Foundations of Design Patterns

.NET Design Patterns
Scott E. Robertson, UCLA Extension

Goal of Design Patterns Class

- Introduce the principles of design patterns
- Study original Gang of Four (GOF) catalog of twenty-three design patterns and some post-GOF design patterns
- Show design pattern's role in architecting complex systems
- Develop proficiency in the use of design patterns

Prerequisites for Design Patterns

- **Minimal C# language programming skills**
  - **Sequential statements, decision logic, and iterative commands**
  - **Classes and Objects, Interfaces, Types**
  - **Overriding/overloading methods**
- **Familiarity with object-oriented programming (OOP)**
  - **Encapsulation, Inheritance, Polymorphism, Abstraction**

Definition of Design Pattern

- **A design pattern systematically names, motivates, and explains a recurring design problem in object-oriented (OO) systems.**
- **It describes the OO design problem and its solution, often giving practical examples.**
  - **The solution is a general arrangement of objects and classes that solve the problem.**
  - **The solution is customized and implemented to solve the problem in a particular context.**

Benefits of Design Patterns

- **Leverage past experience in designing OOP software**
- **Promote the reuse of common designs and architectures**
- **Clarify and extend our understanding of successful OOP methodologies**

Features of Design Patterns

- Implemented in standard object-oriented languages
- Describe reusable and adaptable solutions to specific software engineering problems
- Emphasize greater reuse and flexibility in software engineering
- Develop simpler and tighter architectures
- Require no programming tricks

Origin of Design Pattern Concept

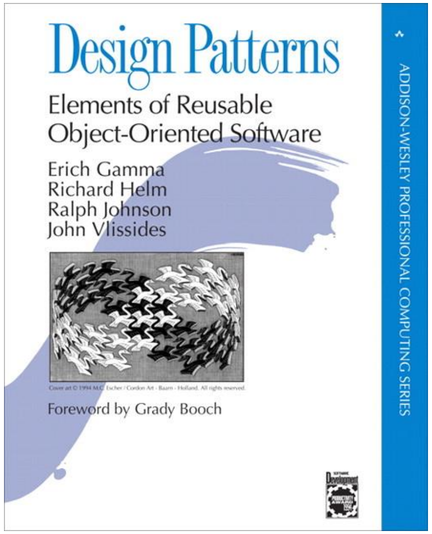
- **Christopher Alexander, an architect, introduced design patterns as a pattern language to architect buildings and cities.**
- **He proposed a practical architectural system emphasizing rules and pictures, describing methods for constructing practical, safe, and attractive designs.**
- **Design-patterns quickly took hold in OOP community.**

Christopher Alexander on Design Patterns

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times, over, without ever doing it the same way twice.” [Christopher Alexander, et. al., A Pattern Language: Oxford University Press, New York, 1977]

The Gang of Four Design Patterns

- **Design patterns were formalized in Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides's seminal work *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley) in 1994.**
- **This book specifies and describes the 23 core patterns that form the foundation of design patterns.**



Elements of Design Patterns

- **Name**
  - **the handle used to describe a design problem, solutions, and consequences in word or two, providing a design vocabulary**
- **Problem**
  - **describes when to apply the pattern, explaining the problem and its context**

Elements of Design Patterns (cont'd)

- **Solution**
  - **describes elements that make up the design, their relationships, responsibilities, and collaborations, but does not describe particular concrete design but, instead, the template to be applied in different situations**
- **Consequences**
  - **identifies results and trade-offs of applying a design pattern, including impact on system's flexibility, extensibility, or portability**

Features of Design Patterns

- **Describe how communicating objects and classes are customized to solve a general object-oriented design problem in a particular context**
- **Name, abstract, identify key aspects of common design structures making them useful for creating a reusable OO design**
- **Identify the participating classes and objects, the roles and collaborations, and the distribution of responsibilities**

Classification of Design Patterns: Purpose

- **Purpose: reflects what pattern does**
  - **Creational patterns: concerned with the process of object creation**
  - **Structural patterns: deal with the composition of classes and objects**
  - **Behavioral patterns: characterize the ways in which the classes or objects interact and distribute responsibility**

Classification of Design Patterns: Scope

- **Scope: whether the pattern applies primarily to classes or objects**
  - **Class patterns deal with inheritance relationships between classes and their subclasses, which are fixed at compile time.**
  - **Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.**

Catalog of Design Patterns: Creational

- **Abstract Factory: Creates an instance of several families of classes**
- **Builder: Separates object construction from its representation**
- **Factory Method: Creates an instance of several derived classes**
- **Prototype: A fully initialized instance to be copied or cloned**
- **Singleton: A class of which only a single instance can exist**

Catalog of Design Patterns: Structural

- **Adapter: Match interfaces of different classes**
- **Bridge: Separates an object's interface from its implementation**
- **Composite: A tree structure of simple and composite objects**
- **Decorator: Add responsibilities to objects dynamically**
- **Façade: A single class that represents an entire subsystem**
- **Flyweight: A fine-grained instance used for efficient sharing**
- **Proxy: An object representing another object**

Catalog of Design Patterns: Behavioral

- **Chain of Responsibility: A way of passing a request between a chain of objects**
- **Command: Encapsulate a command request as an object**
- **Interpreter: A way to include language elements in a program**
- **Iterator: Sequentially access the elements of a collection**
- **Mediator: Defines simplified communication between classes**
- **Memento: Capture and restore an object's internal state**

Catalog of Design Patterns: Behavioral (cont'd)

- Observer: A way of notifying change to a number of classes
- State: Alter an object's behavior when its state changes
- Strategy: Encapsulates an algorithm inside a class
- Template Method: Defer the exact steps of an algorithm to a subclass
- Visitor: Defines a new operation to a class without change

Design Pattern Space: Purpose and Scope

- **Patterns classified by Purpose (Creational, Structural, Behavioral) and Scope (Class, Object)**
  - **Class Design Patterns**
    - ***Creational: Factory Method***
    - ***Structural: Adapter (class)***
    - ***Behavioral: Interpreter, Template Method***
  - **Object Design Patterns**
    - ***Creational: Abstract Factory, Builder, Prototype, Singleton***
    - ***Structural: Adapter (object), Bridge, Composite, Decorator, Façade, Flyweight, Proxy***
    - ***Behavioral: Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor***

Design Pattern Purpose and Scope

- Creational class patterns defer some part of object creation to subclasses; creational object patterns defer it to another object.
- Structural class patterns use inheritance to compose classes; structural object patterns describe ways to assemble objects.
- Behavioral class patterns use inheritance to describe algorithms and flow of control; behavioral object patterns describe how a group of objects cooperate to perform tasks.

Object-Oriented Programming Review (OOP)

- **An application design and programming framework emphasizing the use of objects, discrete, reusable units of programming logic and data.**

OOP Concepts

- **Abstraction: what are essential details?**
- **Three Pillars of OOP**
  - **Encapsulation: how does object represent internal data and implementation?**
  - **Inheritance: how does language promote code reuse?**
  - **Polymorphism: how are related objects treated in similar fashion?**

Abstraction

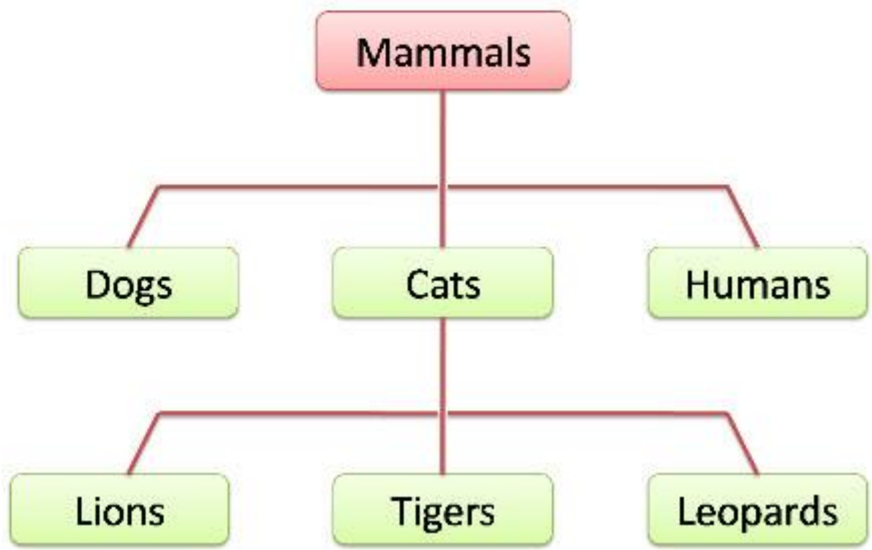
- **Approach recognizing and focusing on the essential details of a situation or entity and filtering out the nonessential details**
- **Example is roadmap, an abstraction of the roads, geographical features, and places of interest for a geographical region, which does not include every feature**

Pillar I: Encapsulation

- **Hides data (state) and implementation (operations) details**
- **Simplifies complexity, allowing for modularization and compartmentalization**

Pillar II: Inheritance

- Build new classes from existing class definitions
- Extends parent classes; enables derived classes to inherit core functionality
- Describes an “is-a” relationship, e.g., a triangle “is-a” shape



Example of Inheritance

```
public class BaseClass
  {
      public virtual void DoWork() { }
      public virtual int WorkProperty
      {
          get { return 0; }
      }
  }
  public class DerivedClass : BaseClass
  {
      public override void DoWork() { }
      public override int WorkProperty
      {
          get { return 0; }
      }
  }
```

Pillar III: Polymorphism

- Treat related objects in the same way
- Allows base class to define set of members (interface) implemented by all descendents

Example of Polymophism

```
static void Main(string[] args)
      {
          DerivedClass B = new DerivedClass();
          B.DoWork();  // Calls the new method.

          BaseClass A = (BaseClass)B;
          A.DoWork();  // Also calls the new method.
      }
```

What are Objects?

- Object-oriented programs are built from objects, packages of data (state) and instructions operating on that data (operations).
- An object's internal state is encapsulated, contained within the object.
- A client calls an operation to request a change in the object's state.
- Objects are defined (described) by classes.
- Objects may interact with other objects through their interfaces.

What is a Class?

- **An object's implementation is defined by a class.**
- **A class specifies object's internal data representation (state) and defines operations the object can perform.**
- **Process of instantiating class allocates storage for object's instance data, associating operations with data.**

Approaches to Modeling Objects

- Noun/Verb analysis
- Focus on collaborations and responsibilities
- Modeling real world entities

Specifying Object Interfaces

- **Operations defined by object must specify a signature: an operation name, parameters, and return value.**
- **The set of all signatures defined by an object's operations is called the interface to the object.**
- **Any request sent to the object must match a signature in the object's interface.**

Interface Example

```
public interface IEquatable<T>
  {
      bool Equals(T other);
  }

public class Car : IEquatable<Car>
  {
      public string Make { get; set; }
      public string Model { get; set; }
      public string Year { get; set; }

 public bool Equals(Car car)
      {
          if (this.Make == car.Make &&
                      this.Model == car.Model && this.Year == car.Year)
              return true;
          else
              return false;
      }
  }
```

What is a Type?

- **A type is the name identifying a particular interface.**
- **Type “Window”, e.g., accepts requests for operations on “Window” objects.**
- **A type is a subtype of another type if its interface contains, or inherits, the interface of the other type.**
- **An object can have many types, and types can be shared.**

Interfaces in OOP

- Interfaces are fundamental to object-oriented development and to design patterns.
- All interaction with an object is through interface members.
- Interface is pure protocol: it does not specify implementation, just definition.
- Different objects may have different implementations for the same interface member.

Dynamic Binding

- A request doesn't commit to particular implementation until run-time.
- Programs can be written with particular interface knowing an object with that interface will accept the request.
- This interchangeability is known as polymorphism.

Polymorphism

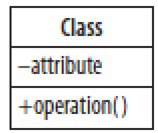
- Pillar of object-oriented programming
- Objects with identical interfaces substitute for each other at runtime
- Simplifies definition of clients, decouples objects from each other, lets relationships between objects vary at runtime
- Client makes no assumptions about objects beyond supporting particular interface

Interfaces and Design Patterns

- **Define interfaces by identifying their key members and the kinds of data that are sent to them.**
- **Design patterns specify relationships between interfaces and may place constraints on interfaces.**
- **Interfaces can be implemented as either abstract base classes or interface types.**

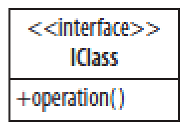
UML Class Diagrams

Class



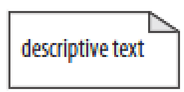
Types and parameters specified when important; access indicated by + (public), (private), and # (protected).

Interface



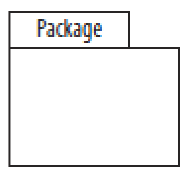
Name starts with I. Also used for abstract classes.

Note



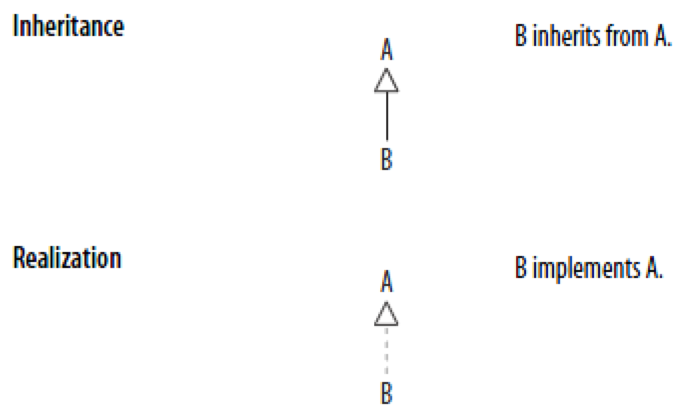
Any descriptive text.

Package

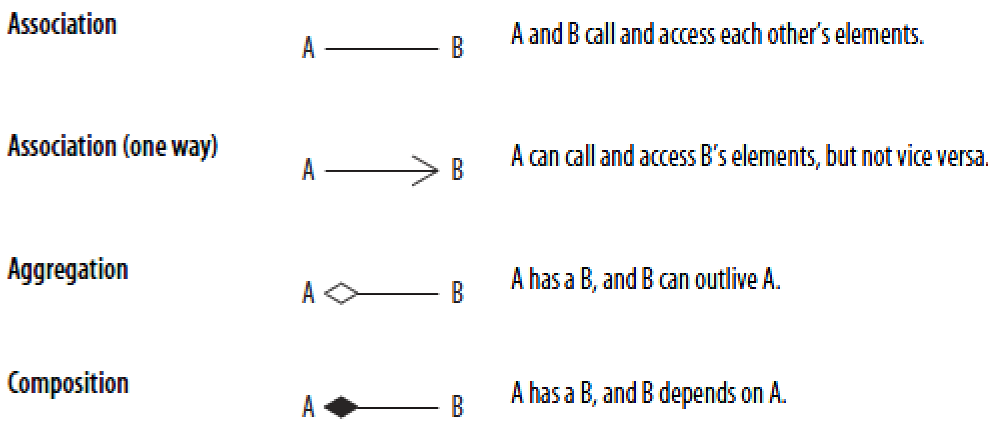


Grouping of classes and interfaces.

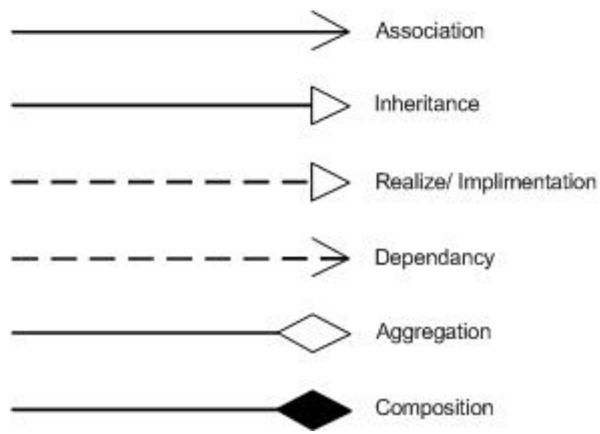
UML Class Diagrams



UML Class Diagrams



UML Class Diagrams



UML Class Diagram Notation

- Class depicted as rectangle with class name in bold-faced text.
- Operations appear in normal type below class name.
- Data that class defines comes after the operations in the box.
- Lines separate class name from operations and operations from data.
- Return type is optional.

Class Inheritance

- Classes defined in terms of existing classes use class inheritance.
- A child class inherits the data and operations from the parent class.
- The child class can override parent-class operations and add some of its own.
- Inheritance class relationship is indicated with a vertical line and a triangle.

Value of Inheritance

- Inheritance is a mechanism for extending application functionality by reusing functionality defined in the parent class.
- New kinds of objects can be defined in terms of old ones.
- Polymorphism depends on families of objects having identical interfaces.
- Subclasses share the interface of the base class, adding or overriding operations.

Abstract Classes

- **Purpose of abstract class is to define common interface for its subclasses.**
- **Abstract operations defers some or all of their implementation to subclasses.**
- **Abstract classes cannot be instantiated.**
- **Non-abstract classes or methods are referred to as concrete rather than as abstract.**

Other Notations

- **Abstract classes appear in slanted type to distinguish them from concrete classes.**
- **Abstract operations also appear in slanted type.**
- **Pseudocode appears in a separate dog-eared box connected by a dashed line.**

Class versus Interface Inheritance

- **A difference exists between object's class and its type.**
- **Object's class defines how object is implemented, how internal state and implementation are defined.**
- **Object's type refers only to its interface, the set of requests to which it can respond.**
- **An object can have many types, and there can be many objects of the same type.**

More on Inheritance

- Class inheritance defines object’s implementation in terms of another object’s implementation.
- Class can define both abstract and virtual members.
- Interface inheritance defines an object that can be used in place of another.
- Interface is pure protocol, i.e., member definitions only.

Programming to an Interface

- **Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.**
- **Clients remain unaware of the classes that implement these objects; they know only about the abstract classes defining the interface.**

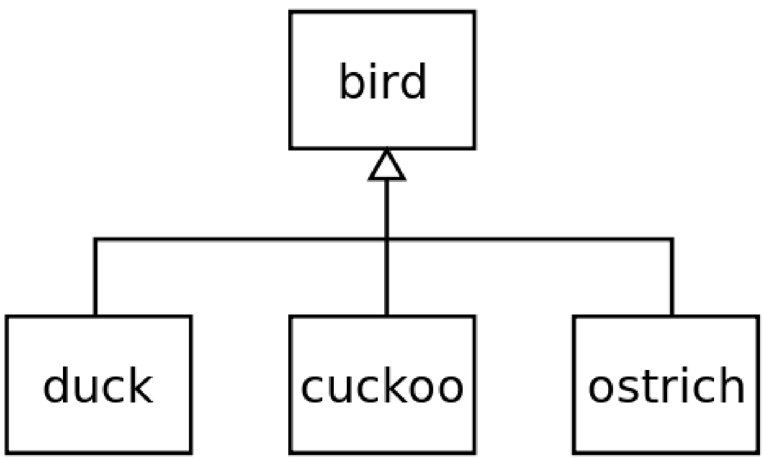
First Principle of OO Design

- **Program to an interface, not an implementation.**
  - **Don’t declare variables as instances of concrete classes.**
  - **Instead commit only to an interface defined by an abstract class.**

Inheritance and Composition

- **Class inheritance and object composition are two most common techniques for reusing functionality.**

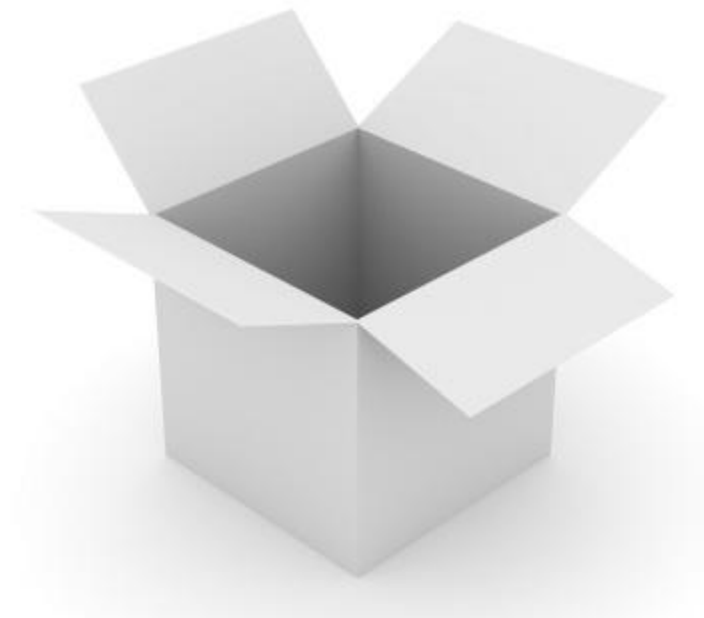
Inheritance



Composition

White-Box Reuse (Inheritance)

- **White-box reuse refers to the use of class inheritance, i.e., defining the implementation of one class in terms of another; the internals of parent classes are often visible to subclasses.**



Black-Box Reuse (Composition)

- **Black-box reuse refers to the use of new functionality by assembling or composing objects to get more complex functionality; no internal details of objects are visible.**

Advantages of Class Inheritance

- **Defined statically at compile-time and easy to use**
- **Supported directly by programming language**
- **Easy to modify implementation**

Disadvantages of Class Inheritance

- Cannot change implementations inherited from parent classes at run-time.
- Parent classes often define part of subclasses' physical representation.
- Inheritance exposes subclass to details of parent's implementation.
- Inheritance can break encapsulation.

More on Disadvantages of Class Inheritance

- **Dependency limits flexibility and ultimately reusability.**
- **The Fragile Base Class Problem**

Object Composition

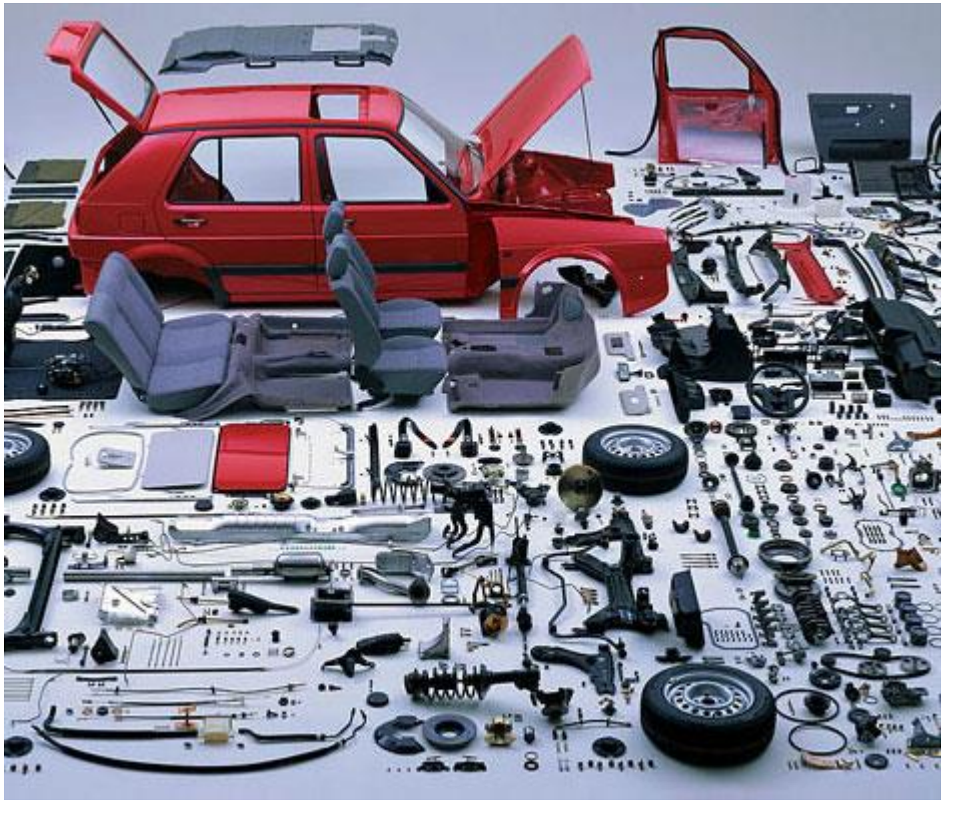
- **Object composition is defined dynamically at run-time through objects acquiring references to other objects.**
- **Composition defined by relationships between types, not by inheritance, but by internal references between objects.**
- **Composition usually implemented as an object containing another object(s).**
- **Objects must respect each other's interfaces, not breaking encapsulation.**

More on Object Composition

- Design based on object composition will have more objects and behavior will, instead, depend on interrelationships.
- Each class is encapsulated and focused on one task.
- Objects referred to must be considered parts of the whole and have no independent existence.
- Fewer implementation dependencies exist because implementation written in terms of interfaces.

Real World Example of Composition

- **Automobile and its parts (steering wheel, seats, engine): automobile 'has or is composed from' parts.**

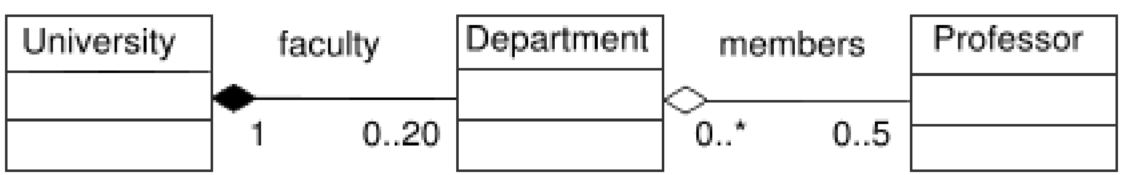


Related Concept of Aggregation

- **Aggregation also allows combination of simple objects and data types into more complex objects without using inheritance.**
- **Aggregation differs from ordinary composition in that it does not imply ownership.**
- **With composition, when the owning object is destroyed, so are contained objects; with aggregation, this is not necessarily true.**
- **With Aggregation, the object may only contain a reference to another object and not have lifetime responsibility for it.**

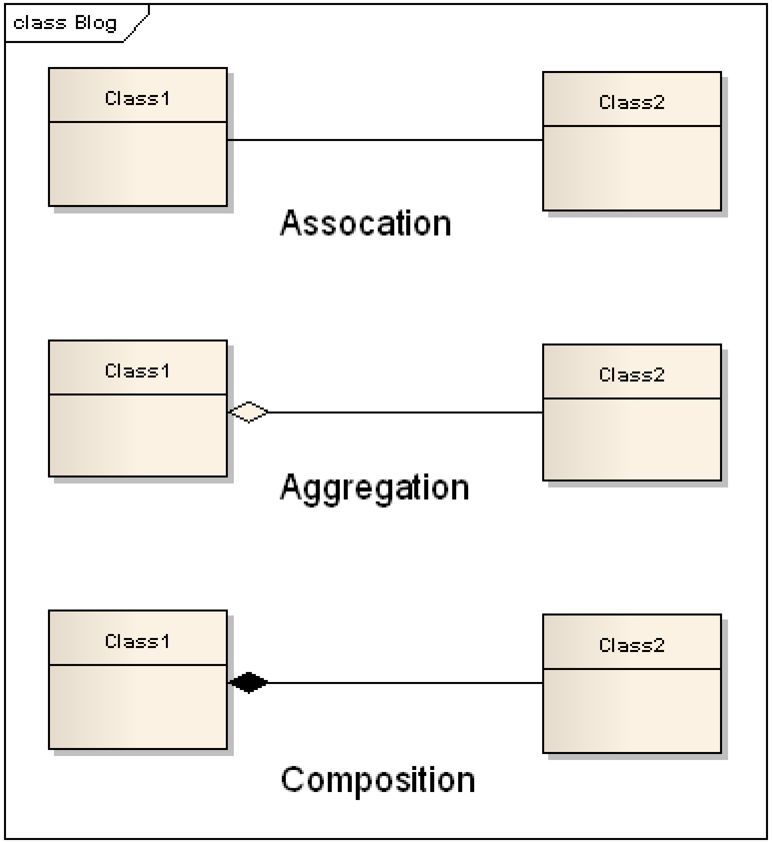
Real World Example Contrasting Composition and Aggregation

- University or College
  - The university has departments (e.g., Economics).
  - Each department has professors.
  - If the university closes, the departments no longer exist, but the professors continue to exist.
  - University can be seen as composition of departments; departments can be seen as an aggregation of professors.
  - A professor could work in more than one department, but a department cannot be part of more than one university.



Aggregation and Composition UML Notation

- **Composition depicted as filled diamond and solid line; implies multiplicity of 1 or 0..1: no more than one object can have lifetime responsibility for another object.**
- **Aggregation is depicted a unfilled diamond and solid line.**



Other Related Concepts

- Containment
  - When composition is used to store several instances of composed data type
  - Containment depicted with multiplicity of 1 or 0..n
  - Examples: arrays, linked lists, binary trees, and associative arrays
- Association
  - Represents ability of one instance to send message to another
  - Typically implemented with reference instance variable, but can be implemented as method argument, or creation of local variable

Second Principle of OO Design

- **Favor object composition over class inheritance**
  - **Should be able to get all necessary functionality and polymorphic behavior by assembling existing components through object composition**
  - **Set of available components may not be enough, so reuse by inheritance makes it easier to create new components**
  - **Inheritance is often overused as a reuse technique.**

Benefits of Composition

- Composition simplifies initial design of business classes; to provide new functionality, just provide another class implementing behavior and reference it.
- Composition provides more isolation of interests than can a class hierarchy.
- Composition accomodates changes more easily because it avoids restructuring the inheritance model.

Delegation and Composition Relationship

- **Two objects are involved in handling a request: the receiving object delegates operations to its delegate, another object.**
- **Sometimes both objects reference each other; i.e., the references are bi-directional.**
- **Delegation makes composition as powerful for reuse as inheritance.**
- **Instead of calling a method in the parent class, the receiver calls a delegated operation.**

More on Delegation and Composition

- For example: Instead of making class Window a subclass of Rectangle, Window class might reuse the behavior of Rectangle by keeping a Rectangle instance variable and delegating Rectangle-specific behavior to it.
- Window now forwards request to Rectangle explicitly, rather than inheriting those operations.
- Delegation is depicted by a plain arrowhead, indicating a class keeps a reference to an instance of another class.

Advantages and Disadvantages of Delegation

- **Advantages**
  - **Easy to compose behaviors at run-time and change way they're composed**
  - **Makes software more flexible at run-time**
- **Disadvantages**
  - **Software is harder to understand than static software**
  - **May be run-time inefficiencies**

Delegation versus Inheritance

- **Delegation is good design choice when it simplifies more than complicates.**
- **Inheritance can always be replaced with object composition as a mechanism for code reuse.**

Parameterized Types

- **Parameterized types, or generics in C#, reuse functionality, too.**
- **Types can be defined without specifying all of the types they use.**
- **Unspecified types are supplied as parameters at the point of use.**

Aggregation versus Acquaintance

- Aggregation
  - One object owns and is responsible for another object.
  - Aggregate object and its owner may have identical lifetimes.
  - Relationships between objects are fewer and longer-lasting.
- Acquaintance
  - Implies an object merely knows of another object; it is not responsible for it.
  - Sometimes called association or "using" relationship.
  - Relationships made more frequently, existing for only the duration of an operation, i.e., they are dynamic.

More on Aggregation and Acquaintance

- **Aggregation and Acquaintance are implemented in the same way, i.e., with references.**
- **Relationships determined more by intent than explicit language mechanisms.**
- **A plain arrowhead denotes acquaintance; an arrowhead with a diamond at its base denotes aggregation.**

Dependency Injection (DIP)

- Software pattern allowing choice of component to be made at run-time rather than compile-time.
- Objects can be injected into a class, rather than allowing the class to create the objects, themselves.
- Different implementations of single component can be created at run-time and passed to the same code.
- Sometimes called Inversion of Control.

Advantages of Dependency Injection

- Simple way to load plug-ins dynamically
- Facilitates the writing of testable code
- Make decisions at run-time rather than compile time

Disadvantages of Dependency Injection

- **Leaks internal implementation details of a class, possibly violating encapsulation**
- **Prevents deferred creation because dependencies must be created before they're needed**

Benefits of DIP

- Reduces boilerplate code in application objects
- Work to initialize objects is handled by provider
- Application and configuration flexibility
- Useful for testing mock objects in test environments vs. real objects in production environment
- Real-world example is stock market application

Three Elements of Dependency Injection

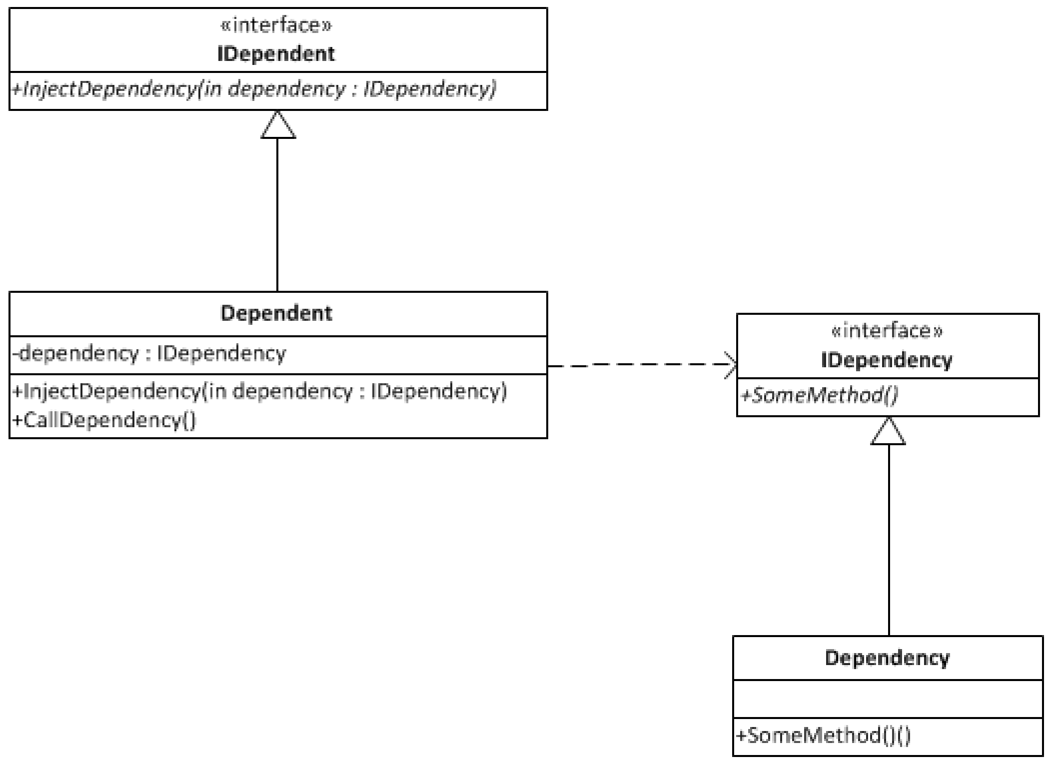
- **Dependent consumer**
- **Declaration of dependencies (interface)**
- **Injector (provider or container)**

Martin Fowler on DIP

- **Three ways an object can get reference to external object**
  - **Type 1: interface injection-exported module provides an interface users must implement to get dependencies at runtime.**
  - **Type 2: setting injection-the dependent module exposes a setter method used to inject dependency.**
  - **Type 3: constructor injection-dependent object passed through parameterized class constructor.**

Type 1 Injection: Interface

Implementing Interface Injection

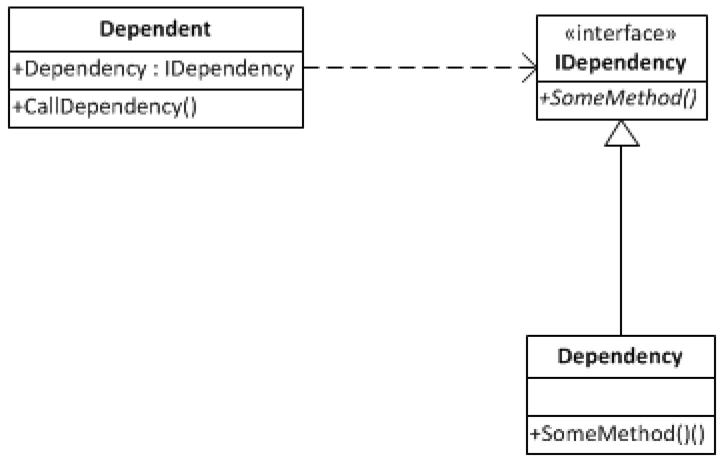


Type 1 Injection Sample Code

- See sample code.

Type 2 Injection: Setter

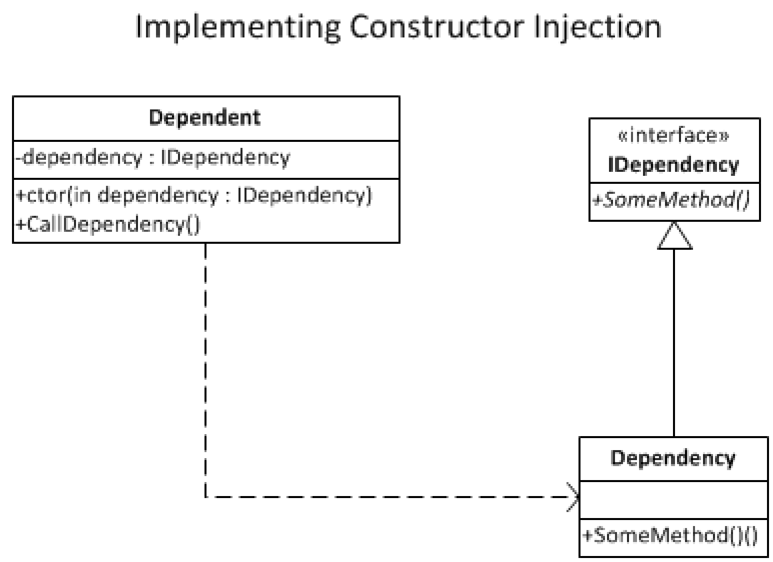
Implementing Setter Injection



Type 2 Injection Sample Code

- See sample code.

Type 3 Injection: Constructor



Type 3 Injection Sample Code

- See sample code.

Other Types of Dependency Injection

- Delegation
- Configuration file
- XML file

Designing for Change

- Changes to existing requirements and new requirements need to be expected and planned for.
- The system should be robust for changes.
- Design patterns ensure that system can change by allowing aspects of the system to vary independently of other aspects.

Common Causes of Redesign

- **Creating an object by specifying a class explicitly**
- **Dependence on specific operations**
- **Dependence on hardware and software platform**
- **Dependence on object representations or implementations**

More Causes of Redesign

- Algorithmic dependencies
- Tight coupling
- Extending functionality by subclassing
- Inability to alter classes conveniently

Tight Coupling versus Loose Coupling

- **Classes tightly coupled are hard to reuse; they have dependencies on other classes, so cannot be changed easily.**
- **Loose coupling promotes reuse and allows the system to be modified or extended more easily.**
- **Design Patterns encourage the development of loosely coupled systems.**

Subclassing versus Composition

- Customizing an object by subclassing requires in-depth understanding of the parent class.
- Subclassing can lead to an explosion of child classes.
- Object composition and delegation provide flexible alternatives to inheritance.
- New functionality can be added to application by composing existing objects in new ways.

Frameworks Defined

- Frameworks define a set of cooperating classes that make up a reusable design for a specific class of software, in the process, dictating the architecture of the application.
- Frameworks define the overall structure, partition classes and objects, and allocate key responsibilities and collaborations.
- Frameworks allow application designer to focus on application, instead.

Contrasting Design Patterns and Frameworks

- **Design patterns are more abstract than frameworks.**
- **Design patterns are smaller architectural elements than frameworks.**
- **Design patterns are less specialized than frameworks.**

Selecting a Design Pattern

- Consider how design patterns solve design problems.
- Revisit your intent.
- Study how patterns interrelate to each other.
- Study patterns of a like purpose.
- Examine the cause of redesign.
- Consider what should be variable in the design.

How to Use a Design Pattern

- **Read the pattern once through for an overview.**
- **Go back and study the structure, participants, and collaborations.**
- **Look for a concrete example of the pattern in code.**
- **Choose names for pattern participants that are meaningful in the application context.**

How to Use a Design Pattern (cont'd)

- **Define the classes.**
- **Define application-specific names for operations in the pattern.**
- **Implement the operations to carry out the responsibilities and collaborations in the pattern.**
- **Design patterns should only be applied when the flexibility they afford is needed.**

Summary

- **What are Design Patterns?**
- **What are the benefits and costs of design patterns?**
- **What are the types of design patterns?**