

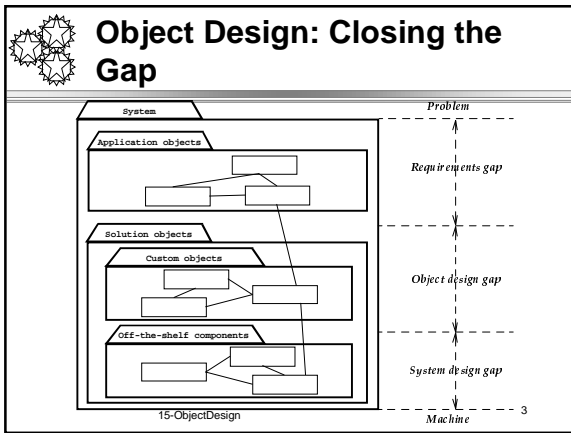
Chapter 7, Object Design

15-ObjectDesign 1

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

15-ObjectDesign 2



Object Design Issues

- Full definition of associations
- Full definition of classes
- Choice of algorithms and data structures
- Detection of new application-domain independent classes (example: Cache)
- Optimization
- Increase of inheritance
- Decision on control
- Packaging

15-ObjectDesign 4

Terminology of Activities

- Object-Oriented Methodologies
 - *System Design*
 - ◆ Decomposition into subsystems
 - *Object Design*
 - ◆ Implementation language chosen
 - ◆ Data structures and algorithms chosen
- SA/SD 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
 - Transforms the object design model to address performance criteria such as response time or memory utilization.

15-ObjectDesign 6



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

15-ObjectDesign

7



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.

15-ObjectDesign

8



Information Hiding Heuristics

- Build firewalls around classes
 - Carefully define public interfaces for classes as well as subsystems
- 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

15-ObjectDesign

9



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.

15-ObjectDesign

10



Add Type Signature Information

```

classDiagram
    class Hashtable {
        -numElements:int
        +put()
        +get()
        +remove()
        +containsKey()
        +size()
    }
  
```

```

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

15-ObjectDesign

11



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.

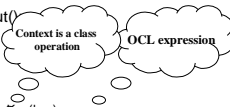
15-ObjectDesign

12



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 inv: numElements >= 0
 - Precondition:
 - context Hashtable::put(key, entry) pre: containsKey(key)
 - Post-condition:
 - context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry



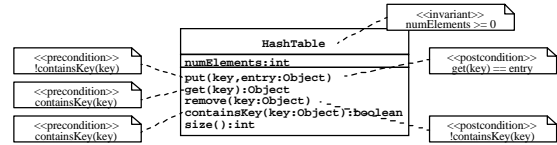
15-ObjectDesign

13



Expressing Constraints in UML

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



15-ObjectDesign

14



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.

15-ObjectDesign

15



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

15-ObjectDesign

16



Reuse...

- Look for existing classes in class libraries
 - JSAPI, JTAPl,
- 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

15-ObjectDesign

17



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.

15-ObjectDesign

18



Restructuring Activities

- Realizing associations
- Revisiting inheritance to increase reuse
- Revising inheritance to remove implementation dependencies

15-ObjectDesign

19



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.

15-ObjectDesign

20



Building a super class from several classes

- 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

15-ObjectDesign

21



Implement Associations

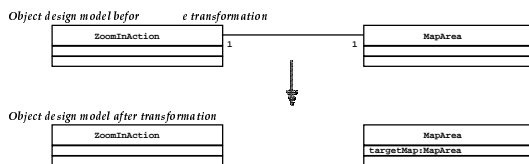
- Strategy for implementing associations:
 - Be as uniform as possible
 - Individual decision for each association
- Example of uniform implementation
 - 1-to-1 association:
 - ◆ Role names are treated like attributes in the classes and translate to references
 - 1-to-many association:
 - ◆ Translate to Vector
 - Qualified association:
 - ◆ Translate to Hash table

15-ObjectDesign

22



Unidirectional 1-to-1 Association

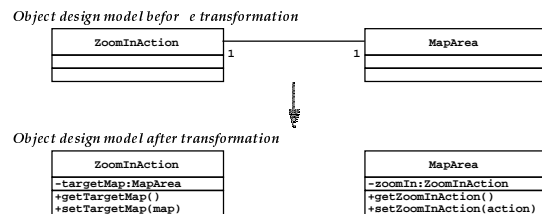


15-ObjectDesign

23

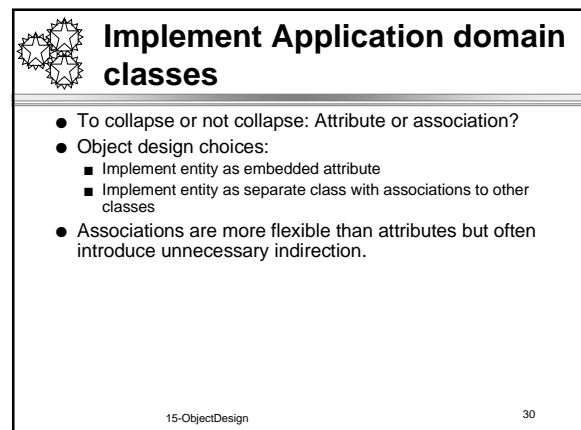
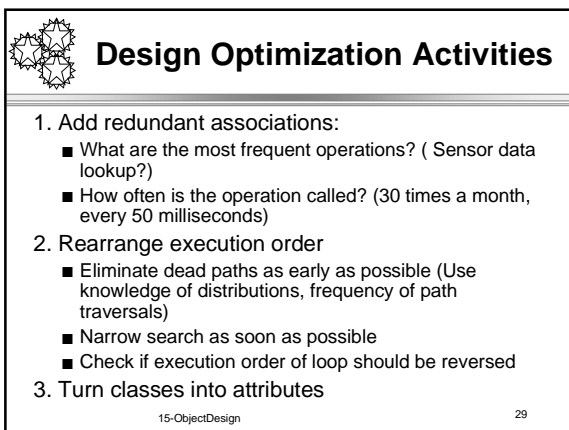
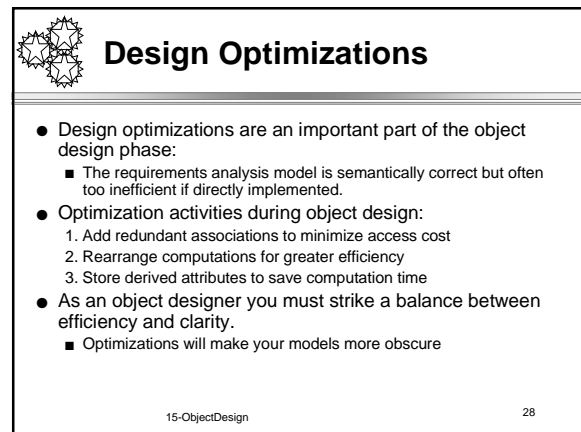
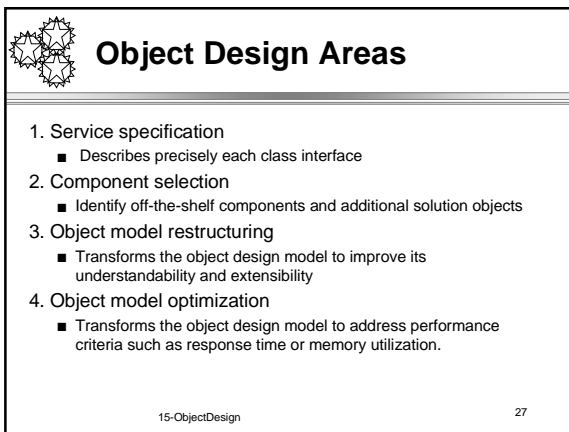
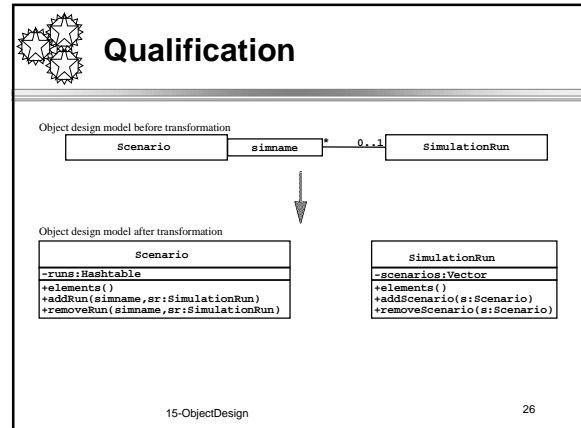
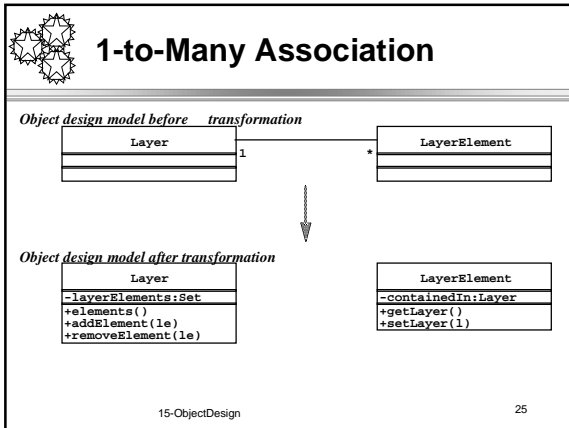


Bidirectional 1-to-1 Association



15-ObjectDesign

24



Optimization Activities: Collapsing Objects

The diagram illustrates the process of collapsing an object into an attribute. On the left, a `Person` class is associated with a `SocialSecurity` class. The `SocialSecurity` class has an attribute `ID:String`. An arrow points to the right, where the `SocialSecurity` class is collapsed into an attribute `SSN:String` within the `Person` class.

15-ObjectDesign 31

To Collapse or not to Collapse?

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

15-ObjectDesign 32

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)

15-ObjectDesign 33

Optimization Activities: Delaying Complex Computations

The diagram shows a class hierarchy and association. At the top is the `Image` class with attributes `filename:String`, `data:byte[]` and methods `width()`, `height()`, and `paint()`. Below it is another `Image` class with attributes `filename:String` and methods `width()`, `height()`, and `paint()`. This second `Image` class is associated with `ImageProxy` and `RealImage`. `ImageProxy` has attributes `filename:String` and methods `width()`, `height()`, and `paint()`. `RealImage` has attributes `data:byte[]` and methods `width()`, `height()`, and `paint()`. The association between `ImageProxy` and `RealImage` is labeled `image` with multiplicity `1` and `0..1`.

15-ObjectDesign 34

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>

15-ObjectDesign 35

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.

15-ObjectDesign 36



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
 */
```

15-ObjectDesign

37



More on Java Doc

- 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

15-ObjectDesign

38



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).`

15-ObjectDesign

39



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.

15-ObjectDesign

40



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 {
    ...
}
```

15-ObjectDesign

41



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 StringIndexOutOfBoundsException
 *         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) {
    ...
}
```

15-ObjectDesign

42



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;
```

15-ObjectDesign

43



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();

    ....
}
```

15-ObjectDesign

44



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

15-ObjectDesign

45

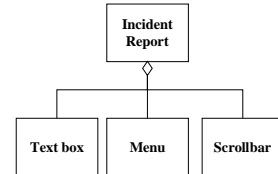


Application Domain vs Solution Domain Objects

Requirements Analysis
(Language of Application Domain)



Object Design
(Language of Solution Domain)



15-ObjectDesign

46



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

15-ObjectDesign

47



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)

15-ObjectDesign

48



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.