



# How Design Patterns Solve Design Problems

Designed by Rita Ganatra

# How Design Patterns Solve Design Problems

## 1. Finding Appropriate Objects

Design patterns help you identify the objects that can capture them.


## 2. Determining Object Granularity(state)

**Facade** pattern describes how to represent **complete subsystems as objects**.

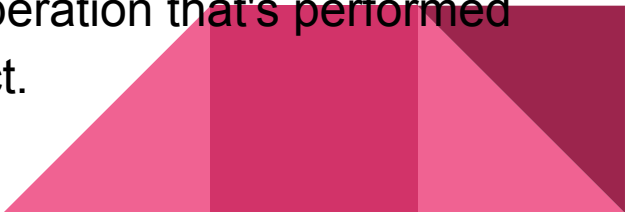
**Flyweight** pattern describes how to **support huge numbers of objects at the finest granularities**.

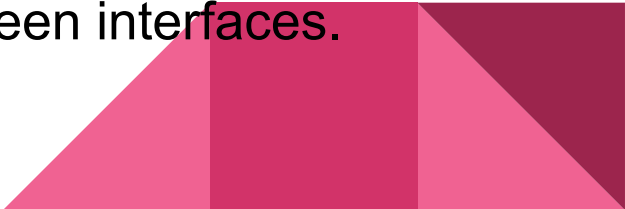
**Abstract Factory and Builder** yield **objects whose only responsibilities are creating other objects**.

**Visitor and Command** yield objects whose only responsibilities are **to implement a request on another object or group of objects**.



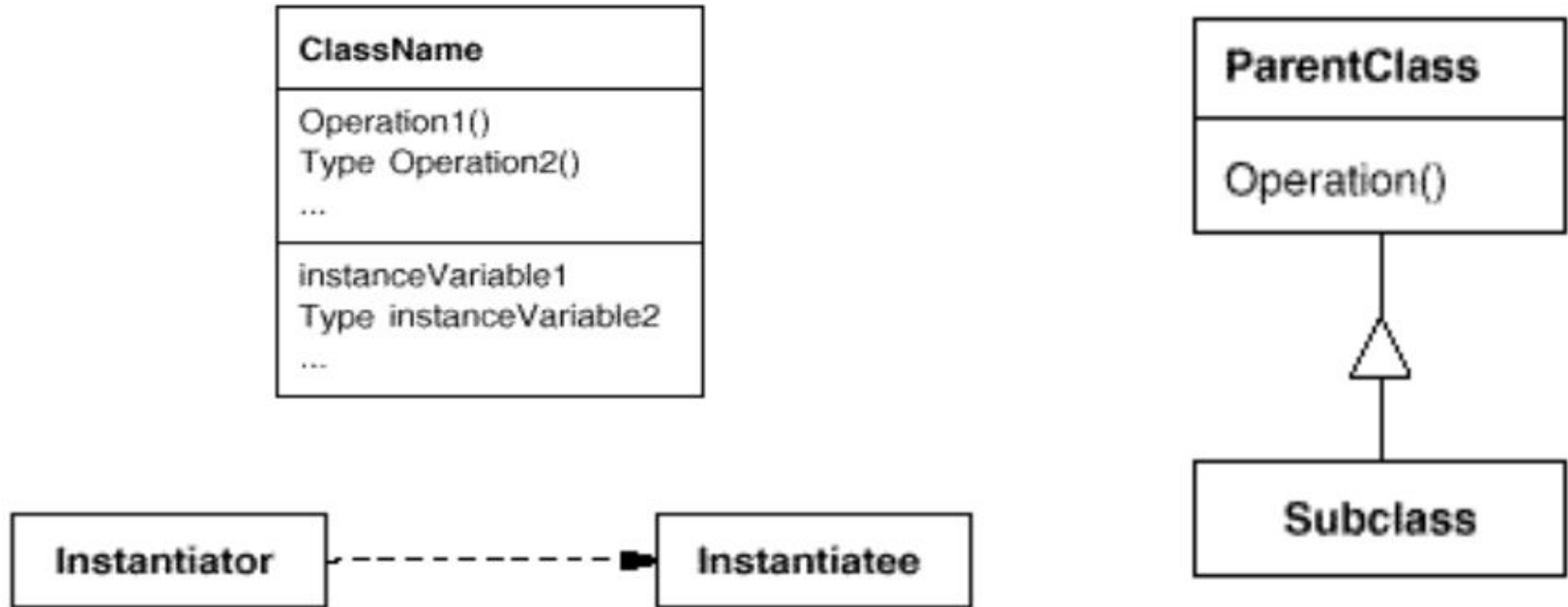
### 3. Specifying Object Interfaces

- An interface is a collection of method signatures without any implementation.
  - A **type** is a name used to denote a particular interface.(For instance, if a class implements an interface, it is said to be of that interface type.)
  - An object may have many types, and widely different objects can share a type.
  - A type is considered a subtype of another if it inherits or extends the interface of its supertype.
  - Objects are known only through their interfaces. An object's interface says nothing about its implementation
  - When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object.
- 

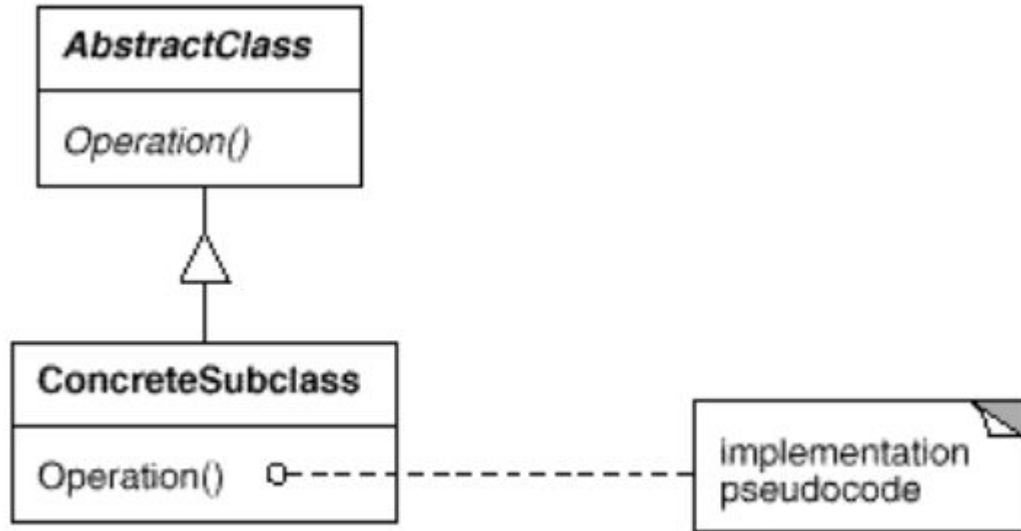
- **Dynamic binding:** the run-time association of a request to an object with one of the appropriate method of that object, based on its actual type.
  - **Polymorphism:** dynamic binding can substitute objects that have identical interfaces for each other at run-time
  - Dynamic binding is closely associated with polymorphism, where a base class reference can point to an object of a derived class, and the method that gets executed depends on the actual object type.
  - Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.
  - Design patterns also specify relationships between interfaces.
- 

## 4. Specifying Object Implementations

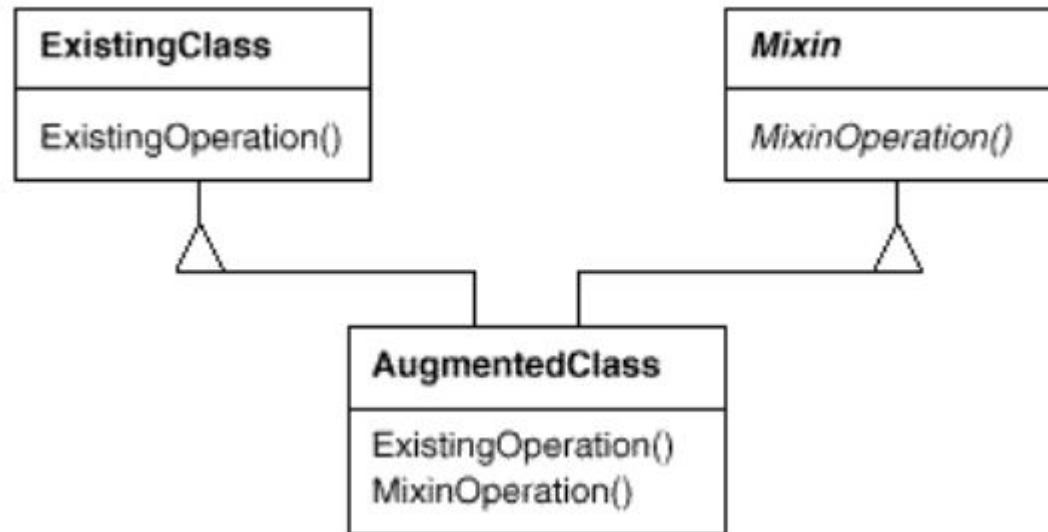
- An object's implementation is defined by its **class**.



- An **abstract class** is one whose main purpose is to define a **common interface for its subclasses**.



- A **mixin class** is a class that's intended to provide an **optional interface or functionality to other classes.**
- A mixin class is designed to offer additional methods or properties that can be used by other classes, thereby providing optional functionality or an extended interface without the need for a class hierarchy or inheritance. Mixins are often used to add reusable features to multiple classes without duplicating code.



# Class Inheritance versus Interface Inheritance

Feature	Class Inheritance	Interface Inheritance
Purpose	Share code between classes	Define common methods for different classes
Flexibility	Inherit from one class (less flexible)	Implement many interfaces (more flexible)
Implementation	Inherits both methods and their code	Only defines methods; classes provide the code
Coupling	Strong link between parent and child classes	Loose link; classes just need to follow the rules
Polymorphism	Limited due to single inheritance	High; can treat objects as instances of their interface
Use Cases	When classes are closely related	When classes need to share abilities



- **First Principle of reusable object-oriented design: Programming to an Interface, not an Implementation**
- Class inheritance-based implementation reuse is only half the story. Inheritance ability to define families of objects with **identical interfaces** is also important, because polymorphism depends on it.
- Two benefits to manipulating objects solely in terms of the interface defined by abstract classes:
  - 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.




# Putting Reuse Mechanisms to Work


- **Inheritance versus Composition**
  - **White-box reuse**: class inheritance.
  - **Black-box reuse**: object composition


## Class inheritance:Advantages

- supported by programming languages, defined statically at compile-time and is straightforward to use
- make it easier to modify the implementation being reused, when a subclass overrides some but not all operations.

## Class inheritance:Disadvantages

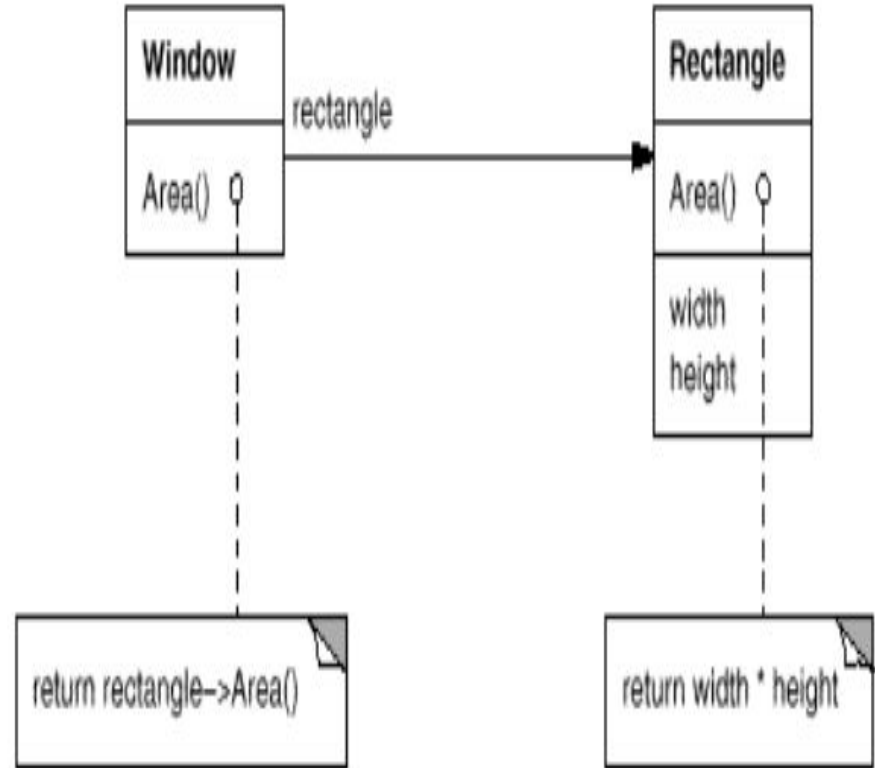
- Cannot change the implementations/representations inherited from parent classes at run-time
  - Implementation dependency between a subclass and its parent class.
- 

- Object composition Advantages
    - Defined dynamically at run-time by referencing interfaces of objects.
    - Access other objects through their interfaces only, not break encapsulation.
    - Fewer implementation dependencies.
    - Small class hierarchies
  - Object composition Disadvantages
    - More objects
    - The system's behavior will depend on their interrelationships instead of being defined in one class
  - The second principle of object-oriented design:  
***Favor object composition over class inheritance.***
- 

Feature	White-Box Reuse (Class Inheritance)	Black-Box Reuse (Object Composition)
Purpose	Reuse code by extending a class	Reuse code by combining objects
Understanding	Requires understanding the internal code	No need to understand the internal code
Coupling	Tightly coupled to the base class	Loosely coupled; uses well-defined interfaces
Flexibility	Limited by single inheritance (in some languages)	Flexible; can combine various objects easily
Code Reusability	Reuses implementation details of the superclass	Reuses functionality by composing objects
Maintenance	Changes in superclass affect all subclasses	Changes in objects don't affect others
Complexity	Can become complex with deep inheritance hierarchies	Can add complexity with many interacting objects
Performance	Generally more direct (less overhead)	May have performance overhead due to object interactions
Design	Defines a clear hierarchy 	Defines a modular structure

# Delegation

- In delegation, an object handles a request by passing it to another object (the delegate). This allows the delegating object to focus on its core responsibilities while relying on the delegate to perform specific tasks.
- In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.
- The receiver passes itself to the delegate to let the delegated operation refer to the receiver.



- Advantage: it makes it easy to compose behaviors at run-time and to change the way they're composed.
- Disadvantage: harder to understand than more static software, and run-time inefficiencies.
- Delegation works best when it's used in standard patterns.
- Design patterns that use delegation: State, Strategy, Visitor, Mediator, Chain of Responsibility, and Bridge patterns.



## Relating Run-Time and Compile-Time Structures

- An object-oriented program's runtime structure often bears little resemblance to its code structure.
- **Aggregation**
  - Manifested at run-times.
  - One object owns (having) or is responsible for another object (being part).



- **Acquaintance**
  - Manifested at compile-times.
  - An object merely *knows of* another object (association, using).
  - A weaker relationship than aggregation.
- In implementation or code, aggregation and acquaintance cannot be distinct.
- Many design patterns capture the distinction between compile-time and run-time structures explicitly..
- The run-time structures aren't clear from the code until you understand the patterns.

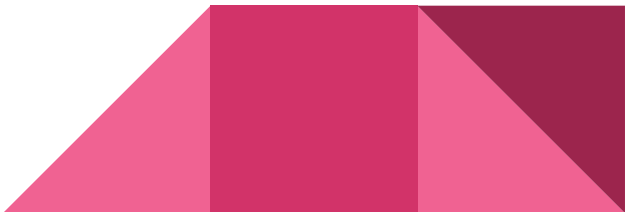




Aspect	Compile-Time Structures	Run-Time Structures
Definition	Static aspects known before the program runs	Dynamic aspects created while the program runs
Class Definitions	Defines class blueprints, attributes, and methods	Creates objects based on these class definitions
Method Signatures	Specifies method names, return types, and parameters	Executes methods and determines actual behavior
Type Checking	Ensures type compatibility before running code	Type issues are handled at compile time, not run time
Inheritance	Defines class hierarchies and relationships	Uses inheritance to determine object behaviors at runtime
Interfaces	Specifies method requirements for implementing classes	Classes implement these interfaces to provide actual functionality
Object Instances	Not created at compile time	Actual objects are created and interact during execution
Method Calls	Checked for validity before execution	Methods are invoked and executed on objects during runtime
Dynamic Binding	Not applicable (resolved at compile time)	Determines method implementation to call based on actual object type
Memory Management	Not directly managed at compile time ↓	Manages allocation and deallocation of memory for objects

# Designing for Change

Common causes of redesign along with the design pattern(s) that address them:

- **Creating an object by specifying a class explicitly-** Design patterns: Abstract Factory, Factory Method, Prototype.
  - **Dependence on specific operations-** Design patterns: Chain of Responsibility, Command
  - **Dependence on hardware and software platform -** Design patterns: Abstract Factory, Bridge
  - **Dependence on object representations or implementations -** Design patterns: Abstract Factory, Bridge, Memento, Proxy
- 

- **Algorithmic dependencies.** Design patterns: Builder, Iterator , Strategy, Template Method , Visitor
- **Tight coupling.** Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer-
- **Extending functionality by subclassing.**Design patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- **Inability to alter classes conveniently-**Design patterns: Adapter, Decorator, Visitor.



# Role of Design Patterns

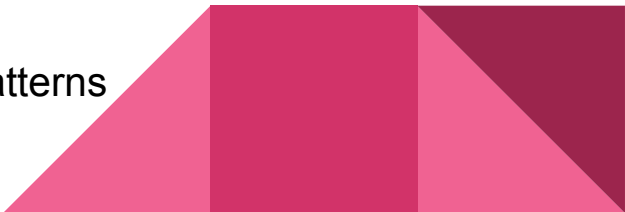
- The role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.
- - **Application Programs**
    - Internal reuse, maintainability, and extension are high priorities.
    - Design patterns that reduce dependencies can increase internal reuse.
    - Design patterns make an application more maintainable when they're used to limit platform dependencies and to layer a system.
    - Design patterns enhance extensibility.



- **Toolkits (class/component libraries)**

- Code reuse
- Application-general design

- **Frameworks**

- A framework is a set of cooperating classes that make up a reusable design for a specific class of software
  - The framework dictates the architecture of your application.
  - Frameworks emphasize design reuse over code reuse.
  - Frameworks are implemented as class hierarchies.
  - Reuse on framework level leads to an inversion of control between the application and the software on which it's based.
  - Mature frameworks usually incorporate several design patterns
- 

Aspect	Design Patterns	Frameworks
Definition	General solutions to common design problems	Predefined structures and libraries for building applications
Purpose	Provide reusable solutions to specific design issues	Provide a foundation with tools and libraries for development
Scope	Focused on specific design problems and solutions	Broad and comprehensive, covering multiple aspects of application development
Implementation	Describes how to solve a problem but doesn't provide implementation	Includes code and infrastructure to be used directly in applications
Flexibility	Highly flexible; can be adapted to various contexts	Less flexible; often imposes certain conventions and structures
Use	Used to guide the design and architecture of software	Used to build applications by leveraging the provided structure and tools
Example	Singleton, Observer, Factory Method	Django (web framework), Spring (Java framework)
Code Reusability	Promotes reusability of design ideas and approaches	Provides reusable components and tools for development
Learning Curve	Generally has a low learning curve; requires understanding design concepts	Often has a steeper learning curve due to its comprehensive nature and conventions
Integration	Can be used independently in various systems	Often dictates how different parts of the system integrate and interact

# Design Patterns vs. Frameworks

1. Design patterns are more abstract than frameworks.
2. Design patterns are smaller architectural elements than frameworks.
3. Design patterns are less specialized than frameworks.



# Thank You

