Object-oriented software engineering

Bernard Boigelot

E-mail : boigelot@montefiore.ulg.ac.be
WWW : http://www.montefiore.ulg.ac.be/~boigelot/
http://www.montefiore.ulg.ac.be/~boigelot/courses/se/

References:

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1994.
Chapter 1

Software development methodologies
Introduction

Subject of the course: Methodologies, techniques and tools for tackling large-scale software development.

Main issues:

- Software products can get very complex.
- High-quality results are expected.
- The development teams can be large and distributed.
- Most projects deal with adding functionalities to an existing product, instead of creating new software from scratch.
- The available amount of development resources is always limited.
Development activities

The activities performed during software development can be classified as follows:

- Specifying the requirements.
- Analyzing the problems to be solved.
- Finding solutions: Designing the architecture of the system, solving problems, . . .
- Implementing the results.
- Testing the product.
The development process

Those development activities must not necessarily be performed in sequential order.

Definition: A development process is an organization of development activities:

- When are the activities performed? For how long? In what order?
- By whom?
- In what form are the results of each activity expressed?
- How are the unexpected situations handled?

For software engineering, we need a process that:

- minimizes the risks of not delivering a suitable product,
- produces high-quality software that corresponds to the needs of the user, and
- manages resources efficiently.
The waterfall process

Principles:

• Each activity is **completed** before the following one starts.

• The **results** of each activity are input to the next one.
Advantages: The waterfall process is well adapted to disciplines such as civil engineering, where:

- Requirements are easy to define.
- The costs of applying modifications to existing products are high.
- Powerful tools are available for checking the quality of intermediate results.

Drawbacks: For large-scale software engineering, this approach fails because:

- Specifying the requirements is a non-trivial problem.
- Requirements can change during the lifetime of a project.
- The consequences of bad design choices are sometimes only apparent at a very late stage.

In particular, some correctness and performance issues are usually only detected during testing.
When flaws are detected during testing, one has to start again the corresponding development activities. This leads to:

- missing deadlines,
- releasing flawed products, and
- exceeding the allocated resources.
The iterative process framework

Principles:

- The development is organized as a sequence of cycles.
- Each development cycle is aimed at producing a usable product that implements a small set of additional requirements.
- Development cycles have a short duration.
- The requirements are dealt with in decreasing order of relevance. The idea is to first address the problems that:
  - lead to important design choices (e.g., the system architecture), or
  - present a risk of not succeeding.
- In each development cycle, one adapts the requirements, analyzes the current problems, designs solutions, implements them, and tests the final result.
**Advantages:** For software engineering, this approach works well because:

- Valuable **feedback** is obtained early.
- A functional version of the product is always available for evaluation or demonstration purposes.
- The risks of not meeting the specifications or exceeding the allocated resources are mitigated.
- Dealing with changing requirements becomes possible.

**Drawbacks:** During iterative development, one sometimes has to **refactor** (i.e., remove and/or rebuild) parts of the system. This is impractical, unfeasible or too expensive for some disciplines such as civil engineering.
Example 1: Unified Software Development Process

The development cycles are structured in four phases:

- **Inception:**
  - Define the main goals of the project.
  - Estimate the main risks.
  - Decide whether the project is worth being continued.

- **Elaboration:**
  - Define a base architecture for the product.
  - Specify detailed requirements.
  - Estimate the costs and plan the schedule.

- **Construction:** Build the product.

- **Transition:** Make the product ready for commercial release.
Notes:

- Each phase consists of several development cycles.
- The ends of development cycles and phases correspond to minor and major milestones, where management decisions are taken.

Illustration:
Example 2: Scrum

The people involved in the project are organized as follows:

- The *Product Owner*:
  - interacts with the client or stakeholder, and
  - defines and manages the global requirements into a *Product Backlog*, the contents of which are sorted in decreasing order of priority.

- The *Development Team*:
  - builds incremental versions of the product, and
  - estimates the risks, costs and resources of each iteration.

- The *Scrum Master*:
  - deals with methodological issues.
The development cycles are called *Sprints*, and have the following features:

- Their **duration** is fixed (≈ one month), and **never extended**.
- Their goal is to deliver a **useable product** that implements one or more additional items from the Product Backlog. The scope of these items is **negotiated** between the Development Team and the Product Owner.

A Sprint proceeds in the following way:

- At the beginning, a *Sprint Planning* meeting is held:
  - limited to **one day**,  
  - attended by the **whole team**, and  
  - aimed at defining collectively **what** will be done and **how**.
At the end of a Sprint, the whole team holds:

- a *Sprint Review*:
  * one half day,
  * aimed at evaluating the result and adapting the Product Backlog.

- a *Sprint Retrospective*:
  * one half day,
  * for discussing the methodological issues.

Each morning during a Sprint, the Development Team holds a 15-minute *Daily Scrum*, discussing:

- what has been done the day before,
- what will be done today, and
- what are the current blocking problems and how to overcome them.
Models

Modern software can be very complex. Project managers and developers need to be able to work on a system without knowing all its details.

**Definition:** A *model* is a description of some part of a system:

- at some *level of abstraction*, and
- limited to a subset of its *relevant features*.

In software development, models are used:

- as a *communication medium* between development activities, and
- as *documentation* of problems, design choices, and solutions.
Several classes of models are employed:

- **Conceptual model**: Describes the problems to be solved, and their relation to the needs of the user.

- **Design model**: Specifies design and architectural choices, such as the main system components and the interfaces between them, communication protocols, data representation structures, . . .

- **Implementation model**: Documents the structure and principles of operations of the executable code.

- **Test model**: Keeps track of the test scenarios and strategies used for assessing the quality of the resulting product.
The object-oriented paradigm

**Principles:** At each perspective (conceptual, design, implementation) and level of abstraction, models will be expressed in terms of entities that:

- represent classifiers that can be instantiated,
- are simple and well defined,
- are as much as possible decoupled from each other, and
- encapsulate both data management and operational capacities.

In the framework of software development:

- **Conceptual model:** Entities are concepts, i.e., elements of the problem that are:
  - relevant to the project, and
  - expressed in a language that is understood by the user.
• **Design model:** Entities are the main architectural components of the system:
  – building **blocks** and **subsystems**,  
  – **interfaces** between them,  
  – **protocols**,  
  – ...  

• **Implementation model:** Entities are **classes** in an object-oriented program.

**Illustration:** Cash register software.

- **Conceptual:** *Customer, Cashier, Store, Purchase, Product, ...*
- **Design:** *ProductDatabase, Inventory, UserInterface, BarcodeScanner, ...*
- **Implementation:** *Product, ProductID, ProductLookupTable, ...*
The Unified Modeling Language (UML)

• Collection of several **graphical formalisms**, that can sometimes be intermixed.

• Industrial **standard**.

• Most types of diagrams can be employed at **different levels** of precision:
  – **Sketch**: Specification and discussion of the **important aspects** of the system.
  – **Blueprint**: Complete and **detailed** description, that can directly be translated into code by experienced programmers.
  – **Programming**: Executable code can be **automatically generated** from the model.

**Note**: In this course, we will mainly use UML at the **sketch** level.

• The **syntactic rules** have evolved over successive versions of the language. At the **sketch level**, the emphasis is on **clarity** and **relevance** rather than on strict adherence to those rules.
UML in practice

- Each type of diagram shows *one particular aspect* of a system, subsystem, or component.

  For most problems or parts of a problem, not all diagrams are thus relevant!

- Diagrams can always be *incomplete*: Hypotheses are never made on features that are not represented.

- It is allowed to supplement UML models with *other types of diagrams*, or with textual information.

- In this course, we will only study a *subset* of UML.
Chapter 2

Requirements analysis
Use cases

The first step of requirements analysis consists in collecting and documenting use cases.

Definition: A use case represents a typical interaction between a user and the system.

Example: Development of a word processor. Possible use cases:

- Typing a new text.
- Editing an existing text.
- Generating a table of contents.
- Setting a paragraph in boldface.
Notes:

- The interactions captured in a use case must be observable at the interface between the system and the user.

- Use cases must be expressed at a level of abstraction that:
  - highlights the relevant features of the problem, but
  - leaves away unnecessary details.

- A use case shows how a user’s goal is satisfied by the system.

A use case must always correspond to a high-level goal.

For instance, the operations performed in order to select a new font do not form a valid use case.
Obtaining use cases

A powerful method for finding use cases consists in performing a thought experiment in which:

- one assumes that the development has been successfully completed, and
- a virtual camera films interactions between the system and its users.

Advantages:

- The result contains only observable actions and is naturally consistent.
- The requirements obtained in this way can easily be discussed with the client.

Note: In use-case terminology, an actor corresponds to the role played by an user with respect to the system.
## Example of use case

### Cash purchase

<table>
<thead>
<tr>
<th>Actors</th>
<th>Customer, Cashier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Purchasing products by paying cash.</td>
</tr>
<tr>
<td>Overview</td>
<td>A Customer arrives at the cash register with products to purchase. The Cashier encodes these products, announces the total price, receives payment from the Customer, and gives back change.</td>
</tr>
</tbody>
</table>
### Detailed interaction

<table>
<thead>
<tr>
<th>Actor actions</th>
<th>System actions</th>
</tr>
</thead>
</table>
| 1. A *Customer* arrives at the cash register with products to purchase.  
2. The *Cashier* encodes the identifier of each product.  
If there are more than one unit of the product, then the *Cashier* enters the number of items. | 3. The cash register identifies the product, and displays its name as well as its unit price.  
This number is then displayed. |
<table>
<thead>
<tr>
<th>Actor actions</th>
<th>System actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. When all products have been encoded, the Cashier signals that the encoding is complete to the cash register.</td>
<td>5. The cash register computes and displays the total price.</td>
</tr>
<tr>
<td>6. The Cashier announces the total price to the Customer.</td>
<td></td>
</tr>
<tr>
<td>7. The Customer pays the Cashier.</td>
<td></td>
</tr>
<tr>
<td>8. The Cashier inputs the amount paid by the Customer.</td>
<td>9. The cash register displays the change due to the Customer. The cash register prints a ticket.</td>
</tr>
<tr>
<td>Actor actions</td>
<td>System actions</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>10.</strong> The <em>Cashier</em> cashes the payment, and gets change from the money tray. The <em>Cashier</em> gives back change to the <em>Customer</em>, as well as its ticket.</td>
<td><strong>11.</strong> The purchase is registered.</td>
</tr>
<tr>
<td><strong>12.</strong> The <em>Customer</em> leaves the cash register with his/her purchases.</td>
<td></td>
</tr>
<tr>
<td>Alternatives</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td><strong>2.</strong> The product identifier is unknown. The cash register displays an error message.</td>
<td></td>
</tr>
<tr>
<td><strong>7.</strong> The <em>Customer</em> is unable to pay. The <em>Cashier</em> cancels the whole transaction.</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- The *formalism* in which use cases are written is not strictly specified. It is allowed to group together scenarios sharing the same goal, as well as to include branching paths or exceptional situations, provided that the use cases remain readable and simple to understand.

- A detailed interaction is usually only provided for the *most important* use cases.
A use case diagram depicts the relations between the use cases and the actors of the system. The aim is to provide a high-level view of requirements.

Syntax:
Example

Cashier

Manager

Identification

Cash purchase

Product return

Initialization

Customer
Use case generalization

It is sometimes convenient to express some requirements as abstract use cases that generalize several specialized interactions.

Example: A payment made by a customer is a general concept that can be specialized into

- a cash payment,
- a payment by credit card,
- the opening of a line of credit,
- ... 

In line with the object-oriented approach, the corresponding use cases are then linked to each other by a generalization relation.
Illustration and notation:
Use case inclusion

In the previous example, the interaction represented by the use case *Payment* is likely to be part of other use cases, such as *Purchase* or *Rental*.

Such an inclusion of a use case in another is denoted by a relation *stereotyped* by the qualifier <<include>>.

Illustration:
Notes:

- Remember that use cases correspond to high-level user’s goals. Generalization and inclusion relations must only be used when they improve the readability of the model. (In doubt, it is better to omit them.)

- Other stereotypes can be employed in use case diagrams, such as <<extend>>, but they are not useful in most cases.

- The generalization relation can also be used between actors.
Use case prioritization

An important activity in use-case driven development processes is to select the use cases that will be tackled during the next development cycle.

Guidelines:

- The use-case diagram(s) provide a suitable level of abstraction for reasoning about use case priorities.
- In order to keep development cycles short, each of them should address only a small number of use cases, aimed at a common goal.
- The priority should be given to problems that:
  - influence architectural decisions, and
  - present the highest risk of not being successfully solved.
Example: Cash register.

1. By starting with the use case *Cash purchase*, one first defines a basic architecture for:
   - managing *products*, and
   - handling *sales*,

   which are two essential features.

2. The realization of the use case *Initialization* leads to designing additional components such as an *inventory*, an *accounting* system, . . .

3. The use case *Product return* can now be implemented.

4. Finally, addressing the use case *Identification* only requires minor modifications to the existing product.
Chapter 3

Static diagrams
Introduction

Goal: Static diagrams are aimed at describing the object-oriented features of:

- components of the problem (conceptual model),
- software subsystems, main components, and interfaces (design model), or
- actual code elements (implementation model).

Notes:

- At each perspective, diagrams can be drawn at various levels of abstraction.
- The semantics of a diagram depends on the perspective.
Class diagrams

Principles:

The main graphical element represents a classifier, the nature of which (concept, subsystem, interface, class) depends on the current perspective.

In the rest of this chapter, we use the term class to refer to any of those classifiers, depending on the perspective.

Notation:
Notes:

- The **attributes** correspond to **information** managed by an instance of the class:
  - **data** relevant to the problem of interest at the conceptual perspective,
  - **variables** at the design or implementation perspectives.

- The **operations** define the **behavior** of an instance:
  - **main functionalities** for concepts,
  - **responsibilities** and **methods** in the design and implementation models.

**Note:** Instance **data** management operations are **implicitly defined**, and do not need to appear on diagrams.

- The **visibility** ("+": **public**, "-": **private**, ": **protected**) can have a **platform-dependent** semantics.

- **Class** (as opposed to instance) elements are **underlined**.
Associations

In addition to employing attributes, data can also be represented by associations linking two or more classes.

Principles:

- An association represents data that jointly involves instances of several classes.
- An instance of an association is an element of the Cartesian product of the relevant sets of instances.

Notation:

- Binary association:
• *n*-ary association (*n* > 2):
Multiplicity

The multiplicity of an association branch represents a constraint on the possible number of instances of this branch when the value of the other branches is fixed.

Notation:

- \( 1 \) exactly one
- \( \ast \) any number
- \( 1..\ast \) at least one
- \( 2..10 \) between two and ten
Notes:

- In addition to multiplicities, association branches can also be labeled with constraints. In particular, the constraint "\{ordered\}" specifies that the instances must be kept in some given order.

**Example:**

![Diagram showing Movie, Sequence, and Scene with multiplicities and constraints]

- Multiplicities can also label associations in use-case diagrams.

**Example:**

![Diagram showing Player and Play a game with multiplicities]

46
Attributes vs associations

Information involving **multiple classes** can sometimes be represented either by **attributes** or by **associations**.

**Example:**

- items: `OrderLine [*] {ordered}`

**Alternative diagram:**

```
Order
   ──items: OrderLine [*] {ordered}
       multiplicity
       constraint

Order
    1
    items
    * {ordered}
```
Guidelines:

- Primitive data is better represented by attributes.
- Information related to classes that are present in a given diagram must be represented by an association in that diagram.

Illustration:

Invalid

<table>
<thead>
<tr>
<th>Cashier</th>
<th>Cash Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>num</td>
</tr>
<tr>
<td>id</td>
<td>idCashier [0..1]</td>
</tr>
<tr>
<td>numRegister [0..1]</td>
<td></td>
</tr>
</tbody>
</table>

Valid

<table>
<thead>
<tr>
<th>Cashier</th>
<th>Uses</th>
<th>Cash Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>0..1</td>
<td>num</td>
</tr>
<tr>
<td>id</td>
<td>0..1</td>
<td></td>
</tr>
</tbody>
</table>

Navigability expresses the ability of an object to compute the set of instances of an association to which it belongs.

Example and notation:
Notes:

- Navigability is usually only relevant at the design and implementation perspectives.
- **Bidirectional navigability** is denoted by a double arrow ("←→").
- In order for an instance of a class $C_1$ to be able to send a message to an instance of a class $C_2$, it must often be possible to navigate from $C_1$ to $C_2$. 
Examples:

```java
class Customer {
    ...  
}

class Product {
    private Customer owner;
    ...  

    public Customer getOwner() {
        return this.owner;
    }
}
```
class Customer
{
    private HashSet<Product> ownedProducts;
    ...

    public Iterator<Product> getOwnedProducts() { ... }  
    public boolean isOwned(Product p) { ... }  
    public void addOwnedProduct(Product p) { ... }  
    public void removeOwnedProduct(Product p) { ... }  
    ...
}
Association classes

It is possible to add **attributes** and/or **operations** to an association, which then become part of **every instance** of this association.

**Example and notation:**

![UML Diagram](attachment:image.png)
Notes:

- Defining an association class **does not amount** to replacing an association by a class.

For instance, the following diagram is **not equivalent** to the previous one:

![Diagram](image)

Indeed, the former diagram prevents a person from working **more than once** for a given company, which is allowed by the latter.
• The semantics of a ternary association is also different.

Indeed, with the diagram on the previous transparency, if a person $P_1$ works for a company $C_1$ and a person $P_2$ works for a company $C_2$ during the same period, then $P_1$ also has to work for $C_2$, and $P_2$ for $C_1$, which is absurd!

The diagram on this page does not have this shortcoming.
Composition and aggregation

*Composition* and *aggregation* are two particular types of associations, expressing membership relations between a component and composite entity.

**Composition:** A component belongs to **one and only one** composite object.

```
Composite Class <---> Component Class
```

**Aggregation:** A component may belong to **several** aggregate objects.

```
Aggregate Class <-> Component Class
```
Example:

Notes:

- Composition makes it possible to model constraints that cannot be expressed with only multiplicity.

Example: An instance of “Point” cannot be associated at the same time to instances of “Triangle” and “Circle”.
• Composition is often used for modeling membership relations between physical objects, e.g., the parts of a car engine.

• On the other hand, aggregation is applicable to abstract concepts. (When in doubt about the nature of an association, it is better to not consider it as aggregation.)

• Multiplicity constraints are only meaningful on the distant branch of composition and aggregation associations. Other constraints such as “{ordered}” can also be employed.

Example:
A dependency relation between classes $C_1$ and $C_2$ indicates that a modification of $C_2$ can potentially affect $C_1$.

Syntax:

![Diagram of dependency relation between $C_1$ and $C_2$]

Notes:

- There can be several causes for dependency: The class $C_2$ may appear as the type of a parameter or return value in a method of $C_1$, the class $C_2$ may be instantiated by operations performed by $C_1$, ...
• The nature of a dependency can be explicited by a stereotype, e.g., “<<create>>” for instantiation.

• Only non-trivial dependency relations have to be explicitly represented.
Generalization

Classes that share common features can be expressed as specializations of a general class that captures those features.

If $C_2$ specializes $C_1$, then $C_1$ is a generalization of $C_2$.

Notation:
Note: The following notation is also allowed (with an identical semantics).
Inheritance

Specializations of a class inherit all its properties, including

- attributes,
- operations, and
- associations with other classes.

Example:
In this model, each of the classes “Cash Payment”, “Credit Card Payment” and “Check Payment”

- has the attribute “amount”, and
- is associated to the class “Purchase” by the relation “Pays”.
The substitution principle

If two classes are linked by the generalization relation, then they must ideally satisfy the following property:

Every instance of the specialized class must be able to play the role of an instance of the general one.

In other words, it must be possible to substitute an instance of the general class with an instance of the specialized one in all applications, without breaking correctness.

Informally, this property can be checked by checking that the sentence

⟨ specialized class ⟩ is a ⟨ general class ⟩.

is both meaningful and valid.

Example: A credit card payment is a payment.
Multiple classification

In some cases, there exist several ways of specializing a general class.

**Example:** Human resources management in a hospital:

Every physician is also either a man or a woman. An instance of the concept *Physician* can therefore belong at the same time to multiple specializations of this concept.
Solution: In a class diagram, one can label generalization relations with generalization set labels, which define the allowed combinations of specializations.

Principles:

- An instance of a general class cannot belong simultaneously to two specialized classes belonging to the same generalization set.
- When a generalization set is labeled with the constraint “{ mandatory }”, every instance of the general class must necessarily be an instance of one and only one specialized class belonging to this generalization set.
Example:

Note: When a generalization set label is missing, the corresponding specialized class belongs to a particular generalization set identified by the empty label.
Abstract classes

Following object-oriented principles, a class that cannot be instantiated is said to be abstract. Abstract classes are often used for grouping together the common elements of a set of specialized classes.

Example:
Note: In diagrams drawn by hand, the constraint “{abstract}” can be used instead of writing class names in italics.
An abstract class can have a **partial implementation**. In contrast, a class that **does not implement** any of its methods is called an **interface**.

In UML class diagrams, interfaces can be represented in two different ways:

- either as regular classes labeled with the stereotype `<<interface>>`

  The following notation is then used for indicating that a class \( C \) **implements** an interface \( I \):
• or by using the **ball and socket** notation.

In the following diagram, the class $C_1$ **requires** the interface $I$, which is **implemented** by the class $C_2$:
Examples:

- The diagram

![Diagram]

...can be condensed into

![Condensed Diagram]
The diagram

simplifies into
Derived attributes and associations are elements of a class diagram whose value can be derived from other elements.

**Notation:** The name of the attribute or association is prefixed with the symbol “/”. A constraint can also be provided for specifying how to compute the value of the element.

**Examples:**

- Derived attributes:
• Derived association: (windowed user interface)
Generic classes

In several object-oriented languages, the programmer can define generic classes or templates, the declaration of which relies on one or several parametric types.

Example and notation:

There are two ways of binding the value of the type parameter:

- by suffixing the name of a class by an expression of the form
  
  \[ < \text{TypeParameter} :: \text{TypeValue}, \ldots > \]
Example:

- by using a generalization relation **stereotyped by** `<<bind>>`.

Example and notation:

```
Set<T :: Product>
```

```
+add(T)
+remove(T)
+boolean contains(T)
```

```
ProductSet
```

```
Set
<<bind>>
<T :: Product>
```

```
+add(T)
+remove(T)
+boolean contains(T)
```
Stereotyped classes

In some cases, it is useful to partition the classes appearing in a diagram according to their properties. Stereotyped classes can be defined either:

- by a stereotype label prefixing their name ("<< ... >>"),
- by a stereotype icon added to their name, or
- by such an icon replacing their class box.

Example: In the conceptual model, it is often useful to distinguish:

- boundary concepts that handle interactions between the actors and the system: 🟢
- control concepts that coordinate operations: ⚙️
- entity concepts that manage persistent data: 📊
Illustration: (cash register)

Note: During the design and implementation activities, each of these concepts will be replaced by one or several classes, with their detailed features documented in the design and implementation models.
Object diagrams

Object diagrams are a formalism similar to class diagrams, but with basic elements that represent instances of classifiers (e.g., objects at the design or implementation perspectives).

Those diagrams are used for providing examples of objects created at runtime, showing the associations between them.

Example and notation:

```
<table>
<thead>
<tr>
<th>toolsCatalog : ProductCatalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>: Product</td>
</tr>
<tr>
<td>name = “hammer”</td>
</tr>
<tr>
<td>Contains</td>
</tr>
<tr>
<td>: Product</td>
</tr>
<tr>
<td>name = “hand saw”</td>
</tr>
<tr>
<td>Contains</td>
</tr>
<tr>
<td>: Product</td>
</tr>
<tr>
<td>name = “pliers”</td>
</tr>
</tbody>
</table>
```
Package diagrams

These diagrams make it possible to group logically classes that heavily depend on each other.

The aim is to simplify the dependency structure between those classes.

Notation:
Example:
Note: Like class diagrams, package diagrams can be drawn at different perspectives (conceptual, design or implementation).
Chapter 4

Dynamic diagrams
Activity diagrams

An activity diagram depicts:

- the actions performed by the system, its components, or the users,
- the order in which these actions can be carried out, and
- the dependencies between actions.

Notes:

- At the conceptual perspective, those diagrams help to define a full specification of the workflows illustrated by use cases.
- At the design and implementation perspectives, activity diagrams are useful for documenting processes.
- The advantage of activity diagrams over flowcharts is their ability to represent independent actions.
Example and notation: (sequence of actions)

Indicates that Action 2 can only start after Action 1 is finished.
Forks and joins

A fork after an action splits the flow into independent branches, which means that these branches can be followed:

- in any order, or
- concurrently.

Notation:
Conversely, a join is used for reuniting independent branches.

**Notation:**

Note: Forks and joins can be generalized into synchronization bars with any number of incoming and outgoing flows:

- A bar can be crossed only when all its incoming actions are completed.
- After crossing a bar, its outgoing actions can be started, and then performed independently from each other.
Decisions

A decision is an action that is followed by one branch among several possibilities, this branch being selected by evaluating guards.

- The guards labeling the outgoing branches from a decision must be mutually exclusive.
- The special guard “[else]” corresponds to the cases that are not handled by the other branches.
- A merge is employed for reuniting decision branches.

Notation:
Look for Beverage

[no coffee]

[found ground coffee]

Place Filter

Fill in Water Tank

Take Cup

Open Can

[found soda can]

Fill in with Ground Coffee

Switch on Coffee Machine

Wait

Pour Coffee

Drink!
Subactivities

In an activity diagram, an action can itself be decomposed into a combination of other actions, described by another activity diagram.

Notation:

- The decomposed action is distinguished by a rake symbol “resents”.
- The corresponding subactivity diagram is enclosed in a box labeled by the action name, with the parameter(s) and returned value(s) of this action appearing at the boundary of this box.

Example:
The details of the action *Check Payment* are then provided by the following diagram:
Activity diagram partitions

It is sometimes useful to specify in an activity diagram which of the actors and system components is responsible for each action.

This can be achieved by partitioning the diagram into different areas associated to these actors and components.

Note: Partitions shaped as vertical bands are often called swim lanes.
Notation:

Component 1

actor, system, or subsystem

Component 2

Component 3
Special actions

In order to represent **interactions** with outside components, as well as **timed behaviors**, the following special actions are defined:

- **Signal emission:**

- **Signal reception:**

- **Time signal:**
Example: (access to a protected resource)
Expansion regions

Actions or group of actions that have to be performed multiple times can be specified with the help of expansion regions.

Notation and example:

Note: The keyword is:

- \texttt{<<concurrent>>} if iterations can be performed in parallel, or
- \texttt{<<iterative>>} if sequential operations are required.
Flow finals

Inside an expansion region, an individual iteration can end without terminating the global flow of action. This is indicated by a flow final.

Notation and example:
State diagrams

State diagrams describe components that switch between several modes of operation, in which they have different behaviors.

A state diagram is associated to an instance of a concept, subsystem, or class, depending on the perspective. It shows:

- the states that can be taken by this instance,
- the transitions that can be followed for moving from a state to another, and
- the events (either internal or external) that lead to following transitions.

Note: A state diagram does not need to represent all the events affecting the underlying object: Events that do not appear on a diagram are ignored, i.e., they do not modify the current state. This feature is useful for drawing state diagrams at different abstraction levels.
States

A state represents a particular mode of operation that an object can have at a given time.

An object always stays in its current state for a non-zero amount of time. During this time, it can

- perform some actions, and/or
- wait for events.

Notation:
Transitions specify the reactions of the object to events, which may include modifying its current state, as well as performing actions. There are two types of transitions:

- **Internal transitions**, that react to an event without modifying the current state. These transitions are defined inside the corresponding state symbol.

- **External transitions**, representing an action that can either be spontaneous, or triggered by an event. These transitions are denoted by simple arrows ("→") linking their origin and destination states.
Transition elements

Transitions labels admit the following general form:

\[ \text{event} [ \text{guard} ] / \text{action}, \]

where:

- "event" is the name of the event that leads to following the transition.
- "guard" expresses a condition that must be satisfied in order to follow the transition.
- "action" defines the operation that is performed when the transition is followed. In the case of an external transition, this operation must have an instantaneous effect.

Note: Some or all of those elements can be omitted.
Special events

Three special events are defined for internal transitions:

- **entry** and **exit** are triggered when (respectively) **entering** and **leaving** the state.

- **do** is used for specifying actions that are carried out whenever the object **stays in the corresponding state**.

Notes:

- Internal transitions triggered by these three special events **cannot have guards**.

- The detail of actions associated to the **do** event can be provided on other diagrams.

- Transitions with **identical origin and destination** states trigger entry and exit events, but **internal transitions do not**.
Example of state diagram

Order management system. The following diagram describes the behavior of the concept “Order”:
Composite states

A subset of states as well as the transitions linking them can be grouped into a composite state, or superstate.

Principles:

- A transition ending in a composite state is immediately followed by the initial transition of this state.
- A transition originating from a composite state can be followed from every state that belongs to it.
Concurrent states

The area inside a composite state can be partitioned into concurrent zones that are independent from each other. At any time, the object of interest then occupies one state in each of these zones. Concurrency ends when leaving such a composite state.

Notation:
Example: Parallel handling of an order and its payment:
Sequence diagrams

Sequence diagrams are a particular type of interaction diagrams. Their goal is to describe how objects collaborate with each other in order to perform a given task.

A sequence diagram depicts:

- a selection of objects involved in a given operation, and
- a sequence of actions performed by these objects, presented in chronological order. Such actions include:
  - sending message to other objects, and
  - creating and deleting objects.
Notation:

- The basic elements are objects, placed horizontally on top of the diagram.
- Each object is associated with a lifeline, on which time progresses from top to bottom.
- Lifelines are decorated with activation bars that show when the corresponding objects are active (i.e., have a method currently executing).
Messages

A message sent by an object to another is denoted by an arrow linking their lifelines.

- The shape of the arrowhead corresponds to the type of method call:
  - “──”: synchronous call (the caller is suspended while the invoked method is being executed).
  - “→”: asynchronous call (the invoked methods run concurrently with the calling thread).

- The first message of a sequence may come from an unspecified source.

  Notation:  

  The instant at which a method returns can optionally be specified by a dashed arrow (“→”) towards the caller, possibly labeled with a returned value.
Example:
Nested activation bars

When an object needs to invoke its own methods, the following notation makes it possible to distinguish the operations of such methods from those of their caller:

The invoked method is active.

The caller method is active.
Objects creation and deletion

Notation:

- **Instantiation:**
  - object
  - `new`
  - instantiated object
  - constructor execution

- **Deletion:**
  - by the object itself:
– by another object:
Example: (transaction authentication)
Interaction frames

Actions that need to be performed multiple times, or that are subject to a condition can be enclosed in interaction frames. Those frames can be nested. Several types of interaction frames are defined:

- **“alt”:** The frame is subdivided in regions corresponding to mutually exclusive guards. Only the region labeled with a true guard is executed.

- **“opt”:** Equivalent to an alt frame with only one internal region.

- **“loop”:** The contents of the frame are executed iteratively, while the specified guard remains true.

  **Note:** The guard can also be used for specifying a set of objects that perform an identical operation.

- **“neg”:** Shows an invalid sequence of actions.
Examples:
Communication diagrams

Like sequence diagrams, communication diagrams are a particular type of interaction diagrams, aimed at describing object interactions.

Compared to sequence diagrams, communication diagrams are better suited for specifying complex patterns of collaborations between objects, whereas sequence diagrams emphasize the chronological order of operations.

Principles: A communication diagram is a graph in which:

- The nodes correspond to the objects involved in the interaction.
- The edges (or links) represent the messages exchanged between objects, as well as special operations such as object instantiations and deletions.
- Sequence numbers define the order in which operations are performed.
Note: In order for an object $e_1$ to send a message to an object $e_2$, one must usually be able to navigate from $e_1$ to $e_2$ on the corresponding class diagram.
Sequence numbers

In each link label, the actions are prefixed by a sequence number that defines the order in which they are performed. Sequence numbers are nested in order to emphasize the structure of object interactions.

Example:
Iterations

Actions that need to be performed multiple times, or by several instances of the same class, can be labeled by iteration markers of the form

\[ \ast[\text{description of the iteration}], \]

following the sequence number.

When two actions

- share the same iteration marker, and
- have sequence numbers that are identical except for their last number,

they are considered to be both performed (sequentially) in the same iteration.
Example 1:

```c
msg1()
{
    for (i = 1; i <= 10; i++)
    {
        b.msg2();
        c.msg3();
    }
}
```

Diagram:

```
msg1()

: A

1° [i := 1..10]: msg2()

b: B

2° [i := 1..10]: msg3()

c: C
```
Example 2:

```c
msg1()
{
    for (i = 1; i <= 10; i++)
        b.msg2();

    for (i = 1; i <= 10; i++)
        c.msg3();
}
```

Diagram:

```
msg1()

A
1.1*[i := 1..10]: msg2()

B

2.1*[i := 1..10]: msg3()

C

: A --→ B

: B --→ C
```
Conditional operations

The different branches of a *conditional choices* are distinguished by a *guard* of the form

\[
[\text{condition}],
\]

as well as by a *branch label* following the sequence number (e.g., 2a, 2b, 2c, . . . ).

Example:

```
2 : msg6()

1b [¬cond1] : msg4()

msg1()

1a [cond1] : msg2()

:E

:A

:B

:D

:C
```
Objects instantiation and deletion

Instantiation and deletion of objects can respectively be denoted by the special messages “new” (or “create”) and “delete”.

In addition, it is possible to specify on the diagram the dynamic status of some objects:

- “<<new>>” for objects that are created in the scenario described by the diagram,
- “<<deleted>>” (or “<<destroyed>>”) for objects deleted during the interaction,
- “<<transient>>” for temporary objects that are created and then deleted.
Example:

order manager

1: prepare()

:Order

1.1*[all ordered products]: prepare()

:OrderedProduct

1.1.3[inStock]: new

<<new>>

:DispatchedProduct

1.1.1: inStock := lookupStock()

1.1.2[inStock]: removeFromStock()

1.1.2.1: toResupply := emptyStock()

:ProductInStock

1.1.2.2[toResupply]: new

<<new>>

:ProductResupply
Chapter 5

Software patterns
Introduction

In many fields of engineering, problems are usually solved by combining well-established solutions to elementary subproblems.

That such elementary solutions are available, well studied, and commonly exploited is a good indicator of the maturity of the domain.

Example: Electronic circuits are generally designed by assembling basic building blocks with well understood properties: power supply, amplifiers, filters, converters, communication buses, ...
Software patterns

Some problems tackled during software development can also be solved by exploiting well-studied elementary solutions.

**Definition:** An object-oriented software pattern (or design pattern) is a general solution to an elementary programming problem.

There are several advantages to using patterns during software development:

- Part of the experience accumulated by developers can be captured and reused.
- Recurring problems can be solved more quickly, and by tried and tested solutions.
- Implementation choices can be consistently documented.
The form of a software pattern

A software pattern is described by the following elements:

- A name that identifies the pattern without risk of ambiguity.
- A description of the problem that it aims to solve.
- A solution to the problem, expressed as a structure of object-oriented software components.
- A discussion of the advantages and potential drawbacks of that solution.
Patterns classification

The software patterns that will be studied in this course can be classified as follows:

- **Creational** patterns deal with object instantiation problems.
- **Structural** patterns are concerned with class or object composition problems.
- **Behavioral** patterns target interaction problems between objects and/or classes.

Software patterns can alternatively be classified depending on the object-oriented mechanisms they depend on:

- **Class** patterns mainly exploit the generalization relation between classes and subclasses.
- **Object** patterns essentially rely on associations between objects.
5.1

Creational patterns
1. Abstract Factory

**Category**: Creational, object.

**Goal**: Providing an interface for instantiating mutually dependent objects without knowing their actual class.

**Motivating problem**: Development of a *Graphical User Interface (GUI)* for an application that needs to be released for several runtime platforms: Windows, X-Window, . . .

Each platform provides a library with its own classes for the GUI objects that are manipulated: Buttons, Menus, Windows, . . .

**Problem**: How can the code responsible for creating GUI components be made independent from the choice of runtime platform?
Bad solution:

Drawbacks:

- Adding a **new runtime platform** requires major modifications.
- The application is responsible for **ensuring consistency** between the GUI components (for instance, guaranteeing that **X buttons** are only used inside **X windows**).
Better solution:

Principles:

- The class *ElementsFactory* provides a **unique interface** for instantiating GUI elements.
- The application has only an **abstract view**, independent from the runtime platform, of the GUI elements.
General form of the pattern:
Advantages:

- The actual classes of the instantiated objects can remain unknown by the application.
- Porting the application to additional runtime platforms becomes easy.
- The dependence constraints between GUI elements are enforced by the pattern.

Drawback:

- Modifying the set of GUI elements handled by the abstract factory is tedious.
2. Prototype

**Category:** Creational, object.

**Goal:** Creating new objects from a prototype instance.

**Motivating problem:** Development of a musical notation editor based on a generic toolbox for building graphical editors. This toolbox contains:

- an abstract class *Glyph* representing the graphical objects that can be manipulated,
- an abstract class *Tool* corresponding to the manipulation tools, and
- a subclass *CreationTool* of *Tool*, representing tools that are able to create new instances of *Glyph*. 
Problem: How can we keep *CreationTool* generic, but make it able to instantiate the classes that are specific to our application (e.g., *Note*, *Rest*, *BarLine*, …)?

Bad solution:
Drawbacks:

• Duplicating the glyphs hierarchy is needlessly complex.

• The solution is too rigid: The glyphs that can be created by *CreationTool* have to be fully specified at compile time.

Better solution:

• Each glyph that can potentially be created by *CreationTool* is defined by a prototype instance, used as a blueprint by the instantiation operation.

• Instances of *CreationTool* are associated to glyph prototype objects at runtime.
Illustration:

```
p = prototype.clone();
while (mouseIsDragged())
  p.draw(mousePosition());
  canvas.add(p);
returns a copy of the object
```
General form of the pattern:

Prototype

- clone()

ConcretePrototype1

- clone()

ConcretePrototype2

- clone()

returns a copy of the object

Client

- operation()

... prototype.clone(); ...

...
Advantages:

- Like in the Abstract Factory, the actual classes of instantiated objects can remain unknown by the application.
- Prototype objects can be created dynamically, be parameterized, or even be obtained by composing simpler objects at runtime, ... 

Drawback:

- The cloning operation can be tricky to implement.
3. Factory Method

Category: Creational, class.

Goal: Providing an interface for instantiating objects, the class of which is chosen by a subclass.

Motivating problem: Development of an office application composed of several programs sharing part of their user interface: text processor, spreadsheet software, ... One has defined:

- an abstract class *Program* that implements the common functionalities of all programs, and
- an abstract class *Document* that specifies the common features between all the documents handled by the programs.
Problem: When a method implemented by *Program* needs to create a new *Document*, how can it know the class to be instantiated?

Solution: The responsibility of instantiating a *Document* can be delegated to the concrete subclasses of *Program*:
General form of the pattern:

```
... product = factoryMethod(); ...
```
Advantages:

- With this pattern, the code responsible for creating objects can be shared between different classes.
- This pattern can also be used for connecting together parallel hierarchies of classes.

**Example:** Graphical editor in which each glyph is manipulated in its own way:
Drawback: In the case where the instantiation operation is the only one to be shared by the different classes, the class hierarchy becomes excessively complex.

Note: Abstract Factories can be implemented with the help of Factory Methods, or Prototypes . . .
4. Builder

**Category:** Creational, object.

**Goal:** Separating the operations for building a complex object from the representation of this object.

**Motivating problem:** Development of a word processor, that can export documents in several output formats: PDF, HTML, ASCII, . . .

**Problem:** How can we organize the code responsible for exporting a document in such a way that adding new output formats will become easy?
Bad solution:

```java
boolean stop = false;
while (!stop)
{
  t = getNextToken();
  switch(t)
  {
    case END:
      stop = true; break;
    case CHAR:
      outputHTMLChar(t.c); break;
    case PARA:
      outputHTMLPara(); break;
    ...
  }
}
...

boolean stop = false;
while (!stop)
{
  t = getNextToken();
  switch(t)
  {
    case END:
      stop = true; break;
    case CHAR:
      outputASCIIChar(t.c); break;
    case PARA:
      outputASCIICarriageReturn(); break;
    ...
  }
}
...

boolean stop = false;
while (!stop)
{
  t = getNextToken();
  switch(t)
  {
    case END:
      stop = true; break;
    case CHAR:
      outputPDFChar(t.c); break;
    case PARA:
      outputPDFPara(); break;
    ...
  }
}
...
```

**Drawback:** The code is redundant. If the internal representation of documents is modified in a subsequent development cycle, the code of all export classes will need to be adapted.
Better solution:

```java
boolean stop = false;
while (!stop)
{
    t = getNextToken();
    switch(t)
    {
    case END:
        stop = true; break;
    case CHAR:
        builder.convertChar(t.c); break;
    case PARA:
        builder.startParagraph(); break;
    ...
    }
}
```

**Advantage:** The operations over the internal structure of documents are well decoupled from those related to the output format.
General form of the pattern:

```java
for (...) {
    builder.buildPart(...);
}
```

```
Director
  build()

Builder
  buildPart()
  builder

ConcreteBuilder
  buildPart()
  getResult()

Product
```
Typical interaction:
5. Singleton

Category: Creational, object.

Goal: Ensuring that a class can only be instantiated once, and providing global access to the resulting object.

Motivating problem: Development of a desktop environment. Some objects such as the window manager or the printing manager must be unique, as well as accessible to all applications.

Bad solution: Making these objects accessible by means of global variables.

Drawback: Multiple instantiation remains possible.
Better solution:

```
Singleton

-uniqueInstance

+instance()

+instanceOperation()

return uniqueInstance;
```

**Principles:** The class itself is responsible for its own instantiation, as well as for providing access to its unique instance.
Advantages:

- Compared to using global variables, namespace conflicts are naturally prevented.

- This pattern can easily be generalized to classes that can only be instantiated a bounded amount of times.

Notes: The patterns Abstract Factory, Prototype and Builder can include Singleton in their implementation.
5.2

Structural patterns
1. Bridge

**Category:** Structural, object.

**Goal:** Decoupling an abstraction from its implementation, in order to allow them to evolve independently from each other.

**Motivating problem:** Like the Abstract Factory example (cf. Ch. 5.1), one develops an application with a GUI that needs to be ported to several runtime platforms: Windows, X-Window, . . .

**Problem:** The code responsible for drawing window contents needs to call different primitive functions for each runtime platform. How can this code be organized so as to keep it manageable when the set of runtime platforms is updated?
Bad solution:

Drawbacks:

- The class hierarchy is too complex. Indeed, for each type of window, a different class has to be defined for each runtime platform.

- The code implementing the drawing operations is redundant.
Better solution:

Principles: The abstraction of the primitive drawing operations performed by the windows is decoupled from the implementation of these operations.
General form of the pattern:

```
impl.implOperation();
...  ...  ...
abstractOperation();
```

```
Impl::abstractOperation();
```

```
implOperation();
```

```
Implementation
implOperation();
```

```
ConcreteImpl1
implOperation()  ConcreteImpl2
implOperation()
```

```
Impl::implOperation();
```

```
RefinedAbstraction()
operation();
```

```
Implementation
implOperation();
```

```
ConcreteImpl1
implOperation()  ConcreteImpl2
implOperation()
```

```
impl::implOperation();
```

```
...  ...  ...
abstractOperation();
...  ...
```
**Advantage:** The code that needs to call the primitive drawing functions and the code that implements those functions can evolve independently from each other.

**Drawback:** The common interface to the primitive operations that are available on all runtime platforms must be carefully specified, and cannot be easily modified.

**Note:** The objects composing a bridge can be instantiated thanks to an Abstract Factory (cf. Ch. 5.1).
2. Decorator

**Category:** Structural, object.

**Goal:** Make it possible to add dynamically responsibilities to an object.

**Motivating example:** Development of a generic **toolbox** for building graphical user interfaces. Some **common functionalities** can (optionally) be assigned to GUI components: **window borders, scrollbars, ...**

**Problem:** How can the implementation of those common functionalities be **shared** between the involved classes?
Bad solution:

```
GUIComponent
  draw()

TextArea
  draw()

TextAreaBorder
  draw()

Canvas
  draw()

Border
  drawBorder()
  --width

Scrollbar
  goUp()
  goDown()

CanvasBorderScrollbar
  draw()

ScrollBarOption
```

Drawbacks:

- Use of multiple inheritance.
- The large number of possible combinations of options complicates the class hierarchy.
- The possible choices of options are fixed at compile time.
Better solution:

![Class diagram with GUIComponent, TextArea, Canvas, Decorator, BorderDecorator, and ScrollbarDecorator]

Principles:

- The interface of a `Decorator` includes the interface of an `GUIComponent`. A `Decorator` can thus dynamically take the place of a `GUIComponent`.
- Decorators can be chained together.
- Decorators are free to add specific operations to their interface.
General form of the pattern:

```
Component
 (operation())

ConcreteComponent
  (operation())

Decorator
  (operation())

ConcreteDecorator
  (operation())
  (additionalOperation())
  (additionalData)

super.operation();
additionalOperation();
```

(component)
Advantages:

- The choice of optional functionalities can easily evolve.
- The class hierarchy remains simple.

Note: If the problem consists in adding a unique functionality to an existing class hierarchy, it is not necessary to define an abstract class *Decorator*.

Drawbacks:

- Since decorators take the place of the manipulated objects, comparing such objects becomes more difficult.
- If the class located on top of the hierarchy of manipulated elements stores a large amount of data, this pattern leads to wasting away memory.
3. Adapter

Category: Structural, object or class.

Goal: Modifying the interface of a class in order to make it compatible with the needs of a specific application.

Motivating problem: Consider once again the development of a GUI toolbox. In order to simplify the code, it would be helpful for the class `TextArea` to reuse the implementation of a class `Text` developed for another application. How can this be achieved if the classes `TextArea` and `Text` have different interfaces?
Solution:

Principles: The class *TextArea* maintains an association to the class *Text*, used for sending messages to this class, but implements a different interface.

Note: Another possibility is to use multiple inheritance, and then define the class *TextArea* as a subclass of *Text*. 
General form of the pattern:

(object)

(class)
4. Flyweight

Category: Structural, object.

Goal: Handling efficiently a large combination of similar objects.

Motivating problem: Development of a word processor. Texts are composed of characters assembled into rows and columns.

Problem: How can a text be efficiently represented?
Bad solution:
**Drawback:** The amount of memory needed for representing all the *Character* objects is prohibitively large.

**Better approach:** Identical characters can be represented by a single object.
Good solution:

Principles:

- For each object, one distinguishes:
  - its **intrinsic** state (which is stored in the object), and
  - its **extrinsic** state (managed by other objects).

- Objects that share the **same intrinsic state** do not have to be replicated.
General form of the pattern:

```java
FlyweightFactory

getFlyweight(key)

if (flyweights[key] == null)
{
    f = createFlyweight(key);
    flyweights[key] = f;
}
return flyweights[key];

ConcreteFlyweight

operation(extrinsicState)

intrinsicState

Flyweight

operation(extrinsicState)

UnsharedConcreteFlyweight

operation(extrinsicState)

intrinsicState

Client

completeState
```
Advantages:

- The object-oriented approach can be followed, even if the number of objects that have to be manipulated is very large.

- The extrinsic state of objects does not necessarily have to be stored, it can also be computed on the fly.

Drawbacks:

- It must be possible to represent efficiently the extrinsic state of sets of objects.

- Comparison operations between objects cannot depend on data stored in their extrinsic state.
5. Proxy

**Category:** Structural, object.

**Goal:** Replacing an object by a placeholder that controls access to this object.

**Motivating problem:** Development of a WWW browser. Downloading the images contained in a HTML page can be long and costly. A good solution is to restrict this operation to the images that are currently visible.

**Problem:** In order to layout the text of the page, the dimension of images has to be known in advance. How can this layout operation be performed without downloading all the images of the page?
Good solution:

Principles: Images are replaced by proxy objects that download them only when needed.
General form of the pattern:

Client \rightarrow Subject

Subject
  \rightarrow RealSubject
  \rightarrow Proxy

RealSubject \rightarrow Proxy

... 

realSubject.request();
...
**Advantages:** A proxy object can provide various services:

- Ensuring that **costly objects** are only created when they are needed.
- Providing a **local representation** of objects that exist outside of the program.
- Managing **access rights** to objects.
- Adding functionalities to **object pointers**: reference counters, locks, . . .
5.3

Behavioral patterns
1. Iterator

**Category:** Behavioral, object.

**Goal:** Accessing the elements of a composite object in sequential order, independently from the internal representation of this object.

**Motivating problem:** Development of a data structure for representing an ordered list of objects, that can be enumerated. Many implementation choices are possible: array, linked list, . . .

**Problem:** How can the methods responsible for enumerating list elements be made independent from the representation of lists?
Bad solution:

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbElements()</td>
</tr>
<tr>
<td>add(element)</td>
</tr>
<tr>
<td>remove(element)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>beginEnumeration()</td>
</tr>
<tr>
<td>next()</td>
</tr>
<tr>
<td>finished?()</td>
</tr>
<tr>
<td>currentElement()</td>
</tr>
</tbody>
</table>

**Drawback:** Two objects cannot traverse simultaneously the *same list.*
Better solution:

Principles: The responsibility of enumerating the elements of composite objects is assigned to a separate class.

Drawback: The class ListIterator is specific to List (and thus depends on its implementation details), even though the concept of list traversal is more general.
Improved solution:

Principles:

- The common features to all list variants are shared in a single class;
- A Factory Method (cf. 5.1) is responsible for instantiating the proper iterator.
General form of the pattern:

```java
class ConcreteComposite
{
    private ConcreteIterator iterator;

    public ConcreteComposite()
    {
        createIterator();
    }

    public ConcreteIterator createIterator()
    {
        return new ConcreteIterator(this);
    }
}
```
Advantages:

- A composite object can be associated with different iterators in order to implement multiple traversal strategies.
- A single object can be simultaneously traversed by several iterators.
- The interface of the class representing composite objects remains simple.

Drawback: Enumerating the elements of a composite object is difficult when the contents of this object can be modified during the operation.

Note: The polymorphic form of the pattern is only relevant when there are several data structures with common features that need to be traversed.
2. Visitor

**Category:** Behavioral, object.

**Goal:** Representing operations to be applied to the elements of a composite object, in such a way that modifying such operations (or adding new ones) does not require to change the classes defining that object.

**Motivating problem:** Development of a compiler carrying out several compilation stages:

2. Type checking for variables and expressions.
3. Object code generation.

**Problem:** The operations that are performed in each compilation stage need to process all nodes of the syntactic tree. In which class can those operations be defined?
Bad solution:

```
<table>
<thead>
<tr>
<th>SyntacticNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkType()</td>
</tr>
<tr>
<td>generateCode()</td>
</tr>
</tbody>
</table>
```

Drawbacks:

- The **algorithms** corresponding to each compilation stage are scattered among several classes, which makes them unmanageable.
- In order to **add a new compilation stage**, one needs to modify a large number of classes.
Better solution:

Principles: Two distinct class hierarchies are defined for the nodes of the syntactic tree and the compilation stages.
General form of the pattern:

Client

Visitor

ConcreteVisitor1

ConcreteVisitor2

ObjectStructure

Element

ConcreteElementA

ConcreteElementB

v.visitConcreteElementA(this);

v.visitConcreteElementB(this);

v.visitConcreteElementA(ConcreteElementA e)

v.visitConcreteElementB(ConcreteElementB e)

visitConcreteElementA(ConcreteElementA e)

visitConcreteElementB(ConcreteElementB e)

visitConcreteElementA(ConcreteElementA e)

visitConcreteElementB(ConcreteElementB e)

accept(Visitor v)

operationA()
Typical interaction:

:ObjectStructure
a:ConcreteElementA
b:ConcreteElementB
v:Visitor

accept(v)

visitConcreteElementA(a)
operationA()

visitConcreteElementB(b)
operationB()
Advantages:

- This pattern avoids to mix in a same class operations that are unrelated to each other.
- Adding new operations remains simple.

Drawback: The structure of the composite object manipulated by the operations cannot easily be modified.
3. Command

**Category:** Behavioral, object.

**Goal:** Encapsulating a request into an object, in order to decouple the operations triggered by this request from the context in which this request is issued.

**Motivating problem:** Development of a GUI toolbox. Clicking on a menu item must trigger the execution of a corresponding operation.

**Problem:** The class that represents menu items has to remain generic. How can then specific operations be associated to menu items?
Good solution:

Principles: The operations to be performed are encapsulated in individual objects, the behavior of which can be defined in a flexible way.
Examples:

- **Paste operation:**
- **Open operation:**

```java
OpenCommand execute()
askUser()
name = askUser();
doc = new Document(name);
appl.add(doc);
doc.open();
```
**Macro operation:**

```java
MacroCommand execute() {
    foreach (c in command)
        c.execute();
}
```
General form of the pattern:
Typical interaction:

- `r`: Receiver
- `c`: Command
- `Client`
- `Invoker`

Actions:
- `new(r)`
- `setCommand(c)`
- `execute()`
- `action()`
Advantages:

- The operations are well decoupled from the objects that are responsible for triggering them.
- The operations to be performed can be stored in order to be executed later.
- The pattern can be refined in order to implement reversible operations, maintain a log of actions, . . .

Drawback: Implementations of reversible operations are usually quite sensitive to error accumulation.
4. Observer

**Category:** Behavioral, object.

**Goal:** Managing a dependency relation between objects, ensuring that when an object changes its state, those that depend on it are automatically updated.

**Motivating problem:** Development of a computer-aided design tool for architectural drawings. One should be able to interact with a drawing by means of multiple windows: orthographic projections, cross sections, 3D view, . . .

**Problem:** Modifications entered in a given window must be reflected in all of them, as well as in the internal representation of the drawing.
Good solution:

Principles:

- A single object is responsible for managing the state of the drawing.
- The objects that need to be informed of changes made to the drawing subscribe to a notification service.
General form of the pattern:

```java
ConcreteSubject
  getState()
  setState()
  subjectState

ConcreteObserver
  observerState
  update()

Subject
  attach(Observer o)
  detach(Observer o)
  notify()

Observer
  update()

foreach (o in observers)
  o.update();

observerState = subject.getState();
```
Typical interaction:
Advantages:

- The subject and the observers remain weakly coupled to each other.
- It is easy to dynamically add or remove observers.

Drawback: When an observer modifies the state of the subject, it is difficult to estimate the cost of the updates that need to be performed by the observers.
5. State

Category: Behavioral, object.

Goal: Modifying the behavior of an object according to its current state.

Motivating problem: Implementation of a network protocol. The behavior of an instance of `NetworkConnection` depends on the state of the connection that it represents: Waiting, established, or closed.

Problem: How can the operations implemented by `NetworkConnection` be programmed?
Bad solution:

```java
switch (state)
{
    case WAITING: ...; break;
    case ESTABLISHED: ...; break;
    case CLOSED: ...; break;
}
```

Drawback: This code is **not modular enough**, since it concentrates in a **single class** the behaviors corresponding to all the states.
Better solution:

Principles: The class *NetworkConnection* delegates its responsibilities to an object, the class of which is updated at each transition between states.
General form of the pattern:
**Advantage:** For every state, the implementation of the operations that need to be performed in that state is specified inside a separate class.

**Drawbacks:**

- This pattern does not describe how the objects corresponding to the states are instantiated when transitions are followed.
- Those instantiation operations can be costly.
Chapter 6

Software testing
Introduction

The aim of software testing is to assess the quality of the developed product: Does it work correctly and efficiently? Is it consistent with the conceptual, design, and implementation models? Does it meet the requirements of the user?

Principles:

- Testing is carried out with the intent of discovering flaws.

- A test case is an execution scenario that is considered during testing, and describes precisely all interactions between the tested system and its environment.

- A good test case is one that is able to show a system flaw. A test run is considered to be successful if it discovers a new flaw.

- The set of test cases obtained up to the current development cycle form the test model.
Notes:

- Testing is an **essential activity**. The amount of resources allocated to testing can typically represent more than 40% of the effort of a project.

- The goal of testing is to **discover flaws**, that can then be corrected. However, testing is **unable to guarantee** that a system is error free!

- The fact that testing is an integral part of every development cycle does not exempt from following **good programming practices**: Writing code that is modular, readable, and **well documented**, proofreading programs, ...
Testing can be performed at different levels of abstraction:

- **Unit**: The target of the test is a single software component (such as a class).
- **Subsystem**: Major components are tested separately.
- **Product**: The complete product is tested.
- **Alpha testing**: Testing is carried out in a realistic setting.

In practice, a good methodology consists in considering these levels of abstraction

- in bottom-up order when testing is performed after development activities, and
- in top-down order when investigating defects reported by users.
Blackbox testing

Blackbox testing is a first way of obtaining test cases, by working at the interface between the tested unit (or system) and its environment. The goal is to check whether functional requirements are satisfied.

Blackbox testing is focused on the following type of defects:

- absent or incorrectly implemented features,
- interface errors,
- errors in data structures,
- inefficient implementations,
- initialization and termination errors.
Blackbox testing is carried out by building a set of test cases that sufficiently covers all expected modes of operation of the tested system. The test cases are obtained thanks to the following guidelines:

- Each functionality documented in the set of requirements or in the design or implementation models must be covered by at least one test case.

- When input data values can be partitioned into subsets leading to different expected behaviors, the test cases must consider at least one value from each of these subsets.

- For data values that are constrained by a validity domain, good practice is to consider values that are close to the boundaries of this domain, as well as at least one typical value chosen inside the domain.
Glassbox testing complements blackbox testing by producing test cases derived from a study of the code of the tested system.

Ideally, the test cases would correspond to all possible execution paths, but the number of scenarios would then be prohibitively large. A reasonable approximation is to build test cases in such a way that:

- each executable instruction is at least run once,
- each decision is at least tested for each of its possible outcomes, and
- independent execution paths are sufficiently well covered.

Question: Is it possible to define a notion of independence between execution paths that

- leads to a good coverage, and
- yields only a manageable number of test cases?
Control graphs

Given a fragment of executable code (e.g., a method), its control graph, or flow graph, is a directed graph constructed as follows:

- Its nodes correspond to the program instructions.
- An edge links the node \( n_1 \) to the node \( n_2 \) if it is possible to execute the instruction corresponding to \( n_2 \) immediately after the one associated to \( n_1 \).
- Two distinct edges cannot share the same origin and the same destination.
- The graph has unique and distinct entry and exit nodes. All the nodes in the graph must be reachable from the entry nodes. Similarly, the exit node must be reachable from all the nodes in the graph.
A control path is a sequence of edges that can be followed between its entry and its exit nodes, in that direction. Such a path can also be described by the sequence of nodes that it visits.

Control paths can be abstracted into a vector representation:

- The dimension of the space corresponds to the number of edges in the control graph.
- The vector representation of a path is obtained by counting the number of times that each edge is visited by the path.
Example:

The path $(1; 3; 7; 3; 9)$ (or $(a; b; c; b; c; g)$) admits the vector representation $(1, 0, 2, 0, 0, 0, 1, 0, 1, 0)$. 
Path independence

The vector representation of control paths makes it possible to define precisely a notion of independence between such paths:

A set of control paths is independent if the vector representations of these paths form a linearly independent set of vectors.

Example: In the previous figure, the paths \( \pi_1 = (a; b; c; g) \), \( \pi_2 = (a; b; c; b; c; g) \) and \( \pi_3 = (a; b; e; f; g) \) are independent.

On the other hand, the paths \( \pi_1, \pi_2, \pi_3 \) and \( \pi_4 = (a; b; c; b; e; f; g) \) are not independent, since they satisfy the linear dependence relation

\[ \pi_4 = \pi_2 + \pi_3 - \pi_1. \]
Cyclomatic complexity

The cyclomatic complexity \( V(G) \) of a control graph \( G \) is defined as

\[
V(G) = E - N + 2,
\]

where

- \( E \) is the number of edges of the graph,
- \( N \) is the number of nodes.

**Theorem:** The cyclomatic complexity of a control graph corresponds to the maximum number of independent paths in this graph.

**Example:** For the control graph in the previous figure, one has \( N = 7 \) and \( E = 10 \). The cyclomatic complexity of this graph is thus equal to 5.
Notes:

- The cyclomatic complexity is often used as a metric for evaluating the syntactic complexity of code.

  A useful guideline is to consider that a code fragment with a cyclomatic complexity larger than 10 is too intricate and must be reworked (except when this high complexity is caused by a decision instruction with a large number of outgoing branches).

- The cyclomatic complexity of a control graph $G$ in which all decisions are binary satisfies the relation

  $$V(G) = D + 1,$$

  where $D$ is the number of decision nodes.
• The cyclomatic complexity of a **planar** control graph (i.e., such that it can be drawn in a plane without crossing edges) is equal to the **number of regions** delimited by the edges in a planar representation of the graph.

**Example:**
Control graph bases

Given a control graph $G$, it is always possible to extract from this graph $V(G)$ paths that are independent, and such that the set of all their linear combinations contains all paths of $G$.

Such a set of $V(G)$ paths is called a basis of the control graph.

Example: The control graph of our first example admits the basis $\{(a; b; c; g), (a; b; c; b; c; g), (a; b; e; f; g), (a; d; e; f; g), (a; d; f; g)\}$.

After computing the cyclomatic complexity $V(G)$ of a control graph, a basis of this graph can be built incrementally, by searching successively for paths $\pi_1, \pi_2, \ldots, \pi_{V(G)}$ such that for all $i \in \{2, 3, \ldots, V(G)\}$, the paths $\pi_1, \pi_2, \ldots, \pi_i$ are independent.
Basis path testing

The following procedure can be used for synthesizing the test cases to be considered during glassbox testing:

1. One draws the control graph of the unit or subsystem being tested.
2. One computes the cyclomatic complexity $V(G)$ of this graph.
3. One extracts a basis of $G$, and builds test cases corresponding to these paths.

Note: Some control paths obtained at Step 3 may turn out to be unfeasible, in which case they do not have to be considered. Extracting from a code fragment the largest possible set of feasible independent paths is a difficult problem.
Formal verification

The drawback of testing methods is that they only provide partial coverage of system executions. Formal verification is a set of techniques aimed at yielding full coverage.

Overview:

- The relevant features of the system being verified are precisely described in a model, expressed in a specific modeling formalism.

  Several modeling formalisms are available, providing a different balance between expressive power and the possibility of deciding interesting properties.

- The properties that can be checked include safety, liveness, and timed properties, and are usually specified in logical formalisms (e.g., temporal logic).

- Formal verification is often costly and complex, and is thus usually reserved for critical applications.
Regression testing

The test cases that are checked at the end of a development cycle must not be limited to the code that has been added or modified during this cycle.

Indeed, a flaw introduced in one part of the code may have an effect on remote components, and the correction of errors may introduce further flaws.

Regression testing then consists in verifying at the end of every development cycle that the system still runs correctly the set of all test cases considered from the beginning of the project.

A simple way of performing this operation is to maintain a test case library, and to use an automated tool for validating them periodically.
Modular regression

In the case of a highly complex or very large system, validating the full set of test cases needed for regression testing may incur an unacceptable cost.

A possible solution is then to perform modular regression testing, which makes it possible to restrict the number of test cases to consider.

**Principles:** The system is decomposed into modules that can be checked individually. One only tests the modules that have potentially been affected by a modification performed since the last time they have been tested.

Let us consider a module $P$ with a behavior that is entirely characterized by data values flowing through its interface (i.e., identical input sequences lead to identical output values).
If the module $P$ has to be modified during the current development cycle, one performs the following operations:

1. **Before modifying $P$.** The sequence of data exchange operations involving $P$ during the current set of test cases is recorded.

2. **After modifying $P$.** One tests the new implementation of $P$, by providing it with the input data sequences that have previously been recorded. One then checks that the data output by $P$ is identical to the values observed before the modification.

In the case of a difference, all the modules that directly or transitively depend on $P$ have to be tested again.