Object Design – Reusing Design Patterns

The Analysis Model contains application domain objects and describes the purpose of the system. During system design, the system is described in terms of its architecture: the system decomposition in subsystems, the global control flow, and persistency management. Also, during system design we define the hardware/software platform on which we build the system. This allows the selection of off-the-shelf components that provide a higher level of abstraction than the hardware.

During the second/last phase of design referred as object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. Object design includes:

- *reuse*, during which we identify off-the-shelf components and design patterns to make use of existing solutions
- *service specification*, during which we precisely describe each class interface
- *object model restructuring*, during which we transform the object design model to improve its understandability and extensibility
- *object model optimization*, during which we transform the object design model to address performance criteria such as response time or memory utilization.

Like system design, object design, is not algorithmic. The identification of existing patterns and components is central to the problem-solving process. In this course, we provide an overview of object design and focus on reuse, that is the selection of components and the application of design patterns.

An Overview of Object Design

Conceptually, software system development fills the gap between a given problem and an existing machine. The activities of system development incrementally close this gap by identifying and defining objects that realize part of the system (Figure 8-1).

Analysis reduces the gap between the problem and the machine by identifying objects representing problem-specific concepts. During analysis the system is described in terms of external behavior such as its functionality (use case model), the application domain concepts it manipulates (object model), its behavior in terms of interactions (dynamic model), and its nonfunctional requirements.

System design reduces the gap between the problem and the machine in two ways. First, system design results in a virtual machine that provides a higher level of abstraction than the machine. This is done by selecting off-the-shelf components for standard services such as middleware, user interface toolkits, application frameworks, and class libraries. Second, system design identifies off-the-shelf components for application domain objects such as reusable class libraries of banking objects.

After several iterations of analysis and system design, the developers are usually left with a puzzle that has a few pieces missing. These pieces are found during object design. This includes identifying new solution objects, adjusting off-the-shelf components, and precisely specifying each subsystem interface and class. The object design model can then be partitioned into sets of classes that can be implemented by individual developers.

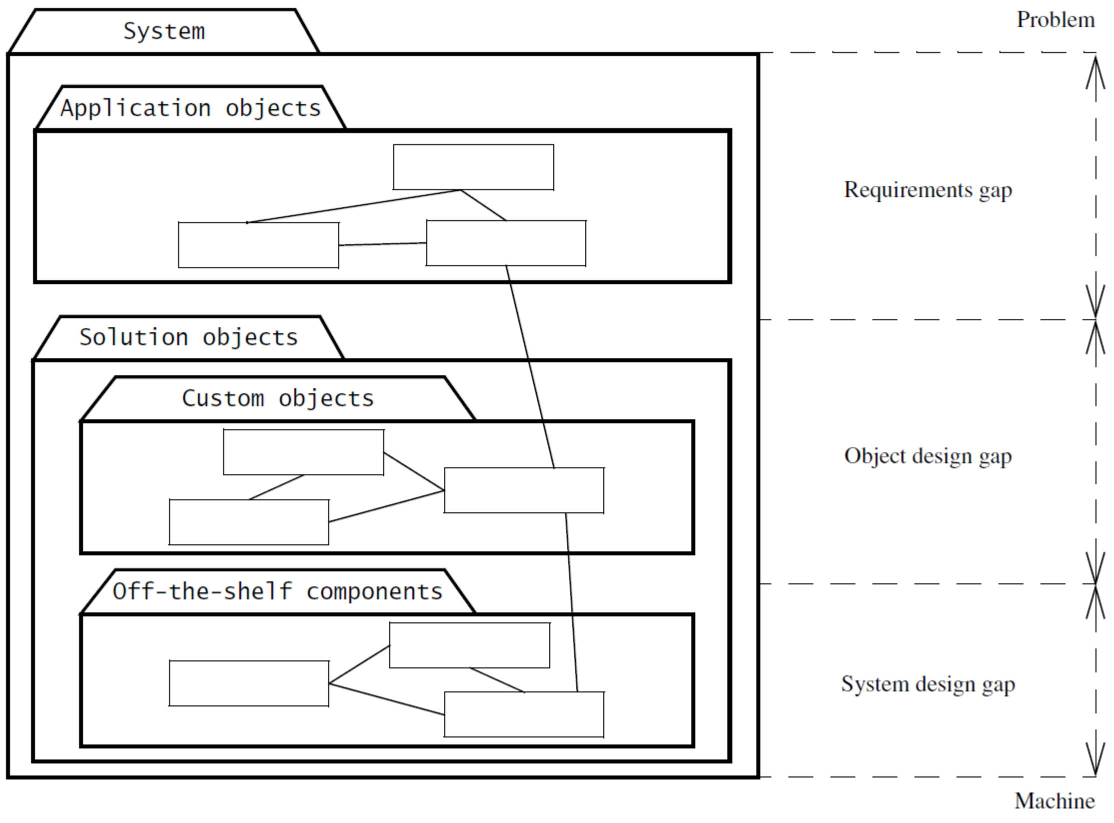


Figure 8-1 Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design (stylized UML class diagram).

As can be see in (Figure 8-2), Object design activities are included in one of the four groups, represented in the figure.

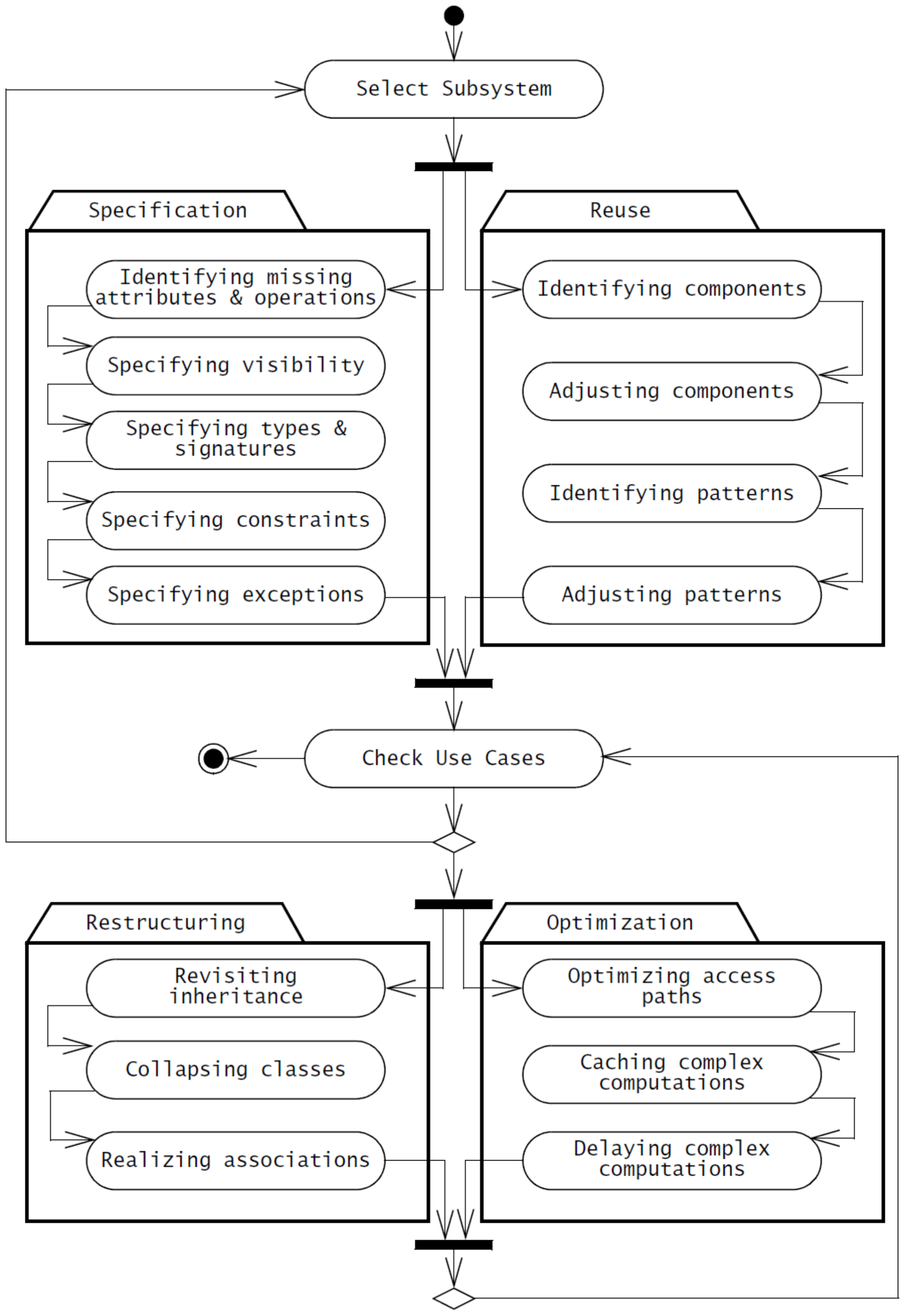


Figure 8-2 Activities of object design (UML activity diagram).

Reuse Concepts: Solution Objects, Inheritance, and Design Patterns

Application Objects and Solution Objects

Application objects, also called "domain objects", or "problem domain objects" represent concepts of the domain that are relevant to the system. **Solution objects** represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.

During analysis, we identify entity objects and their relationships, attributes, and operations. Most entity objects are application objects that are independent of any specific system. During analysis, we also identify solution objects that are visible to the user, such as boundary and control objects representing forms and transactions defined by the system. During system design, we identify more solution objects in terms of software and hardware platforms. During object design, we refine and detail both application and solution objects and identify additional solution objects needed to bridge the object design gap.

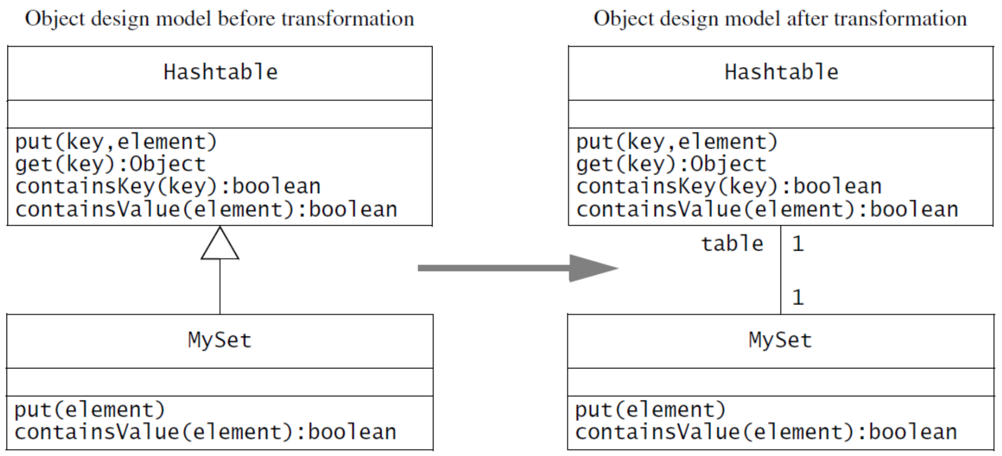
Specification Inheritance and Implementation Inheritance

During analysis, by using inheritance objects are classified into taxonomies. This allows us to differentiate the common behavior of the general case, that is, the **ancestor** (also called the "**base class**"), from the behavior that is specific to specialized objects, that is, the **descendants** (also called the "**derived classes**").

The focus of inheritance during object design is to reduce redundancy and enhance extensibility. By factoring all redundant behavior into a single superclass, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we must make changes only once for all subclasses. By providing abstract classes and interfaces that are used by the application, we can write new specialized behavior by writing new subclasses that comply with the abstract interfaces.

Although inheritance can make an analysis model more understandable and an object design model more modifiable or extensible, these benefits do not occur automatically. On the contrary, inheritance is such a powerful mechanism that novice developers often produce code that is more obfuscated and more brittle than if they had not used inheritance in the first place.

The use of inheritance for the sole purpose of reusing code is called **implementation inheritance**. With implementation inheritance, developers reuse code quickly by sub-classing an existing class and refining its behavior. A Set implemented by inheriting from a Hashtable is an example of implementation inheritance (Figure 8-3). Conversely, the classification of concepts into type hierarchies is called **specification inheritance** (also called "interface inheritance"). The UML class model of Figure 8-4 summarizes the four different types of inheritance we discussed in this section.


Object design model before transformation
Object design model after transformation
Hashtable
put(key,element)
get(key):Object
containsKey(key):boolean
containsValue(element):boolean
MySet
put(element)
containsValue(element):boolean
table
1
1

```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
            put(element, this);
        }
    }
    boolean containsValue(Object
            element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

```
/* Implementation of MySet using
delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object
            element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

Figure 8-3 An example of implementation inheritance. The left column depicts a questionable implementation of MySet using implementation inheritance. The right column depicts an improved implementation using delegation (UML class diagram and Java).

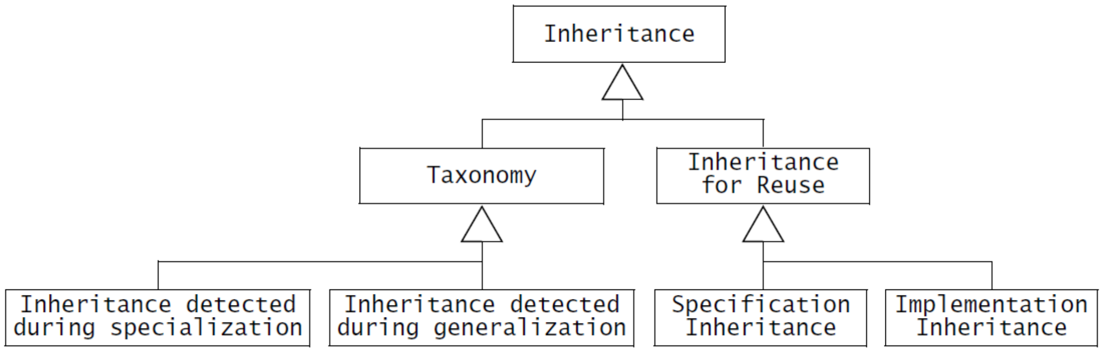

Inheritance
Taxonomy
Inheritance for Reuse
Inheritance detected during specialization
Inheritance detected during generalization
Specification Inheritance
Implementation Inheritance

Figure 8-4 Inheritance metamodel

In object-oriented analysis and design, inheritance is used for achieving several goals, in particular modeling taxonomies and reusing behavior from abstract classes. When modeling taxonomies, the inheritance relationships can be identified either during specializations (when specialized classes are identified after general ones) or during generalizations (when general classes are abstracted out of a number of specialized ones). When using inheritance for reuse, specification inheritance represents subtyping relationships, and implementation inheritance represents reuse among conceptually unrelated classes.

Delegation

Delegation is the alternative to implementation inheritance that should be used when reuse is desired. A class is said to delegate to another class if it implements an operation by resending a message to another class instance. Delegation makes explicit the dependencies between the reused class and the new class. The right column of Figure 8-3 shows an implementation of MySet using delegation instead of implementation inheritance. The only significant change is the private field table and its initialization in the MySet() constructor. This address both problems we mentioned before:

- *Extensibility*. The MySet on the right column does not include the containsKey() method in its interface and the new field table is private. Hence, we can change the internal representation of MySet to another class (e.g., a List) without impacting any clients of MySet.

- *Subtyping*. MySet does not inherit from Hashtable and, hence, cannot be substituted for a Hashtable in any of the client code. Consequently, any code previously using Hashtables still behaves the same way.

Delegation is a preferable mechanism to implementation inheritance as it does not interfere with existing components and leads to more robust code. Note that specification inheritance is preferable to delegation in subtyping situations as it leads to a more extensible design.

Liskov Substitution Principle

If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

Interpretation

In object design, the Liskov Substitution Principle means that if all classes are subtypes of their ancestors/superclasses, all inheritance relationships are specification inheritance relationships. In other words, a method written in terms of a superclass T must be able to use instances of any subclass of T without knowing whether the instances are of a subclass. Consequently, new subclasses of T can be added without modifying the methods of T, hence leading to an extensible system. An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.

Delegation and Inheritance in Design Patterns

In general, when to use delegation or inheritance requires some experience and judgement on the part of the developer. Inheritance and delegation, used in different combinations, can solve a wide range of problems: decoupling abstract interfaces from their implementation, wrapping around legacy code, and/or decoupling classes that specify a policy from classes that provide mechanism. In object-oriented development, **design patterns** are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]. Specification Inheritance and delegation are the 2 relationships used in all **design patterns**.