Object-Oriented Software Engineering

Conquering Complex and Changing Systems

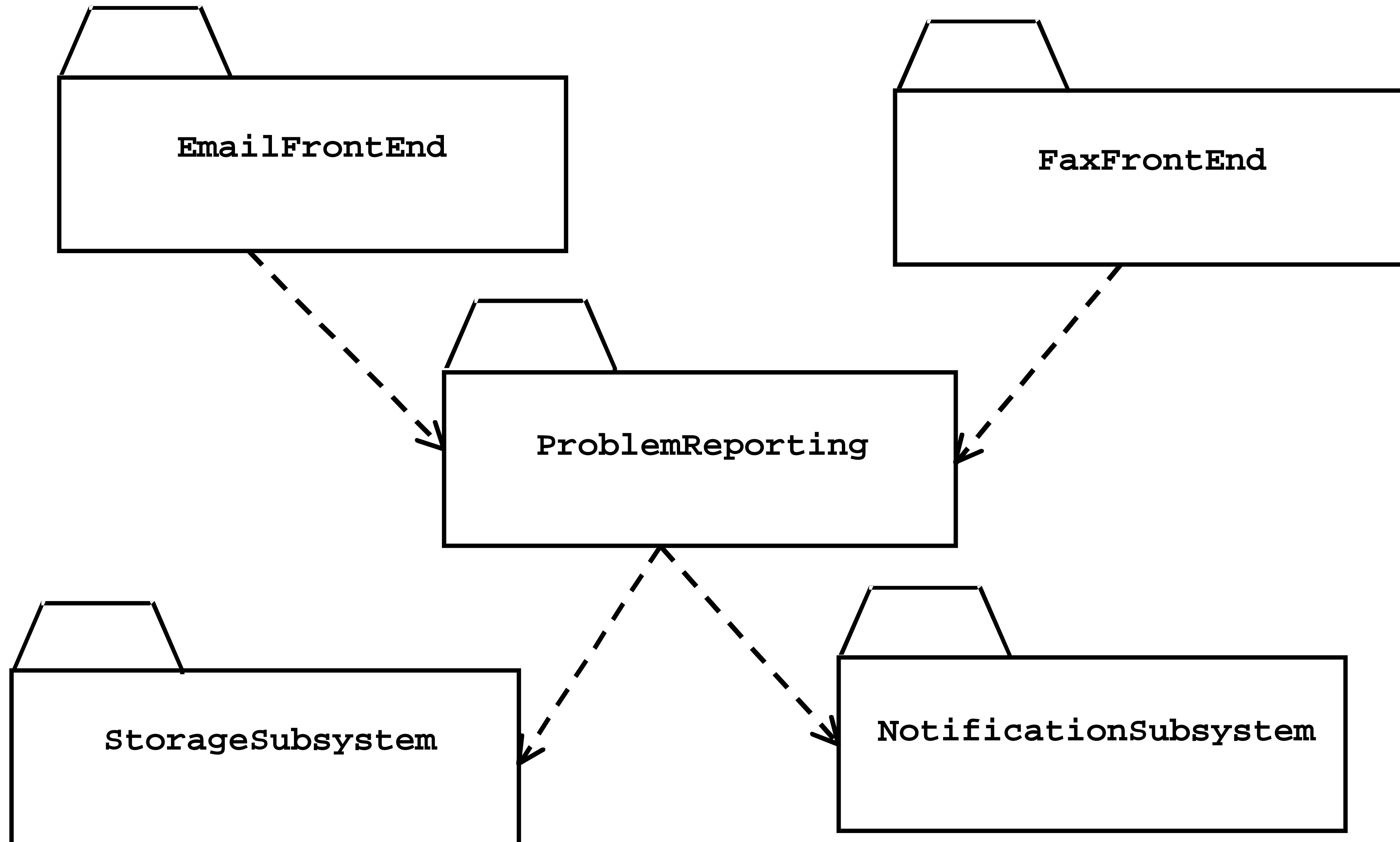
Chapter 7, Object Design

Exercise 6.4

6.4 Consider a legacy, fax-based, problem-reporting system for an aircraft manufacturer. You are part of a reengineering project replacing the core of the system by a computer-based system, which includes a database and a notification system. The client requires the fax to remain an entry point for problem reports. You propose an E-mail entry point.

Describe a subsystem decomposition, and possibly a design pattern, which would allow both interfaces.

Possible solution for exercise 6.4



Exercise 6.5

6.5 You are designing the access control policies for a Web-based retail store:

- **Customers access the store via the Web, browse product information, input their address and payment information, and purchase products.**
- **Suppliers can add new products, update product information, and receive orders.**
- **The store owner sets the retail prices, makes tailored offers to customers based on their purchasing profiles, and provides marketing services.**

You have to deal with three actors: StoreAdministrator, Supplier, and Customer. Design an access control policy for all three actors. Customers can be created via the Web, whereas Suppliers are created by the StoreAdministrator.

Possible solution for exercise 6.5

	Product	CustomerInfo	SupplierInfo	Order
Anonymous	GetInfo()	Create()		
Customer	GetInfo() GetPrice()	UpdateInfo()		Create()
Supplier	Create() GetInfo() UpdateInfo()		UpdateInfo()	Process()
StoreAdministrator	UpdatePrice()	VerifyInfo()	Create()	Examine()

Object Design

Object design is the process of adding details to the requirements analysis and making implementation decisions

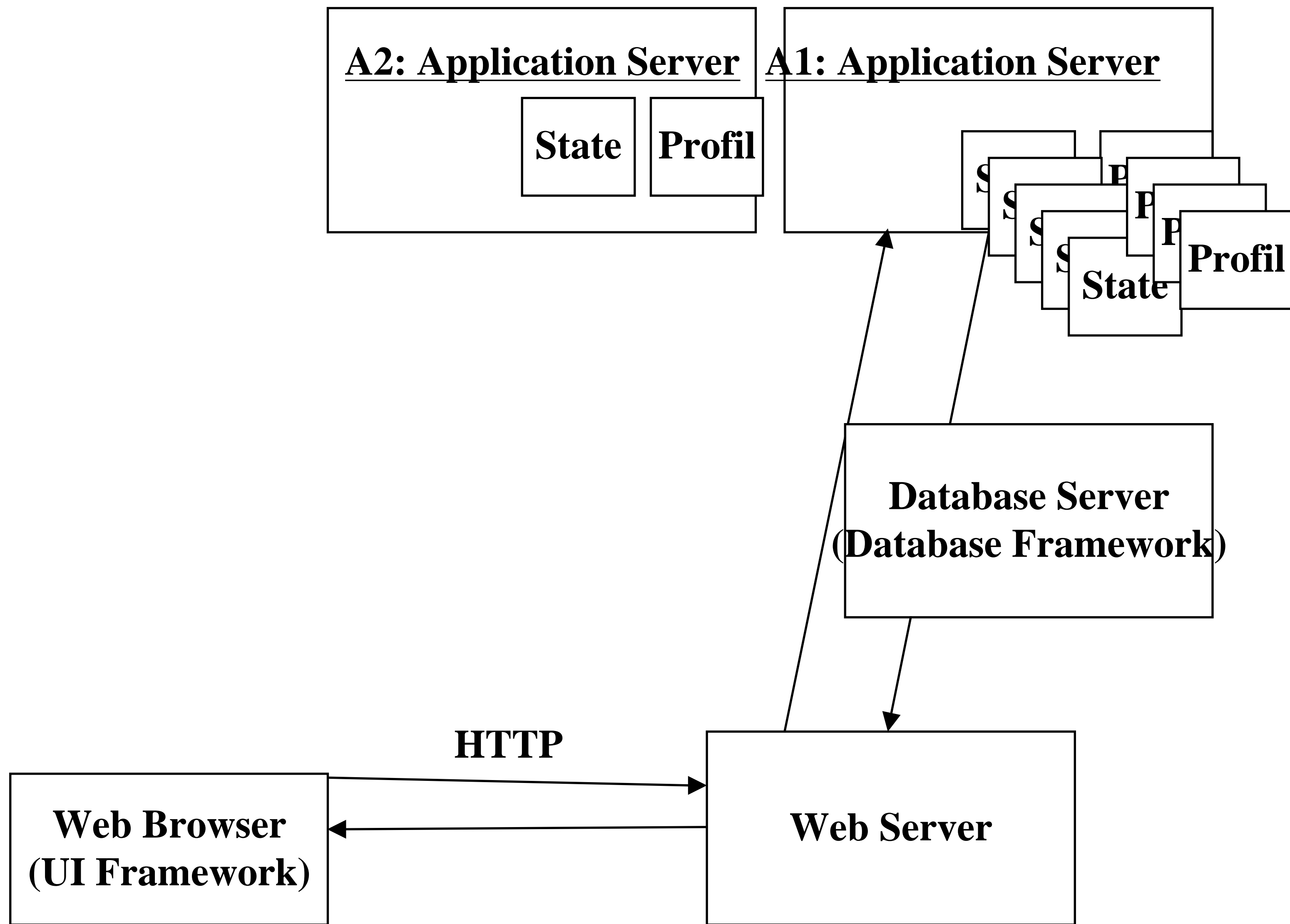
The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.

Requirements Analysis: Use cases, functional and dynamic model deliver operations for object model

Object Design: We iterate on where to put these operations in the object model

Object Design serves as the basis of implementation

4-Tier Architecture


A2: Application Server
State
Profil
A1: Application Server
State
Profil
Database Server
(Database Framework)
HTTP
Web Browser
(UI Framework)
Web Server

Reengineering Terminology

Reverse Engineering:

- **Discovery (or Recovery) of an object model from the code.**

Forward Engineering:

- **Automatic generation of code from an object model**
- **Requirements Engineering, Requirements Analysis, System Design, Object Design, Implementation, Testing, Delivery**

Discipline:

- **Always change the object model, then generate code (for sure do this when you change the interface of a public method/class.)**
  - **Generate code under time pressure**
    - **Patch the code!**

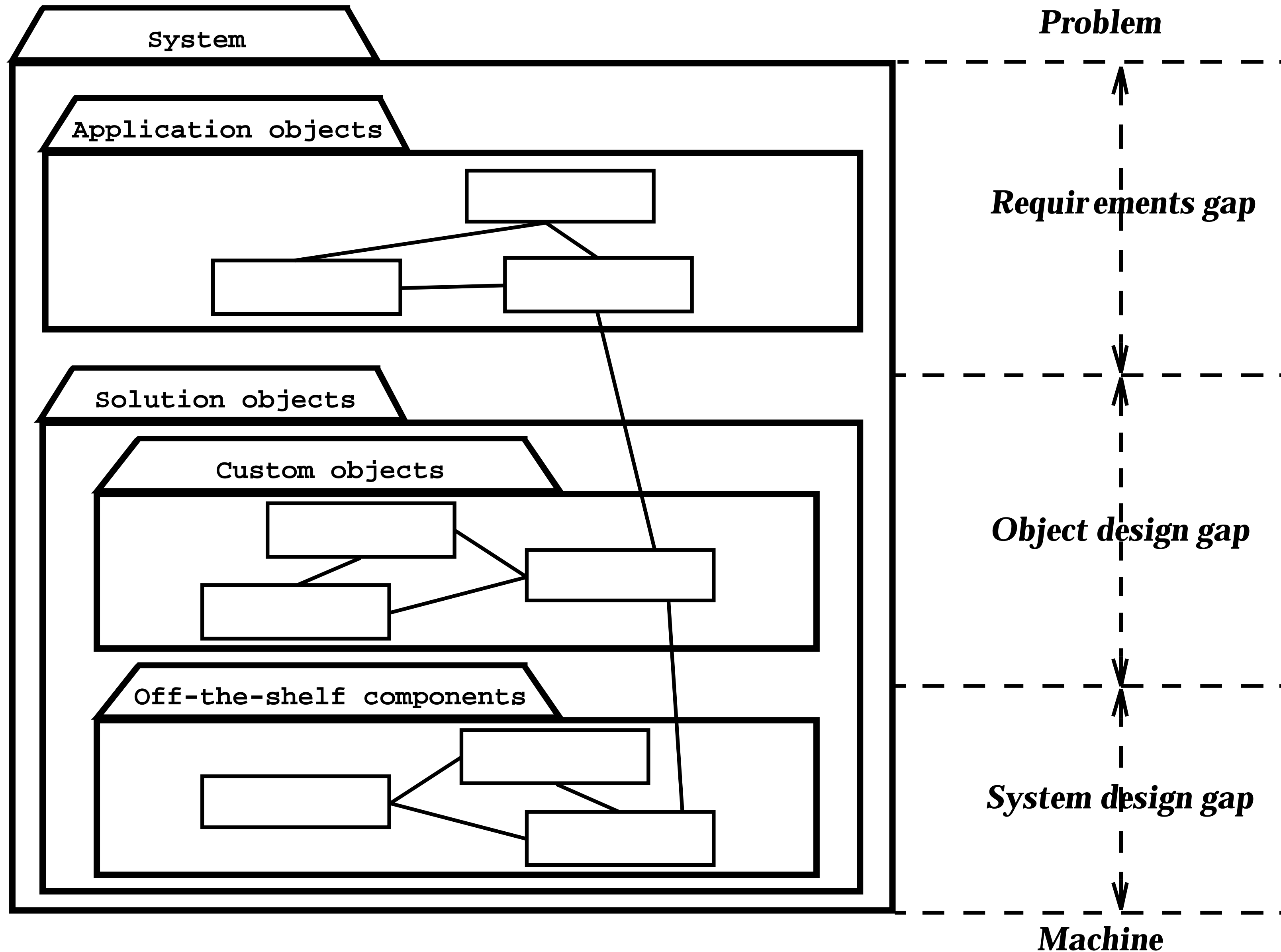
Roundtrip Engineering

- **Forward Engineering + reverse engineering**
- **Inventory analysis: Determine the Delta between OM and Code**
- **Together-J and Rationale have tools for reverse engineering**

Reengineering (Project Management Issue):

- **New functionality (customer dreams up new stuff)**
- **New technology (technology enablers)**

Object Design: Closing the Gap



Object Design Issues

- Full definition of associations
- Full definition of classes (System Design: Service, Object Design: API)
- Specify the contract for each component
- Choice of algorithms and data structures
- Detection of new application-domain independent classes (example: Cache)
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

Terminology of Activities

Object-Oriented Methodologies

- ***System Design***
  - **Decomposition into subsystems**
- ***Object Design***
  - **Implementation language chosen**
  - **Data structures and algorithms chosen**

SA/SD (structured analysis/structured design) uses different terminology:

- ***Preliminary Design***
  - **Decomposition into subsystems**
  - **Data structures are chosen**
- ***Detailed Design***
  - **Algorithms are chosen**
  - **Data structures are refined**
  - **Implementation language is chosen**
  - **Typically in parallel with preliminary design, not separate stage**

Object Design Activities

1. Service specification
   - **Describes precisely each class interface**
2. Component selection
   - **Identify off-the-shelf components and additional solution objects**
3. Object model restructuring
   - **Transforms the object design model to improve its understandability and extensibility**
4. Object model optimization
   - **How to address nonfunctional requirements**
   - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Service Specification

Requirements analysis

- **Identifies attributes and operations without specifying their types or their parameters.**

Object design

- **Add visibility information**
- **Add type signature information**
- **Add contracts (Bertrand Meyer, Eiffel)**

Add Visibility

UML defines three levels of visibility:

Private:

- **A private attribute can be accessed only by the class in which it is defined.**
- **A private operation can be invoked only by the class in which it is defined.**
- **Private attributes and operations cannot be accessed by subclasses or other classes.**

Protected:

- **A protected attribute or operation can be accessed by the class in which it is defined and on any descendent of the class.**

Public:

- **A public attribute or operation can be accessed by any class.**

Information Hiding Heuristics

Build firewalls around classes

- **Carefully define public interfaces for classes as well as subsystems**
- **Never, never, never make attributes public**

Apply “Need to know” principle. The fewer an operation knows

- **the less likely it will be affected by any changes**
- **the easier the class can be changed**

Trade-off

- **Information hiding vs efficiency**

Information Hiding Design Principles

Only the operations of a class are allowed to manipulate its attributes

- **Access attributes only via operations.**

Hide external objects at subsystem boundary

- **Define abstract class interfaces which mediate between system and external world as well as between subsystems**

Do not apply an operation to the result of another operation.

- **Write a new operation that combines the two operations.**

Add Type Signature Information

```
Hashtable
-numElements:int
+put()
+get()
+remove()
+containsKey()
+size()
```

```
Hashtable
-numElements:int
+put(key:Object,entry:Object)
+get(key:Object):Object
+remove(key:Object)
+containsKey(key:Object):boolean
+size():int
```

Contracts

Contracts on a class enable caller and callee to share the same assumptions about the class.

Contracts include three types of constraints:

- ***Invariant*: A predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces. Invariants are used to specify consistency constraints among class attributes.**
- ***Precondition*: A predicate that must be true before an operation is invoked. Preconditions are associated with a specific operation. Preconditions are used to specify constraints that a caller must meet before calling an operation.**
- ***Postcondition*: A predicate that must be true after an operation is invoked. Postconditions are associated with a specific operation. Postconditions are used to specify constraints that the object must ensure after the invocation of the operation.**

Expressing constraints in UML

OCL (Object Constraint Language)

- **OCL allows constraints to be formally specified on single model elements or groups of model elements**
- **A constraint is expressed as an OCL expression returning the value true or false. OCL is not a procedural language (cannot constrain control flow).**

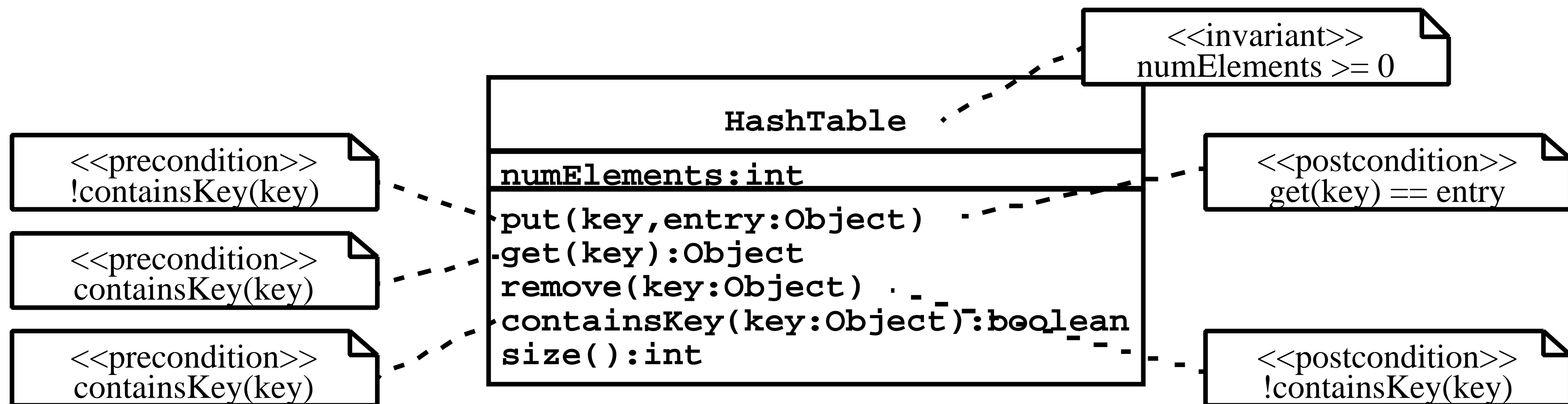
OCL expressions for Hashtable operation put():

- **Invariant:**
  - **context Hashtable ... 0**

- **Precondition:**
  - **context Hashtable::put(key, entry) pre:!containsKey(key)**
- **Post-condition:**
  - **context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry**

Expressing Constraints in UML

A constraint can also be depicted as a note attached to the constrained UML element by a dependency relationship.



Object Design Areas

1. Service specification
   - **Describes precisely each class interface**
2. Component selection
   - **Identify off-the-shelf components and additional solution objects**
3. Object model restructuring
   - **Transforms the object design model to improve its understandability and extensibility**
4. Object model optimization
   - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Component Selection

Select existing off-the-shelf class libraries, frameworks or components

Adjust the class libraries, framework or components

- **Change the API if you have the source code.**
- **Use the adapter or bridge pattern if you don’t have access**

Reuse...

Look for existing classes in class libraries

- **JSAPI, JTAPI, ...**

Select data structures appropriate to the algorithms

- **Container classes**
- **Arrays, lists, queues, stacks, sets, trees, ...**

Define new internal classes and operations only if necessary

- **Complex operations defined in terms of lower-level operations might need new classes and operations**

Object Design Areas

1. Service specification
   - **Describes precisely each class interface**
2. Component selection
   - **Identify off-the-shelf components and additional solution objects**
   - **Use the bridge pattern if the off-the-shelf component comes late**
     - **Use a quick and dirty implementation first**
3. Object model restructuring
   - **Transforms the object design model to improve its understandability and extensibility**
4. Object model optimization
   - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Restructuring Activities

Realizing associations

Revisiting inheritance to increase reuse

Revising inheritance to remove implementation dependencies

Realizing Associations

Strategy for implementing associations:

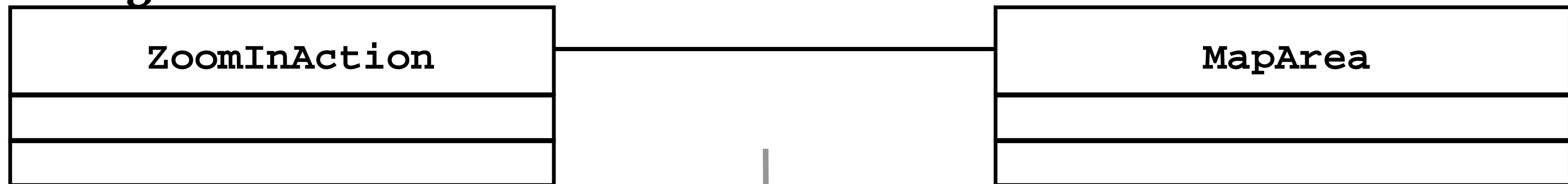
- **Be as uniform as possible**
- **Individual decision for each association**

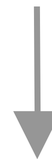
Example of uniform implementation

- **1-to-1 association:**
  - **Role names are treated like attributes in the classes and translate to references**
- **1-to-many association:**
  - **"Ordered many" : Translate to `Vector`**
  - **"Unordered many" : Translate to `Set`**
- **Qualified association:**
  - **Translate to Hash table**

Unidirectional 1-to-1 Association

Object design model before transformation

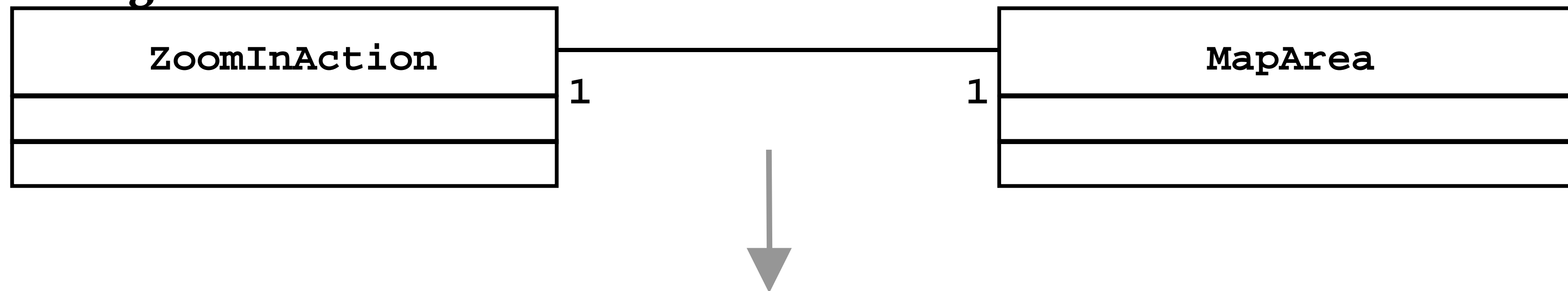




Object design model after transformation

Bidirectional 1-to-1 Association

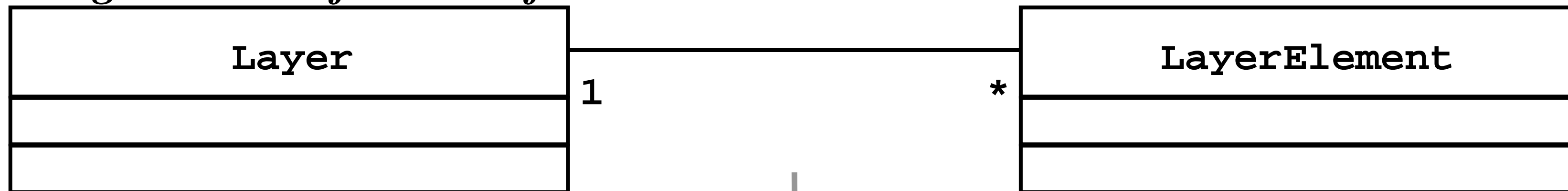
Object design model before transformation

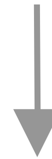


Object design model after transformation

1-to-Many Association

Object design model before transformation

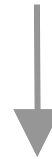




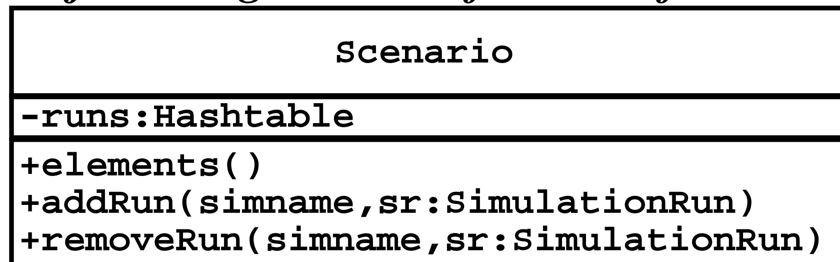
Object design model after transformation

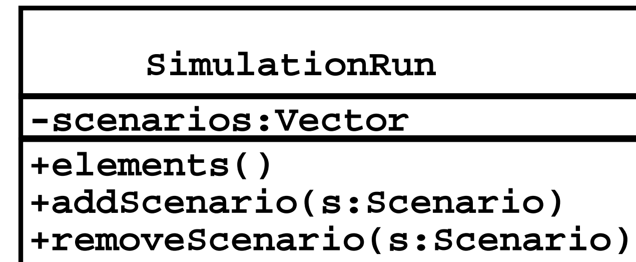
Qualification

Object design model before transformation



Object design model after transformation





Increase Inheritance

Rearrange and adjust classes and operations to prepare for inheritance

Abstract common behavior out of groups of classes

- **If a set of operations or attributes are repeated in 2 classes the classes might be special instances of a more general class.**

Be prepared to change a subsystem (collection of classes) into a superclass in an inheritance hierarchy.

Building a super class from several classes1/11/01

Prepare for inheritance. All operations must have the same signature but often the signatures do not match:

- **Some operations have fewer arguments than others: Use overloading (Possible in Java)**
- **Similar attributes in the classes have different names: Rename attribute and change all the operations.**
- **Operations defined in one class but no in the other: Use virtual functions and class function overriding.**

Abstract out the common behavior (set of operations with same signature) and create a superclass out of it.

Superclasses are desirable. They

- **increase modularity, extensibility and reusability**
- **improve configuration management**

Turn the superclass into an abstract interface if possible

- **Use Bridge pattern**

Object Design Areas

1. Service specification
   - **Describes precisely each class interface**
2. Component selection
   - **Identify off-the-shelf components and additional solution objects**
3. Object model restructuring
   - **Transforms the object design model to improve its understandability and extensibility**
4. Object model optimization
   - **Transforms the object design model to address performance criteria such as response time or memory utilization.**

Design Optimizations

Design optimizations are an important part of the object design phase:

- **The requirements analysis model is semantically correct but often too inefficient if directly implemented.**

Optimization activities during object design:

1. **Add redundant associations to minimize access cost**
2. **Rearrange computations for greater efficiency**
3. **Store derived attributes to save computation time**

As an object designer you must strike a balance between efficiency and clarity.

- **Optimizations will make your models more obscure**

Design Optimization Activities

1. Add redundant associations:
   - **What are the most frequent operations? (Sensor data lookup?)**
   - **How often is the operation called? (30 times a month, every 50 milliseconds)**
2. Rearrange execution order
   - **Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)**
   - **Narrow search as soon as possible**
   - **Check if execution order of loop should be reversed**
3. Turn classes into attributes

Implement Application domain classes

To collapse or not collapse: Attribute or association?

Object design choices:

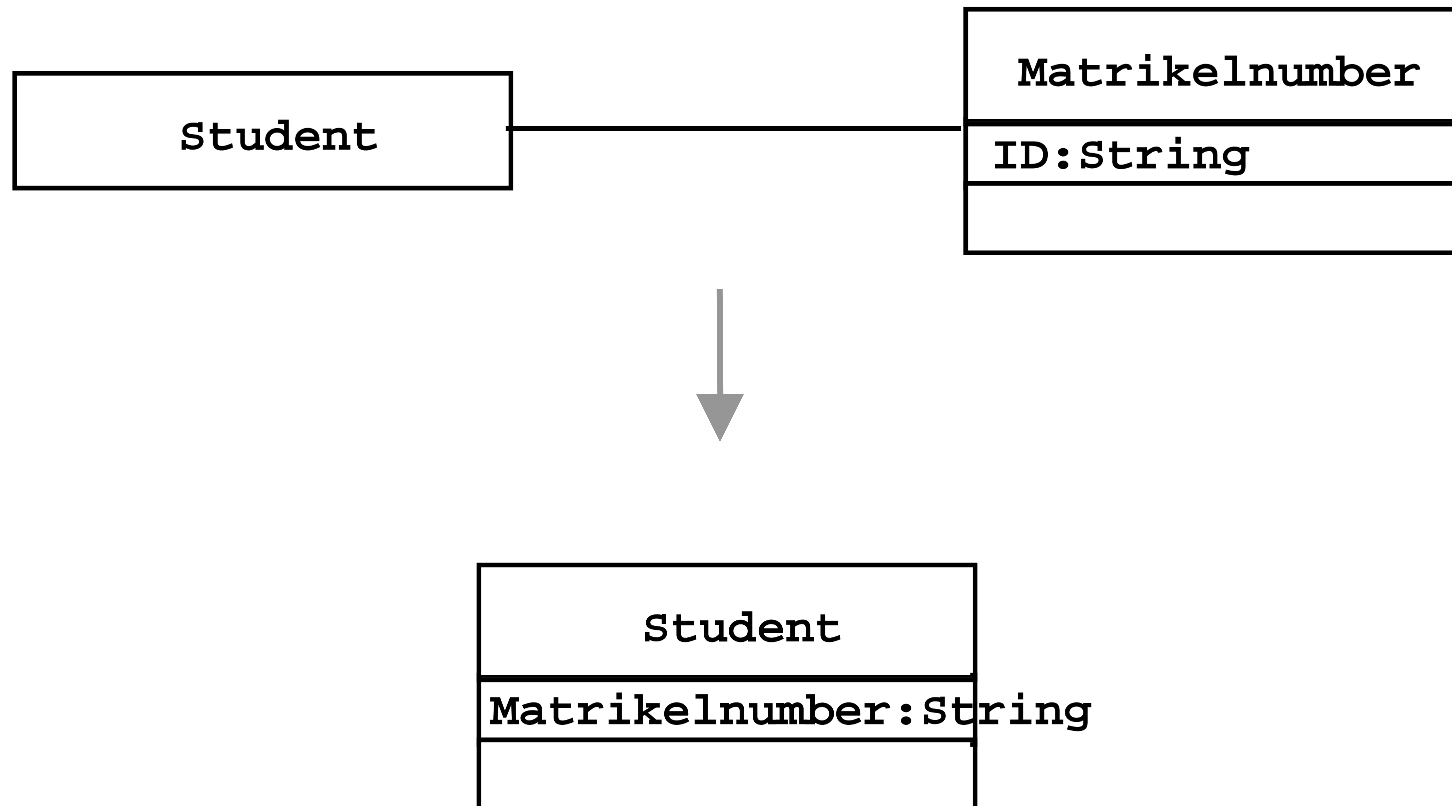
- **Implement entity as embedded attribute**
- **Implement entity as separate class with associations to other classes**

Associations are more flexible than attributes but often introduce unnecessary indirection.

Abbott's textual analysis rules

Every student receives an immatriculationnumber at the first day in TUM.

Optimization Activities: Collapsing Objects



To Collapse or not to Collapse?

Collapse a class into an attribute if the only operations defined on the attributes are Set() and Get().

Design Optimizations (continued)

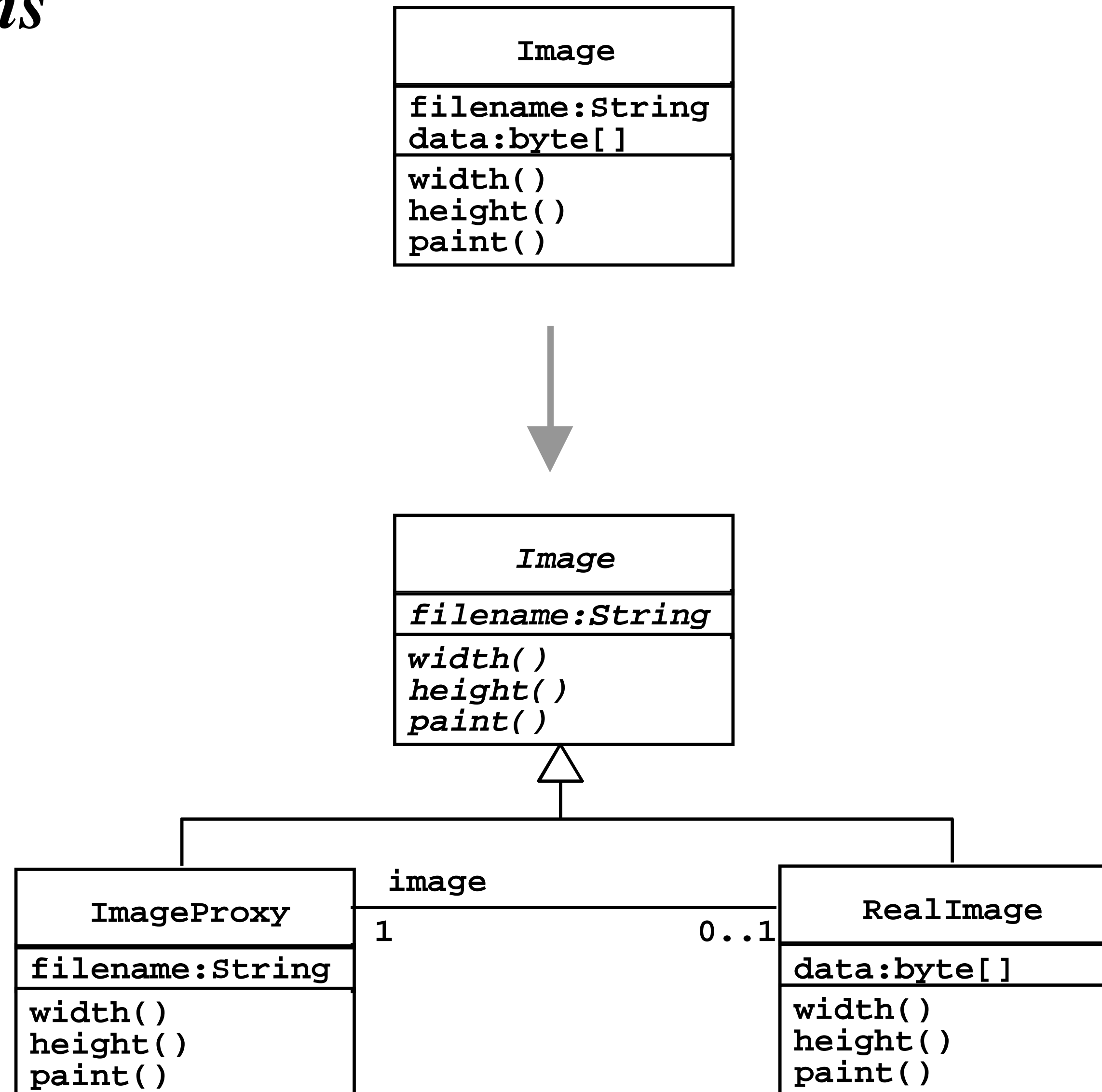
Store derived attributes

- **Example: Define new classes to store information locally (database cache)**

Problem with derived attributes:

- **Derived attributes must be updated when base values change.**
- **There are 3 ways to deal with the update problem:**
  - **Explicit code: Implementor determines affected derived attributes (push)**
  - **Periodic computation: Recompute derived attribute occasionally (pull)**
  - **Active value: An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)**

Optimization Activities: Delaying Complex Computations



Documenting the Object Design: The Object Design Document (ODD)

- Object design document
  - **Same as RAD +...**
  - **… + additions to object, functional and dynamic models (from solution domain)**
  - **… + Navigational map for object model**
  - **… + Javadoc documentation for all classes**
- ODD Management issues
  - **Update the RAD models in the RAD?**
  - **Should the ODD be a separate document?**
  - **Who is the target audience for these documents (Customer, developer?)**
  - **If time is short: Focus on the Navigational Map and Javadoc documentation?**
- Example of acceptable ODD:
  - **http://macbruegge1.informatik.tu-muenchen.de/james97/index.html**

Documenting Object Design: ODD Conventions

Each subsystem in a system provides a service (see Chapter on System Design)

- **Describes the set of operations provided by the subsystem**

Specifying a service operation as

- **Signature: Name of operation, fully typed parameter list and return type**
- **Abstract: Describes the operation**
- **Pre: Precondition for calling the operation**
- **Post: Postcondition describing important state after the execution of the operation**

Use JavaDoc for the specification of service operations.

JavaDoc

Add documentation comments to the source code.

A doc comment consists of characters between **/**** and ***/**

When JavaDoc parses a doc comment, leading * characters on each line are discarded. First, blanks and tabs preceding the initial * characters are also discarded.

Doc comments may include HTML tags

Example of a doc comment:

```
/**
 * This is a <b> doc </b> comment
 */
```

More on JavaDoc

Doc comments are only recognized when placed immediately before class, interface, constructor, method or field declarations.

When you embed HTML tags within a doc comment, you should not use heading tags such as <h1> and <h2>, because JavaDoc creates an entire structured document and these structural tags interfere with the formatting of the generated document.

Class and Interface Doc Tags

Constructor and Method Doc Tags

Class and Interface Doc Tags

@author name-text

- **Creates an "Author" entry.**

@version version-text

- **Creates a "Version" entry.**

@see classname

- **Creates a hyperlink "See Also classname"**

@since since-text

- **Adds a "Since" entry. Usually used to specify that a feature or change exists since the release number of the software specified in the "since-text"**

@deprecated deprecated-text

- **Adds a comment that this method can no longer be used. Convention is to describe method that serves as replacement**
- **Example: @deprecated Replaced by setBounds(int, int, int, int).**

Constructor and Method Doc Tags

Can contain @see tag, @since tag, @deprecated as well as:

@param parameter-name description

Adds a parameter to the "Parameters" section. The description may be continued on the next line.

@return description

Adds a "Returns" section, which contains the description of the return value.

@exception fully-qualified-class-name description

Adds a "Throws" section, which contains the name of the exception that may be thrown by the method. The exception is linked to its class documentation.

@see classname

Adds a hyperlink "See Also" entry to the method.

Example of a Class Doc Comment

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
 * @author  Sami Shaio
 * @version %I%, %G%
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
   ...
}
```

Example of a Method Doc Comment

```
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *              if the index is not in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
   ...
}
```

Example of a Field Doc Comment

A field comment can contain only the @see, @since and @deprecated tags

```
/**
 * The X-coordinate of the window.
 *
 * @see window#1
 */
int x = 1263732;
```

Example: Specifying a Service in Java

/** Office is a physical structure in a building. It is possible to create an instance of a office; add an occupant; get the name and the number of occupants */

```
public class Office {
        /** Adds an occupant to the office */
        * @param  NAME  name is a nonempty string */
        public void AddOccupant(string name);

        /** @Return Returns the name of the office. Requires, that Office has
            been initialized with a name */
        public string GetName();

    ....

}
```

Implementation of Application Domain Classes

New objects are often needed during object design:

- **Use of Design patterns lead to new classes**
- **The implementation of algorithms may necessitate objects to hold values**
- **New low-level operations may be needed during the decomposition of high-level operations**

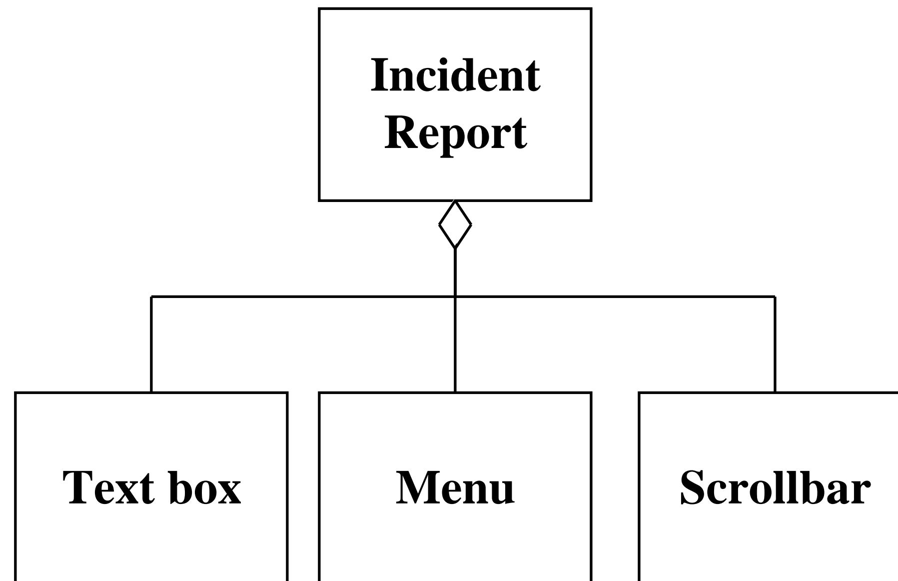
Example: The EraseArea() operation offered by a drawing program.

- **Conceptually very simple**
- **Implementation**
  - **Area represented by pixels**
  - **Repair () cleans up objects partially covered by the erased area**
  - **Redraw() draws objects uncovered by the erasure**
  - **Draw() erases pixels in background color not covered by other objects**

Application Domain vs Solution Domain Objects

Requirements Analysis (Language of Application Domain)

Object Design (Language of Solution Domain)



Package it all up

Pack up design into discrete physical units that can be edited, compiled, linked, reused

Construct physical modules

- **Ideally use one package for each subsystem**
- **System decomposition might not be good for implementation.**

Two design principles for packaging

- **<u>Minimize coupling:</u>**
  - **Classes in client-supplier relationships are usually loosely coupled**
  - **Large number of parameters in some methods mean strong coupling (> 4-5)**
  - **Avoid global data**
- **<u>Maximize cohesiveness:</u>**
  - **Classes closely connected by associations => same package**

Packaging Heuristics

Each subsystem service is made available by one or more interface objects within the package

Start with one interface object for each subsystem service

- **Try to limit the number of interface operations (7+-2)**

If the subsystem service has too many operations, reconsider the number of interface objects

If you have too many interface objects, reconsider the number of subsystems

Difference between interface objects and Java interfaces

- ***Interface object :*** **Used during requirements analysis, system design and object design. Denotes a service or API**
- ***Java interface:*** **Used during implementation in Java (A Java interface may or may not implement an interface object)**

Summary

Object design closes the gap between the requirements and the machine.

Object design is the process of adding details to the requirements analysis and making implementation decisions

Object design includes:

1. **Service specification**
2. **Component selection**
3. **Object model restructuring**
4. **Object model optimization**

Object design is documented in the Object Design Document, which can be generated using tools such as JavaDoc.