Object-Oriented Design

Lecturer: Raman Ramsin

Lecture 18:
Interfaces and Components
Design Workflow: *Architecture and Subsystems*

- Place in the *Design Workflow:*
  - Architectural Design
  - Design a Use Case
  - Design a Class
  - **Design a Subsystem**

- Concerned with breaking a system up into subsystems that are as independent as possible.
  - Interactions between subsystems are mediated by interfaces.
Interfaces

- Interfaces allow software to be designed to a contract rather than to a specific implementation.

- An interface specifies a named set of public features.
  - Interfaces separate specification of functionality from implementation.
  - Interfaces may be attached to classes, subsystems, components, and any other classifier and define the services offered by these.
  - If a classifier inside a subsystem realizes a public interface, the subsystem or component also realizes the public interface.
  - Anything that realizes an interface agrees to abide by the contract defined by the set of operations specified in the interface.
## Interface Semantics

- A classifier realizing an interface has the following responsibilities for each feature:

<table>
<thead>
<tr>
<th>Interface specifies</th>
<th>Realizing classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Must have an operation with the same signature and semantics</td>
</tr>
<tr>
<td>Attribute</td>
<td>Must have public operations to set and get the value of the attribute – the realizing classifier is not required to actually have the attribute specified by the interface, but it must behave as though it has</td>
</tr>
<tr>
<td>Association</td>
<td>Must have an association to the target classifier – if an interface specifies an association to another interface, the implementing classifiers of these interfaces must have an association between them</td>
</tr>
<tr>
<td>Constraint</td>
<td>Must support the constraint</td>
</tr>
<tr>
<td>Stereotype</td>
<td>Has the stereotype</td>
</tr>
<tr>
<td>Tagged value</td>
<td>Has the tagged value</td>
</tr>
<tr>
<td>Protocol (e.g., as defined by a protocol state machine – see Section 21.2.1)</td>
<td>Must realize the protocol</td>
</tr>
</tbody>
</table>
Alternative Design Approaches

- **Designing to an implementation:**
  - specific classes are connected;
  - to keep things simple (but rigid), design to an implementation.

- **Designing to a contract:**
  - a class is connected to an interface that may have many possible realizations;
  - to make things flexible (but possibly more complex), design to a contract.
Provided Interface

- An interface provided by a classifier:
  - the classifier realizes the interface;
  - use the "class" style notation when you need to show the operations on the model;
  - use the shorthand "lollipop" style notation when you just want to show the interface without operations.
Required Interface

- An interface required by a classifier:
  - the classifier requires another classifier that realizes the interface;
  - show a dependency to a class style interface, a lollipop style interface, or use an assembly connector.
Assembly Connector

- Joins provided and required interfaces.
Interface Hierarchies

```
Iterable
  Collection
    List
    Set

Map
  SortedMap
  ConcurrentHashMap

SortedSet
```
Interface Realization vs. Inheritance

- Interface realization - "realizes a contract specified by".

- Inheritance - "is a".

- Both inheritance and interface realization generate polymorphism.

- Use interfaces to specify the common protocols of classes that should not normally be related by inheritance.
Interface Realization vs. Inheritance: Example
Ports

- Port - groups a semantically cohesive set of provided and required interfaces:
  - may have a name, type, and visibility.

![Diagram of ports in Object-Oriented Design](image-url)
Components

Component - a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment:

- may have attributes and operations;
- may participate in relationships;
- may have internal structure;
- its external behavior is completely defined by its provided and required interfaces;
- components manifest one or more artifacts.
## Components: Standard Stereotypes

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>«buildComponent»</td>
<td>A component that defines a set of things for organizational or system-level development purposes</td>
</tr>
<tr>
<td>«entity»</td>
<td>A persistent information component representing a business concept</td>
</tr>
<tr>
<td>«implementation»</td>
<td>A component definition that has no specification itself – it is an implementation for a separate «specification» to which it has a dependency</td>
</tr>
<tr>
<td>«specification»</td>
<td>A classifier that specifies a domain of objects without defining the physical implementation of those objects – for example, a component stereotyped by «specification» only has provided and required interfaces and no realizing classifiers</td>
</tr>
<tr>
<td>«process»</td>
<td>A transaction-based component</td>
</tr>
<tr>
<td>«service»</td>
<td>A stateless, functional component that computes a value</td>
</tr>
<tr>
<td>«subsystem»</td>
<td>A unit of hierarchical decomposition for large systems</td>
</tr>
</tbody>
</table>
CBD and Subsystems

- Component-Based Development (CBD) is about constructing software from plug-in parts:
  - you use interfaces to make components "pluggable";
  - by designing to an interface, you allow the possibility of many different realizations by many different components.

- Subsystem - a component that acts as a unit of decomposition for a larger system:
  - a component stereotyped «subsystem»;
  - is used to decompose a large system into manageable chunks;
  - breaking a system down into subsystems is a key to successful CBD using UP.
Subsystems: Applications

- Subsystems are used to:
  - separate design concerns;
  - represent large-grained components;
  - wrap legacy systems.
Designing with Interfaces

- Use interfaces to hide the implementation details of subsystems:
  - the *Facade* pattern hides a complex implementation behind a simple interface;
  - the *layering* pattern organizes subsystems into semantically cohesive layers:
    - dependencies between layers should only go one way;
    - all dependencies between layers should be mediated by an interface;
    - example layers include presentation, business logic, and utility layers.
Designing with Interfaces: *Layering* Pattern
Finding Interfaces: Guidelines

- challenge associations;
- challenge message sends;
- factor out groups of reusable operations;
- factor out groups of repeating operations;
- factor out groups of repeating attributes;
- look for classes that play the same role in the system;
- look for possibilities for future expansion;
- look for dependencies between components.
Reference