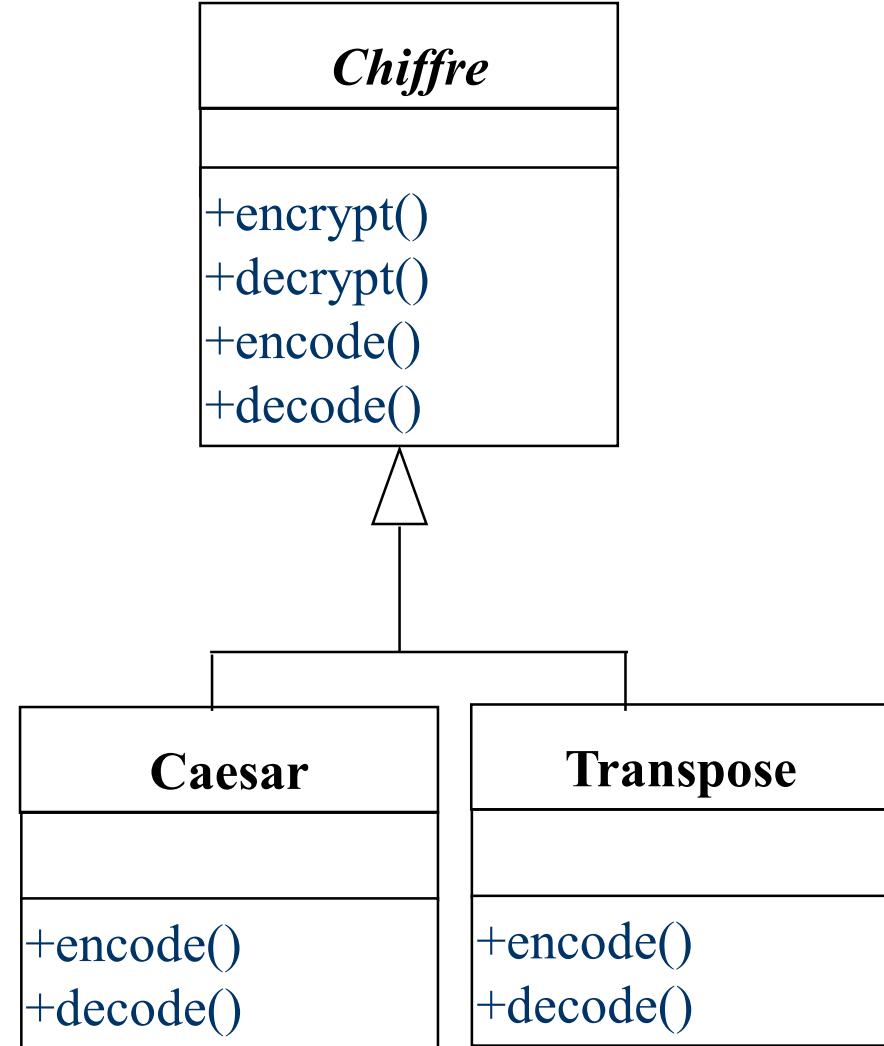
- Realisation of specialization and generalization
 - Definition of subclasses
 - Java keyword: **extends**
- Realisation of simple inheritance
 - Overwriting of methods is not allowed
 - Java keyword: **final**
- Realisation of implementation inheritance
 - Overwriting of methods
 - No keyword necessary:
 - Overwriting of methods is default in Java
- Realisation of specification inheritance
 - Specification of an interface
 - Java keywords: **abstract, interface**

Example for the use of Abstract Methods: Cryptography

- Problem: Delivery a general encryption method
- Requirements:
 - The system provides algorithms for existing encryption methods (e.g. Caesar, Transposition)
 - New encryption algorithms, when they become available, can be linked into the program at runtime, without any need to recompile the program
 - The choice of the best encryption method can also be done at runtime.

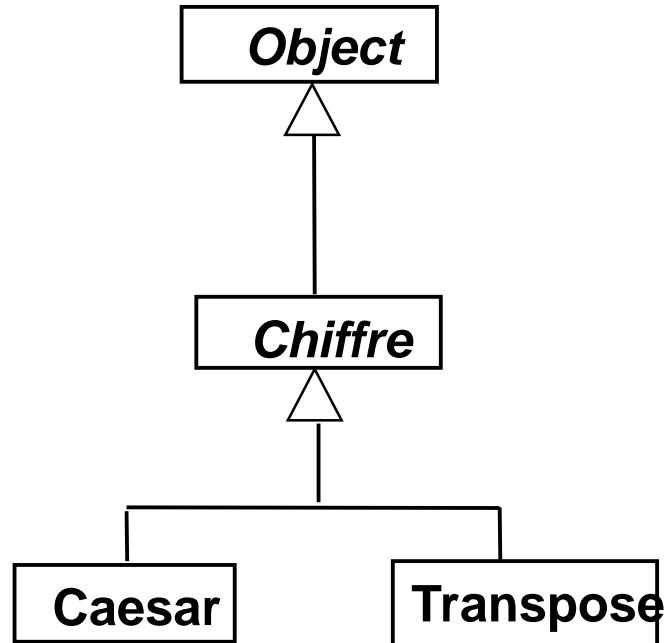
Object Design of Chiffre

- We define a super class **Chiffre** and define subclasses for the existing existing encryption methods
- 4 public methods:
 - **encrypt()** encrypts a text of words
 - **decrypt()** deciphers a text of words
 - **encode()** uses a special algorithm for encryption of a single word
 - **decode()** uses a special algorithm for decryption of a single word.



Implementation of Chiffre in Java

- The methods `encrypt()` and `decrypt()` are the same for each subclass and can therefore be *implemented* in the superclass **Chiffre**
 - **Chiffre** is defined as subclass of **Object**, because we will use some methods of **Object**
- The methods `encode()` and `decode()` are specific for each subclass
 - We therefore define them as *abstract methods* in the super class and expect that they are *implemented* in the respective subclasses.

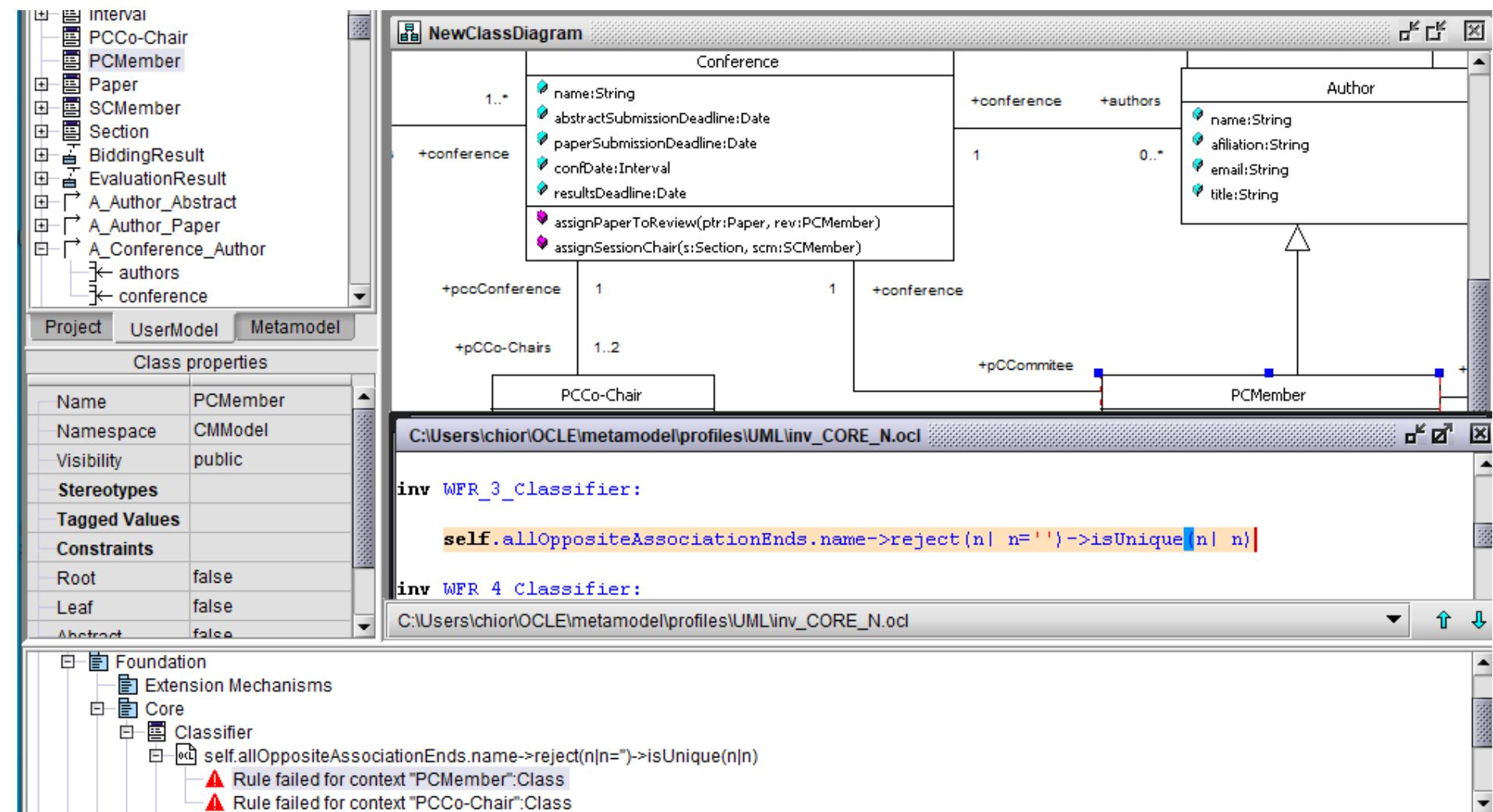


Exercise: Write
the corresponding Java
Code!

Examples of Model Transformations and Forward Engineering

- Model Transformations
 - Goal: Optimizing the object design model
 - ✓ Collapsing objects
 - ✓ Delaying expensive computations
- Forward Engineering
 - Goal: Implementing the object design model in a programming language
 - • Is it the model compilable?
 - ✓ Mapping inheritance
 - Mapping associations
 - Mapping contracts to exceptions
 - Mapping object models to tables

Is it the model compilable?



Is it the model compilable? cont.

File Model Project Edit Tools Options Help

PCMember
Paper
SCMember
Section
BiddingResult
EvaluationResult
A_Author_Abstract
A_Author_Paper
A_Conference_Author
authors
conference
A_Conference_PCCo-Chair_1
A_Conference_PCMember
conference

NewClassDiagram

The diagram shows the following associations:

- Conference** (left) has associations:
 - conference** to **PCMember** (multiplicity 1..0..*)
 - pccConference** to **PCCo-Chair** (multiplicity 1)
 - pCCo-Chairs** to **PCCo-Chair** (multiplicity 1..2)
 - conference** to **SCMember** (multiplicity 1..0..*)
- PCCo-Chair** (bottom left) has associations:
 - +pCCo-Chairs** to **Conference** (multiplicity 1..2)
- PCMember** (bottom right) has associations:
 - +pCommittee** to **PCMember** (multiplicity 0..*)
- SCMember** (right side) has associations:
 - conference** to **SCMember** (multiplicity 1..0..*)

Properties for **Conference**:

- conference**:
 - paperSubmissionDeadline**: Date
 - confDate**: Interval
 - resultsDeadline**: Date
- assignPaperToReview**(ptr: Paper, rev: PCMember)
- assignSessionChair**(s: Section, scm: SCMember)

Properties for **PCMember**:

- affiliation**: String
- email**: String
- title**: String

Project UserModel Metamodel

AssociationEnd properties

Name	conference
Stereotypes	
Tagged Values	
Constraints	
Association	A_Conference_P...
Participant	Conference
Visibility	public
Navigable	true
Multiplicity	1

C:\Users\chior\OCLE\metamodel\profiles\UML\inv_CORE_N.ncl

```
inv WFR_3_Classifier:
    self.allOppositeAssociationEnds.name->reject(n| n='')->isUnique(n| n)

inv WFR_4_Classifier:
    C:\Users\chior\OCLE\metamodel\profiles\UML\inv_CORE_N.ncl
```

Selection: Set(AssociationEnd)=Set{ conference, papers, papersToReview, pcmAbstracts, pcmSection, conference, submittedPapers, abstracts, section }

Selection: Bag(String)=Bag{ 'conference', 'papers', 'papersToReview', 'pcmAbstracts', 'pcmSection', 'conference', 'submittedPapers', 'abstracts', 'section' }

Selection: Bag(String)=Bag{ 'conference', 'papers', 'papersToReview', 'pcmAbstracts', 'pcmSection', 'conference', 'submittedPapers', 'abstracts', 'section' }

Is it the model compilable? Java Profile

ocl 2.0 - OCL Environment

File Model Project Edit Tools Options Help

Project UserModel Metamodel

AssociationEnd properties

Name	abstract
Stereotypes	
Tagged Values	
Constraints	
Association	A_Paper_Abstract
Participant	Abstract
Visibility	public
Navigable	true
Multiplicity	1

NewClassDiagram

```
classDiagram
    class A_Conference {
        +conference
        +paperSubmissionDeadline : Date
        +confDate : Interval
        +resultsDeadline : Date
        &assignPaperToReview(ptr:Paper, rev:PCMember)
        &assignSessionChair(s:Section, scm:SCMember)
    }
    class +pcoConference {
        1
    }
    class +pCCo-Chairs {
        1..2
    }
    class +pCCommittee {
        0..*
        +affiliation : String
        +email : String
        +title : String
    }
    A_Conference "1" -- "0..*" +pCCommittee : +conference
```

C:\Users\chior\OCLE\metamodell\profiles\Java\Java_Profile.ocl

```
context AssociationEnd
    inv:
        isNavigable implies [isId and javaResWord->excludes(self.name)]
```

```
context Class
    inv singleInheritance:
```

Insert 54:77 Write enabled

C:\Users\chior\OCLE\metamodell\profiles\Java\Java_Profile.ocl

Is it the model compilable? Java Profile

The screenshot shows a UML modeling environment with the following components:

- Project Explorer:** On the left, it displays a tree view of model elements. Some visible nodes include: `authors`, `conference`, `A_Conference_PCCo-Chair_1`, `A_Conference_PCMember`, `pCCommitee`, `A_Conference_Paper`, `A_Conference_SCMember`, `A_Conference_Section`, `A_PCMember_Abstract`, `A_Paper_Abstract`, `paper`, and `abstract`.
- NewClassDiagram:** This is the active workspace window. It contains a class diagram with two classes:
 - Author:** Represented by a rectangle. It has attributes: `+name:String`, `+affiliation:String`, `+email:String`, and `+title:String`. It has associations: `+authors` (multiplicity 0..*) to `conference` and `+abtracts` (multiplicity 0..*) to `Abstract`.
 - Abstract:** Represented by a rectangle. It has attributes: `+nbOfChars:Integer` and `+keyWords:String`. It has associations: `+abtracts` (multiplicity 1) to `conference` and `+pcmAbtracts` (multiplicity 0..*) to `Author`.
- Toolbars:** Standard UML modeling toolbars are visible at the top of the workspace.
- Status Bar:** The status bar at the bottom shows the path `C:\Users\chior\OCLE\metamodel\profiles\Java\Java_Profile.ocl` and the current time `54:55`.
- Properties View:** A floating properties view is open, showing the details for the selected `abstract` association end. The table includes columns for Name, Stereotypes, Tagged Values, Constraints, Association, Participant, Visibility, Navigable, and Multiplicity. The `Name` column shows `abstract`, and the `Association` column shows `A_Paper_Abstract`.
- Code Editor:** Below the workspace, there are two code editors showing OCL constraints. The top editor is titled `C:\Users\chior\OCLE\metamodel\profiles\Java\Java_Profile.ocl` and contains:

```
context AssociationEnd
inv:
    isNavigable implies (isId and javaResWord->excludes(self.name))
```



```
context Class
inv singleInheritance:
```
- Log View:** At the bottom, a log view displays several selection statements:
 - `Selection: OrderedSet(ModelElement)= OrderedSet{ Unnamed DataValue , Unnamed DataValue }`
 - `Selection: OrderedSet(ModelElement)= OrderedSet{ 1, 10, 10, 11, 12, 13, 15, 20, 2020, 2020, 21, 25, 30, 5, 5, 7, 7, 9, <undefined>, <undefined>, <undefined>, Abstract, Au }`
 - `Selection: AssociationEnd= abstract`
 - `Selection: String='abstract'`
 - `Selection: Set(String)= Set{ 'package', 'import', 'byte', 'char', 'short', 'int', 'long', 'float', 'double', 'boolean', 'void', 'class', 'interface', 'abstract', 'final', 'private', 'protected', 'pub }`

Is it the model compilable? Common behavior

The diagram shows a UML Class Diagram with two main classes: **Author** and **Abstract**. The **Author** class has attributes: name(String), affiliation(String), email(String), and title(String). It has a multiplicity of 0..* at the +authors end of the association named **ce**, which connects to the +abstracts end of the **Abstract** class. The **Abstract** class has attributes: nbOfChars(Integer) and keyWords(String). It has a multiplicity of 1 at the +abs traits end of the association named **+abs traits**, which connects to the +authors end of the **Author** class. There is also a +pcmAbstracts association from **Abstract** to **Author** with multiplicity 0..*. A stereotype **ce** is applied to the **Author** class.

Project **UserModel** **Metamodel**

LinkObject properties

Name	A_p1_pc11
Namespace	Collaboration
Visibility	public
Stereotypes	
Tagged Values	
Constraints	
Association	EvaluationResult
Connections	

C:\Users\chior\OCLE\metamodel\profiles\UML\inv_Common_Behavior.ocl

```
for each Association in which an Instance is involved, the number of opposite LinkEnds must match the multiplicity of the AssociationEnd.

classifier.allOppositeAssociationEnds->forAll( ae| ae.multiplicity.range->exists( mr |
    f.selectedLinkEnds( ae)->size >= mr.lower and (mr.upper = 2147483647 or
    .upper <> 2147483647 and self.selectedLinkEnds(ae)->size <= mr.upper.oclAsType(Integer))))
```

Once WFR 6:

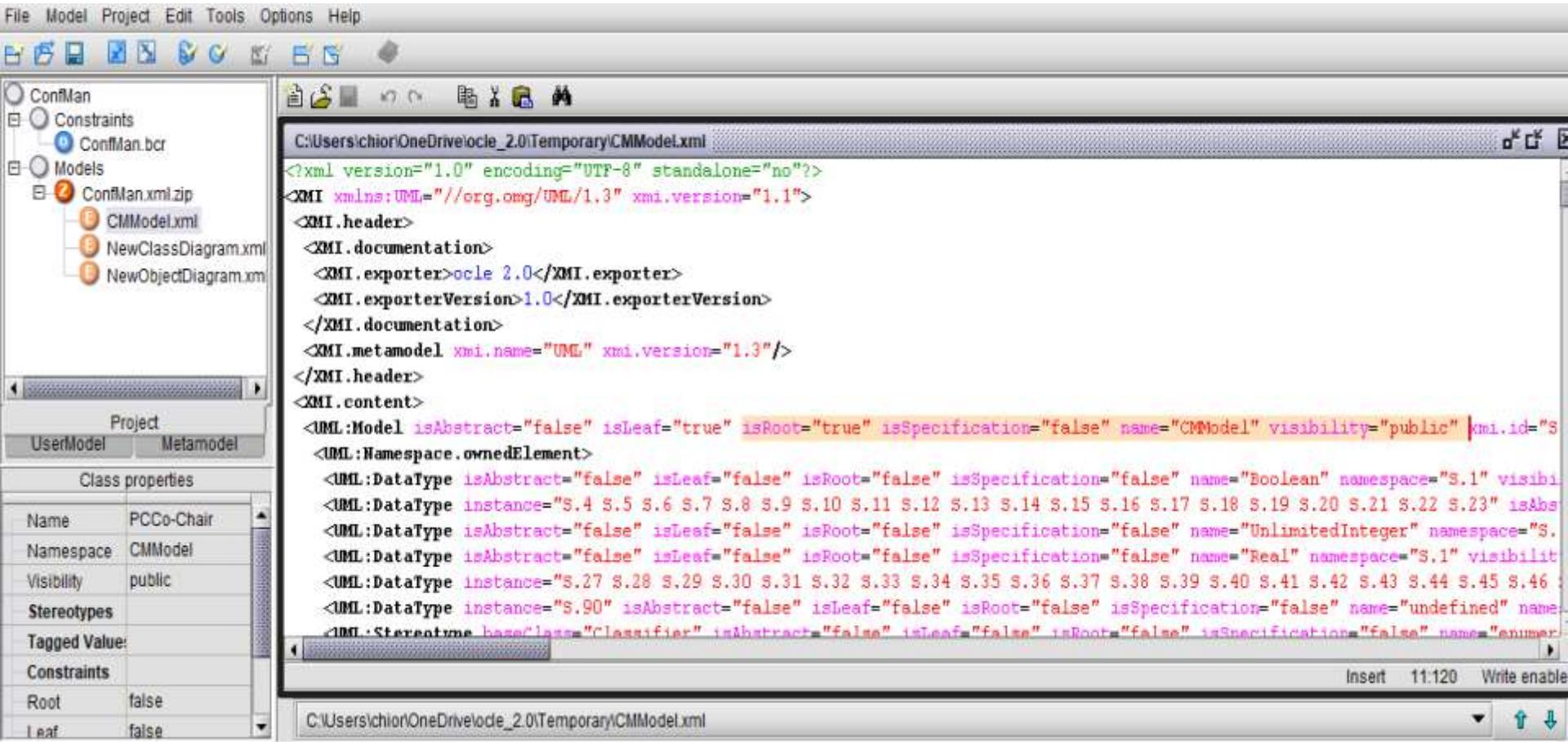
Insert 103:97 Write enabled

C:\Users\chior\OCLE\metamodel\profiles\UML\inv_Common_Behavior.ocl

Selection: Instance= [A_p1_pc11](#)
Selection: Set(Classifier)= Set{ [EvaluationResult](#) }
Selection: Bag(AssociationEnd)= Bag{ [reviewers](#), [papersToReview](#) }
Selection: Boolean= false
Selection: Boolean= false

LOG **Messages** **OCL output** **Evaluation** **Search results**

Model serialization in OCLE - CMMModel



Velocity Template Engine 1.3rc1

```
## Velocity template file for a public class declaration
## Keys:
*      classname - the name of the class, not qualified
*      packagename - the qualified name of the package where the class is declared, such as ro.ubbcluj.lci.utils
*      importstatements - the import statements list required by the class
*      modifiers - the list of modifiers applied to the class; this must always include the "class" or "interface" modifier
* but not both simultaneously
*      extclasses - the list of classes extended by the class; each class is specified using its name, which may be qualified
*      implinterfaces - the list of interfaces implemented by the class; each interface is specified using its name, which
* may be qualified
*#
/*
* ${classname}.java
*
* Generated by <a href="http://lci.cs.ubbcluj.ro/ocle/>OCLE 2.0</a>
* using <a href="http://jakarta.apache.org/velocity/">
* Velocity Template Engine 1.3rc1</a>
*/
#if (${packagename.length()} > 0)package ${packagename};
#end
#import_list(${importstatements})

/**
*
* @author unasccribed
*/
#list(" " ${modifiers}) ${classname}#if (${extclasses.size()} > 0) extends #argument_list(${extclasses})
#end#if (${implinterfaces.size()} > 0)
    implements #argument_list(${implinterfaces})#end#opening_brace()
```

Mapping Associations

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association
5. Bidirectional qualified association.

Unidirectional one-to-one association

Object design model before transformation:

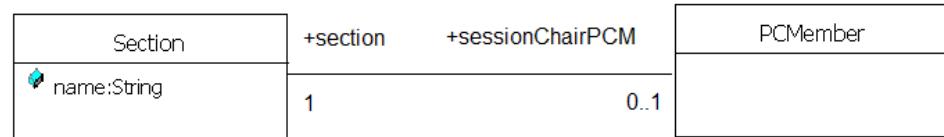


Source code after transformation:

```
public class Advertiser {  
    private Account account;  
    public Advertiser() {  
        account = new Account();  
    }  
    public Account getAccount() {  
        return account;  
    }  
}
```

The source code shows a Java class named "Advertiser". It contains a private attribute "account" of type "Account". The constructor initializes "account" to a new instance of "Account". The class also has a public method "getAccount" which returns the value of "account". Red arrows point from the "Advertiser" class name in the first line to the "Advertiser" box in the UML diagram, and from the "account" variable in the second line to the "Account" box in the UML diagram.

Bidirectional one-to-one association



```
public final PCM getSessionChairPCM() {  
    return sessionChairPCM;  
}  
  
public final void setSessionChairPCM(PCM arg) {  
  
    if (sessionChairPCM != arg) {  
        PCM temp = sessionChairPCM;  
        sessionChairPCM = null; //to avoid infinite recursion  
        if (temp != null) {  
            temp.setPcmSection(null);  
        }  
        if (arg != null) {  
            sessionChairPCM = arg;  
            arg.setPcmSection(this);  
        }  
    }  
}
```

Bidirectional one-to-one association



```
public final Section getPcmSection() {
    return pcmSection;
}

public final void setPcmSection(Section arg) {
    if (pcmSection != arg) {
        Section temp = pcmSection;
        pcmSection = null;//to avoid infinite recursion
        if (temp != null) {
            temp.setSessionChairPCM(null);
        }
        if (arg != null) {
            pcmSection = arg;
            arg.setSessionChairPCM(this);
        }
    }
}
```

Bidirectional one-to-many association



```
public class PCMember extends Author {
```

```
    public final Conference getConference() {
        return conference;
    }

    public final void setConference(Conference arg) {
        if (conference != arg) {
            Conference temp = conference;
            conference = null; //to avoid infinite recursions
            if (temp != null) {
                temp.removePCCommitee(this);
            }
            if (arg != null) {
                conference = arg;
                arg.addPCCommitee(this);
            }
        }
    }
}
```

Bidirectional one-to-many association



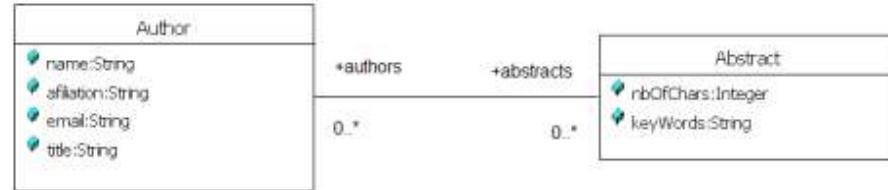
```
public final Set getAuthors() {  
    if (authors == null) {  
        return java.util.Collections.EMPTY_SET;  
    }  
    return java.util.Collections.unmodifiableSet(authors);  
}  
  
public final void addAuthors(Author arg) {  
  
    if (arg != null) {  
        if (authors == null) {  
            authors = new LinkedHashSet();  
        }  
        if (authors.add(arg)) {  
            arg.setAutConference(this);  
        }  
    }  
}
```

Bidirectional one-to-many association



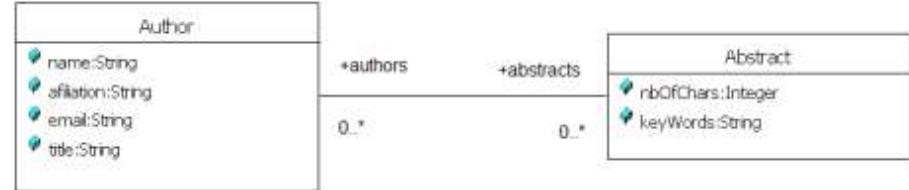
```
public final void removeAuthors(Author arg) {  
    if (authors != null && arg != null) {  
        if (authors.remove(arg)) {  
            arg.setAutConference(null);  
        }  
    }  
}
```

Bidirectional many-to-many association



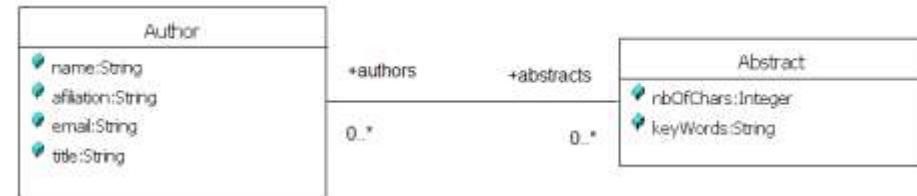
```
public final Set getAbstracts() {  
    if (abstracts == null) {  
        return java.util.Collections.EMPTY_SET;  
    }  
    return java.util.Collections.unmodifiableSet(abstracts);  
}  
  
public final void addAbstracts(Abstract arg) {  
  
    if (arg != null) {  
        if (abstracts == null) abstracts = new LinkedHashSet();  
        if (abstracts.add(arg)) {  
            arg.addAuthors(this);  
        }  
    }  
}
```

Bidirectional many-to-many association



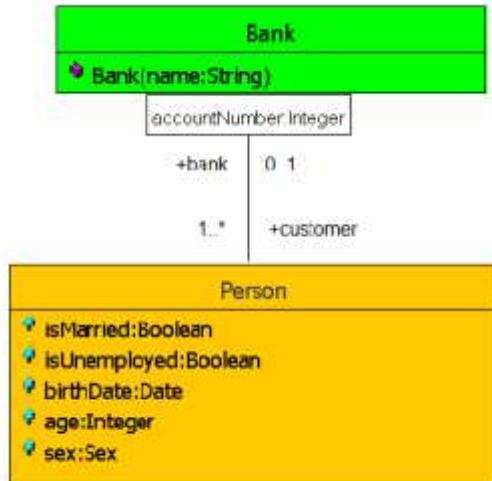
```
public final Set getAuthors() {  
  
    if (authors == null) {  
        return java.util.Collections.EMPTY_SET;  
    }  
    return java.util.Collections.unmodifiableSet(authors);  
}  
  
public final void addAuthors(Author arg) {  
  
    if (arg != null) {  
        if (authors == null) authors = new LinkedHashSet();  
        if (authors.add(arg)) {  
            arg.addAbstracts(this);  
        }  
    }  
}
```

Bidirectional many-to-many association



```
public final void removeAbstracts(Abstract arg) {  
  
    if (abstracts != null && arg != null) {  
        if (abstracts.remove(arg)) {  
            arg.removeAuthors(this);  
        }  
    }  
  
}  
  
public final void removeAuthors(Author arg) {  
  
    if (authors != null && arg != null) {  
        if (authors.remove(arg)) {  
            arg.removeAbstracts(this);  
        }  
    }  
  
}
```

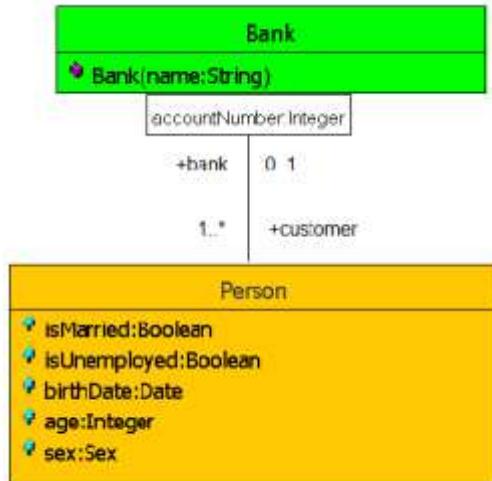
Bidirectional qualified association



```
//File Bank.java (class Bank)

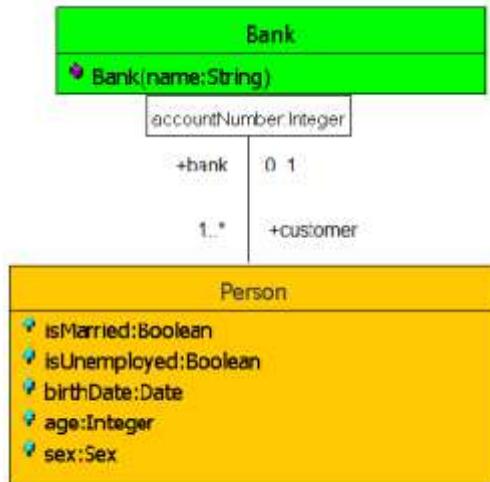
public final Set getCustomer() {
    java.util.Set temp = new LinkedHashSet();
    if (customer != null) {
        temp.addAll(customer.values());
    }
    return temp;
}
```

Bidirectional qualified association



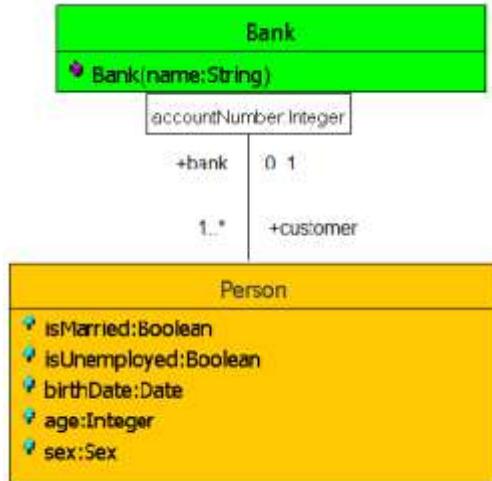
```
public final Person getCustomer(int accountNumber) {  
    if (customer == null) return null;  
    ArrayList key = new ArrayList();  
    key.add(Integer.toInteger(accountNumber));  
    return (Person)customer.get(key);  
}
```

Bidirectional qualified association



```
public final void addCustomer(int accountNumber, Person arg) {
    if (arg != null) {
        ArrayList key = new ArrayList();
        key.add(Integer.toInteger(accountNumber));
        if (customer == null) customer = new HashMap();
        Person temp = (Person)customer.put(key, arg); //the previous value, if any
        if (temp != arg) {
            arg.setBank(this);
            if (temp != null) {
                temp.setBank(null);
            }
        }
    }
}
```

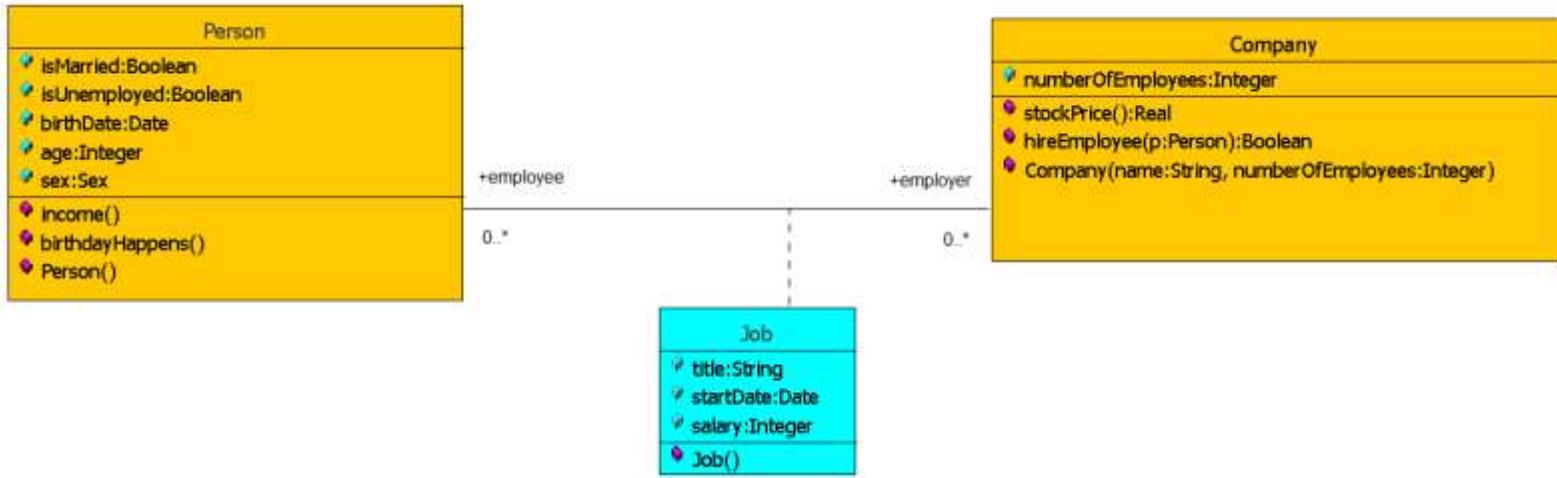
Bidirectional qualified association



```
public final void removeCustomer(int accountNumber) {
    if (customer != null) {
        ArrayList key = new ArrayList();
        key.add(Integer.toInteger(accountNumber));
        Person temp = (Person)customer.remove(key);
        if (temp != null) {
            temp.setBank(null);
        }
    }
}

public final void removeCustomer(Person arg) {
    if (customer != null || arg != null) {
        if (customer.values().remove(arg)) {
            arg.setBank(null);
        }
    }
}
```

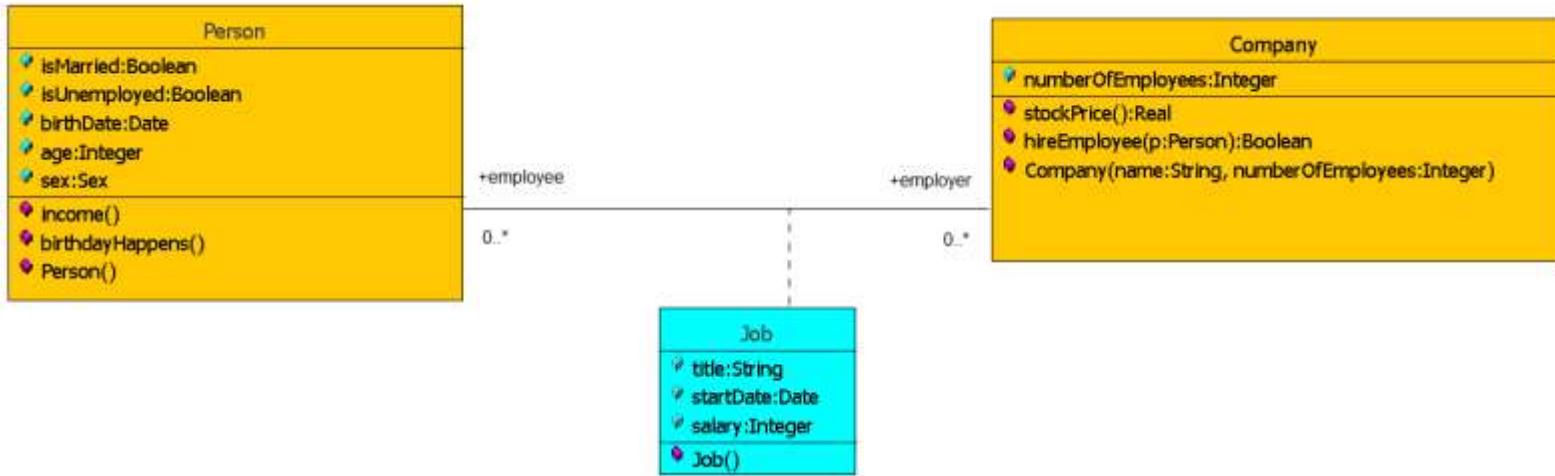
Association class



//File Person.java

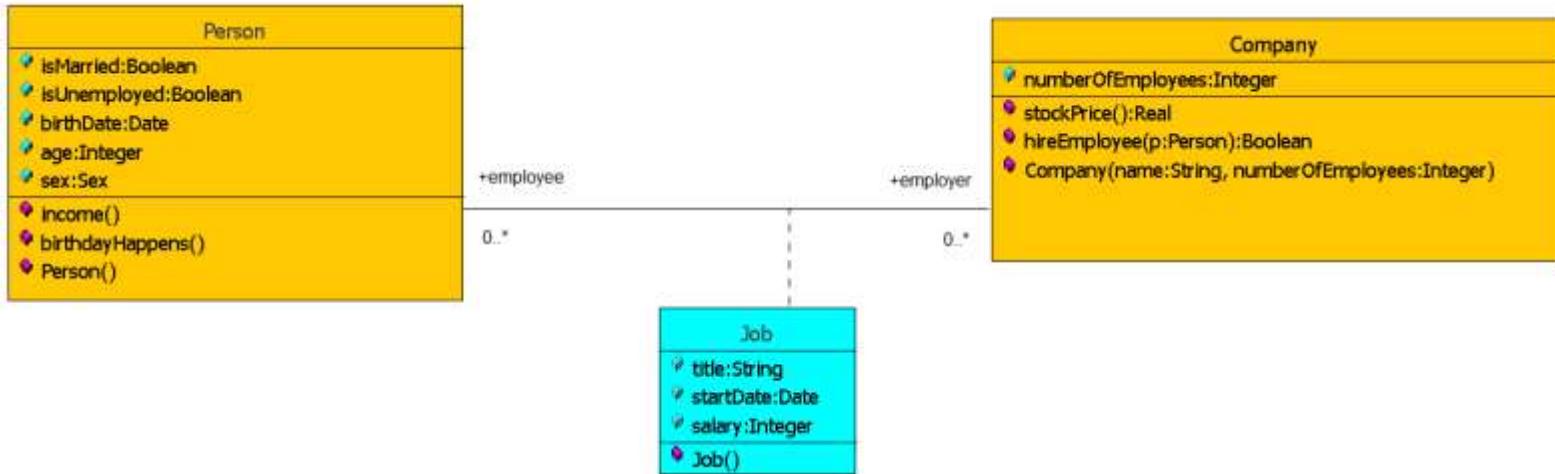
```
//the declaration for the opposite end 'employer'  
public Set employer;  
...  
public final Set getEmployer() {  
    if (employer == null) {  
        return java.util.Collections.EMPTY_SET;  
    }  
    return java.util.Collections.unmodifiableSet(employer);  
}
```

Association class



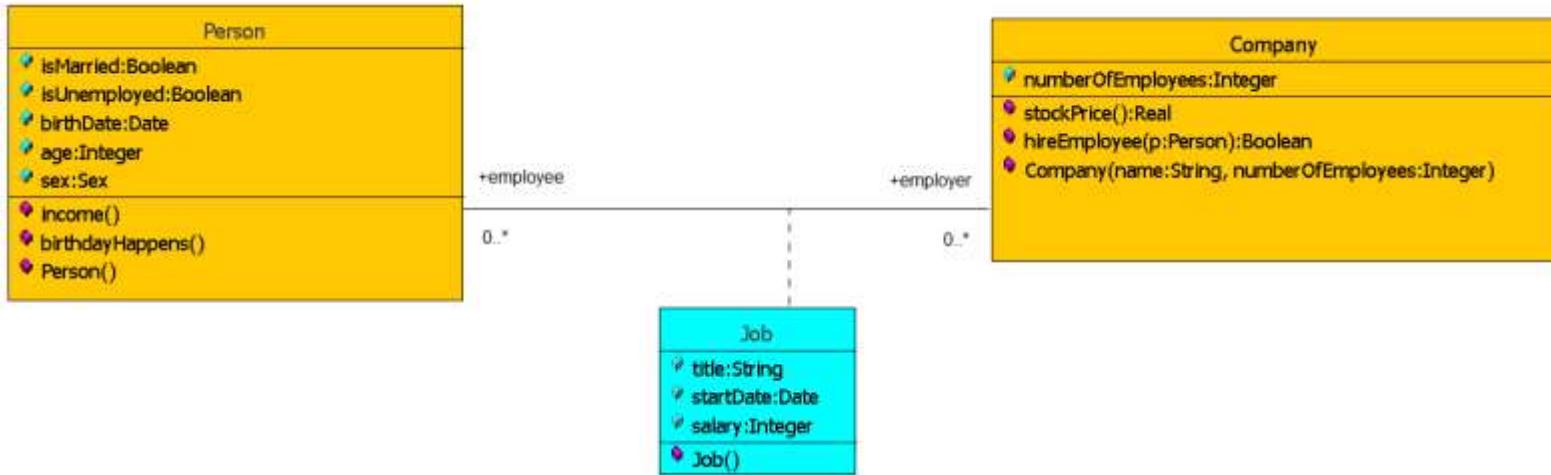
```
public final Set directGetEmployer() {
    java.util.Set temp = new LinkedHashSet();
    if (employer != null) {
        Iterator it = employer.iterator();
        while (it.hasNext()) {
            temp.add(((Job)it.next()).getEmployer());
        }
    }
    return temp;
}
```

Association class



```
public final void addEmployer(Job arg) {
    if (arg != null) {
        if (employer == null) {
            employer = new LinkedHashSet();
        }
        if (employer.add(arg)) {
            arg.setEmployee(this);
        }
    }
}
```

Association class



```
public final void removeEmployer(Job arg) {  
    if (employer != null && arg != null) {  
        if (employer.remove(arg)) {  
            arg.setEmployee(null);  
        }  
    }  
}
```

Association class

```
//File Job.java

public Company employer;
public Person employee;

public final Person getEmployee() {
    return employee;
}

public final void setEmployee(Person arg) {
    if (employee != arg) {
        Person temp = employee;
        employee = null; //to avoid infinite recursions
        if (temp != null) {
            temp.removeEmployer(this);
        }
        if (arg != null) {
            employee = arg;
            arg.addEmployer(this);
        }
    }
}
```

Examples of Model Transformations and Forward Engineering

- Model Transformations
 - Goal: Optimizing the object design model
 - ✓ Collapsing objects
 - ✓ Delaying expensive computations
- Forward Engineering
 - Goal: Implementing the object design model in a programming language
 - ✓ Mapping inheritance
 - ✓ Mapping associations
 - ➡ **Transforming observers into code**
 - Mapping contracts to exceptions
 - Mapping object models to tables

✓ Transforming observers into code

```
public Set rejectedPapersC() {  
  
    Set setAuthors = Conference.this.getAuthors();  
    //evaluate 'collect(submittedPapers)':  
    List bagCollect = CollectionUtilities.newBag();  
    final Iterator iter = setAuthors.iterator();  
    while (iter.hasNext()) {  
        final Author decl = (Author)iter.next();  
        Set setSubmittedPapers = decl.getSubmittedPapers();  
  
        bagCollect.add(setSubmittedPapers);  
    }  
    bagCollect = CollectionUtilities.flatten(bagCollect);  
  
    Set setAsSet = CollectionUtilities.asSet(bagCollect);  
    //evaluate 'select(p:Paper|Set{EvResult::strongReject,EvResult::reject,EvResult::weakReject,EvResult::borderlinePaper})'  
    Set setSelect = CollectionUtilities.newSet();  
    final Iterator iter0 = setAsSet.iterator();|
```

✓ Transforming observers into code _ cont

```
while (iter0.hasNext()) {  
    final Paper p = (Paper)iter0.next();  
    Set set = CollectionUtilities.newSet();  
    CollectionUtilities.add(set, EvResult.strongReject);  
    CollectionUtilities.add(set, EvResult.reject);  
    CollectionUtilities.add(set, EvResult.weakReject);  
    CollectionUtilities.add(set, EvResult.borderlinePaper);  
    Set setEvaluationResult = p.getEvaluationResultReviewers();  
    //evaluate 'collect(rezEv)':  
    List bagCollect0 = CollectionUtilities.newBag();  
    final Iterator iter1 = setEvaluationResult.iterator();  
    while (iter1.hasNext()) {  
        final EvaluationResult decl0 = (EvaluationResult)iter1.next();  
        EvResult evResultRezEv = decl0.rezEv;  
  
        bagCollect0.add(evResultRezEv);  
    }  
    bagCollect0 = CollectionUtilities.flatten(bagCollect0);  
  
    boolean bIncludesAll = CollectionUtilities.includesAll(set, bagCollect0);
```

Implementing Contract Violations

- Many object-oriented languages do not have built-in support for contracts
- However, if they support exceptions, we can use their exception mechanisms for signaling and handling contract violations
- In Java we use the try-throw-catch mechanism
- Example:
 - Let us assume the acceptPlayer() operation of TournamentControl is invoked with a player who is already part of the Tournament
 - UML model (see slide 34)
 - In this case acceptPlayer() in TournamentControl should throw an exception of type KnownPlayer
 - Java Source code (see slide 35).

Implementing Contract Violations - invariants

```
public class ConstraintChecker extends BasicConstraintChecker {  
  
    public void checkConstraints() {  
  
        super.checkConstraints();  
        check_PCMember_appoprPapToReview();  
        check_PCMember_sessionChair();  
  
    }  
    public void check_PCMember_sessionChair() {  
  
        Section sectionPcmSection = PCMember.this.getPcmSection();  
        boolean bIsDefined = Ocl.isDefined(sectionPcmSection);  
        boolean bNot = !bIsDefined;  
        Section sectionSection = PCMember.this.getSection();  
        Set setSectionSpeakers = sectionSection.getSectionSpeakers();  
        Author authorOclAsType = PCMember.this;  
        boolean bExcludes = CollectionUtilities.excludes(setSectionSpeakers, authorOclAsType);  
        boolean bImplies = !bNot || bExcludes;  
        if (!bImplies) {  
            System.err.println("invariant 'sessionChair' failed for object "+PCMember.this);  
        }  
    }  
}
```

Implementing Contract Violations – pre&post

```
public class Conference {  
    public void assignPaperToReview(Paper ptr, PCMember rev) {  
        class ConstraintChecker {  
            public void checkPreconditions(Paper ptr, PCMember rev) {  
                check_precondition(ptr, rev);  
            }  
            public void checkPostconditions(Paper ptr, PCMember rev) {  
                check_postcondition(ptr, rev);  
            }  
        }  
    }  
}
```

Implementing Contract Violations – pre

```
public void check_precondition(Paper ptr, PCMember rev) {  
  
    Set setReviewers = ptr.getReviewers();  
    int nSize = CollectionUtilities.size(setReviewers);  
    boolean bLessThan = nSize < 4;  
    Set setReviewers0 = ptr.getReviewers();  
    boolean bExcludes = CollectionUtilities.excludes(setReviewers0, rev);  
    boolean bAnd2 = bLessThan && bExcludes;  
    Set setsSubmittedPapers = Conference.this.submittedPapers();  
    boolean bIncludes = CollectionUtilities.includes(setsSubmittedPapers, ptr);  
    boolean bAnd1 = bAnd2 && bIncludes;  
    Set setPCCommitee = Conference.this.getPCCommitee();  
    boolean bIncludes0 = CollectionUtilities.includes(setPCCommitee, rev);  
    boolean bAnd0 = bAnd1 && bIncludes0;  
    Set set = CollectionUtilities.newSet();  
    CollectionUtilities.add(set, BiddResult.conflict);  
    CollectionUtilities.add(set, BiddResult.refuseToEv);  
    Set setBiddingResult = ptr.getBiddingResultPCMembers();  
}
```

Implementing Contract Violations – pre_2

```
//evaluate 'select(br|br.pCMembers=rev)':
Set setSelect = CollectionUtilities.newSet();
final Iterator iter = setBiddingResult.iterator();
while (iter.hasNext()) {
    final BiddingResult br = (BiddingResult)iter.next();
    PCMember pCMemberPCMembers = br.getPCMembers();
    boolean bEquals = pCMemberPCMembers.equals(rev);

    if (bEquals) CollectionUtilities.add(setSelect, br);
}

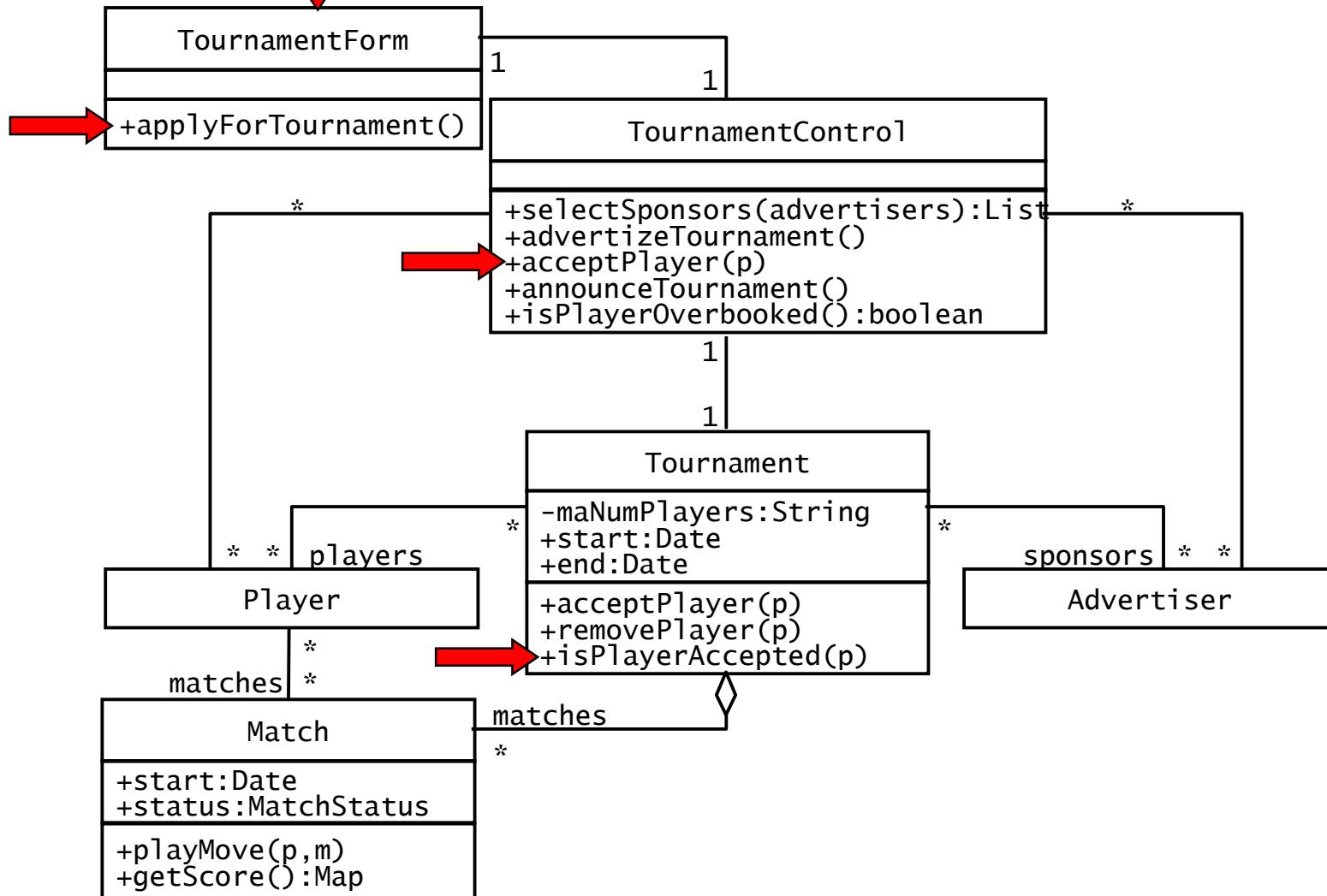
//evaluate 'any(true)':
Object temp = null;
final Iterator iter0 = setSelect.iterator();
while (temp == null && iter0.hasNext()) {
    Object temp0 = iter0.next();
    BiddingResult iter1 = (BiddingResult)temp0;

    if (true) temp = temp0;
}
```

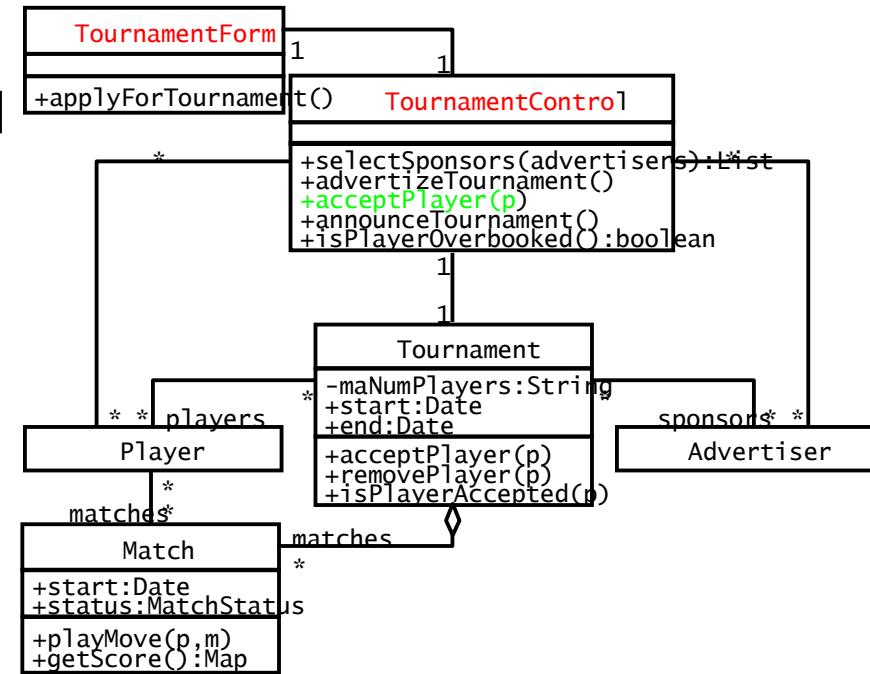
Implementing Contract Violations – pre_3

```
BiddingResult biddingResultAny;  
if (temp == null) biddingResultAny = null;  
else biddingResultAny = (BiddingResult)temp;  
  
BiddResult biddResultResBid = biddingResultAny.resBid;  
boolean bExcludes0 = CollectionUtilities.excludes(set, biddResultResBid);  
boolean bAnd = bAnd0 && bExcludes0;  
if (!bAnd) {  
    System.err.println("precondition 'precondition' failed for object "+Conference.this);  
}  
}  
}
```

UML Model for Contract Violation Example



Implementation in Java

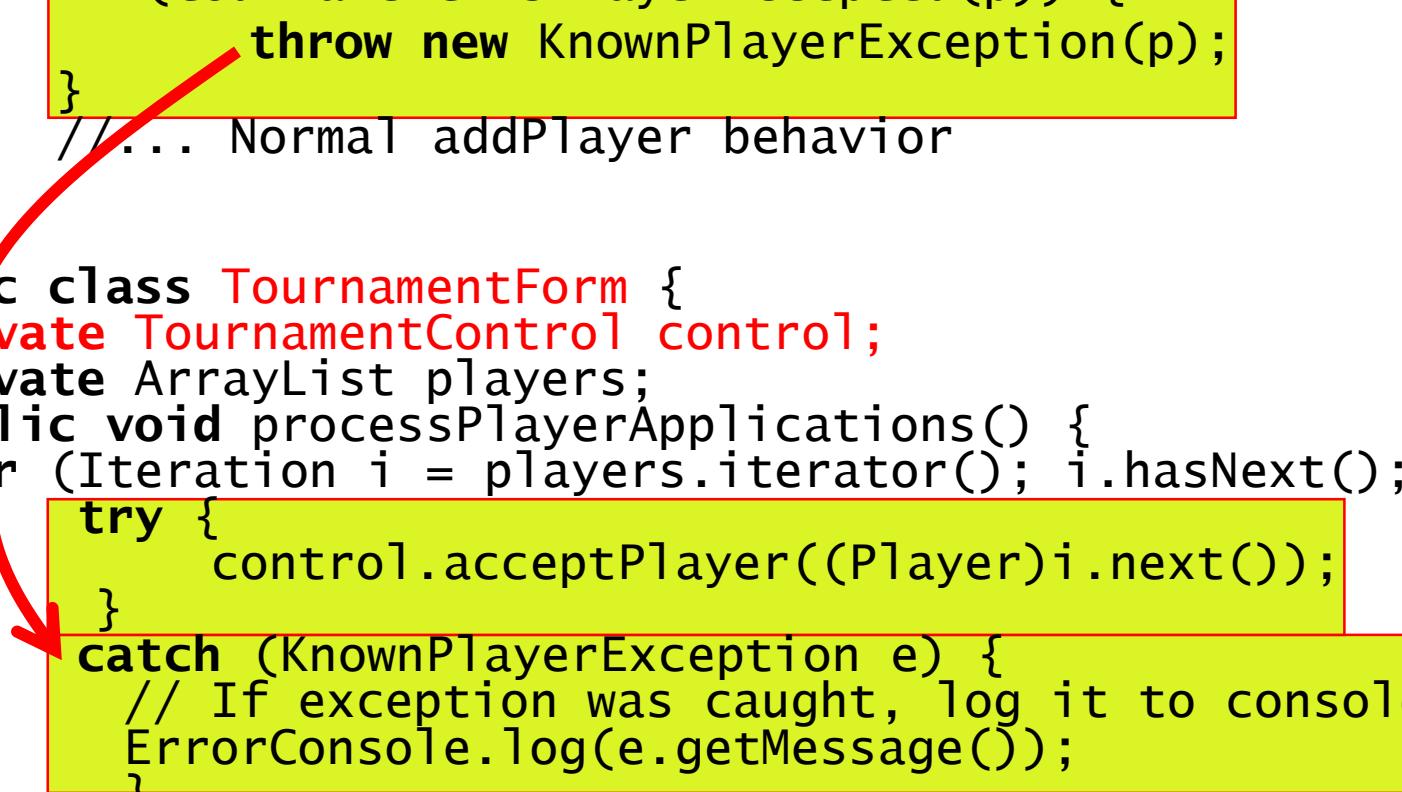


```

public class TournamentForm {
    private TournamentControl control;
    private ArrayList players;
    public void processPlayerApplications() {
        for (Iteration i = players.iterator(); i.hasNext();) {
            try {
                control.acceptPlayer((Player)i.next());
            }
            catch (KnownPlayerException e) {
                // If exception was caught, log it to console
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
  
```

The try-throw-catch Mechanism in Java

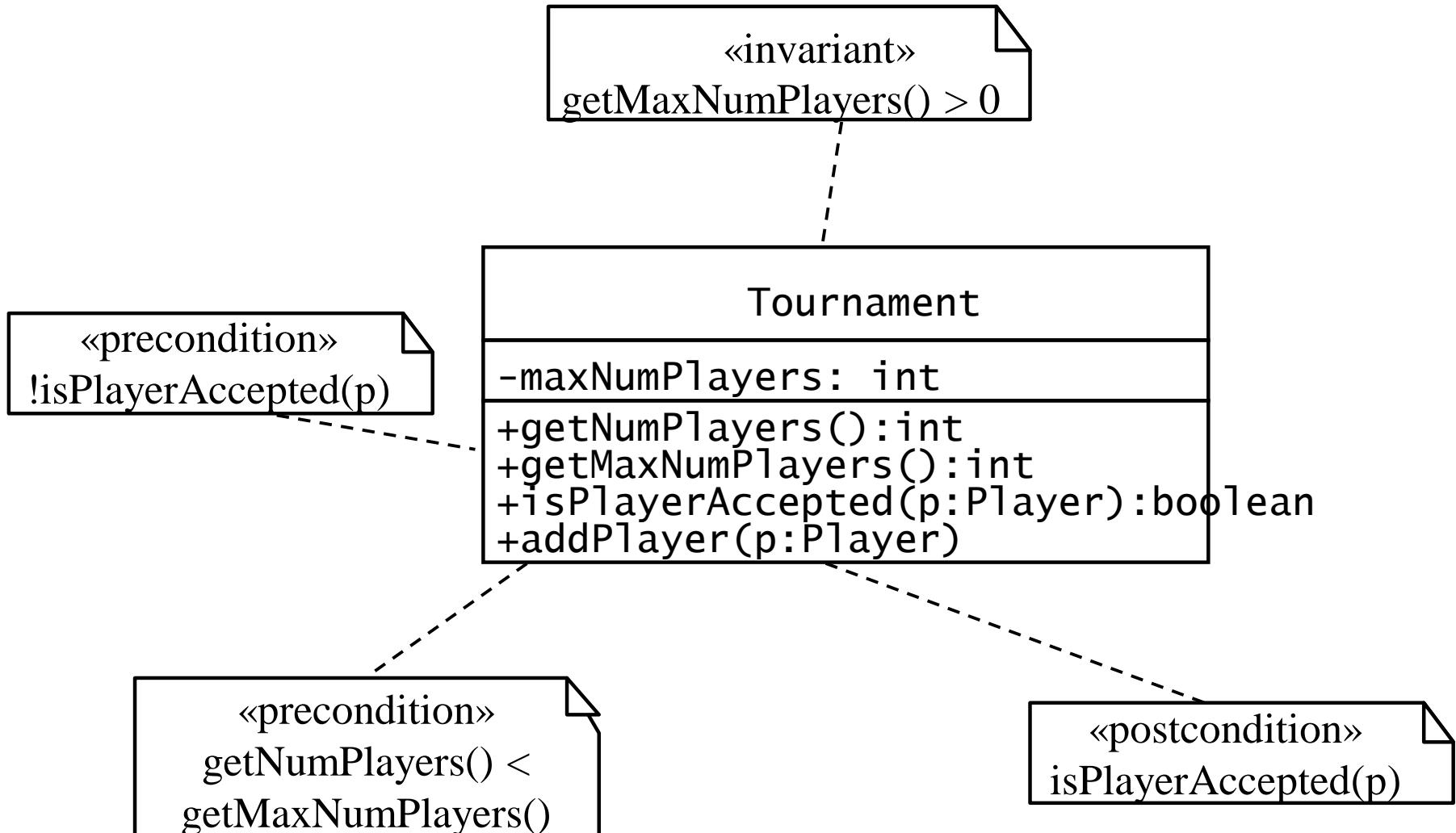
```
public class TournamentControl {  
    private Tournament tournament;  
    public void addPlayer(Player p) throws KnownPlayerException  
    {  
        if (tournament.isPlayerAccepted(p)) {  
            throw new KnownPlayerException(p);  
        }  
        //... Normal addPlayer behavior  
    }  
}  
  
public class TournamentForm {  
    private TournamentControl control;  
    private ArrayList players;  
    public void processPlayerApplications() {  
        for (Iteration i = players.iterator(); i.hasNext();) {  
            try {  
                control.acceptPlayer((Player)i.next());  
            }  
            catch (KnownPlayerException e) {  
                // If exception was caught, log it to console  
                ErrorConsole.log(e.getMessage());  
            }  
        }  
    }  
}
```



Implementing a Contract

- **Check each precondition:**
 - Before the beginning of the method with a test to check the precondition for that method
 - Raise an exception if the precondition evaluates to false
- **Check each postcondition:**
 - At the end of the method write a test to check the postcondition
 - Raise an exception if the postcondition evaluates to false. If more than one postcondition is not satisfied, raise an exception only for the first violation.
- **Check each invariant:**
 - Check invariants at the same time when checking preconditions and when checking postconditions
- **Deal with inheritance:**
 - Add the checking code for preconditions and postconditions also into methods that can be called from the class.

A complete implementation of the Tournament.addPlayer() contract



Heuristics: Mapping Contracts to Exceptions

- Executing checking code slows down your program
 - If it is too slow, omit the checking code for private and protected methods
 - If it is still too slow, focus on components with the longest life
 - Omit checking code for postconditions and invariants for all other components.

Heuristics for Transformations

- For any given transformation always use the same tool
- Keep the contracts in the source code, not in the object design model
- Use the same names for the same objects
- Have a style guide for transformations (Martin Fowler)

Object Design Areas

1. Service specification

- Describes precisely each class interface

2. Component selection

- Identify off-the-shelf components and additional solution objects

3. Object model restructuring

- Transforms the object design model to improve its understandability and extensibility

4. Object model optimization

- Transforms the object design model to address performance criteria such as response time or memory utilization.

Design Optimizations

- Design optimizations are an important part of the object design phase:
 - The requirements analysis model is semantically correct but often too inefficient if directly implemented.
- Optimization activities during object design:
 1. Add redundant associations to minimize access cost
 2. Rearrange computations for greater efficiency
 3. Store derived attributes to save computation time
- As an object designer you must strike a balance between efficiency and clarity.
 - Optimizations will make your models more obscure

Design Optimization Activities

1. Add redundant associations:

- What are the most frequent operations? (Sensor data lookup?)
- How often is the operation called? (30 times a month, every 50 milliseconds)

2. Rearrange execution order

- Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)
- Narrow search as soon as possible
- Check if execution order of loop should be reversed

3. Turn classes into attributes

Implement application domain classes

- To collapse or not collapse: Attribute or association?
- Object design choices:
 - Implement entity as embedded attribute
 - Implement entity as separate class with associations to other classes
- Associations are more flexible than attributes but often introduce unnecessary indirection
- Abbott's textual analysis rules.

To Collapse or not to Collapse?

- Collapse a class into an attribute if the only operations defined on the attributes are Set() and Get().

Design Optimizations (continued)

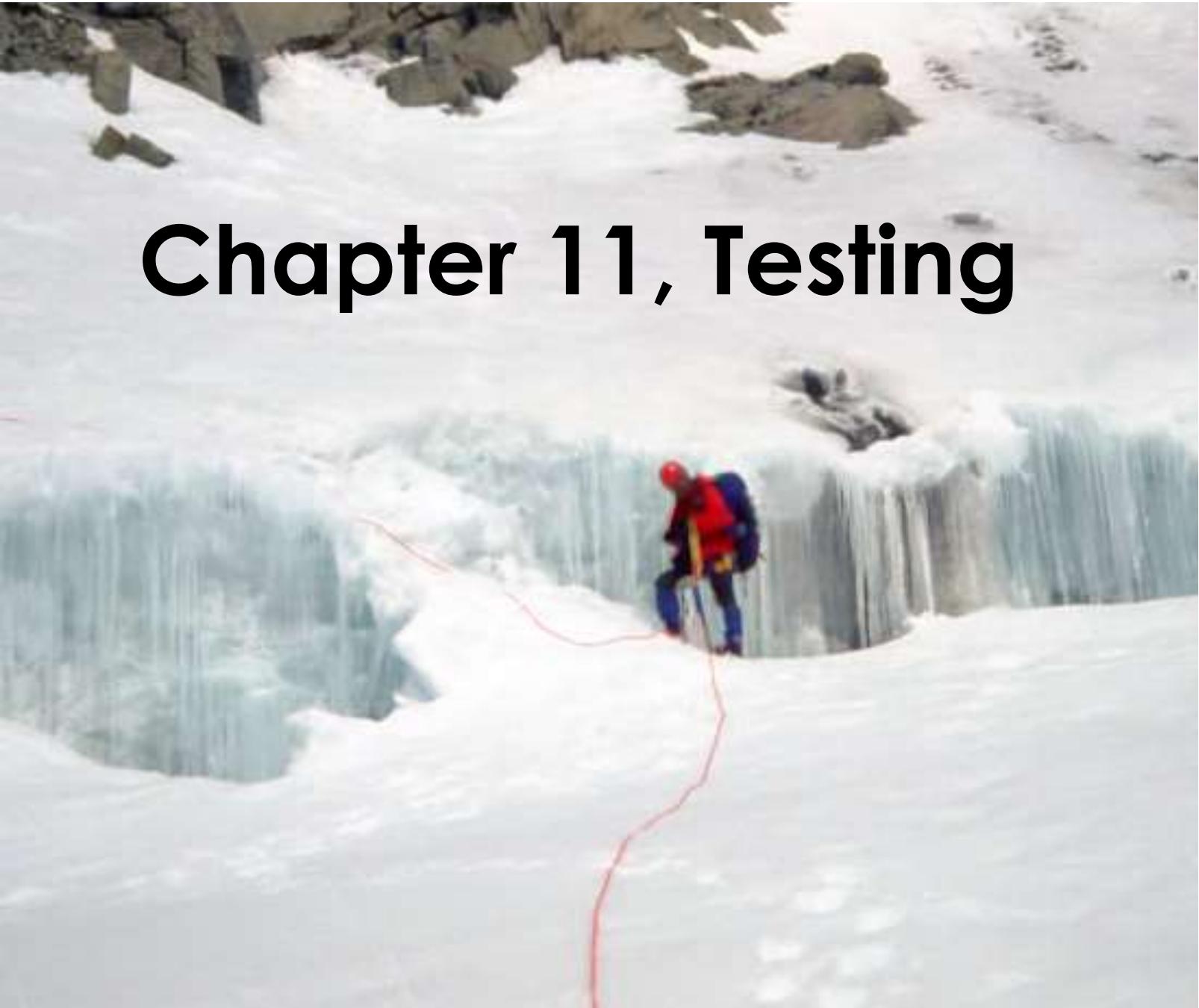
Store derived attributes

- Example: Define new classes to store information locally (database cache)
- Problem with derived attributes:
 - Derived attributes must be updated when base values change.
 - There are 3 ways to deal with the update problem:
 - **Explicit code:** Implementor determines affected derived attributes (push)
 - **Periodic computation:** Recompute derived attribute occasionally (pull)
 - **Active value:** An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 11, Testing



Testing – the problem

Testing is the **process of finding differences between the expected behavior** specified by system models **and the observed behavior** of the implemented system.

- **Unit testing** finds **differences between a specification of an object and its realization** as a component.
- **Structural testing** finds **differences between the system design model and a subset of integrated subsystems**.
- **Functional testing** finds **differences between the use case model and the system**.

Testing – the problem_2

Performance testing finds differences between nonfunctional requirements and actual system performance.

When differences are found, developers identify the defect causing the observed failure and modify the system to correct it. In other cases, the system model is identified as the cause of the difference, and the system model is updated to reflect the system.

The goal of testing is to design tests that exercise defects in the system and to reveal problems. This activity is contrary to all other activities which are constructive activities.

Testing – the problem_3

Unfortunately, it is impossible to completely test a nontrivial system.

- First, **testing is not decidable**.
- Second, **testing must be performed under time and budget constraints**.

As a result, **systems are often deployed without being completely tested, leading to faults discovered by end users**.

Famous Problems

- F-16 : crossing equator using autopilot
 - Result: plane flipped over
 - Reason?
 - Reuse of autopilot software



- The Therac-25 accidents (1985-1987), quite possibly the most serious non-military computer-related failure ever in terms of human life (at least five died)
 - Reason: Bad event handling in the GUI
- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
 - Reason: Unit conversion problem.

Testing – perception

Testing is often viewed as a job that can be done by beginners.

- Managers would assign the new members to the testing team, because the experienced people detested testing or are needed for the more important jobs of analysis and design.

Unfortunately, such an attitude leads to many problems.

- A tester must have a detailed understanding of the whole system, ranging from the requirements to system design decisions and implementation issues.
- A tester must also be knowledgeable of testing techniques and apply these techniques effectively and efficiently to meet time, budget, and quality constraints.

Testing – overview

Reliability is a **measure of success** with which **the observed behavior** of a system **conforms to the specification of its behavior**.

- **Software reliability** is **the probability that a software system will not cause system failure** for a specified time under specified conditions [IEEE Std. 982.2- 1988].
- **Failure** is any **deviation of the observed behavior from the specified behavior**.
- An **erroneous state/error** means the **system is in a state such that further processing by the system will lead to a failure**, which then causes the system to deviate from its intended behavior.

Testing – overview_2

- A **fault/defect/bug**, is the **mechanical or algorithmic cause of an erroneous state**.

The **goal of testing** is to **maximize the number of discovered faults**, which then allows developers to correct them and increase the reliability of the system.

We define **testing** as **the systematic attempt to find faults in a planned way in the implemented software**.

Contrast this definition with another common one:
“testing is the process of demonstrating that faults are not present.”

Testing activities

Test planning allocates resources and schedules the testing.

This activity should occur early in the development phase so that sufficient time and skill is dedicated to testing. Developers can design test cases as soon as the models they validate become stable.

- **Usability testing** tries to **find faults in the user interface design** of the system. Often, systems fail to accomplish their intended purpose simply because their users are confused by the user interface and unwillingly introduce erroneous data.
- **Unit testing** tries to **find faults in participating objects and/or subsystems** with respect to the use cases from the use case model.

Testing activities_2

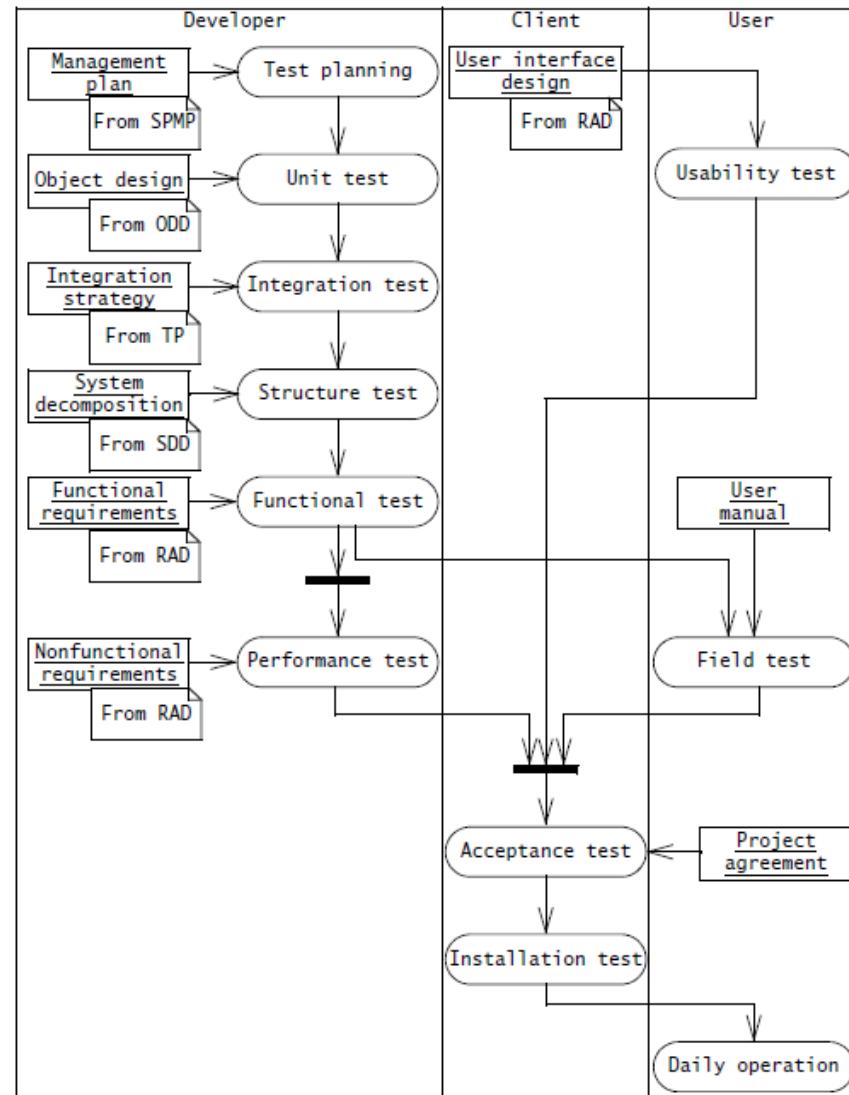
- **Integration testing** is the activity of finding faults by testing individual components in combination. **Structural testing** is the culmination of integration testing involving all components of the system. **Integration tests** and **structural tests** exploit knowledge from the **SDD (System Design Document)** using an integration strategy described in the **Test Plan (TP)**.
- **System testing** tests all the components together, seen as a single system to identify faults with respect to the scenarios from the problem statement and the requirements and design goals identified in the analysis and system design, respectively:

Testing activities_3

- **Functional testing** tests the requirements from the RAD and the user manual.
- **Performance testing** checks the **nonfunctional requirements** and **additional design goals from the SDD**. Functional and performance testing are done by developers.
- **Acceptance testing** and **installation testing** check the system against the project agreement and is done by the client, if necessary, with help by the developers.

Testing activities and assoc. stakeholders

SPMP - Software Project Management Plan
TP - Test Plan
SDD - System Design Document
RAD - Requirements Analysis Document



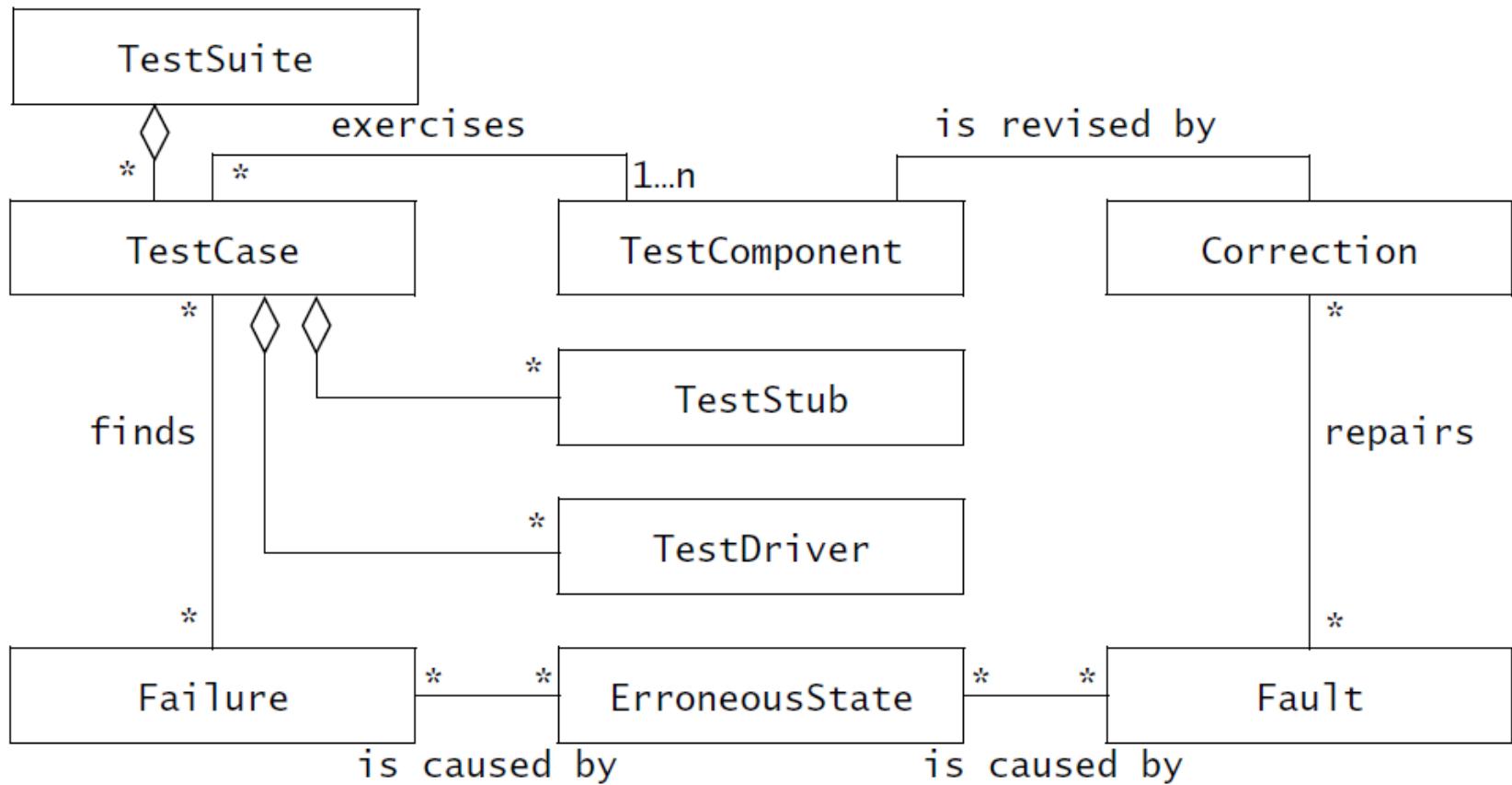
Testing concepts

- **Test component** is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.
- A **fault/bug/defect**, is a **design or coding mistake** that may cause **abnormal component behavior**.
- An **erroneous state** is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.
- A **failure** is a **deviation between the specification and the actual behavior**. A failure is triggered by one or more erroneous states. Not all erroneous states trigger a failure.

Testing concepts_2

- A **test case** is a **set of inputs and expected results** that exercises a test component with the purpose of causing failures and detecting faults.
- A **test stub** is a **partial implementation of components on which the tested component depends**.
- A **test driver** is a **partial implementation of a component that depends on the test component**. **Test stubs** and **drivers** enable components to be isolated from the rest of the system for testing.
- A **correction** is a **change to a component**. The **purpose of a correction is to repair a fault**. Note that a correction can introduce new faults.

Model elements used during testing



Attributes of the class TestCase

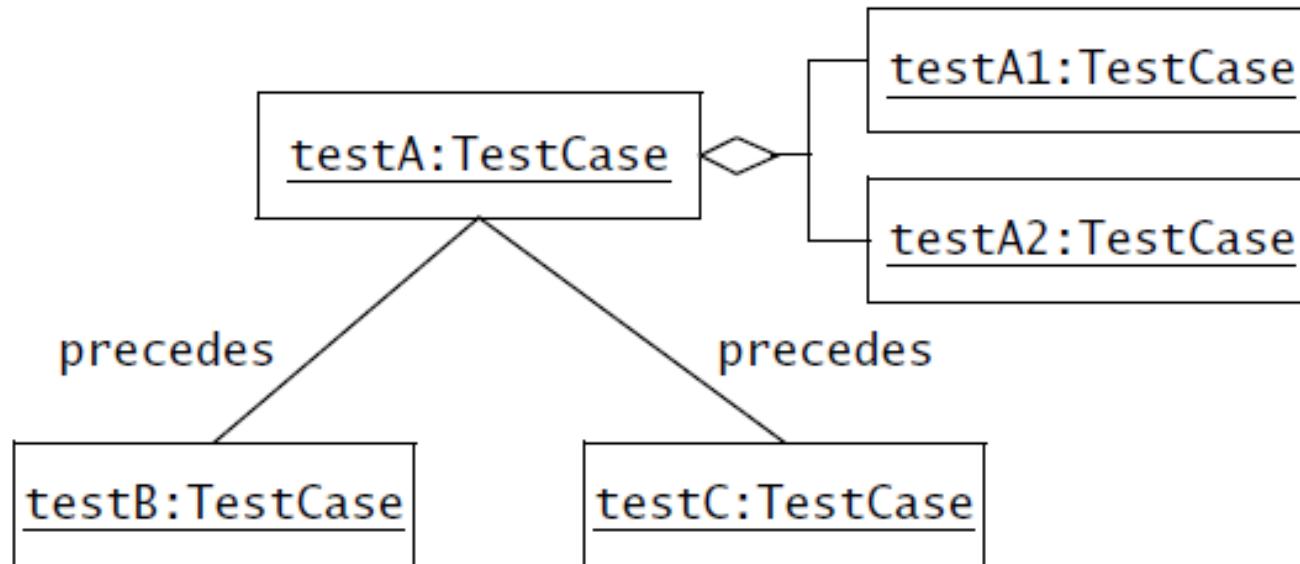
A **test case** is a **set of input data and expected results** that exercises a component with the purpose of causing failures and detecting faults. A test case has **five attributes**: **name**, **location**, **input**, **oracle**, and **log**

Attributes	Description
name	Name of test case
location	Full path name of executable
input	Input data or commands
oracle	Expected test results against which the output of the test is compared
log	Output produced by the test

Test model width test cases

Once test cases are identified and described, relationships among test cases are identified.

Aggregation and the **precede associations** are used to describe the relationships between the test cases. Aggregation is used when a test case can be decomposed into a set of subtests. Two test cases are related via the precede association when one test case must precede another test case.



Test model width test cases

Test cases are classified into **blackbox tests** and **whitebox tests**:

Blackbox tests focus on the **input/output behavior** of the component. **Blackbox tests do not deal with the internal aspects of the component**, nor with the behavior or the structure of the components.

Whitebox tests focus on the internal structure of the component. A **whitebox test** makes sure that, independently from the particular input/output behavior, **every state in the dynamic model of the object and every interaction among the objects is tested**.

Test model width test cases_2

As a result, **whitebox testing goes beyond blackbox testing**. In fact, most of the whitebox tests require input data that could not be derived from a description of the functional requirements alone.

Unit testing combines both testing techniques: **blackbox testing** to test the functionality of the component, and **whitebox testing** to test structural and dynamic aspects of the component.

Executing test cases on single components or combinations of components **requires** the **tested component to be isolated from the rest of the system**. **Test drivers** and **test stubs** are used to substitute for missing parts of the system.

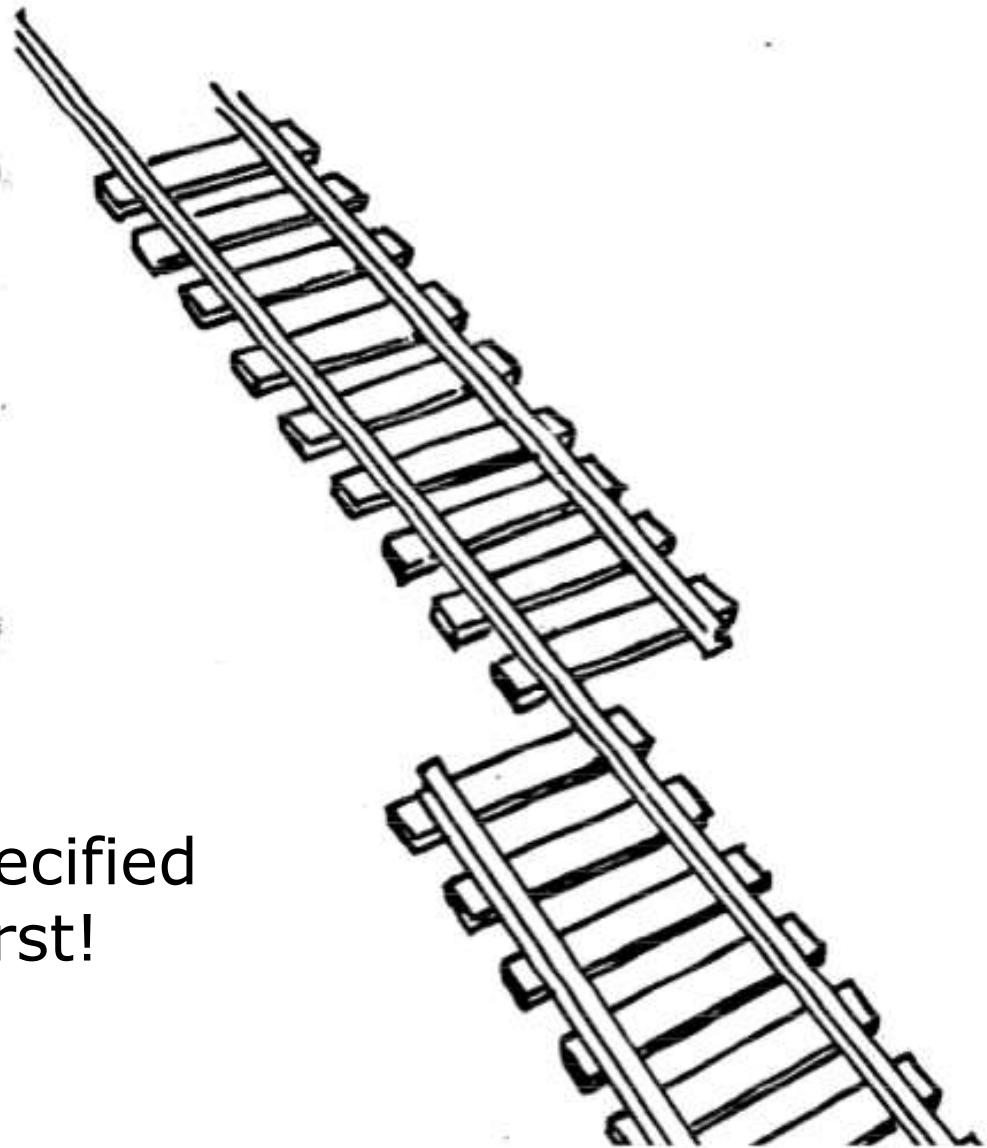
What is this?

A failure?

An error?

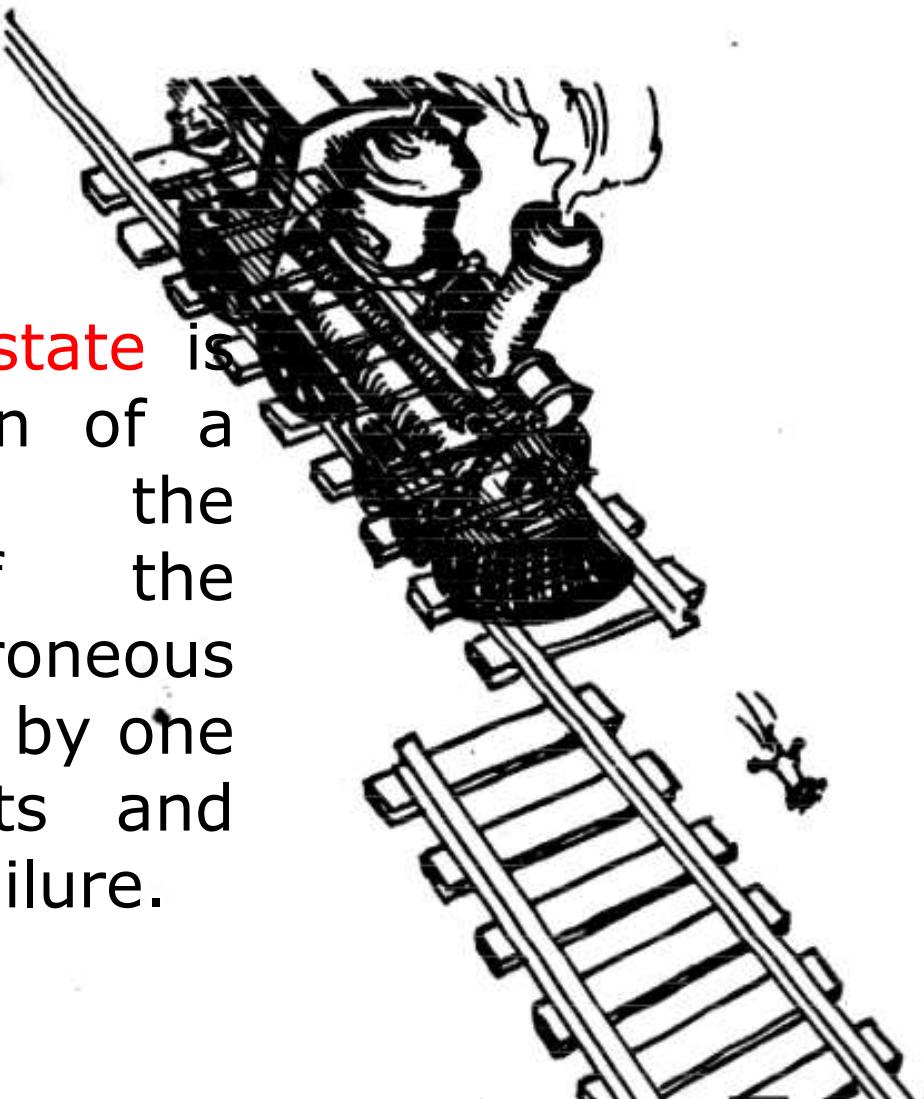
A fault?

We need to describe specified
and desired behavior first!

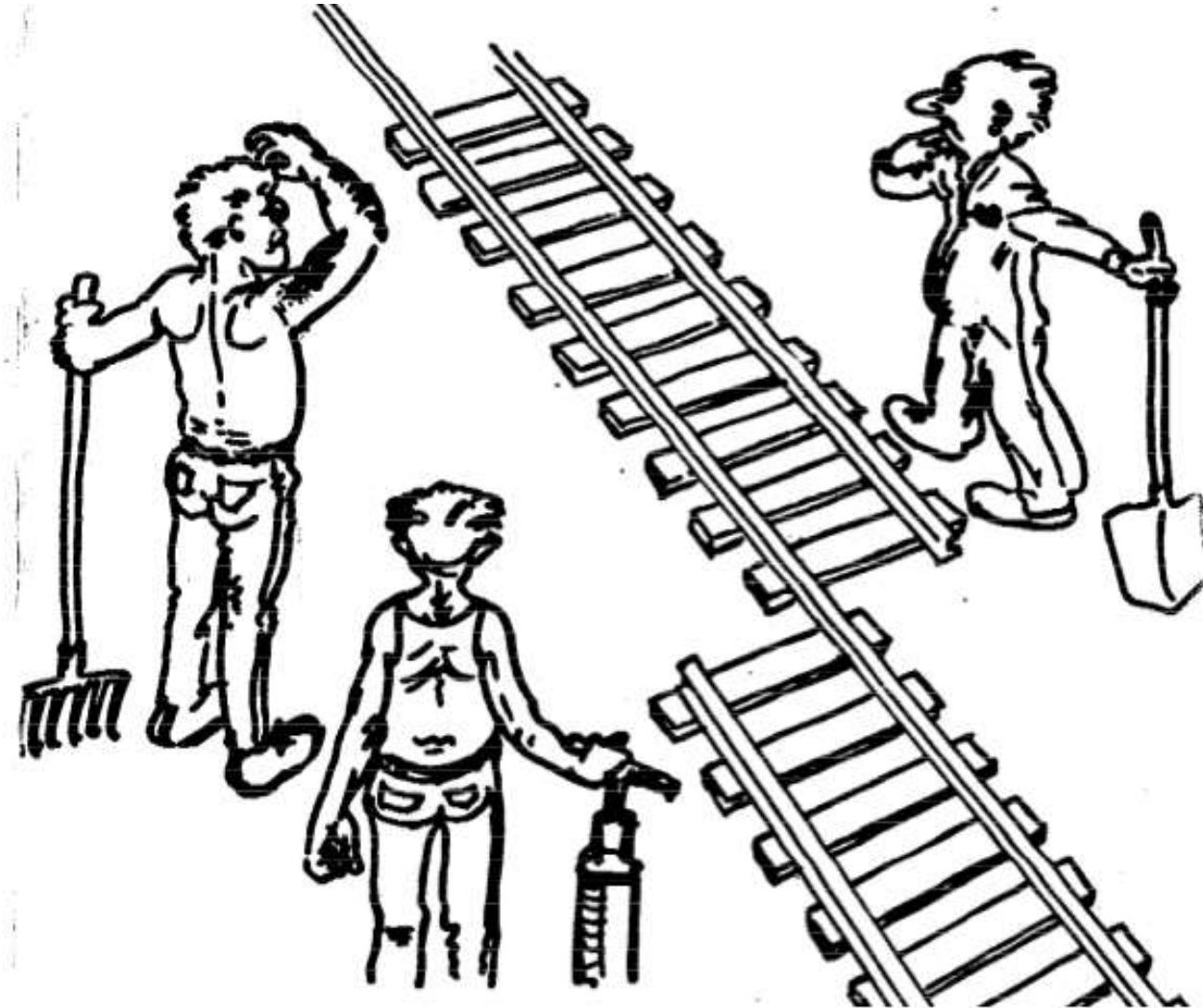


Erroneous State (“Error”)

An **erroneous state** is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.

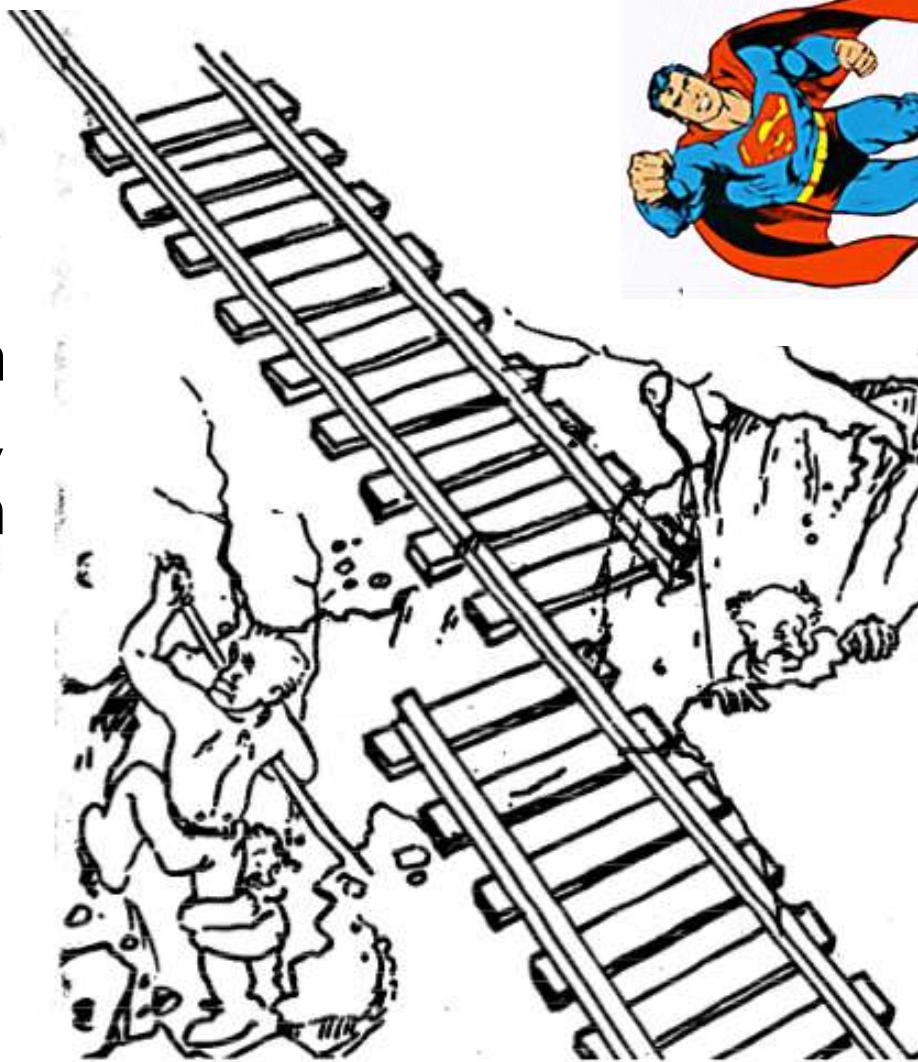


Algorithmic Fault



Mechanical Fault

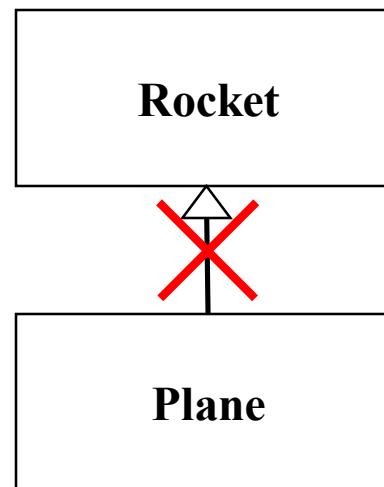
A fault can have a mechanical cause, such as an earthquake



F-16 Bug



- What is the failure?
- What is the error?
- What is the fault?
 - Bad use of implementation inheritance
 - A Plane is **not** a rocket.



Examples of Faults and Errors

- **Faults in the Interface specification**
 - Mismatch between what the client needs and what the server offers
 - Mismatch between requirements and implementation
- **Algorithmic Faults**
 - Missing initialization
 - Incorrect branching condition
 - Missing test for null
- **Mechanical Faults (very hard to find)**
 - Operating temperature outside of equipment specification
- **Errors**
 - Null reference errors
 - Concurrency errors
 - Exceptions.

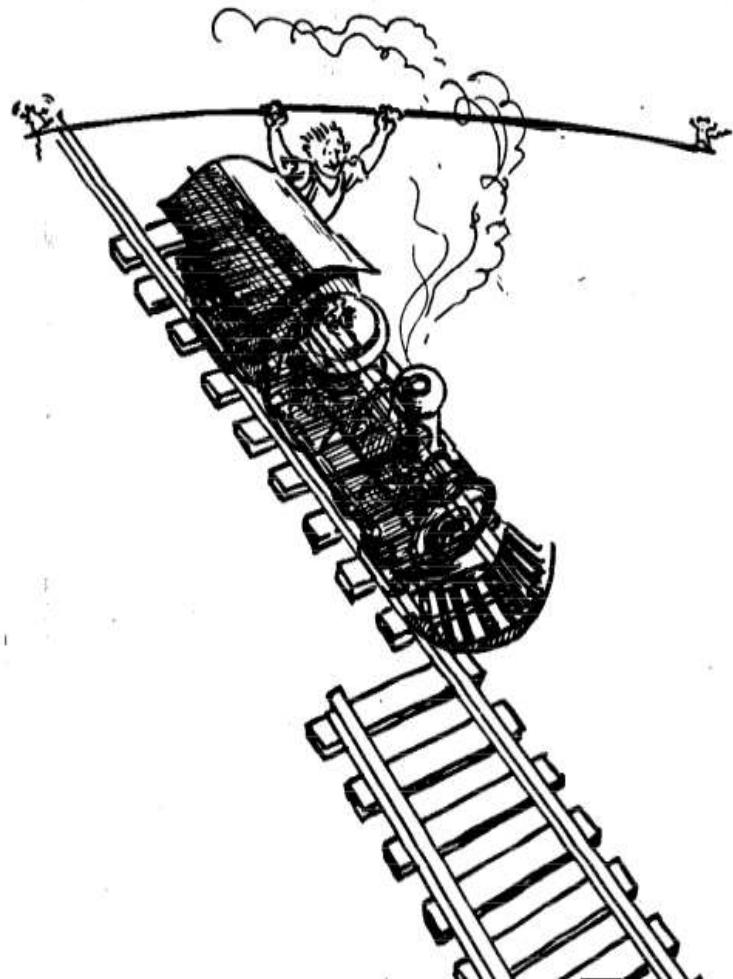
How do we deal with Errors, Failures and Faults?

Modular Redundancy



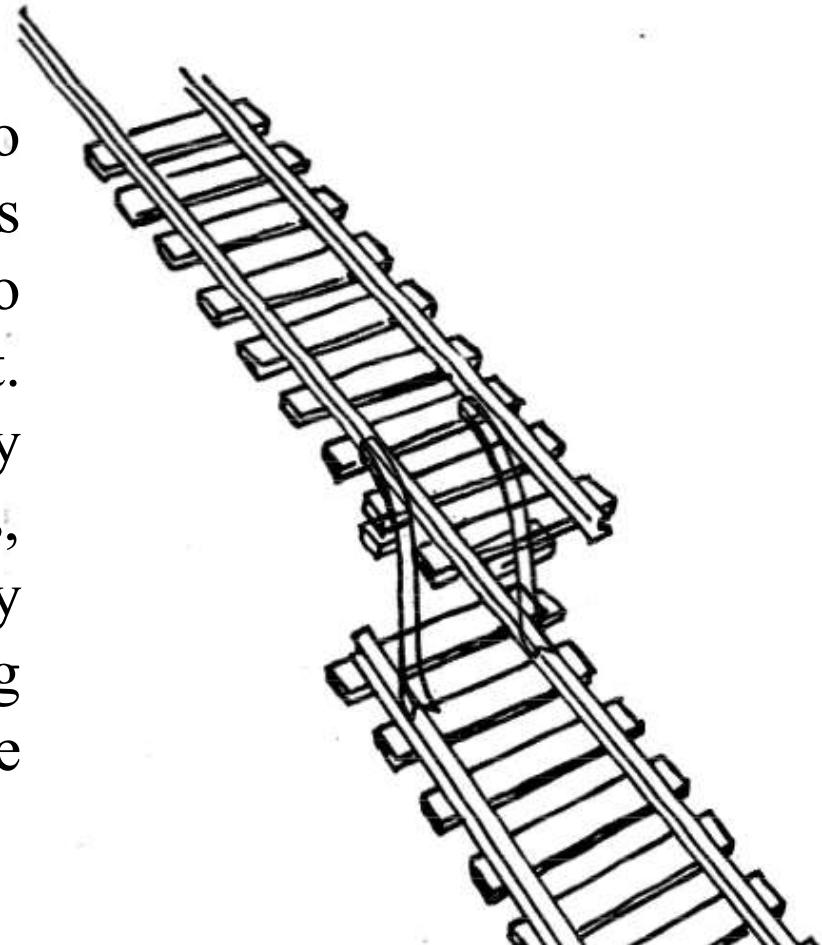
The Saturn rocket that launched the Apollo spacecraft used triple modular redundancy for the guidance system; that is, each of the components in the computer was tripled. The failure of a single component was detected when it produced a different output than the other two.

Declaring the Bug as a Feature

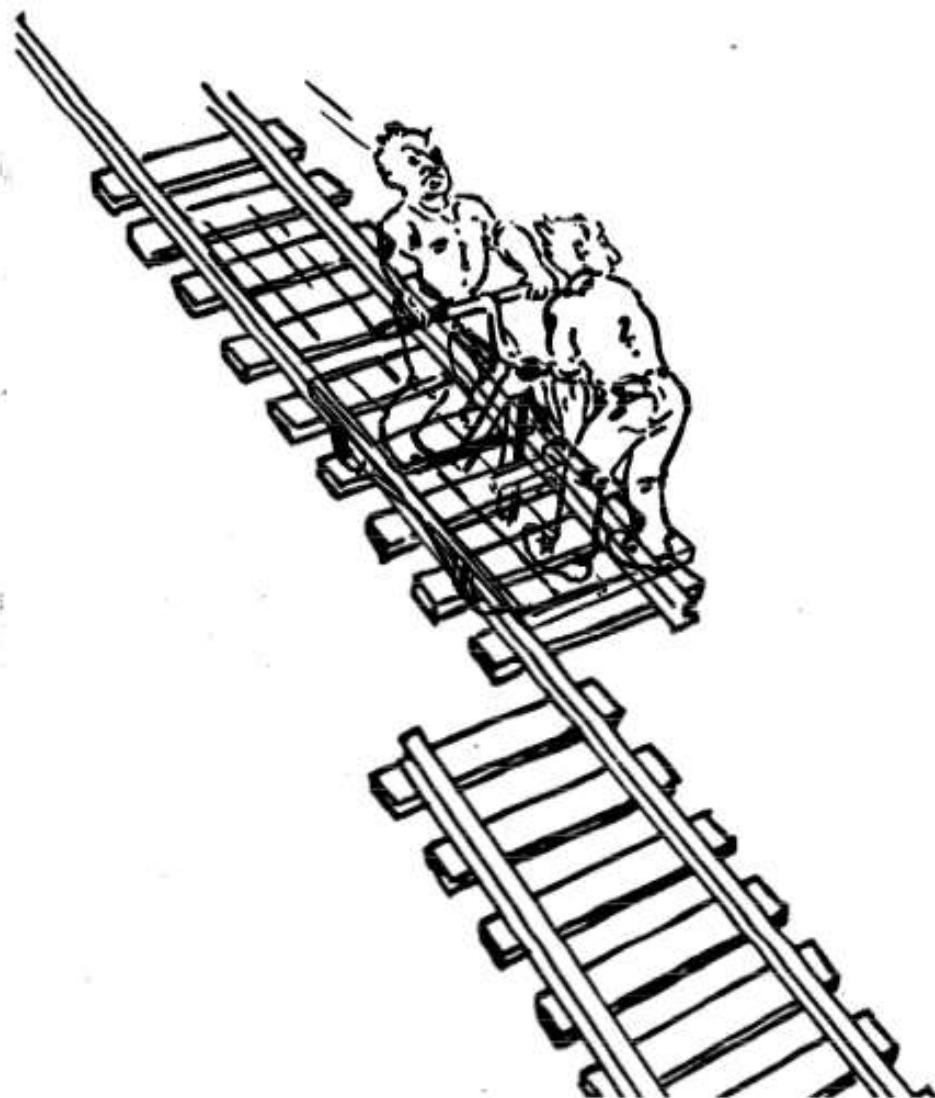


Patching

A **patch** is a set of changes to a **computer** program or its supporting data designed to update, fix, or improve it. This includes fixing security vulnerabilities and other bugs, with such **patches** usually being called bugfixes or bug fixes, and improving the usability or performance.



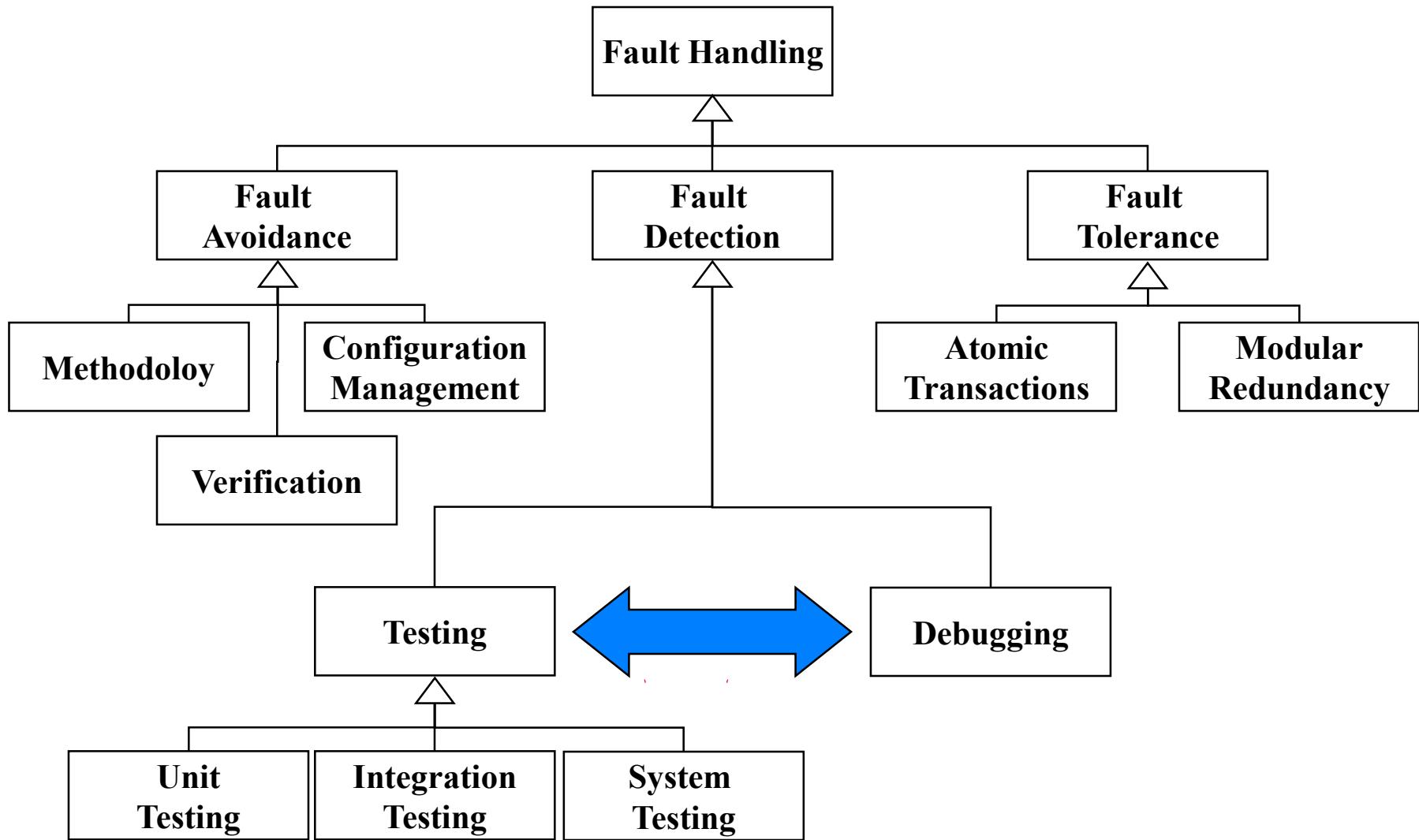
Testing



Another View on How to Deal with Faults

- **Fault avoidance (before the system release)**
 - Use methodology to reduce complexity
 - Use configuration management to prevent inconsistency
 - Apply verification to prevent algorithmic faults
 - Use Reviews
- **Fault detection (while system is running)**
 - **Testing**: Activity to provoke failures in a planned way
 - **Debugging**: Find and remove the cause (Faults) of an observed failure
 - **Monitoring**: Deliver information about state => Used during debugging
- **Fault tolerance (recover from failure after release)**
 - Exception handling
 - Modular redundancy.

Taxonomy for Fault Handling Techniques



Observations

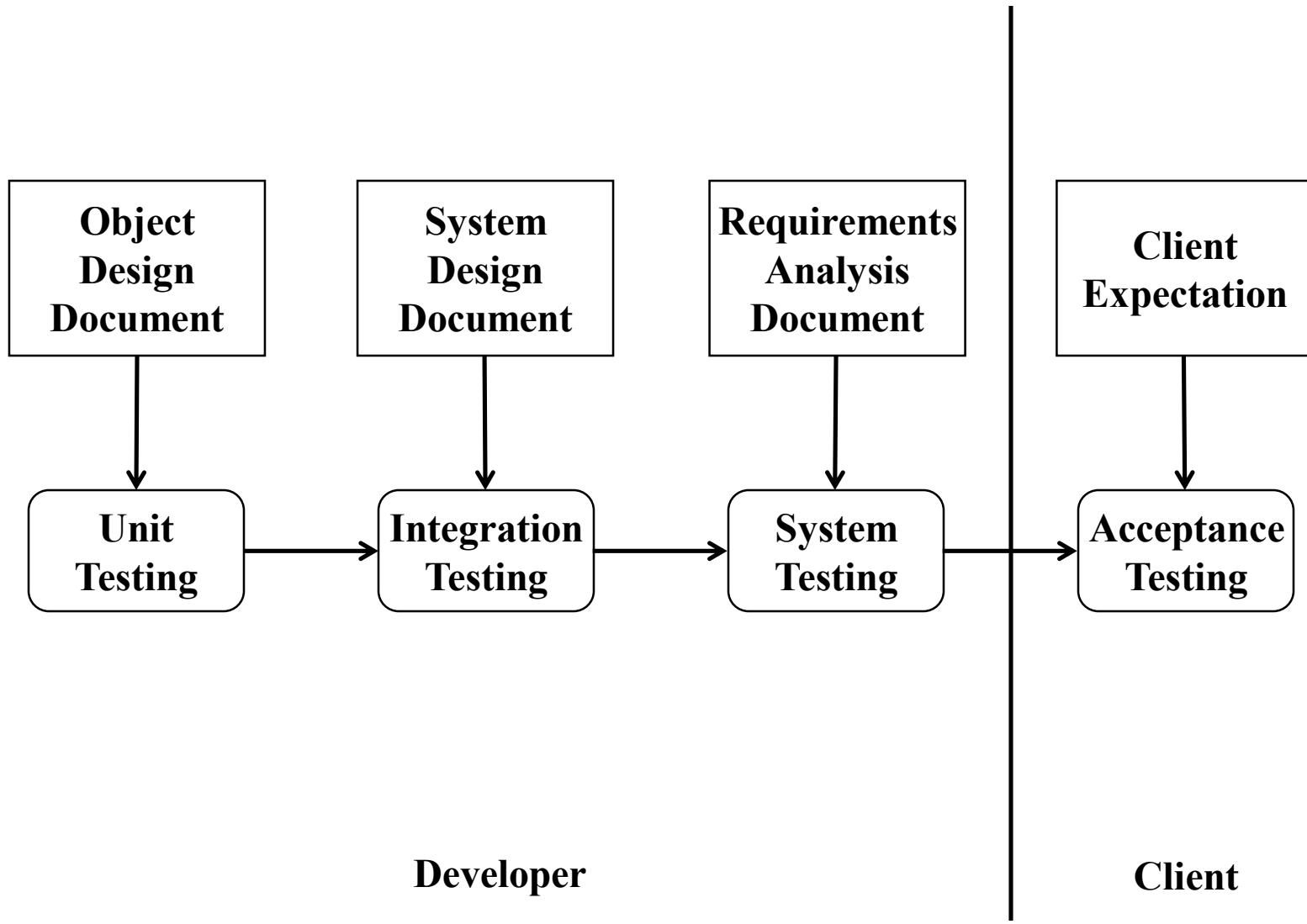
- It is impossible to completely test any nontrivial module or system
 - Practical limitations: Complete testing is prohibitive in time and cost
 - Theoretical limitations: e.g. Halting problem
- “**Testing can only show the presence of bugs, not their absence**” (**Dijkstra**).
- **Testing is not for free**

=> **Define your goals and priorities**

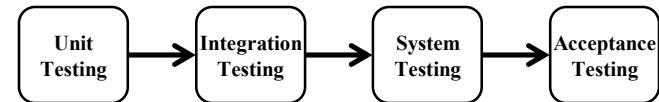
Testing takes creativity

- To develop an effective test, one must have:
 - Detailed understanding of the system
 - Application and solution domain knowledge
 - Knowledge of the testing techniques
 - Skill to apply these techniques
- Testing is done best by independent testers
 - We often develop a certain mental attitude that the program should **behave** in a certain way when in fact it does not
 - Programmers often stick to the data set that makes the program work
 - A program often does not work when tried by somebody else.

Testing Activities

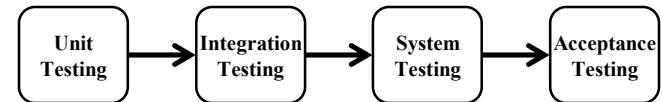


Types of Testing



- **Unit Testing**
 - Individual component (class or subsystem)
 - Carried out by developers
 - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- **Integration Testing**
 - Groups of subsystems (collection of subsystems) and eventually the entire system
 - Carried out by developers
 - Goal: Test the interfaces among the subsystems.

Types of Testing continued...



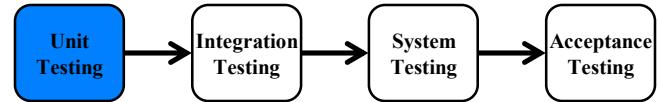
- **System Testing**
 - The entire system
 - Carried out by developers
 - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- **Acceptance Testing**
 - Evaluates the system delivered by developers
 - Carried out by the client. May involve executing typical transactions on site on a trial basis
 - Goal: Demonstrate that the system meets the requirements and is ready to use.

When should you write a test?

- Traditionally after the source code is written
- In XP before the source code written
 - Test-Driven Development Cycle
 - Add a test
 - Run the automated tests
 - => see the new one fail
 - Write some code
 - Run the automated tests
 - => see them succeed
 - Refactor code.

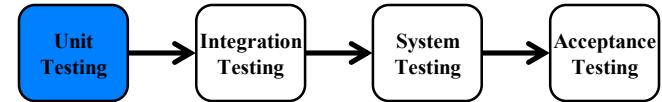


Unit Testing

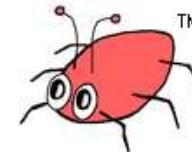


- Static Testing (at compile time)
 - Static Analysis
 - Review
 - Walk-through (informal)
 - Code inspection (formal)
- Dynamic Testing (at run time)
 - Black-box testing
 - White-box testing.

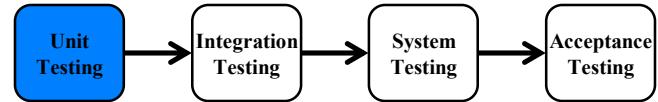
Static Analysis with Eclipse



- Compiler Warnings and Errors
 - *Possibly uninitialized Variable*
 - *Undocumented empty block*
 - *Assignment has no effect*
- Checkstyle
 - Check for code guideline violations
 - <http://checkstyle.sourceforge.net>
- FindBugs
 - Check for code anomalies
 - <http://findbugs.sourceforge.net>
- Metrics
 - Check for structural anomalies
 - <http://metrics.sourceforge.net>



Black-box testing



- Focus: I/O behavior
 - If for any given input, we can predict the output, then the component passes the test
 - Requires test oracle
- Goal: Reduce number of test cases by equivalence partitioning:
 - Divide input conditions into equivalence classes
 - Choose test cases for each equivalence class.

Black-box testing: Test case selection

a) Input is valid across range of values

- Developer selects test cases from 3 equivalence classes:
 - Below the range
 - Within the range
 - Above the range

b) Input is only valid, if it is a member of a discrete set

- Developer selects test cases from 2 equivalence classes:
 - Valid discrete values
 - Invalid discrete values
- No rules, only guidelines.

Black box testing: An example

```
public class MyCalendar {  
  
    public int getNumDaysInMonth(int month, int year)  
        throws InvalidMonthException  
    { ... }  
}
```

Representation for **month**:

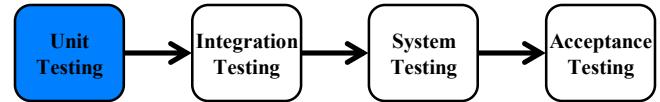
1: January, 2: February,, 12: December

Representation for **year**:

1904, ... 1999, 2000,..., 2006, ...

How many test cases do we need for the black box testing of
`getNumDaysInMonth()`?

White-box testing overview



- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

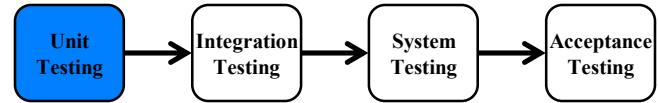
=> Details in the exercise session about testing

Unit Testing Heuristics

1. Create unit tests when object design is completed
 - Black-box test: Test the functional model
 - White-box test: Test the dynamic model
2. Develop the test cases
 - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
 - Don't waste your time!

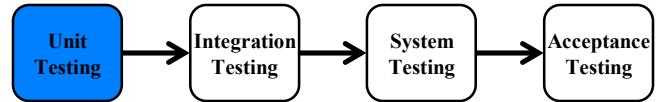
4. Double check your source code
 - Sometimes reduces testing time
5. Create a test harness
 - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
 - Often the result of the first successfully executed test
7. Execute the test cases
 - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
 - Automate this if possible.

JUnit: Overview



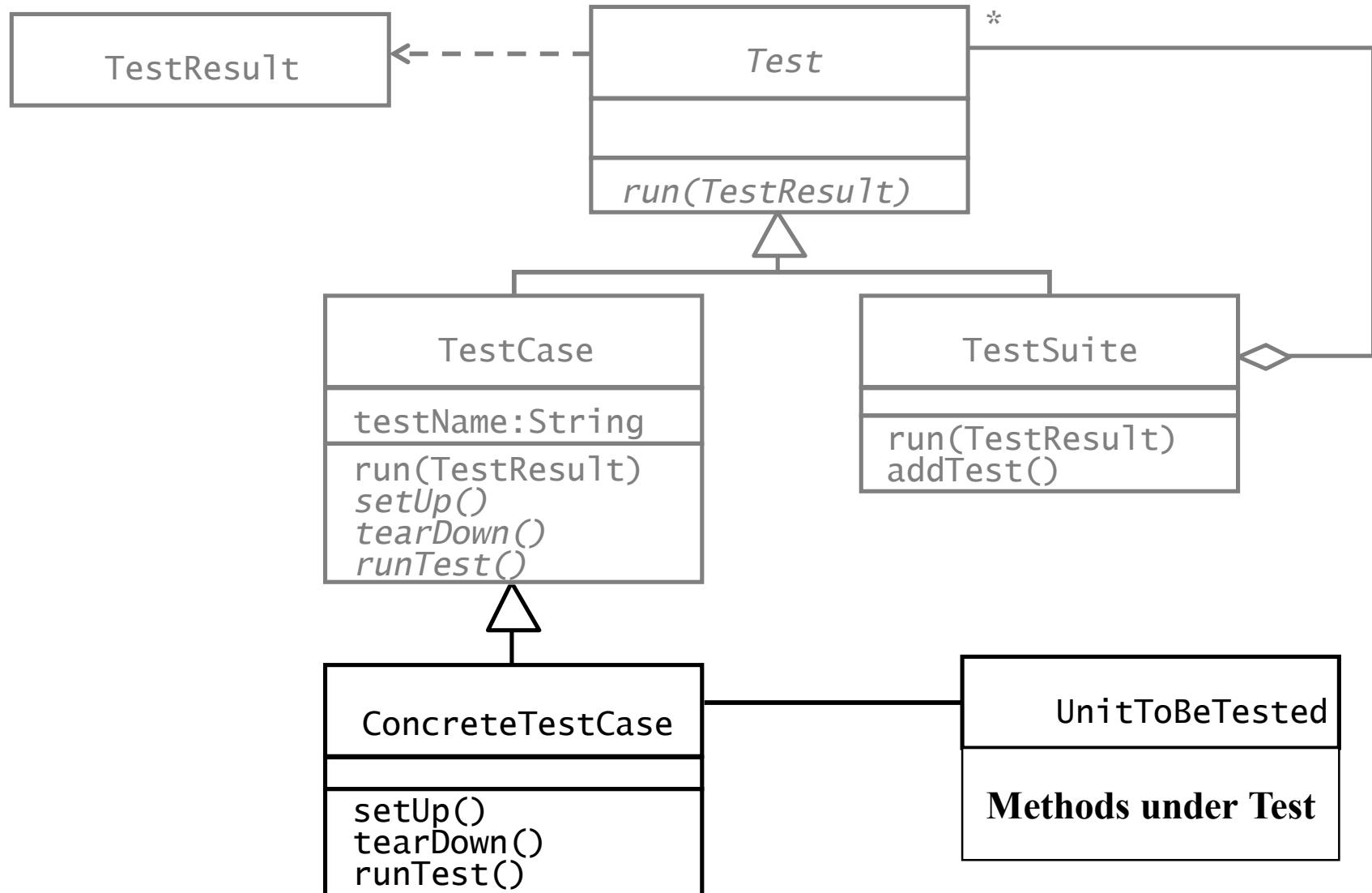
- A Java framework for writing and running unit tests
 - Test cases and fixtures
 - Test suites
 - Test runner
- A **test fixture** (also known as a **test context**) is the set of preconditions or state needed to run a **test**. The developer should set up a known good state before the **tests**, and return to the original state after the **tests**
- Written by Kent Beck and Erich Gamma

JUnit: Overview_2



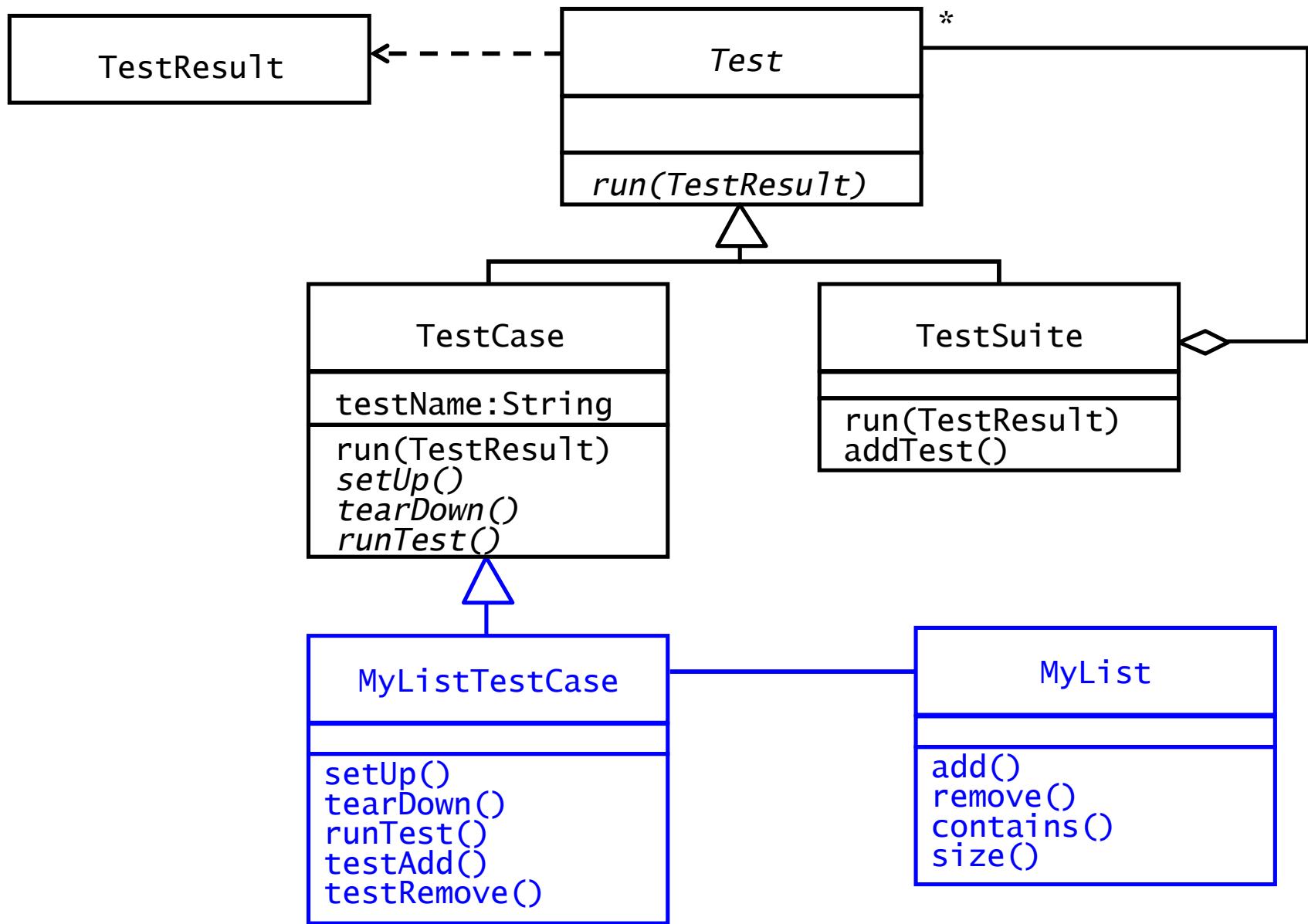
- Written with “test first” and pattern-based development in mind
 - Tests written before code
 - Allows for regression testing
 - Facilitates refactoring
- JUnit is Open Source
 - <https://junit.org/junit5/>
 - JUnit Version 5, was released on OCT 24, 2017

JUnit Classes



An example: Testing MyList

- Unit to be tested
 - MyList
- Methods under test
 - add()
 - remove()
 - contains()
 - size()
- Concrete Test case
 - MyListTestCase



Writing TestCases in JUnit

```
public class MyListTestCase extends TestCase {
```

```
    public MyListTestCase(String name) {
```

```
        super(name);
```

```
}
```

```
    public void testAdd() {
```

```
        // Set up the test
```

```
        List aList = new MyList();
```

```
        String anElement = "a string";
```

```
        // Perform the test
```

```
        aList.add(anElement);
```

```
        // Check if test succeeded
```

```
        assertTrue(aList.size() == 1);
```

```
        assertTrue(aList.contains(anElement));
```

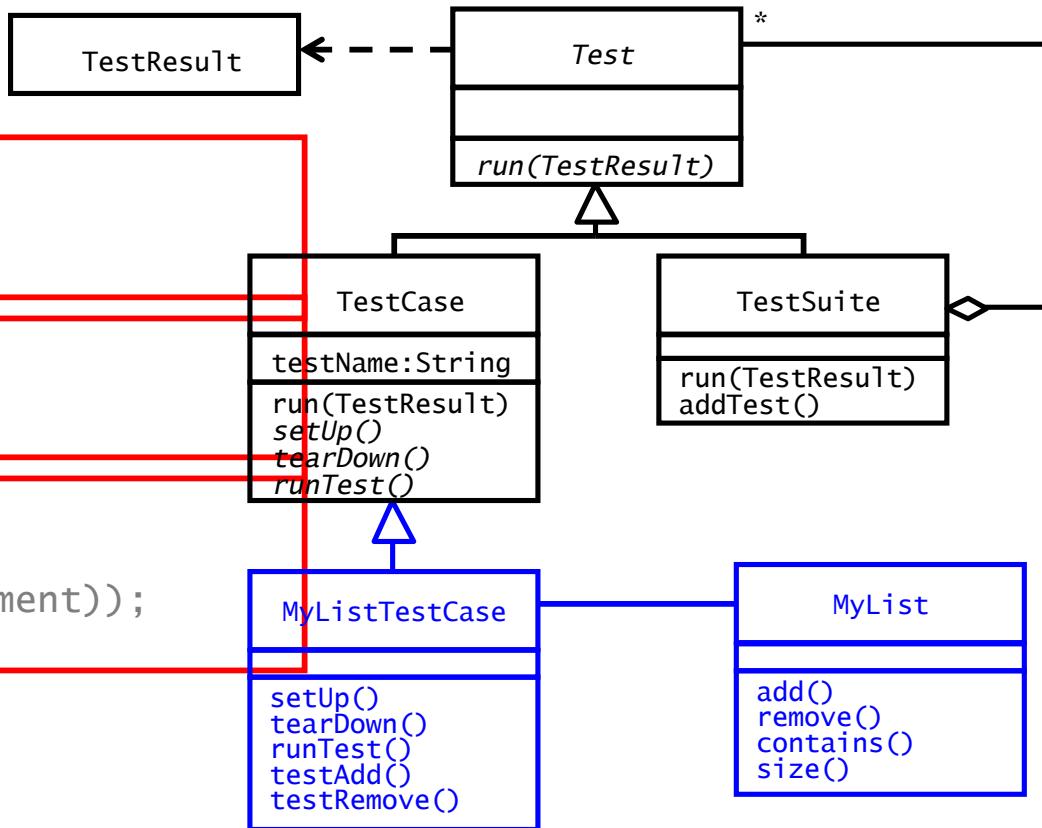
```
}
```

```
protected void runTest() {
```

```
    testAdd();
```

```
}
```

```
}
```



Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {  
    // ...  
  
    private MyList aList;  
    private String anElement;  
    public void setUp() {  
        aList = new MyList();  
        anElement = "a string";  
    }  
}
```

Test Fixture

```
public void testAdd() {  
    aList.add(anElement);  
    assertTrue(aList.size() == 1);  
    assertTrue(aList.contains(anElement));  
}  
}
```

Test Case

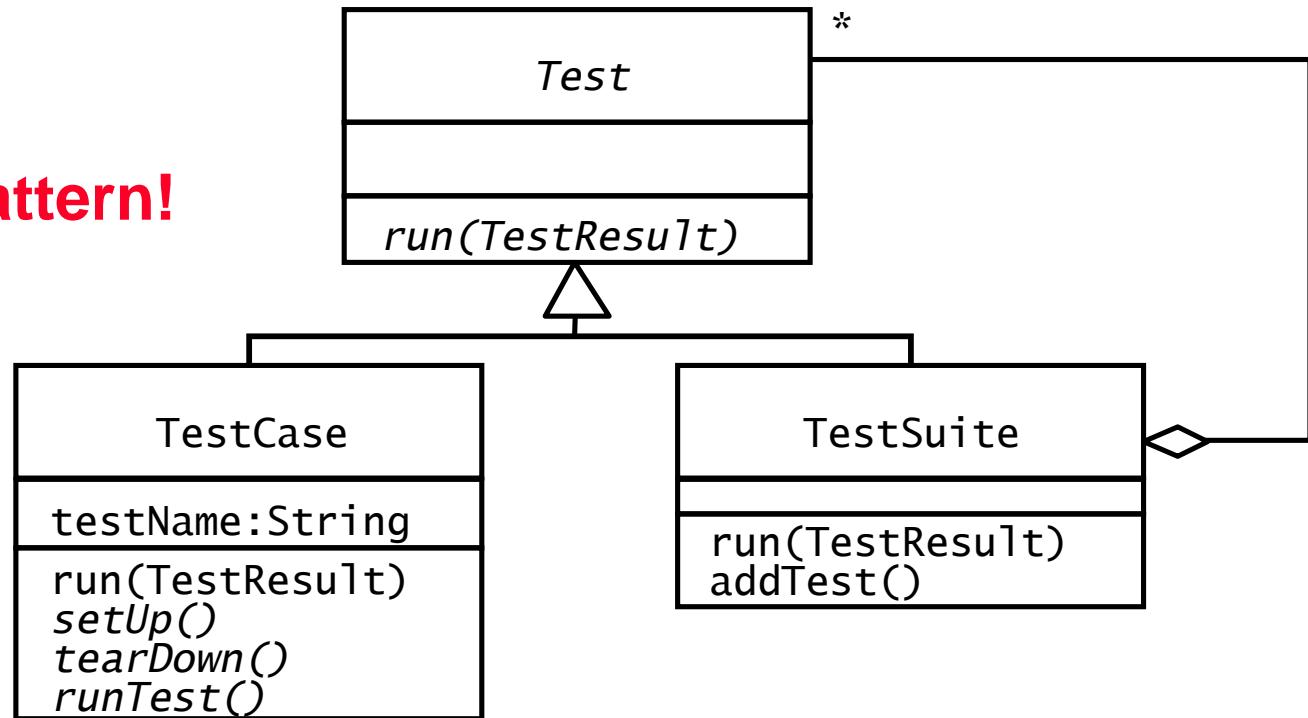
```
public void testRemove() {  
    aList.add(anElement);  
    aList.remove(anElement);  
    assertTrue(aList.size() == 0);  
    assertFalse(aList.contains(anElement));  
}  
}
```

Test Case

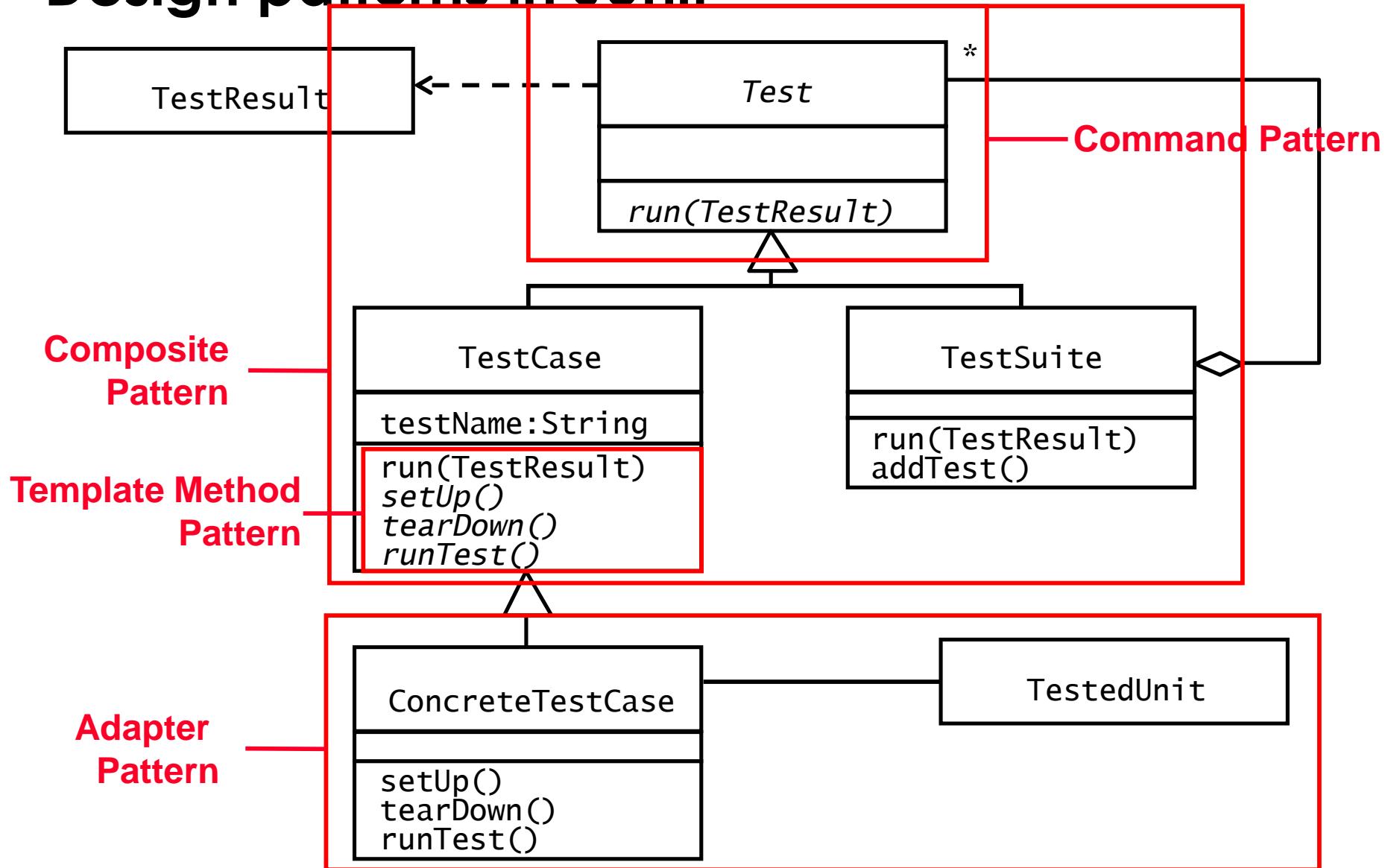
Collecting TestCases into TestSuites

```
public static Test suite() {  
    TestSuite suite = new TestSuite();  
    suite.addTest(new MyListTest("testAdd"));  
    suite.addTest(new MyListTest("testRemove"));  
    return suite;  
}
```

Composite Pattern!



Design patterns in JUnit



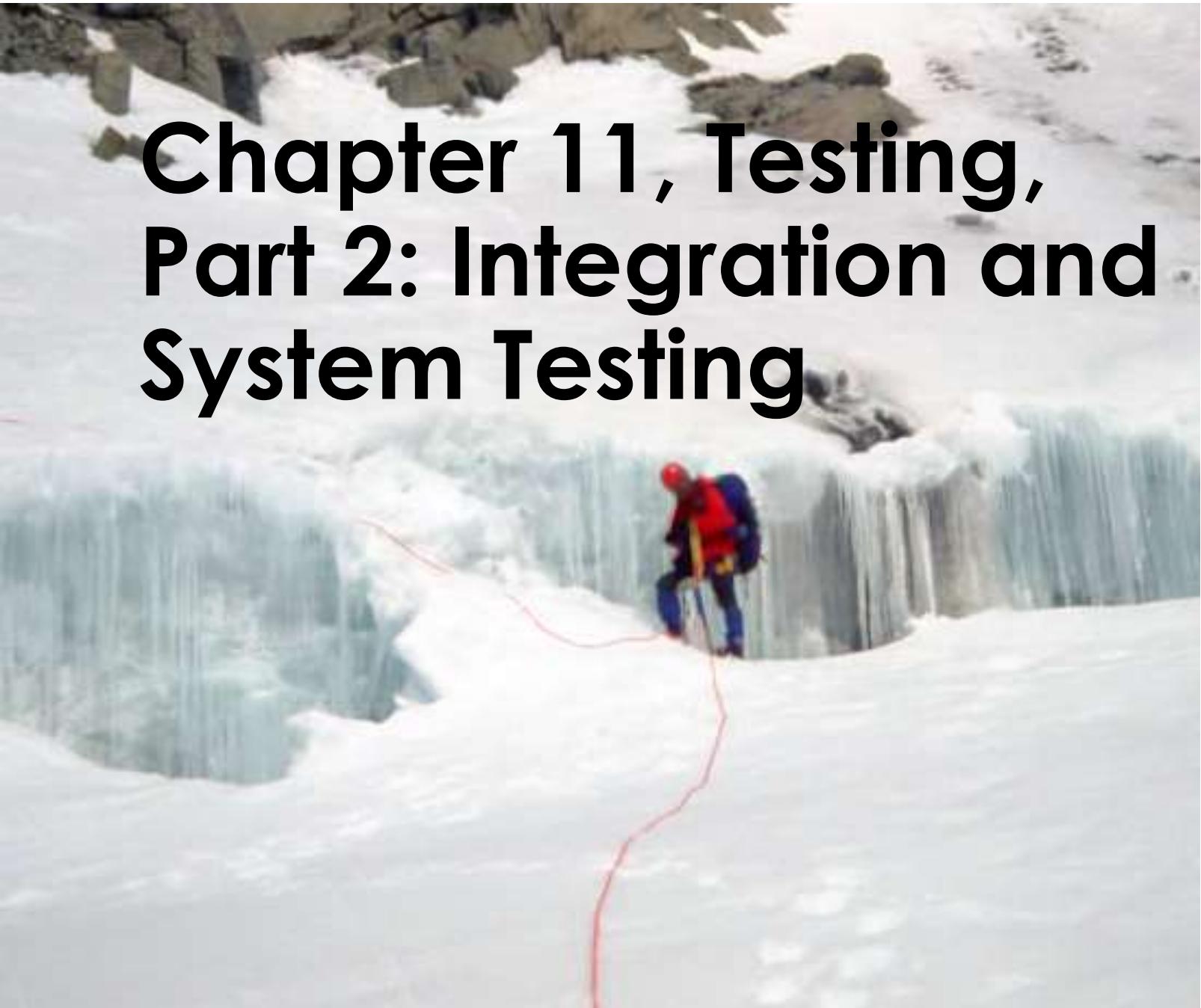
Other JUnit features

- Textual and GUI interface
 - Displays status of tests
 - Displays stack trace when tests fail
- Integrated with Maven and Continuous Integration
 - <http://maven.apache.org>
 - Build and Release Management Tool
 - All tests are run before release (regression tests)
 - Test results are advertised as a project report
- Many specialized variants
 - Unit testing of web applications
 - J2EE applications

Additional Readings

- JUnit Website junit.org/junit5

Chapter 11, Testing, Part 2: Integration and System Testing



Overview

- Integration testing
 - Big bang
 - Bottom up
 - Top down
 - Sandwich
 - Continuous
- System testing
 - Functional
 - Performance
- Acceptance testing
- Summary

Integration Testing

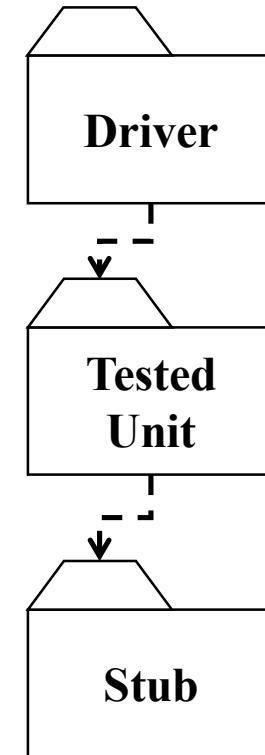
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

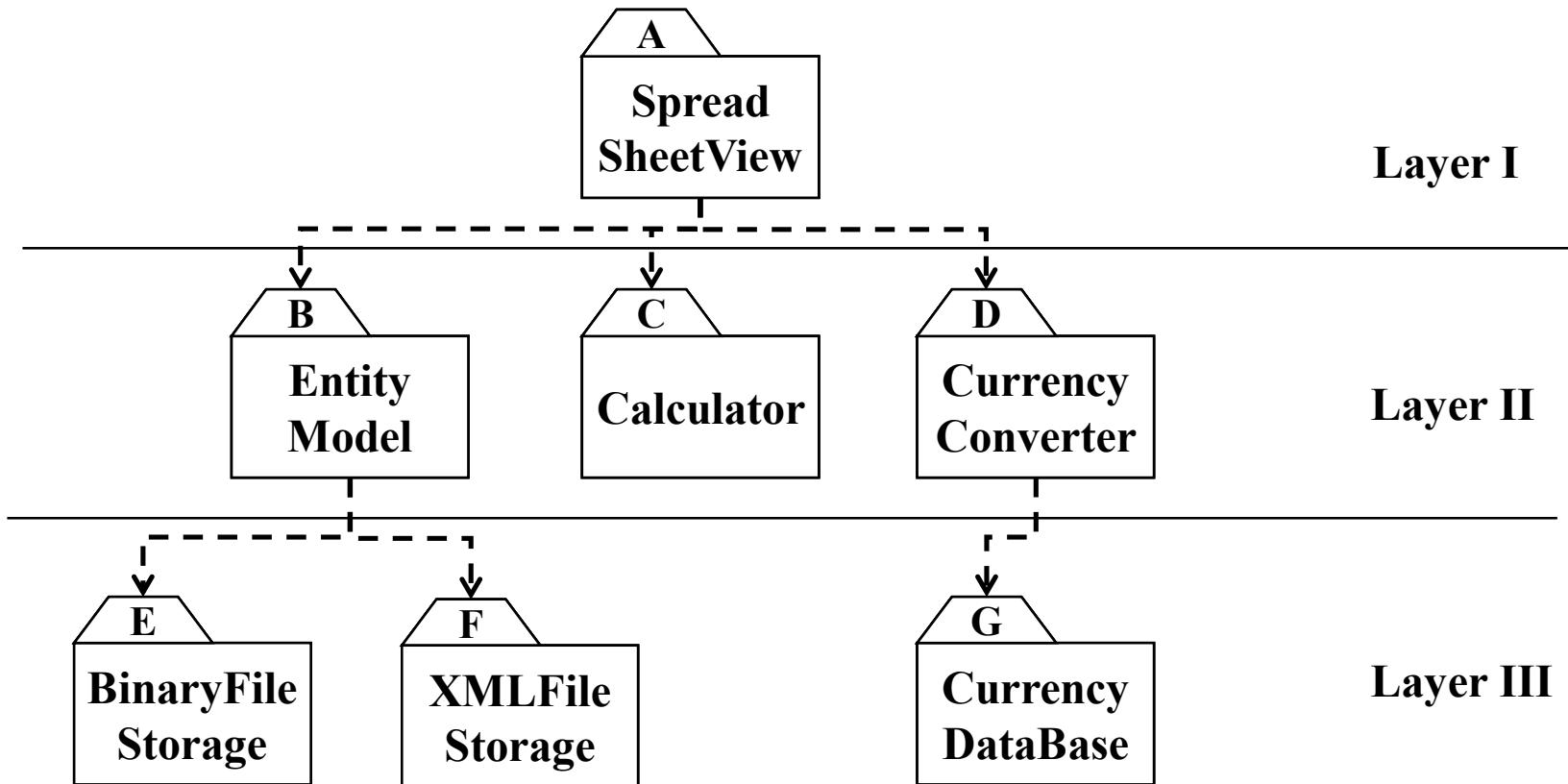
- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

Stubs and drivers

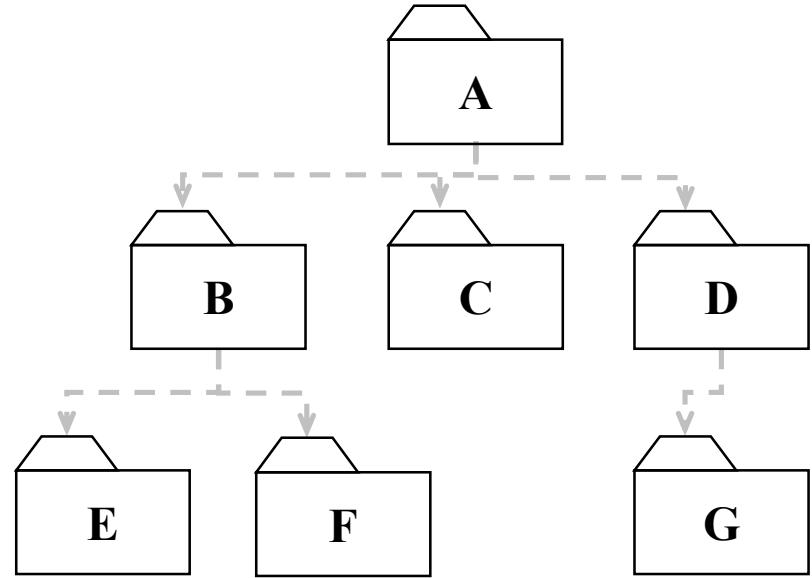
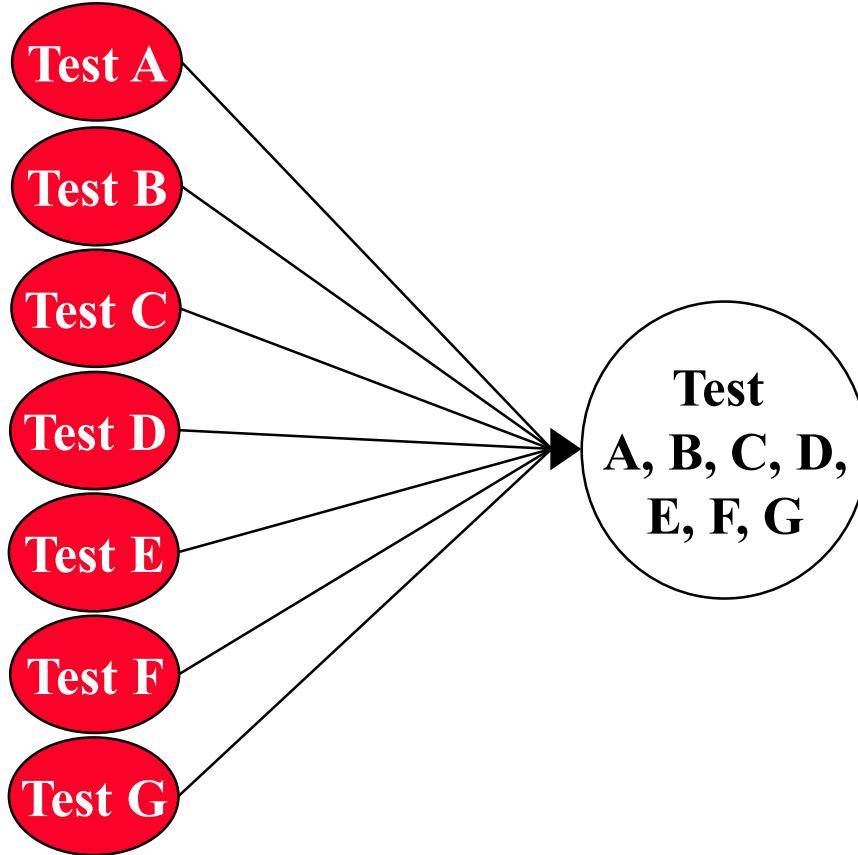
- Driver:
 - A component, that calls the TestedUnit
 - Controls the test cases
- Stub:
 - A component, the TestedUnit depends on
 - Partial implementation
 - Returns fake values.



Example: A 3-Layer-Design (Spreadsheet)



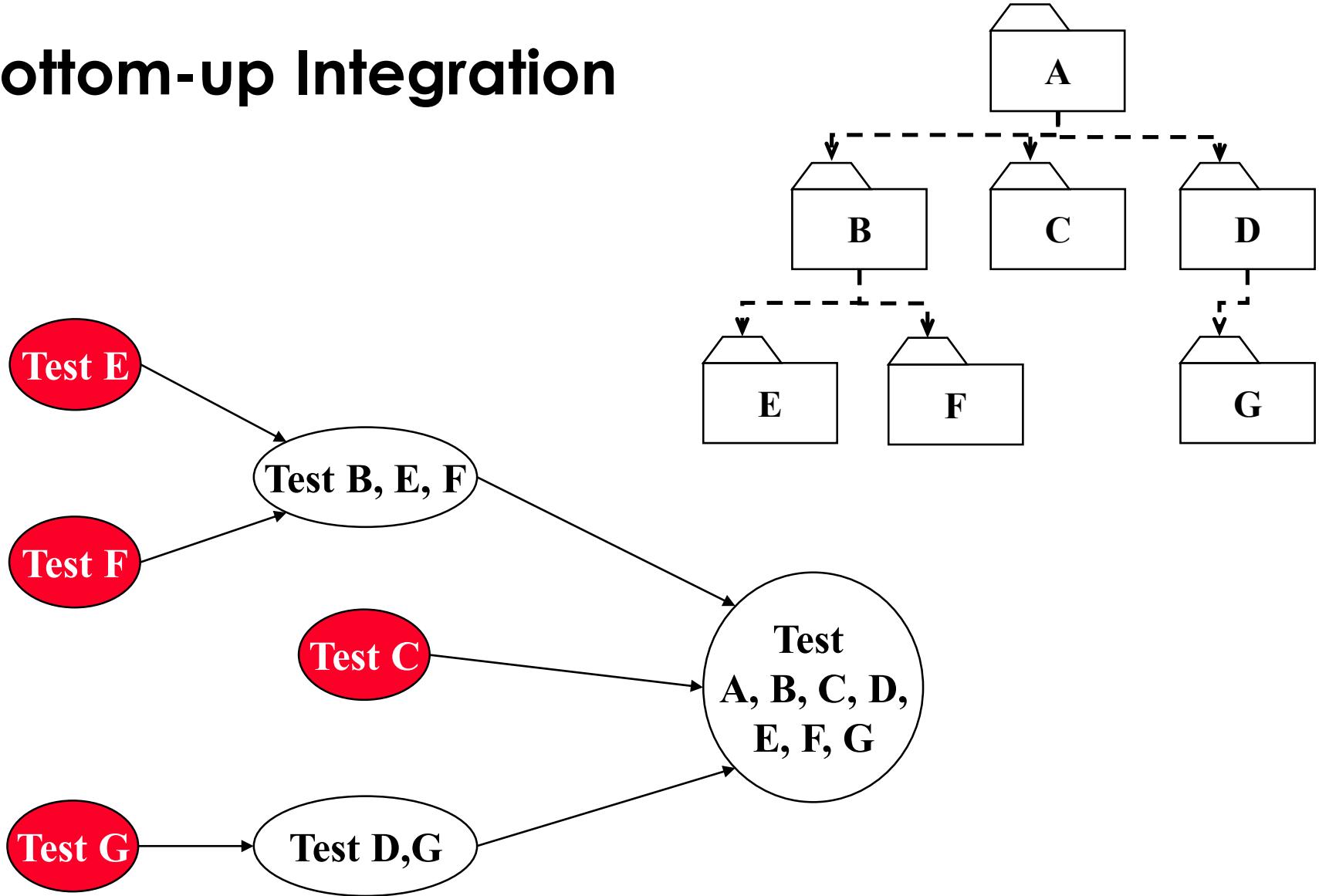
Big-Bang Approach



Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is repeated until all subsystems are included
- Drivers are needed.

Bottom-up Integration



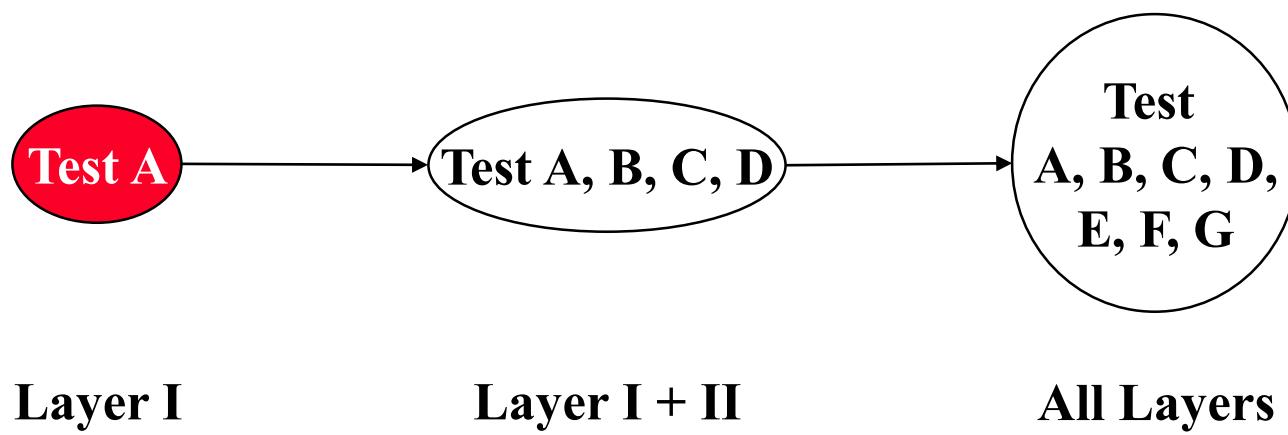
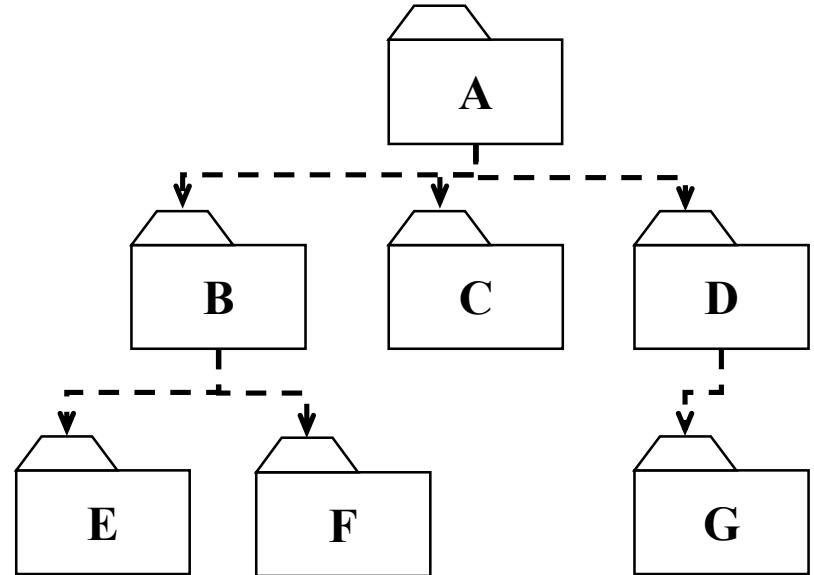
Pros and Cons of Bottom-Up Integration Testing

- Con:
 - Tests the most important subsystem (user interface) last
 - Drivers needed
- Pro
 - No stubs needed
 - Useful for integration testing of the following systems
 - Object-oriented systems
 - Real-time systems
 - Systems with strict performance requirements.

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.

Top-down Integration



Pros and Cons of Top-down Integration Testing

Pro

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

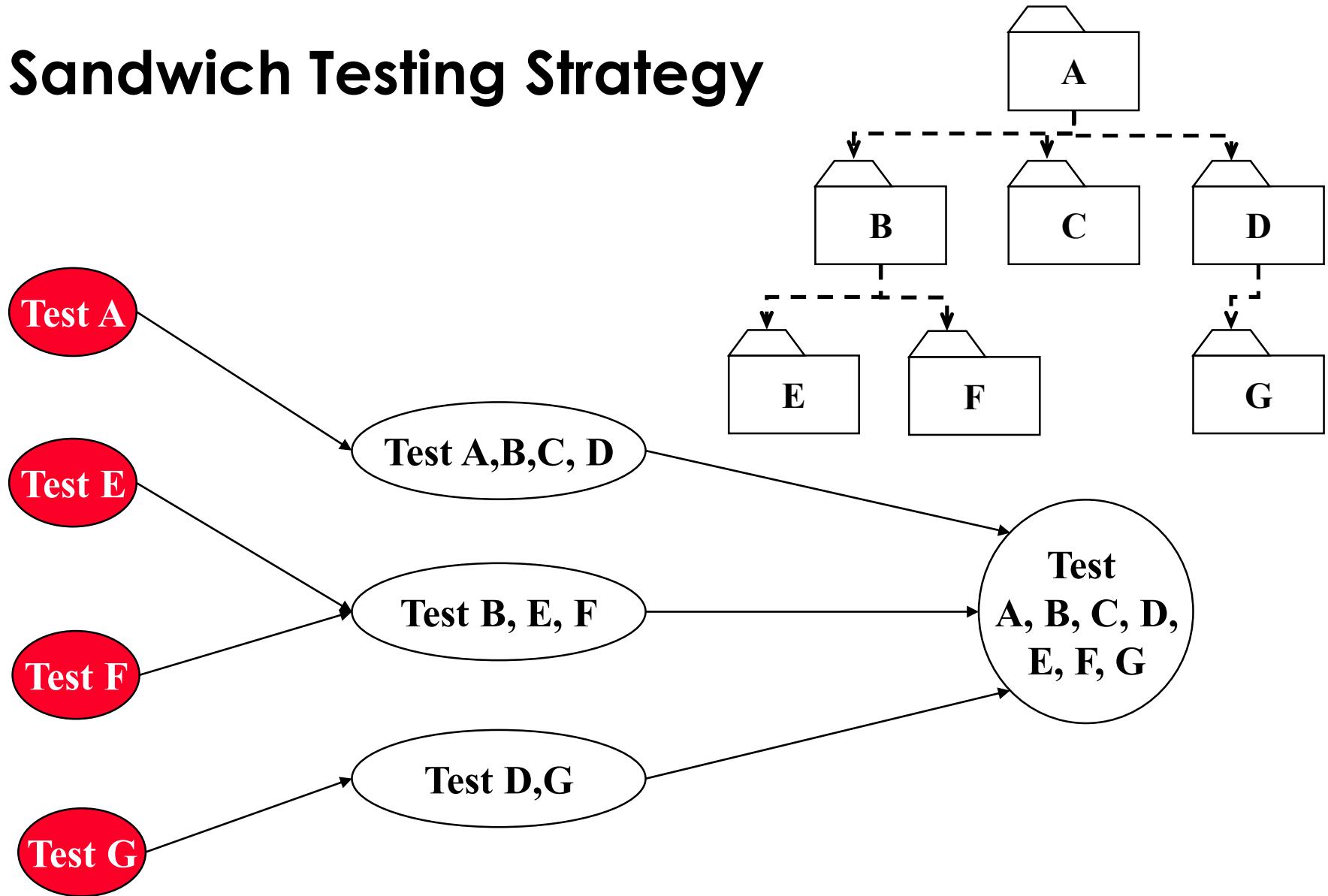
Cons

- Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
- Large number of stubs may be required, especially if the lowest level of the system contains many methods.
- Some interfaces are not tested separately.

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is viewed as having three layers
 - A target layer in the middle
 - A layer above the target
 - A layer below the target
- Testing converges at the target layer.

Sandwich Testing Strategy



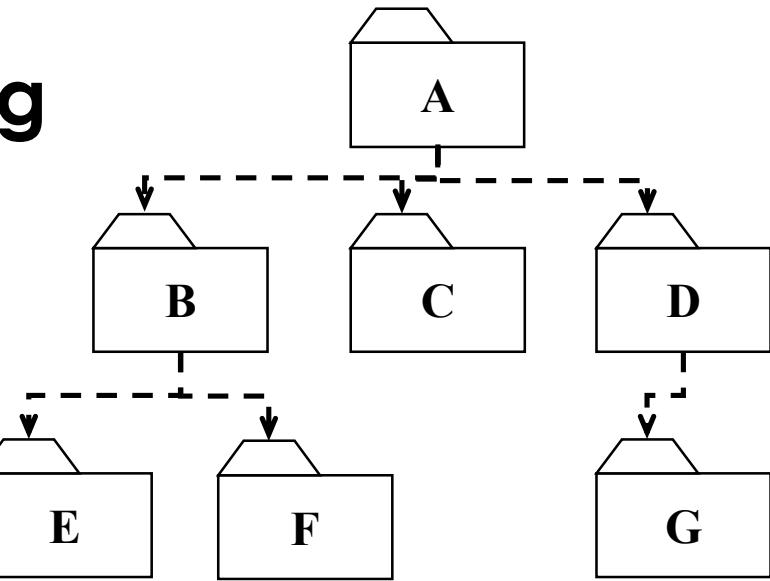
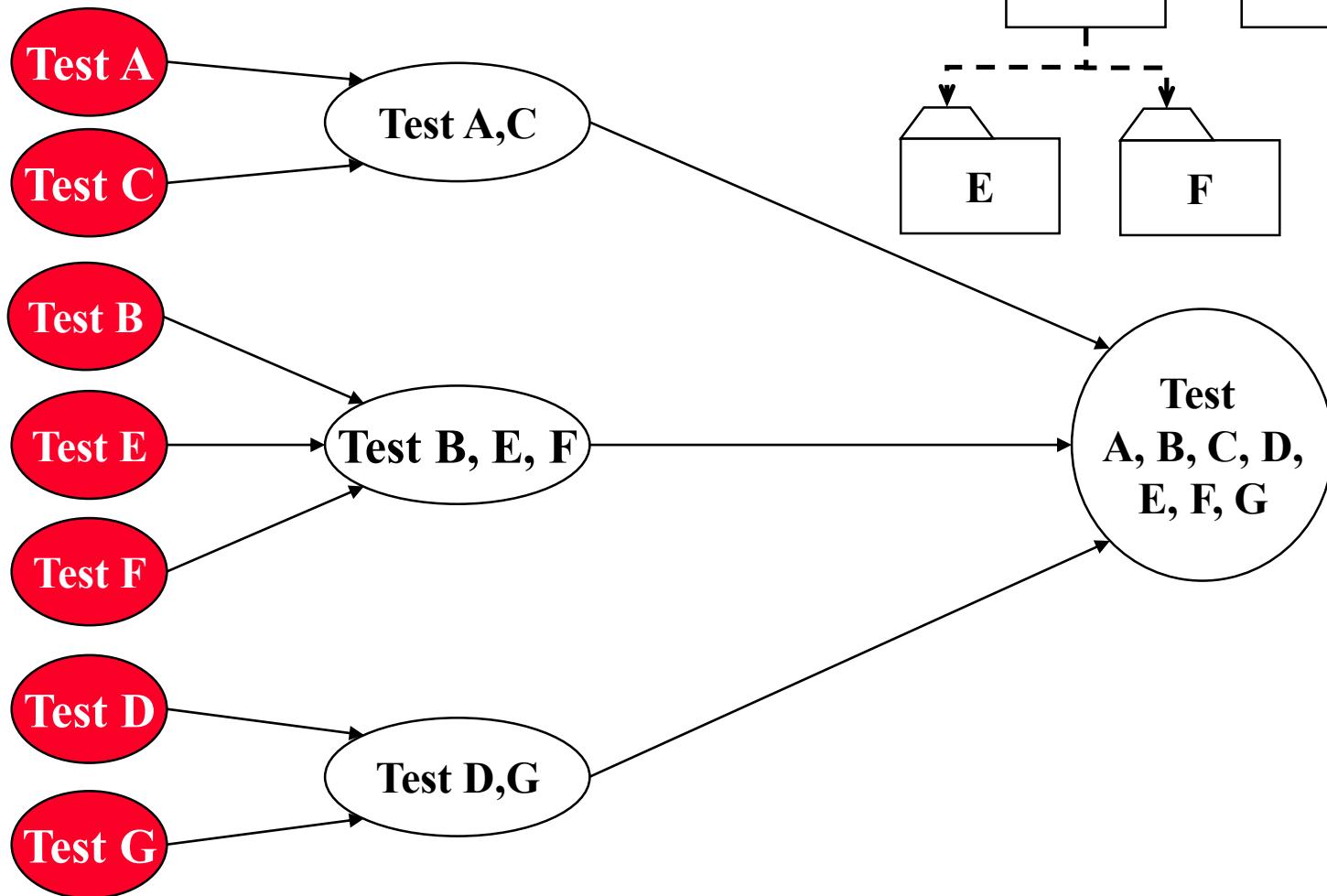
Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Problem: Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy

Modified Sandwich Testing Strategy

- **Test in parallel:**
 - Middle layer with drivers and stubs
 - Top layer with stubs
 - Bottom layer with drivers
- **Test in parallel:**
 - Top layer accessing middle layer (top layer replaces drivers)
 - Bottom accessed by middle layer (bottom layer replaces stubs).

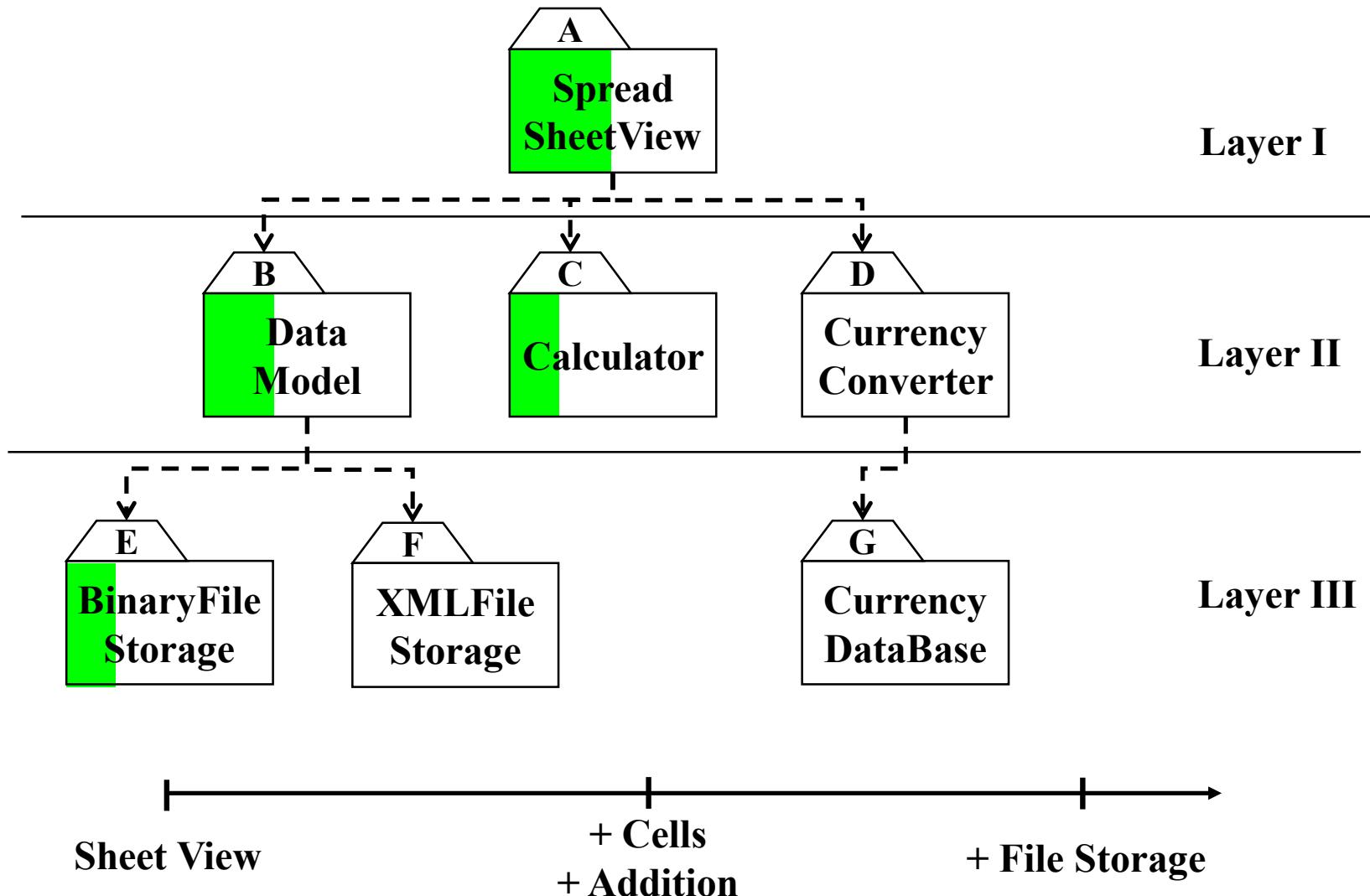
Modified Sandwich Testing



Continuous Testing

- Continuous build:
 - Build from day one
 - Test from day one
 - Integrate from day one
 - ⇒ System is always runnable
- Requires integrated tool support:
 - Continuous build server
 - Automated tests with high coverage
 - Tool supported refactoring
 - Software configuration management
 - Issue tracking.

Continuous Testing Strategy



Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements: Define test cases that exercise all uses cases with the selected component

4. Test subsystem decomposition: Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute *performance tests*
6. Keep records of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify failures with the (current) component configuration.*

System Testing

- Functional Testing
 - Validates functional requirements
- Performance Testing
 - Validates non-functional requirements
- Acceptance Testing
 - Validates clients expectations

Functional Testing

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

Performance Testing

Goal: Try to violate non-functional requirements

- Test how the system behaves when overloaded.
 - Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- Try unusual orders of execution
 - Call a receive() before send()
- Check the system's response to large volumes of data
 - If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?
 - Are typical cases executed in a timely fashion?

Types of Performance Testing

- Stress Testing
 - Stress limits of system
- Volume testing
 - Test what happens if large amounts of data are handled
- Configuration testing
 - Test the various software and hardware configurations
- Compatibility test
 - Test backward compatibility with existing systems
- Timing testing
 - Evaluate response times and time to perform a function
- Security testing
 - Try to violate security requirements
- Environmental test
 - Test tolerances for heat, humidity, motion
- Quality testing
 - Test reliability, maintainability & availability
- Recovery testing
 - Test system's response to presence of errors or loss of data
- Human factors testing
 - Test with end users.

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
 - Choice of tests is made by client
 - Many tests can be taken from integration testing
 - Acceptance test is performed by the client, not by the developer.
- **Alpha test:**
 - Client uses the software at the developer's environment.
 - Software used in a controlled setting, with the developer always ready to fix bugs.
- **Beta test:**
 - Conducted at client's environment (developer is not present)
 - Software gets a realistic workout in target environment

Testing has many activities

Establish the test objectives

Design the test cases

Write the test cases

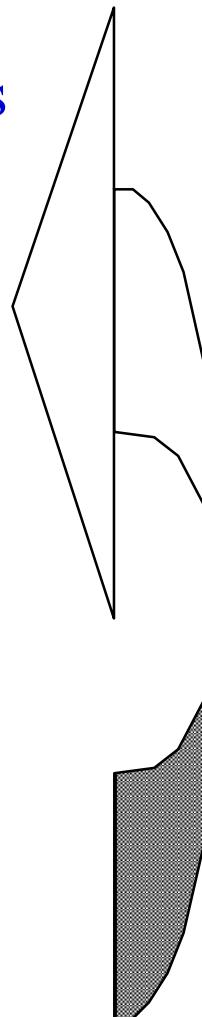
Test the test cases

Execute the tests

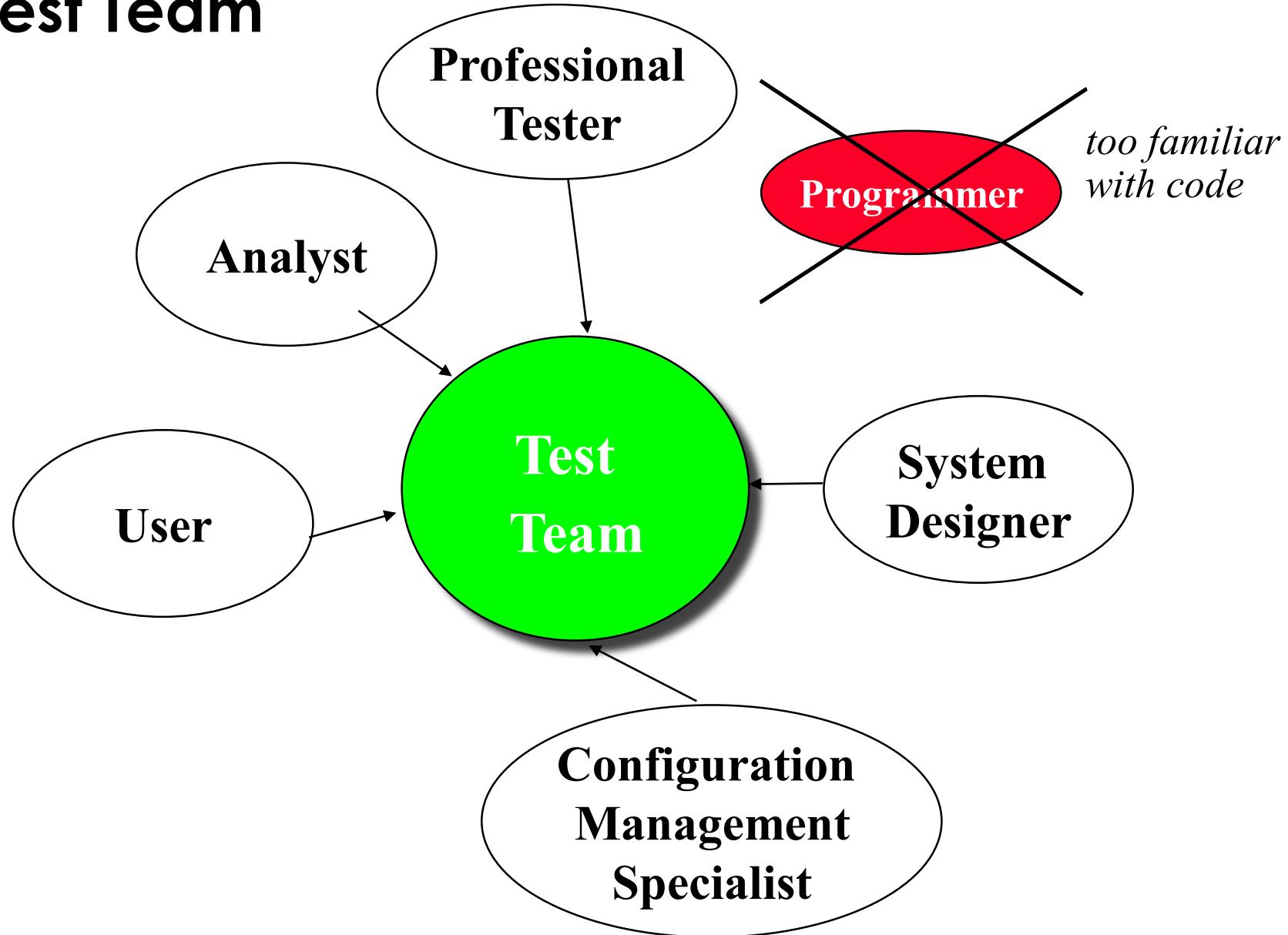
Evaluate the test results

Change the system

Do regression testing



Test Team



The 4 Testing Steps

1. Select what has to be tested

- Analysis: Completeness of requirements
- Design: Cohesion
- Implementation: Source code

2. Decide how the testing is done

- Review or code inspection
- Proofs (Design by Contract)
- Black-box, white box,
- Select integration testing strategy (big bang, bottom up, top down, sandwich)

3. Develop test cases

- A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured

4. Create the test oracle

- An oracle contains the predicted results for a set of test cases
- The test oracle has to be written down before the actual testing takes place.

Guidance for Test Case Selection

- Use *analysis knowledge* about functional requirements (black-box testing):
 - Use cases
 - Expected input data
 - Invalid input data
- Use *design knowledge* about system structure, algorithms, data structures (white-box testing):
 - Control structures
 - Test branches, loops,
...
 - Data structures
 - Test records fields, arrays, ...
- Use *implementation knowledge* about algorithms and datastructures:
 - Force a division by zero
 - If the upper bound of an array is 10, then use 11 as index.

Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of
 - Unit testing
 - Integration testing
 - System testing
 - Acceptance testing
- Design patterns can be used for integration testing
- Testing has its own lifecycle

Additional Reading

- J. Thomas, M. Young, K. Brown, A. Glover, Java Testing Patterns, Wiley, 2004
- D. Saff and M. D. Ernst, An experimental evaluation of continuous testing during development Int. Symposium on Software Testing and Analysis, Boston July 12-14, 2004, pp. 76-85
 - A controlled experiment shows that developers using continuous testing were three times more likely to complete the task before the deadline than those without.