Software Engineering

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_Def_Bruegge

Software Engineering: Definition 1

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

[Brügge]

- Constraints are important

Peter Müller ETH Zurich – SS-06

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

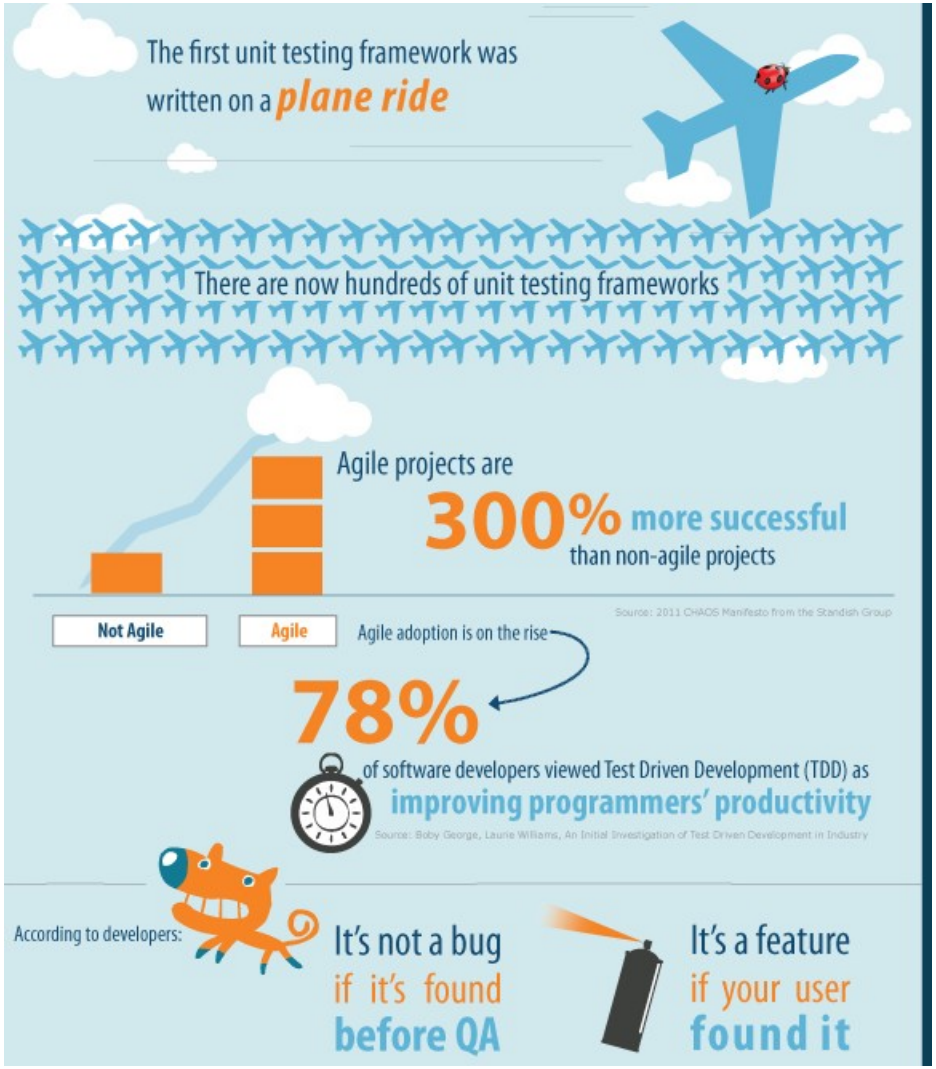
... includes programming, but is more than programming

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

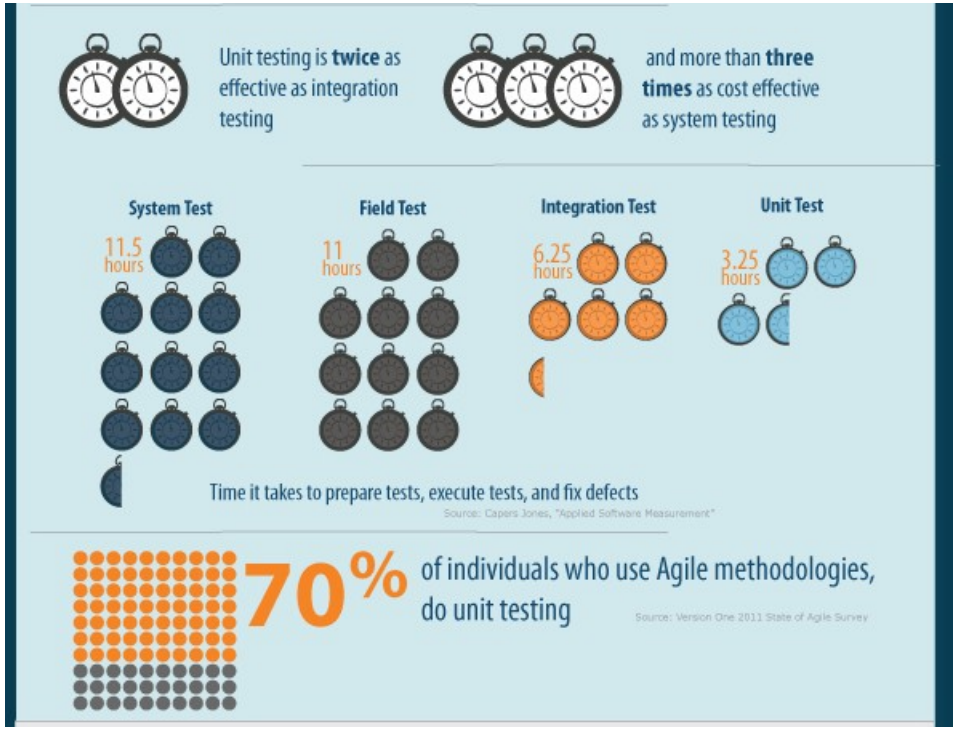
Bertrand Meyer ETH Zurich – SE

The costs of software errors

The costs of software errors_2

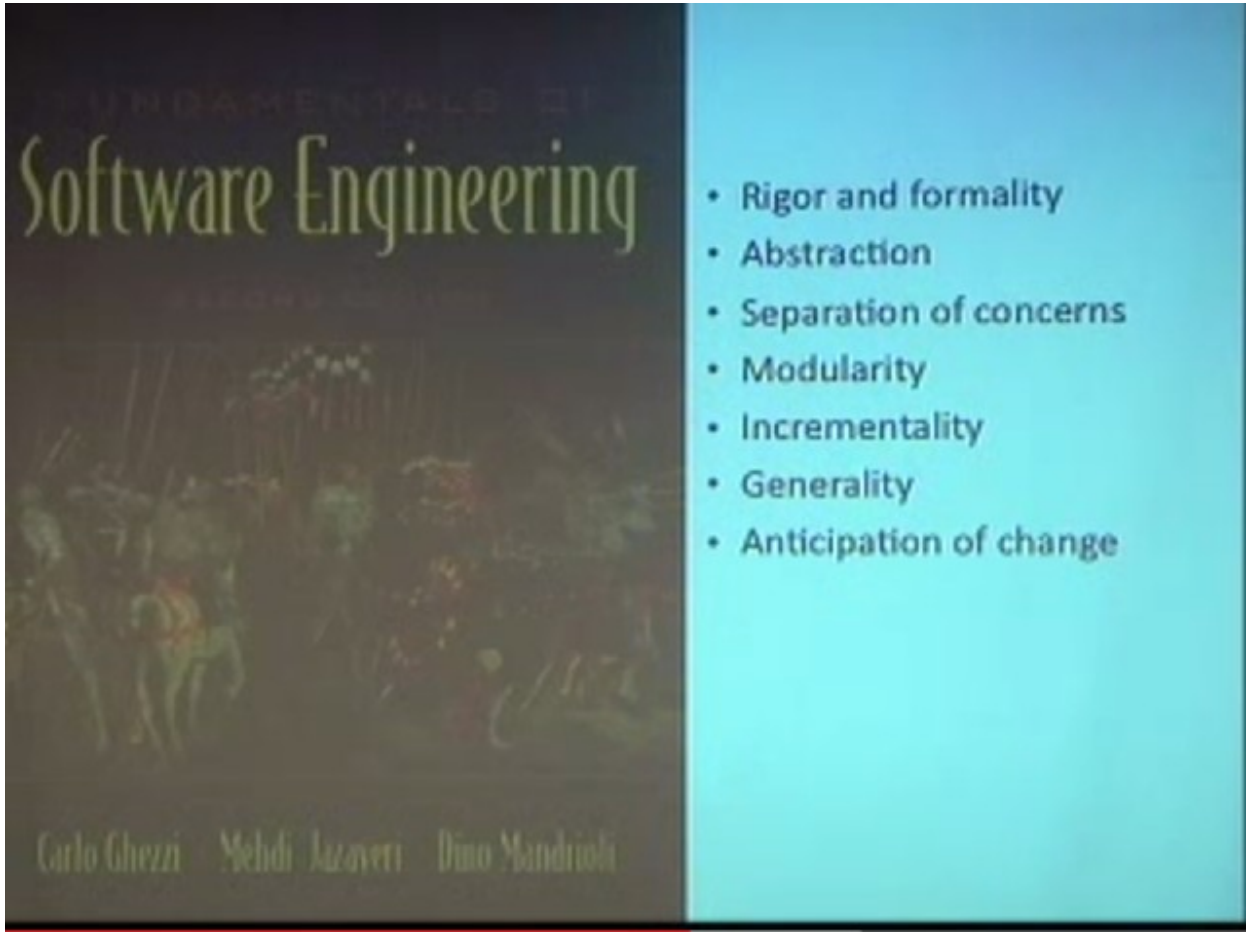


The costs of software errors_3



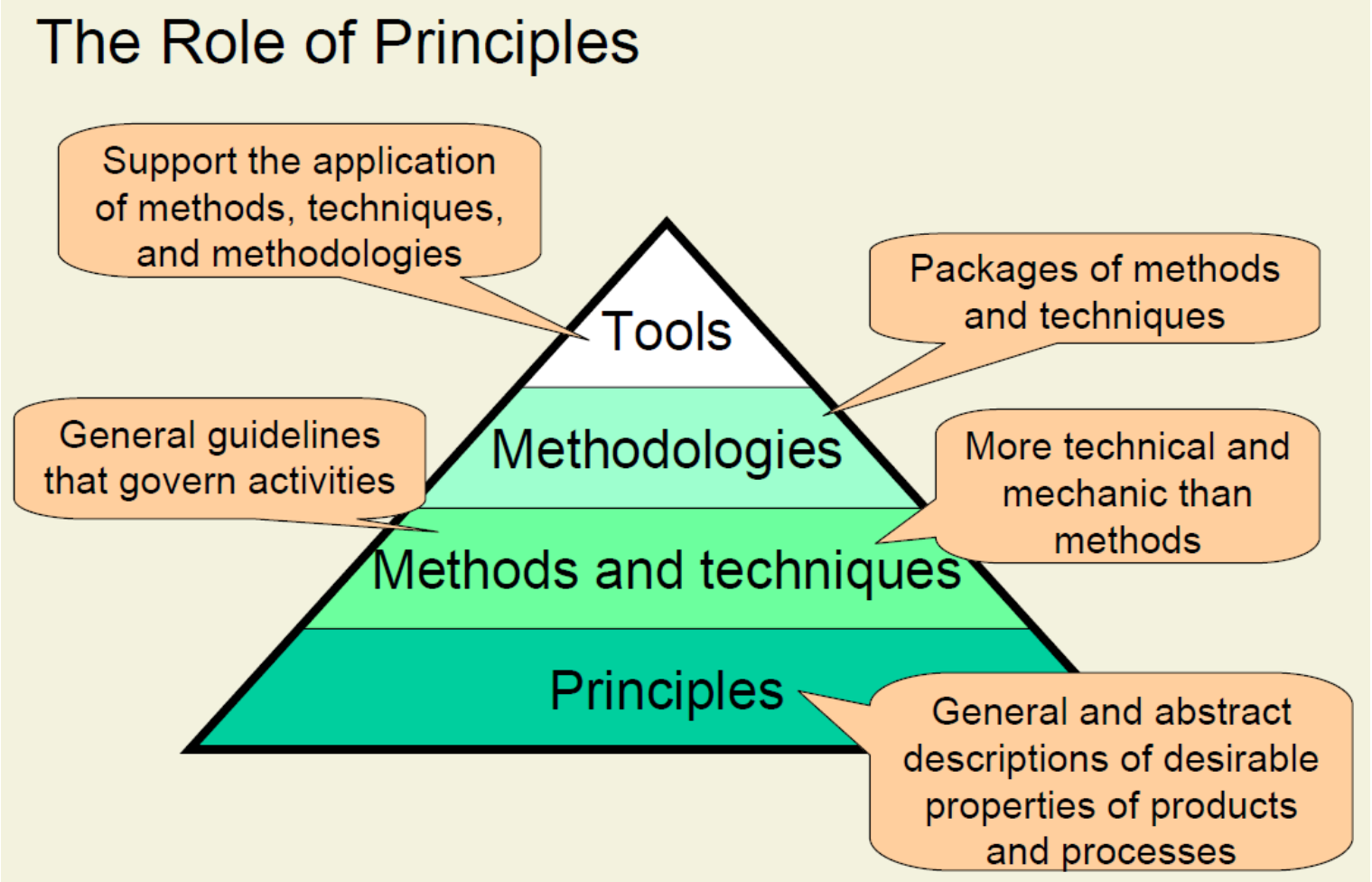
http://www.codeguru.com/blog/category/programming/the-cost-of-bugs.html

Software Engineering Principles



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

Software Engineering Principles_2



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

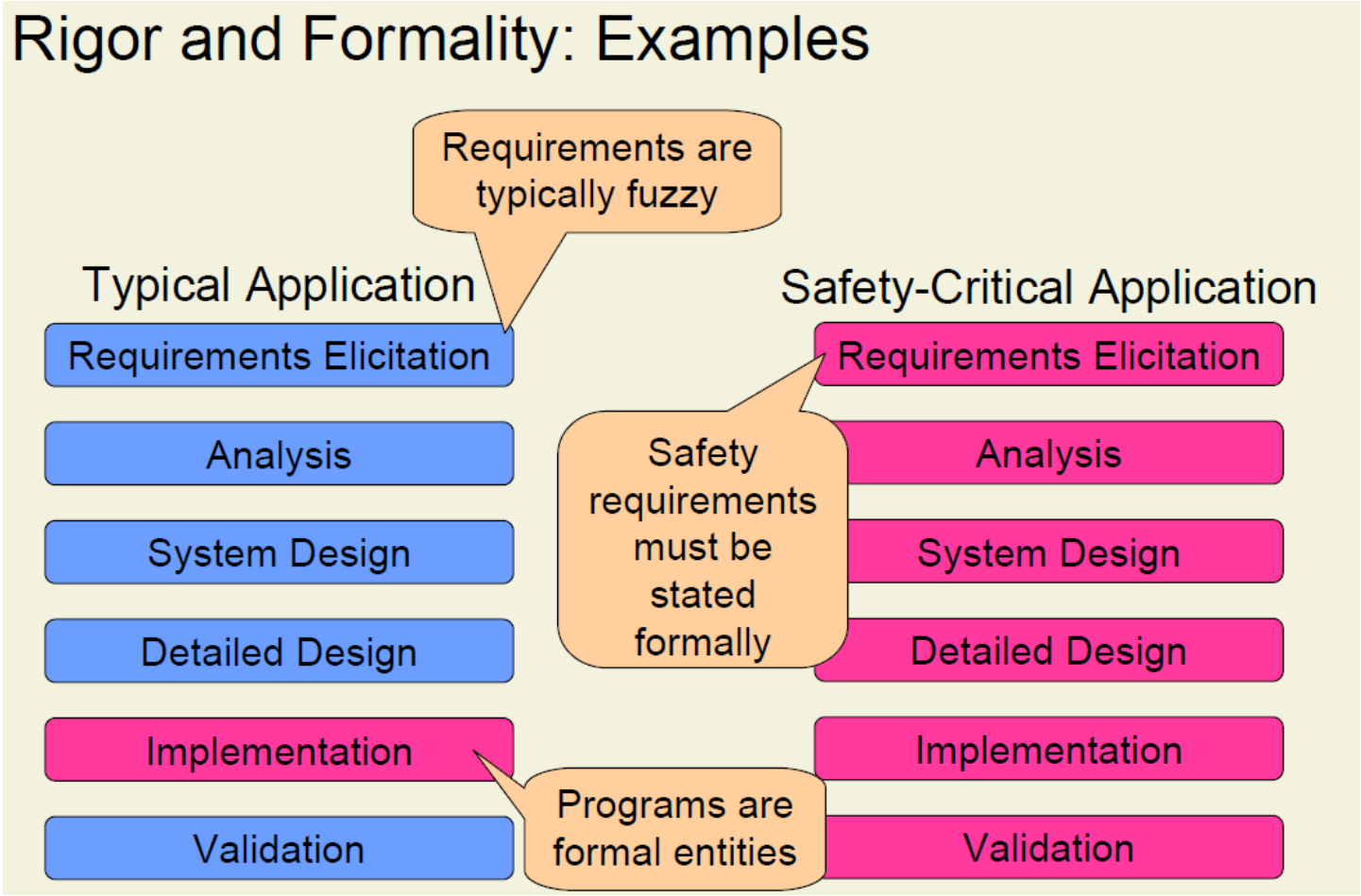
Software Engineering Principles_4

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



Peter Müller ETH Zurich – SS-06

Software Engineering Principles_6

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

Peter Müller ETH Zurich – SS-06

Software Engineering Principles_7

Modularity

- Divide system into modules to **reduce complexity**

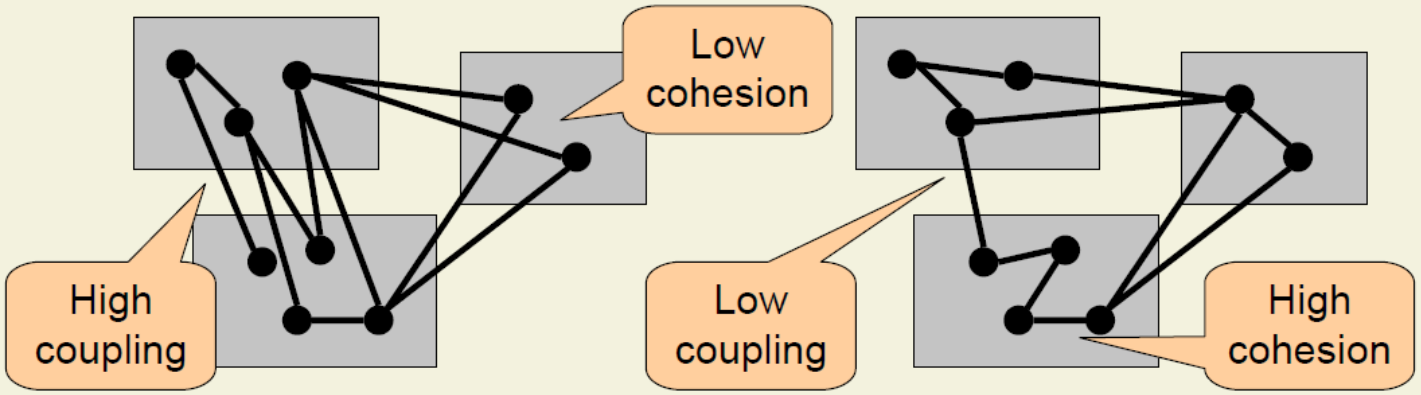
- **Decompose** a complex system into simpler pieces
- **Compose** a complex system from existing modules
- **Understand** the system in terms of its pieces
- **Modify** a system by modifying only a small number of its pieces

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 have to 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

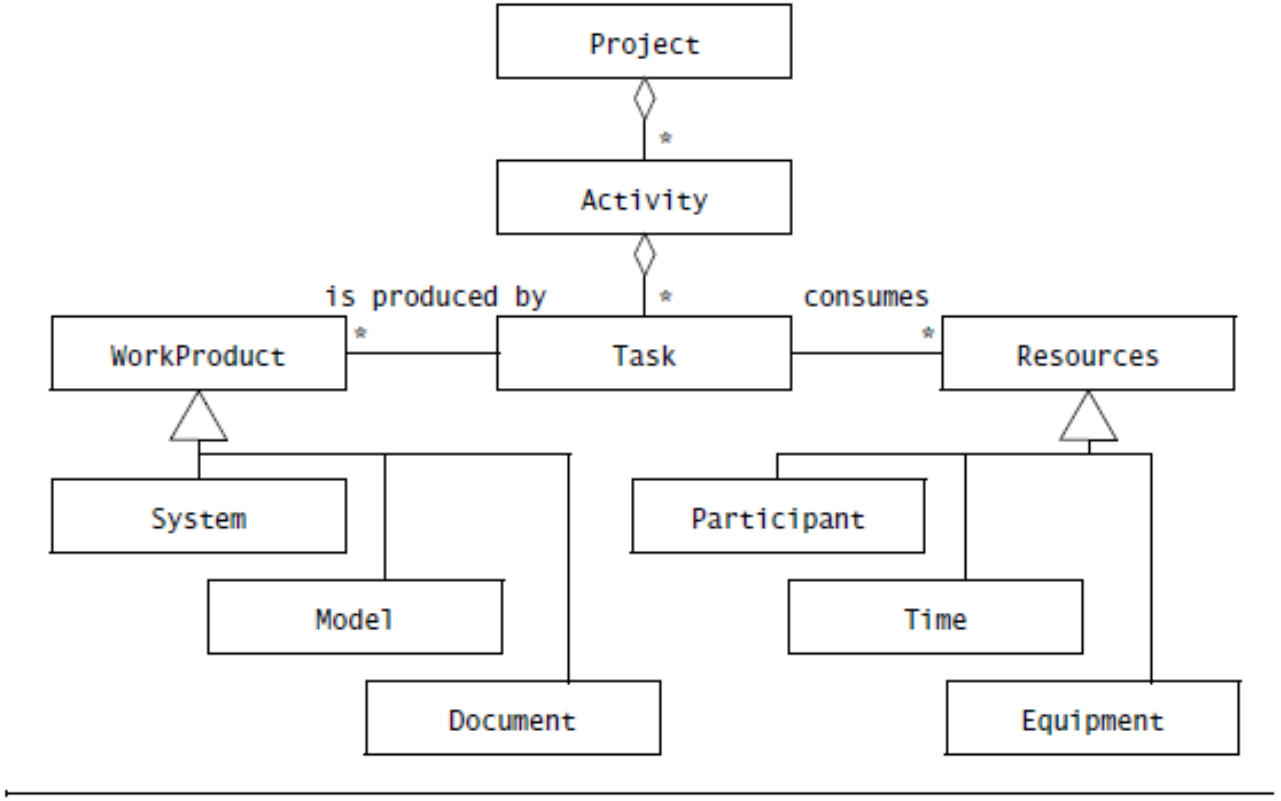


Figure 1-1 Software engineering concepts depicted as a UML class diagram [OMG, 2009].

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 time card 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 `TicketDistributor`.
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 `TicketDistributor`	**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

Nts/Languages, Methods, and Methodologies_2

- 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.

Nts/Languages, Methods, and Methodologies

- 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

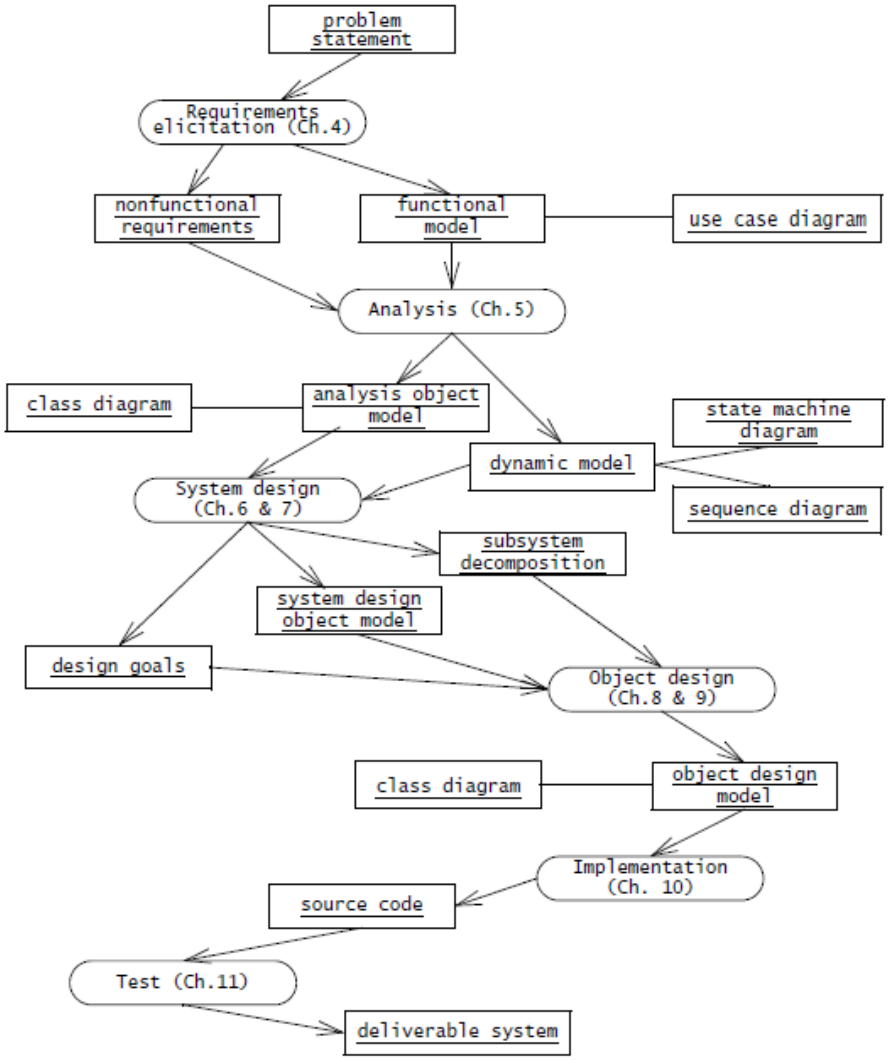


Figure 1-2 An overview of object-oriented software engineering development activities and their products. This diagram depicts only logical dependencies among work products. Object-oriented software engineering is iterative; that is, activities can occur in parallel and more than once.

Software Engineering Development Activities_4

Use case name	`PurchaseOneWayTicket`
Participating actor	Initiated by `Traveler`
Flow of events	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.
Entry condition	The `Traveler` stands in front of the `TicketDistributor`, which may be located at the station of origin or at another station.
Exit condition	The `Traveler` holds a valid ticket and any excess change.
Quality requirements	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

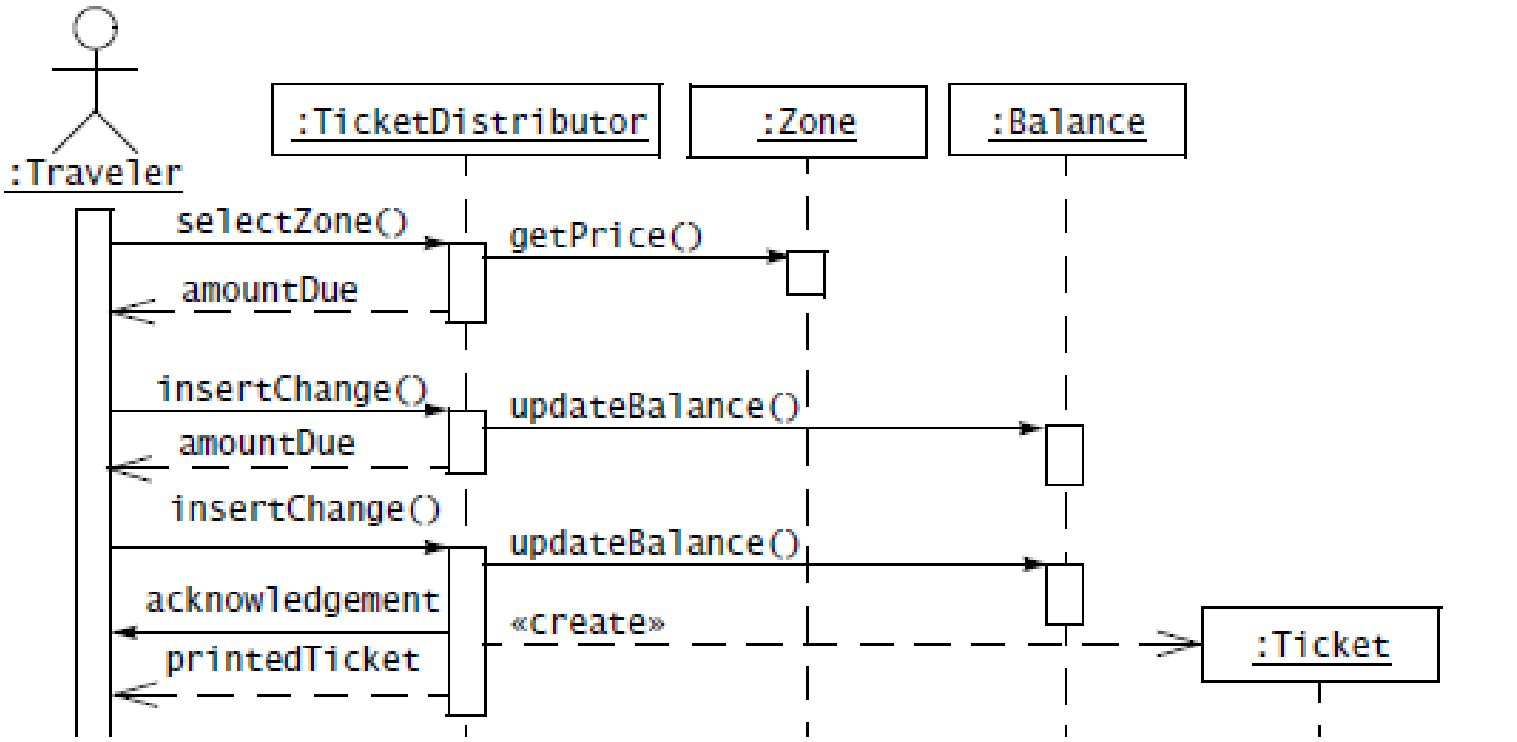


Figure 1-4 A dynamic model for the `TicketDistributor` (UML sequence diagram). This diagram depicts the interactions between the actor and the system during the `PurchaseOneWayTicket` use case and the objects that participate in the use case.

Software Engineering Development Activities_8

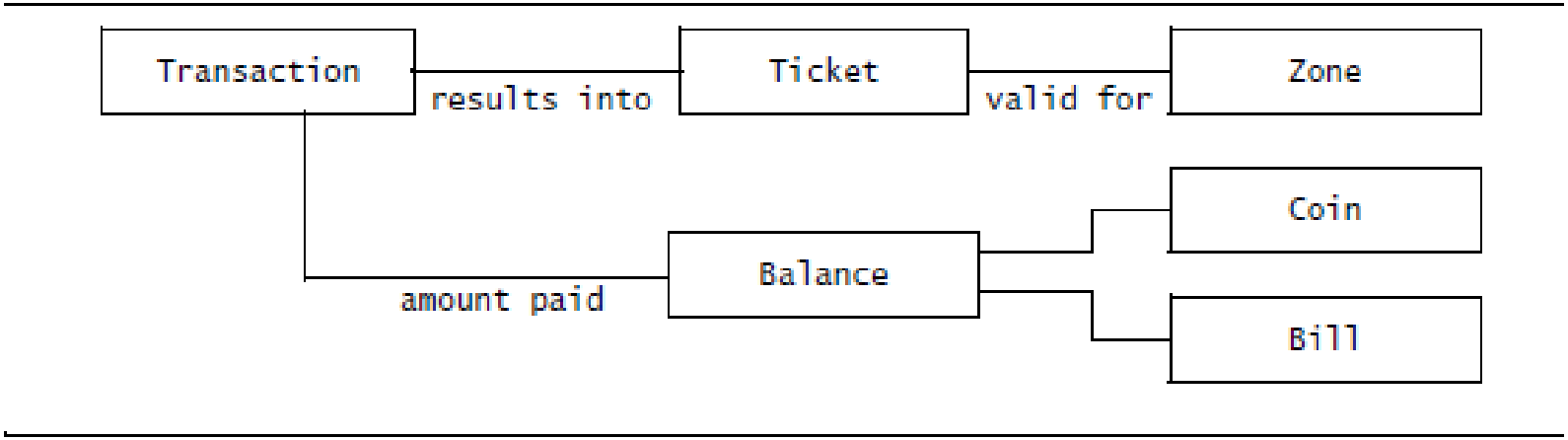


Figure 1-5 An object model for the `TicketDistributor` (UML class diagram). In the `PurchaseOneWayTicket` use case, a `Traveler` initiates a transaction that will result in a `Ticket`. A `Ticket` is valid only for a specified `Zone`. During the `Transaction`, the system tracks the `Balance` due by counting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

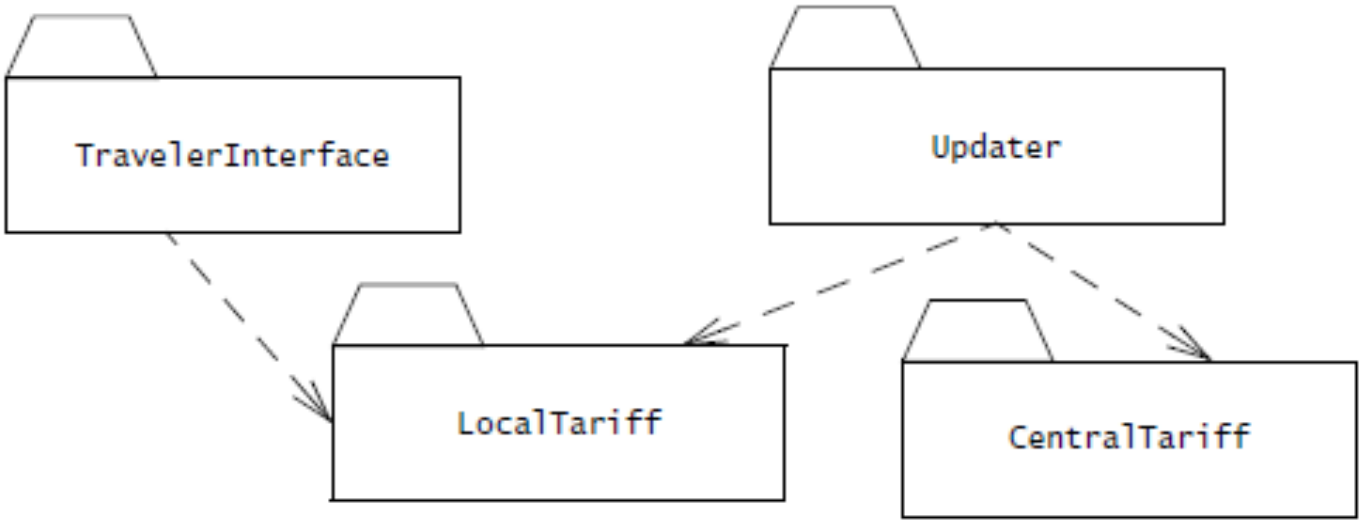


Figure 1-6 A subsystem decomposition for the `TicketDistributor` (UML class diagram, packages represent subsystems, dashed lines represent dependencies). The `TravelerInterface` subsystem is responsible for collecting input from the `Traveler` and providing feedback (e.g., display ticket price, returning 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

Thanks for your patience!