Components Based Design and Development

Computer Engineering Studies

Universidad Carlos III de Madrid

Unit 1: Fundamentals of Components

Juan Llorens

Högskolan på Åland – Finland / Universidad Carlos III de Madrid - Spain

Juan.llorens@uc3m.es

Object Oriented Detailed Design

- Detailed Design is the process of decomposing a module/system into objects.
- This decomposition can be affected by:
  - Encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, etc.

    E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software
- The Inputs to OO Detailed Design must be:
  - The Requirements affecting design
  - The Domain Objects and Classes found in Requirements Analysis (RA)
  - The Domain Objects and Classes collaborations defined in RA
  - The SW Architecture (including Architectural Objects and classes when they exist)
  - The Modules/sub-systems solution principles if they were defined in the Architectural Design

Object Oriented Detailed Design

- The Outputs to be produced in an OO Detailed Design must be
  - The objects structure
  - Their corresponding classes structure (interfaces, Attributes, Operations)
  - The collaborations between the objects (messages, signals,..)
  - The definition of the possible states of the objects
  - The organization of the objects in components and nodes
  - The way the structures, states and collaborations can be tested

Object Oriented Detailed Design

- Therefore, the problem in detailed design can be located in

"*how to organize, integrate and adapt the domain and architectural objects, as well as their structure, interfaces, operative, collaborations and states*".

- Many problems, specific to Object Orientation, may oblige to include new specific objects (Detailed Design Objects) to solve them.

- Many of these problems have already been solved and the objects/classes structures are available for us designers..

They are forming the Design Patterns

Identified problems in Object Oriented Detailed Design

- Not setting the focus on the collaboration
- Finding Appropriate Objects
- Determining Object Granularity
- Specifying Object Implementations
  - Class versus type: Specifying object interfaces
  - Programming to an Interface / Programming to an Implementation
  - The creation of objects
  - Inheritance versus Composition
  - Delegation
  - Inheritances versus Parameterized Types
  - The structures of Objects
  - The behavior of Objects
  - Coupling
- Relating Run-Time and Compile-Time structures
- Designing for Change

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Find Appropriate Objects

- Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays. Others are much higher-level.
  - By using Design Patterns, for example, the Composite pattern introduces an abstraction for treating objects uniformly that doesn't have a physical counterpart. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Determining Object Granularity

- Objects can vary tremendously in size and number. They can represent everythink down to the hardware or all the way up to entire applications
  - Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (315) pattern describes how to implement interchangeable families of algorithms. The State (305) pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class Versus Type

- A Type is a name used to denote a particular interface
  - We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window."
- An object may have many types, and widely different objects can share a type.
- Part of an object's native interface may be characterized by one type, and other parts by other types.
- Two objects of the same type need only share parts of their native interfaces.
- Interfaces can contain other interfaces as subsets. We say that a type is a subtype of another if its interface contains the interface of its supertype.
- Often we speak of a subtype *inheriting* the interface of its supertype.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation

- Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces.
- There is no way to know anything about an object or to ask it to do anything without going through its interface.
- An object's interface says nothing about its implementation—different objects are free to implement requests differently.
- Two objects having completely different implementations can have identical interfaces.
- Dynamic binding means that issuing a request doesn't commit you to a particular implementation until runtime.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation II

- Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.
- Dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as polymorphism [..].
- It lets a client object make few assumptions about other objects beyond supporting a particular interface.
- Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.
- An object's implementation is defined by its class.
  - The class specifies the object's internal data and representation and defines the operations the object can perform.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation III

- Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.
- Subclasses can refine and redefine behaviors of their parent classes. More specifically, a class may override an operation defined by its parent class.
- Overriding gives subclasses a chance to handle requests instead of their parent classes.
  - Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.

"Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface."

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation IV

"Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface."

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Differences between Class and its Type

- An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations.
- An object's type only refers to its interface—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

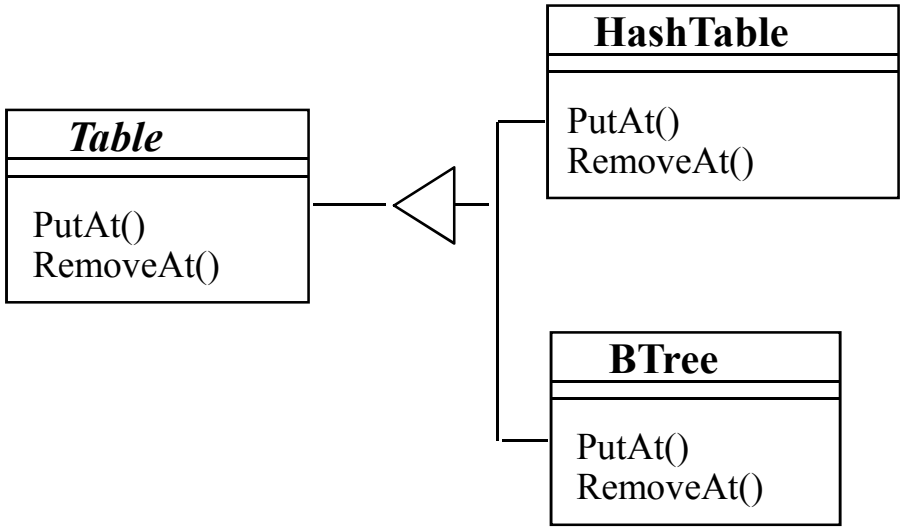
Class inheritance / Interface Inheritance (subtyping)

- Class inheritance defines an object's implementation in terms of another object's implementation.
  - In short, it's a mechanism for code and representation sharing.
- Interface inheritance (or subtyping) describes when an object can be used in place of another.
- The standard way to inherit an interface in C++ is to inherit publicly from a class that has (pure) virtual member functions.
  - Pure interface inheritance can be approximated in C++ by inheriting publicly from pure abstract classes. Pure implementation or class inheritance can be approximated with private inheritance.
- In Smalltalk, inheritance means just implementation inheritance.
  - You can assign instances of any class to a variable as long as those instances support the operation performed on the value of the variable.

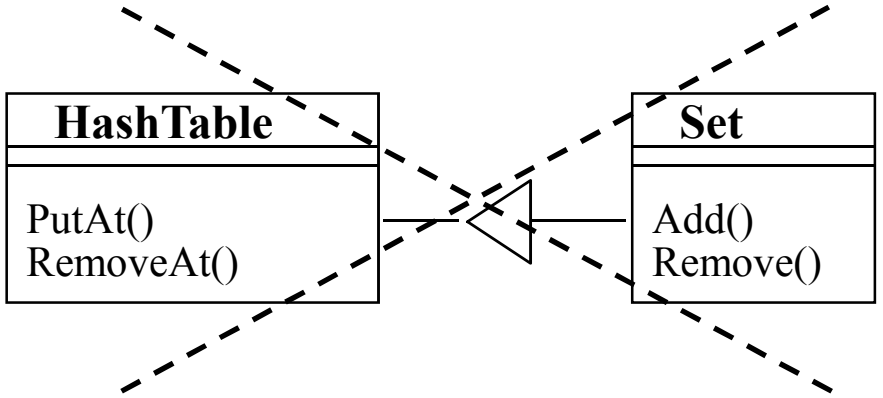
E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class inheritance versus Interface Inheritance

- Interface inheritance (i.e., subtyping):
  - describes when different types of objects can be used in place of each other



- Implementation inheritance:
  - an object's implementation is defined in terms of the implementation of another.



Program to an Interface, not an Implementation

- Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes.
- But inheritance's ability to define families of objects with *identical* interfaces (usually by inheriting from an abstract class) is also important.
  - Why? Because polymorphism depends on it.
- When inheritance is used carefully (some will say *properly*), all classes derived from an abstract class will share its interface.
  - This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. *All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Program to an Interface, not an Implementation: Benefits

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

These are the fundamentals of the creational patterns

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class Inheritance versus object composition

Favor object composition over class inheritance.

- The two most common techniques for reusing functionality in object-oriented
- Class inheritance lets you define the implementation of one class in terms of another's.
  - Reuse by subclassing is often referred to as white-box reuse. The term "whitebox“ refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.
- Object composition is an alternative to class inheritance.
  - New functionality is obtained by assembling or *composing* objects to get more complex functionality. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes.“
- Class Inheritance is defined at compile time while Object composition at run-time

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class Inheritance : advantages and disadvantages

- Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language.
- Class inheritance also makes it easier to modify the implementation being reused.
- When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Disadvantages

- You can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time.
- Parent classes often define at least part of their subclasses' physical representation.
  - The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.
- Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Object Composition : advantages and disadvantages

- Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others.

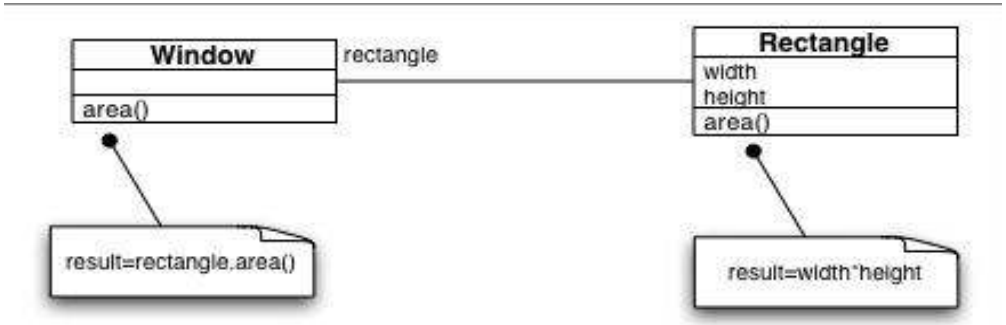
Advantages

- Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at runtime by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.
- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.
- This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the this member variable in C++ and self in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.



E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Delegation : advantages and disadvantages

- The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they're composed.
- Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software.
- There are also run-time inefficiencies.
- Delegation is a good design choice only when it simplifies more than it complicates.
- It isn't easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and on how much experience you have with it.
- Delegation works best when it's used in highly stylized ways—that is, in standard patterns.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software