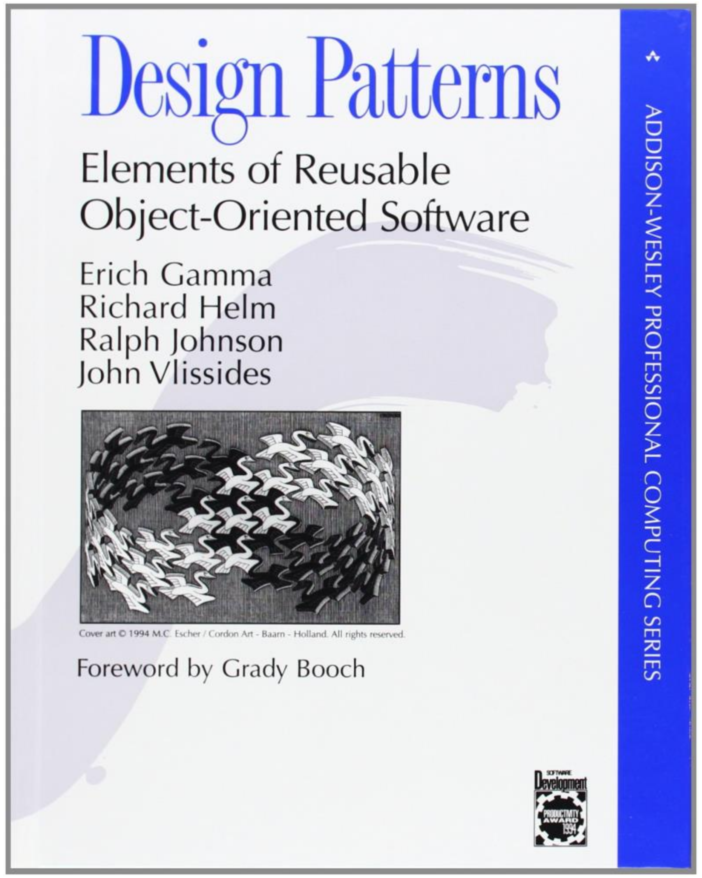
CSE 4513

Lec – 9

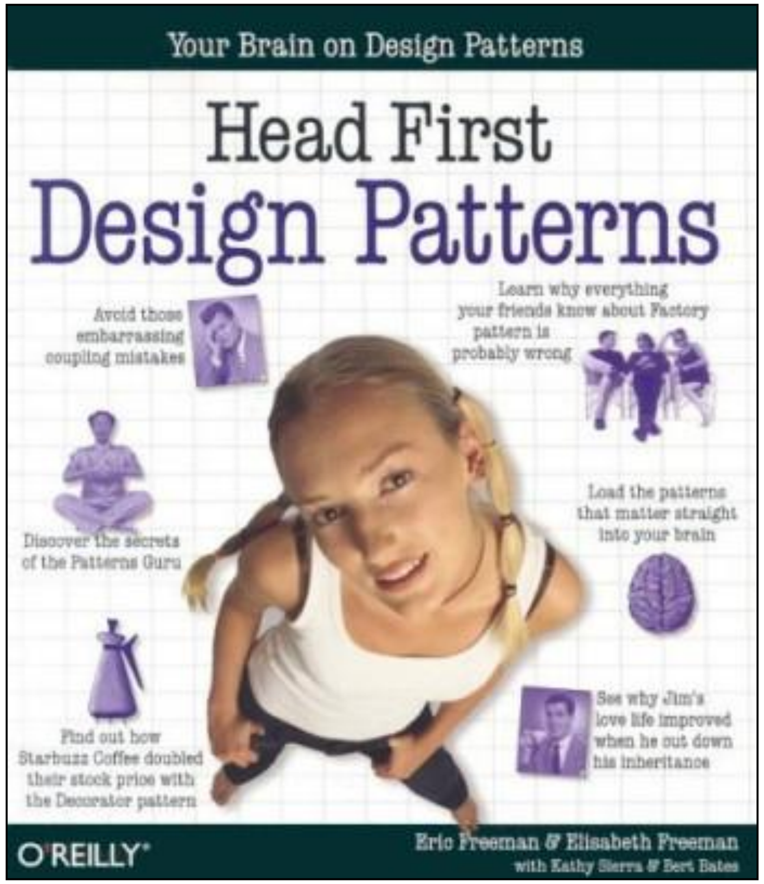
Design Pattern

Someone has already solved your problems

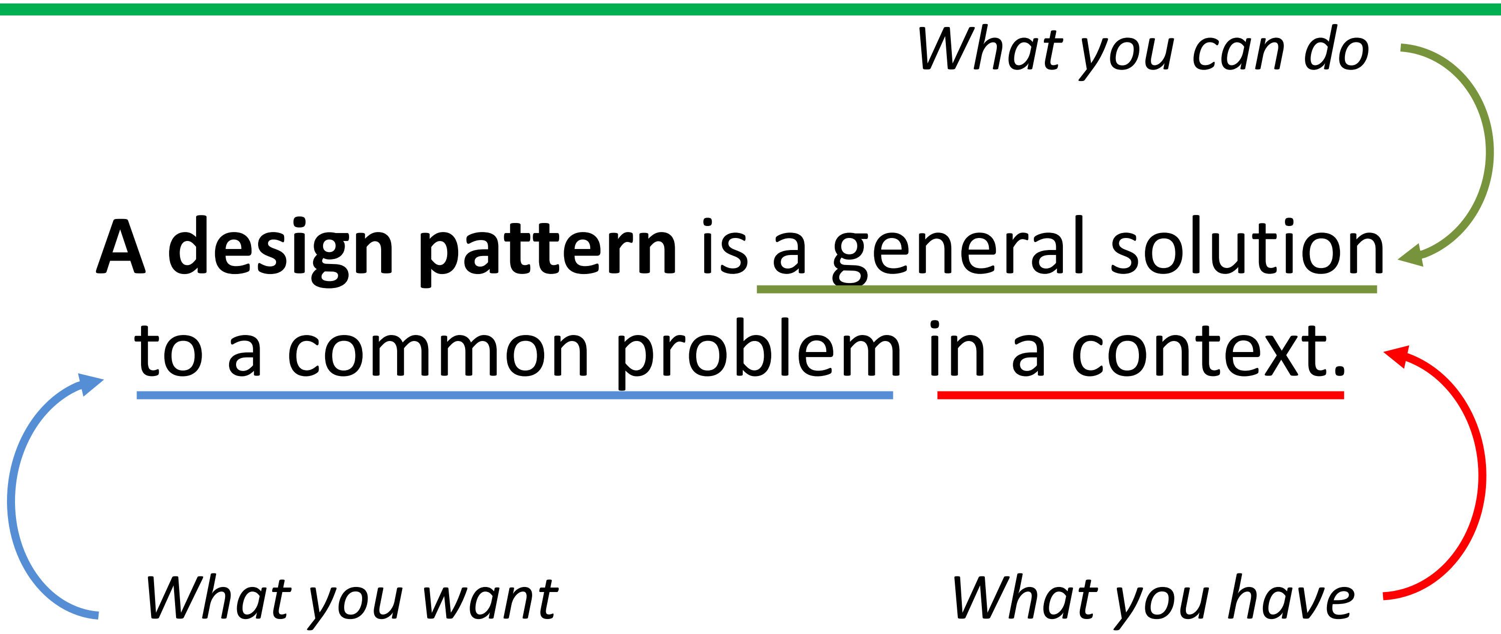
USEFUL BOOKS


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The "Gang of Four" book
1994


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

Head First Design Patterns Book
2004



A design pattern is a general solution to a common problem in a context.

Each pattern Describes a problem which occurs over and over again in our environment ,and then describes the core of the problem

“a design pattern is a general repeatable solution to a commonly occurring problem in software design”

Usage of Design Pattern

Design Patterns have two main usages in software development:

- ✓ **Common platform for developers**
  - Provide a standard terminology and are specific to particular scenario.

- ✓ **Best Practices**
  - Evolved over a long period of time
  - Provide **best solutions** to certain problems faced during software development.
  - helps inexperienced developers to learn software design in an easy and faster way.

Elements of a Pattern

- ✓ four elements to describe a pattern
    - The name of the pattern
    - The purpose of the pattern: what problem it solves
    - How to solve the problem
    - The constraints we have to consider in our solution

TYPES OF DESIGN PATTERNS

There are about 26 Patterns currently discovered …
These 26 can be classified into 3 types:

1. **Creational:** These patterns are designed for class instantiation. They can be either **class-creation** patterns or object-creational patterns.

2. **Structural:** These patterns are designed with regard to a **class's structure** and composition. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition.

3. **Behavioral:** These patterns are designed depending on **how one class communicates with others.**

Design Pattern Index

Creational	Structural	Behavioural
Factory Method	Adapter	Template
Abstract Factory	Bridge	Strategy
Builder	Composite	Command
Singleton	Decorator	State
Multiton	Facade	Visitor
Object pool	Flyweight	Chain of Responsibility
Prototype	Front controller	Interpreter
	Proxy	Observer
		Iterator
		Mediator
		Memento

Warning

Always go with the simplest solution that does the job and **introduce patterns only where the need emerges.**

Overuse of design patterns can lead to code that is downright over-engineered.

Strategy Pattern

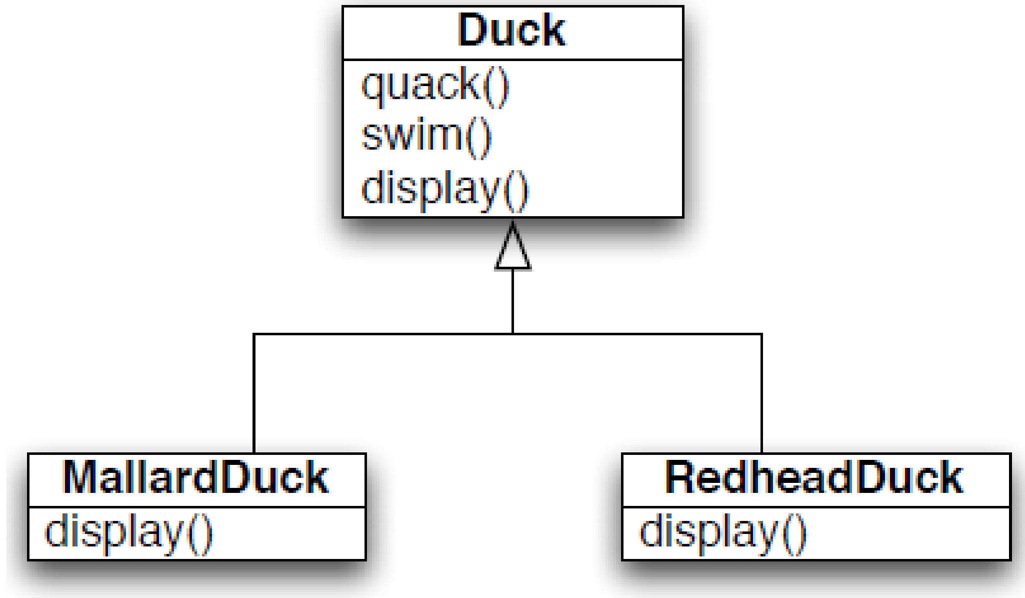
STRATEGY PATTERN

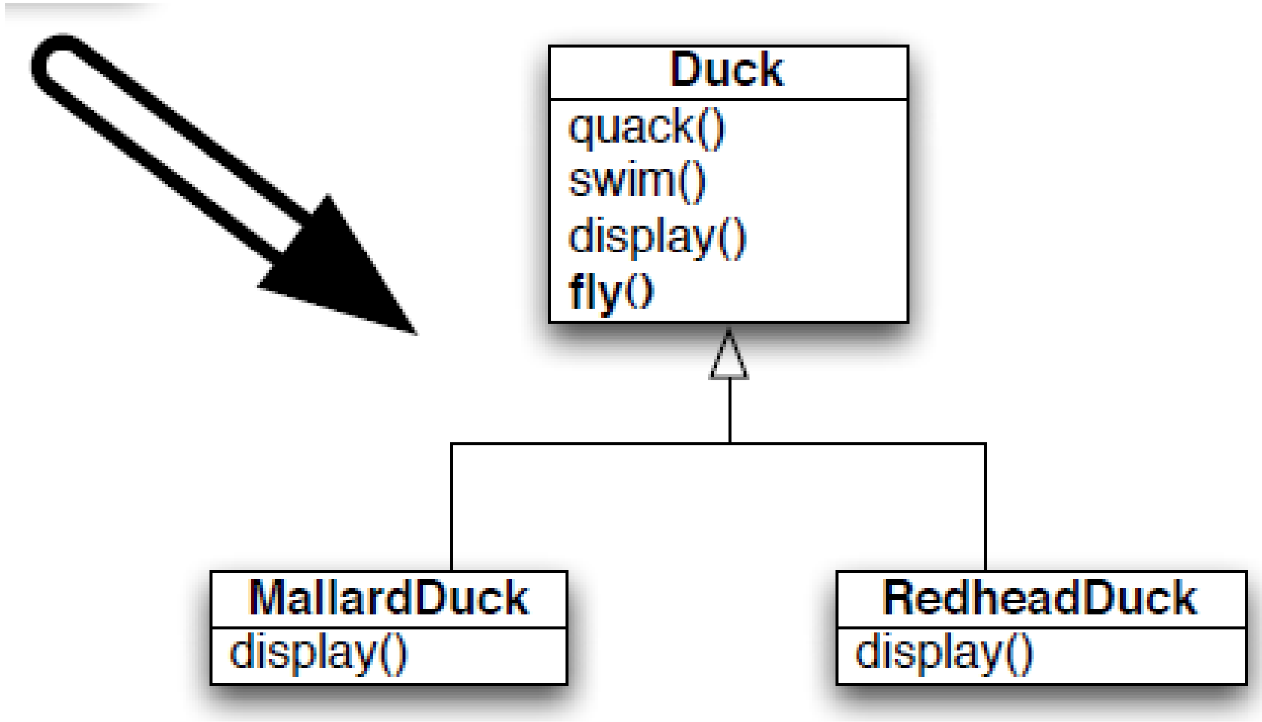
- ✓ it's probably the simplest pattern
- ✓ it's about using composition rather than inheritance
- ✓ it's about understanding that inheritance is not intended for code reuse

DEFINITION

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern

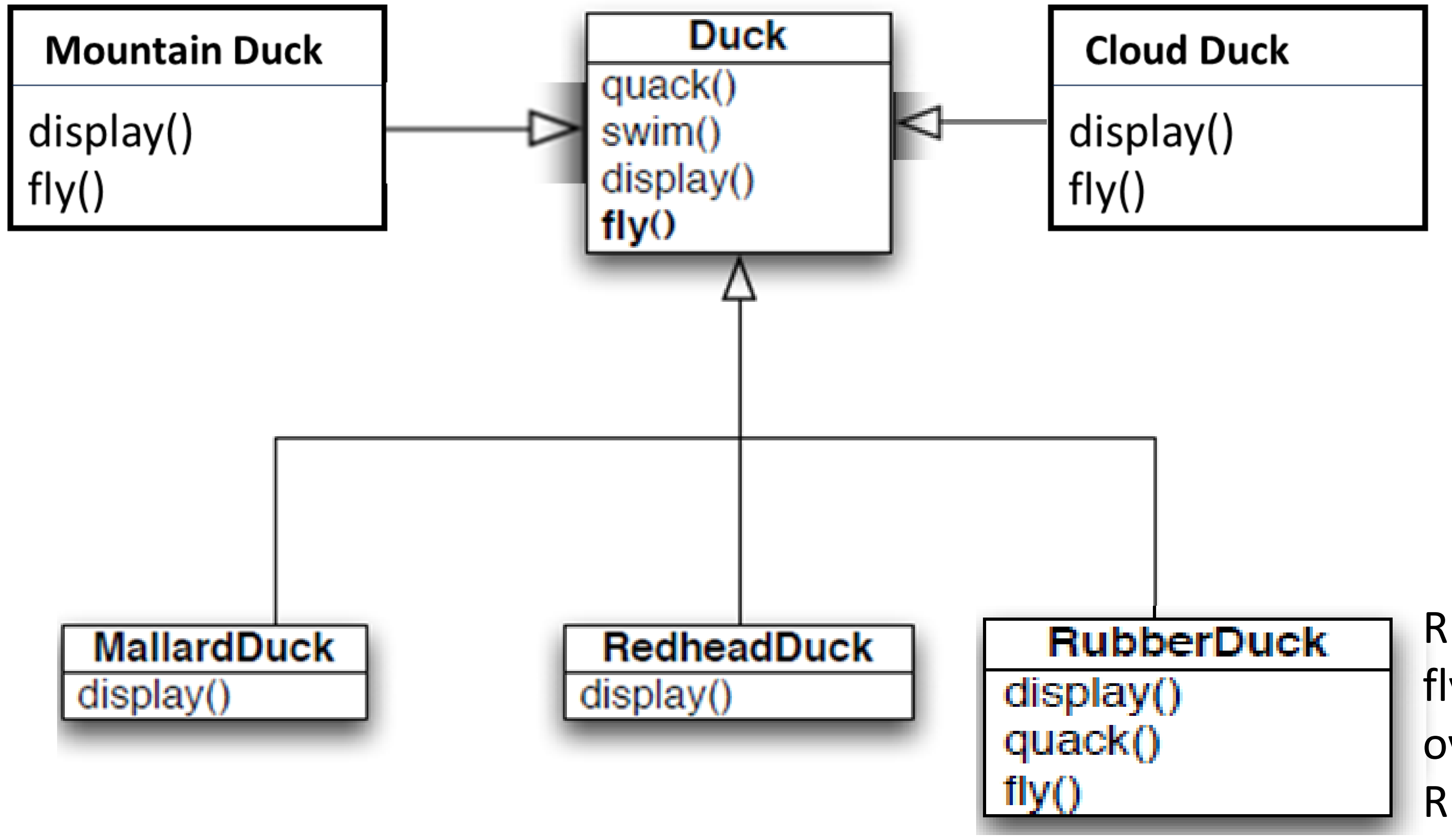




But a request has arrived to allow ducks to also fly.

STRATEGY PATTERN

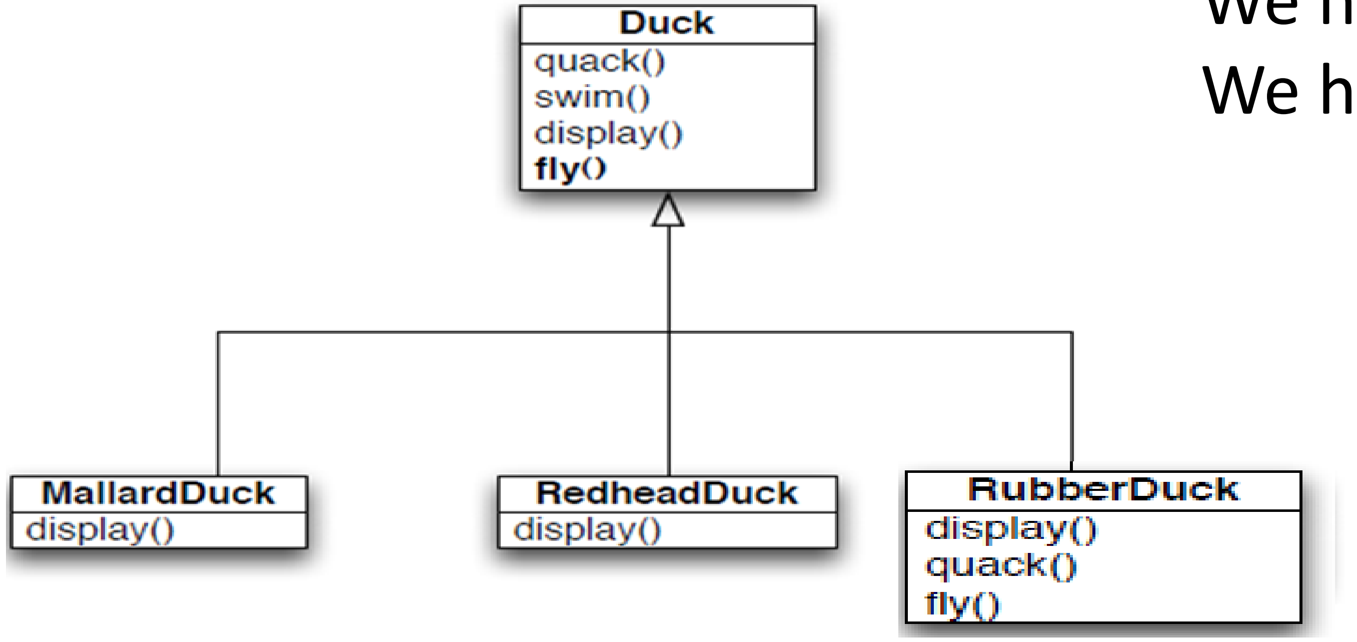
Now we have another duck for example rubber duck which is a fake duck..



Mountain and cloud duck has same flying behavior.

Rubber duck has different flying behavior . We could override fly() in RubberDuck to make it do Nothing..

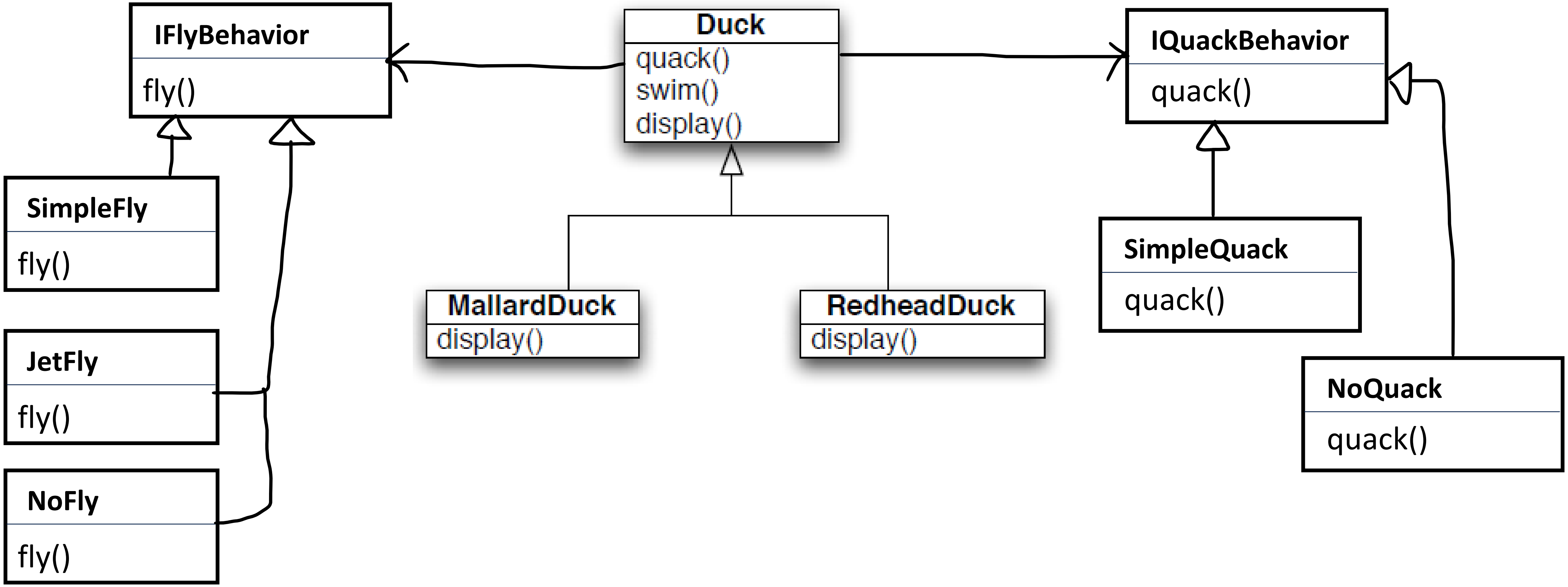
Strategy Pattern



We have algorithm for quacking
We have algorithm for flying…

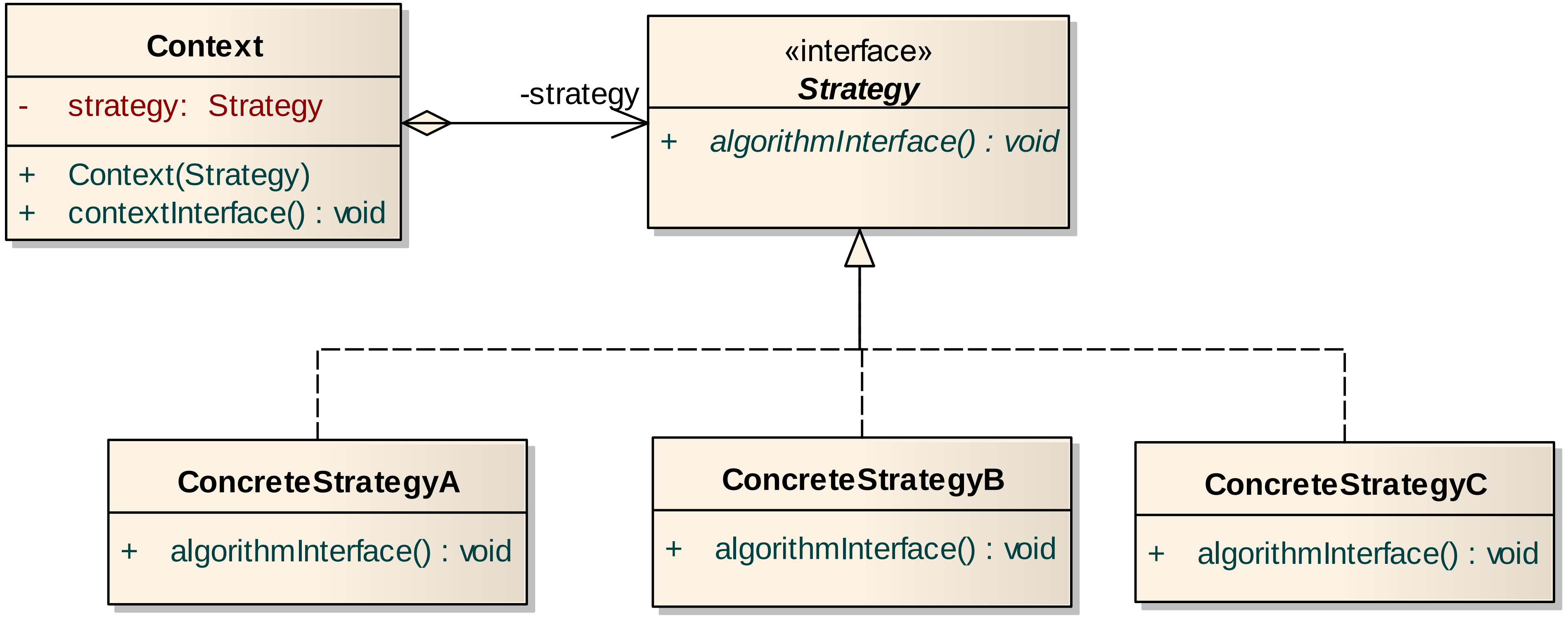
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern



FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck. Duck delegates to each set of behaviors and can switch among them dynamically, if needed.

Strategy Pattern


Context
- strategy: Strategy
+ Context(Strategy)
+ contextInterface() : void
-strategy
«interface»
Strategy
+ algorithmInterface() : void
ConcreteStrategyA
+ algorithmInterface() : void
ConcreteStrategyB
+ algorithmInterface() : void
ConcreteStrategyC
+ algorithmInterface() : void

Observer Pattern

OBSERVER PATTERN

observer: An object that "watches" the state of another object and takes action when the state changes in some way.

Problem: You have a model object with a complex state, and the state may change throughout the life of your program.
You want to update various other parts of the program when the object's state changes.

Solution: Make the complex model object observable.

observable object: An object that allows observers to examine it (notifies its observers when its state changes).
Permits customizable, extensible event-based behavior for data modeling and graphics.

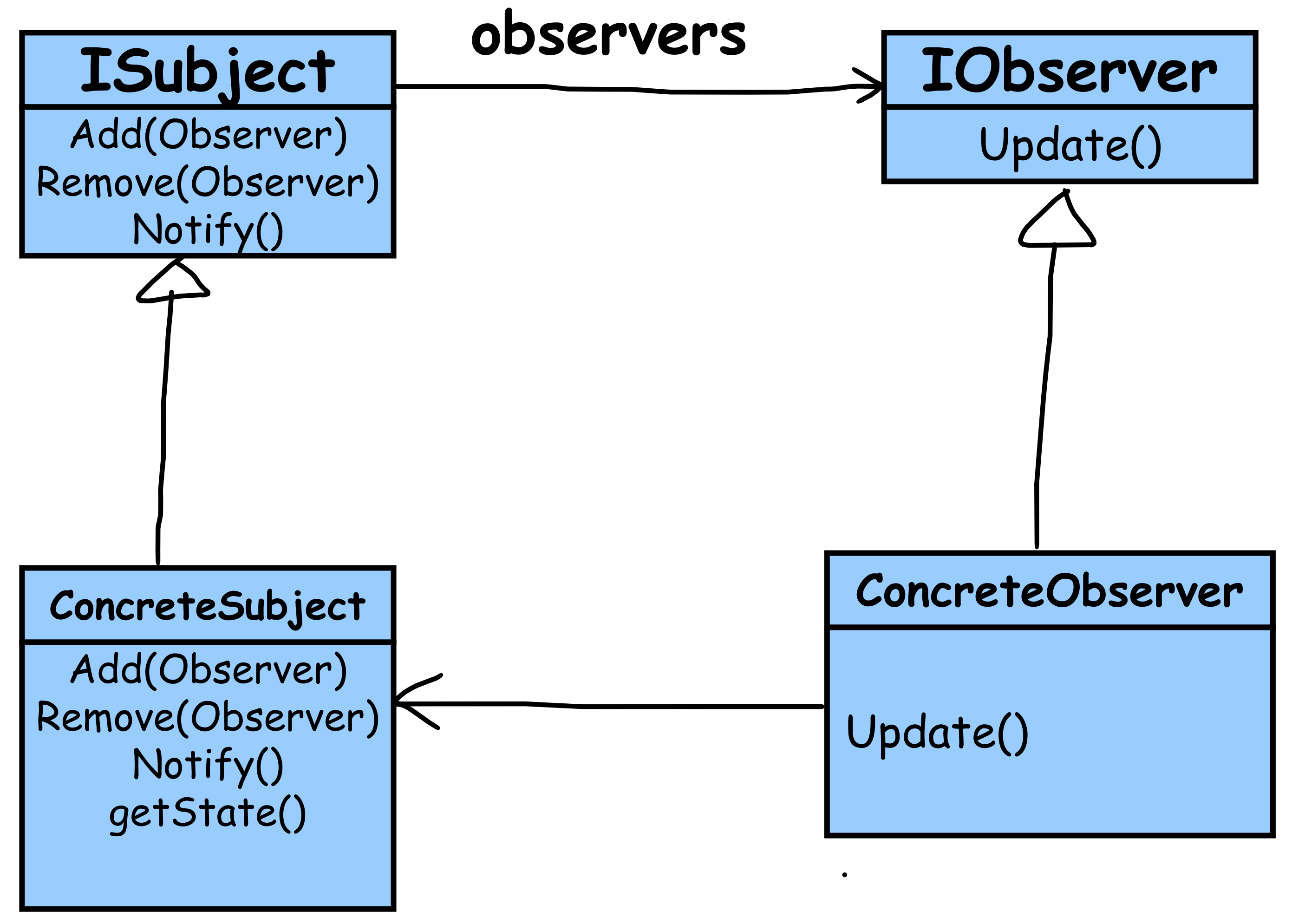
Observer Pattern

- Subject/observable
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that gets notified of changes in a subject
- ConcreteSubject
  - Stores “state of interest” to observers
  - Sends notification when state changes
- ConcreteObserver
  - Implements updating interface

OBSERVER PATTERN

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Observer Pattern



Decorator Pattern

DECORATOR PATTERN

- ✓ The decorator pattern is a pattern where we write wrapper code to let us extend the core code.

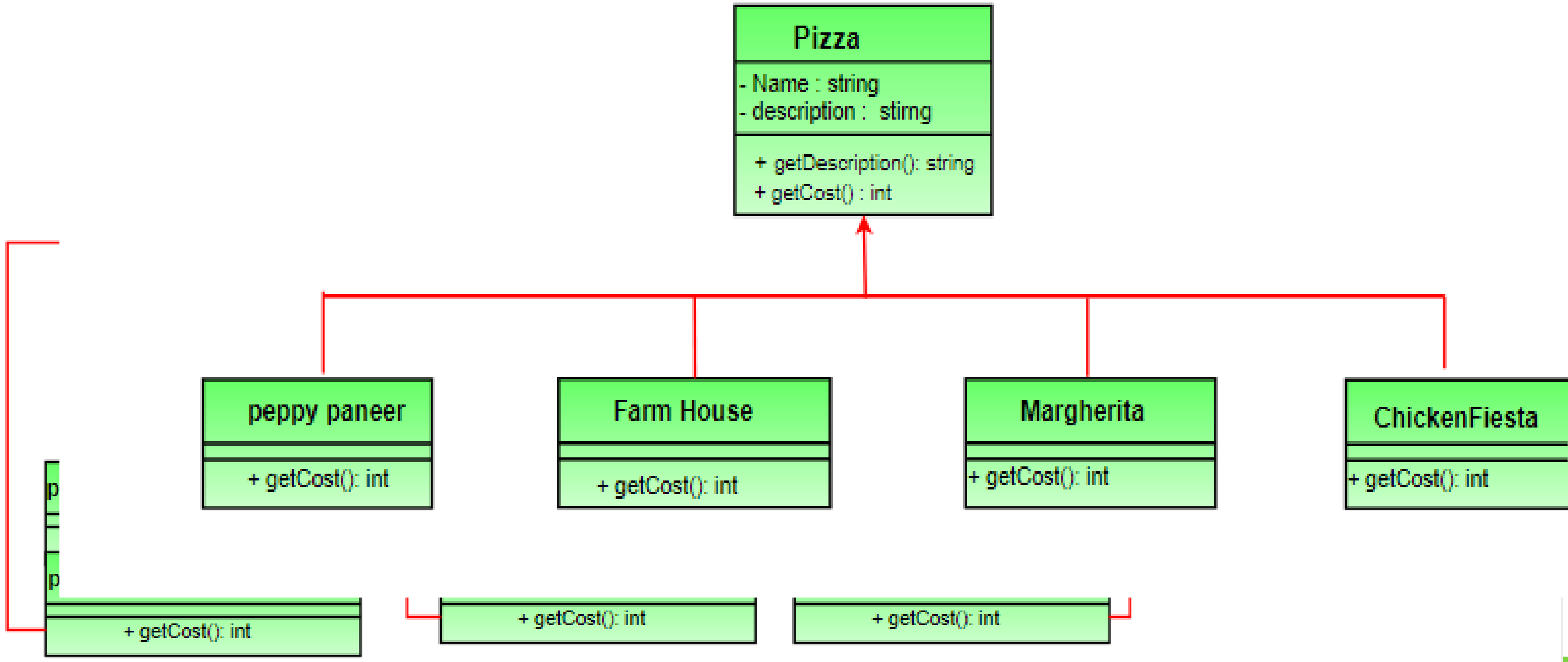
- ✓ We just keep wrapping objects around objects that we want to use so that we keep extending the capabilities of the existing objects by defining new objects that have the capabilities of the existing object.

Decorator Pattern

- ✓ how do we accommodate changes in the below classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

Option 1

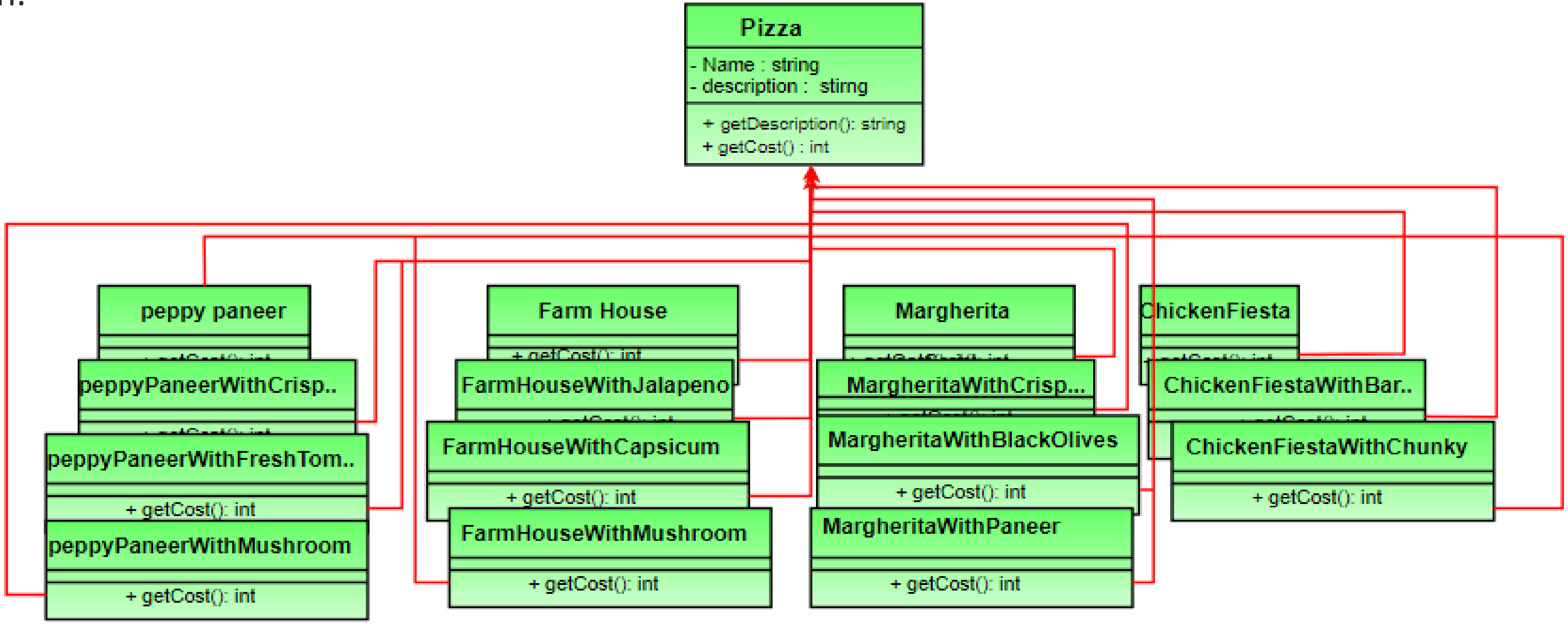
Create a new subclass for every topping with a pizza.



Decorator Pattern

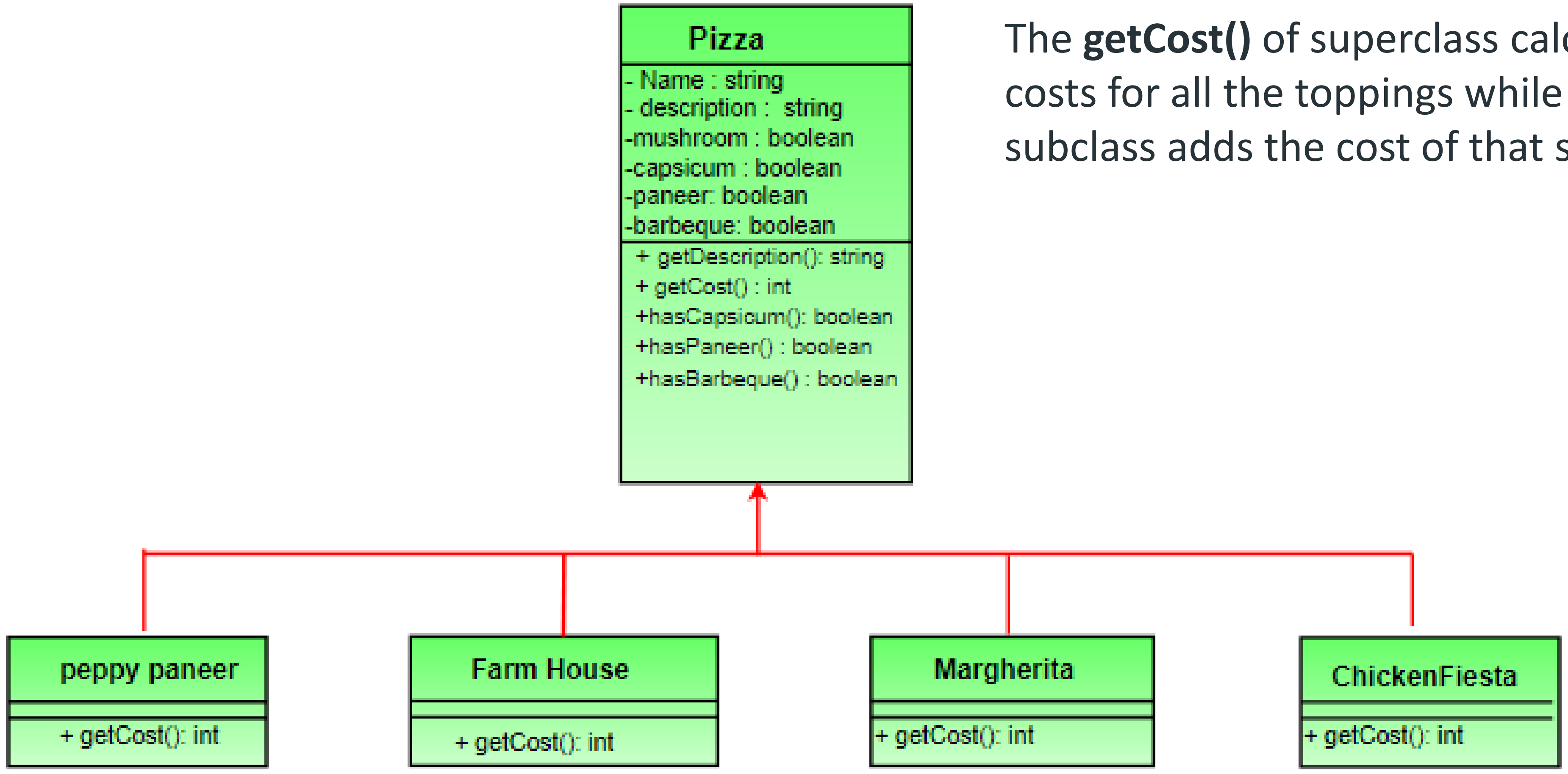
Option 1

- ✓ This looks very complex.
- ✓ There are way too many classes and is a maintenance nightmare.
- ✓ Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.



Decorator Pattern

Option 2:
Let's add instance variables to pizza base class to represent whether or not each pizza has a topping.



The **getCost()** of superclass calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

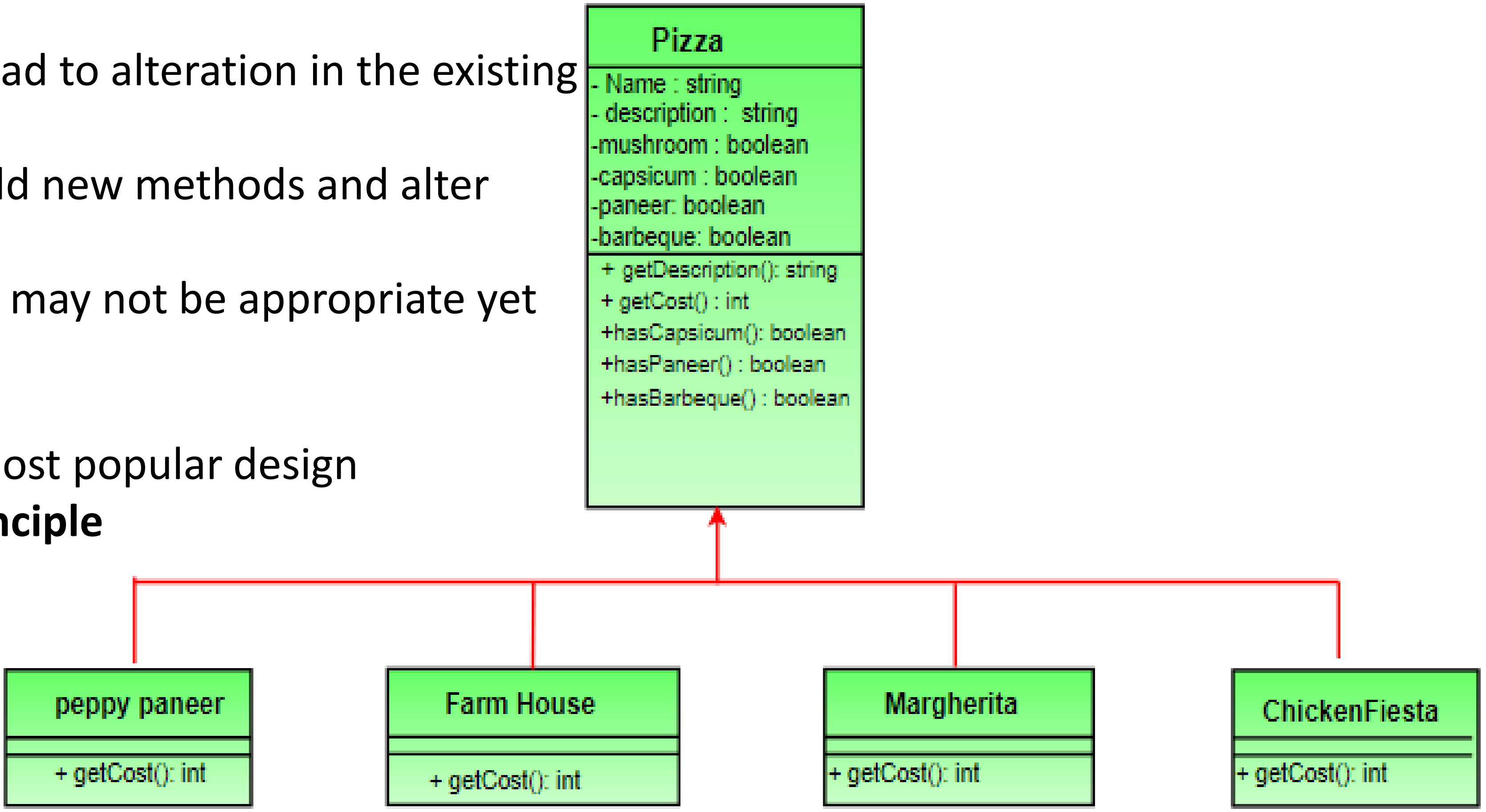
DECORATOR PATTERN

Option 2:

This design looks good at first but problems associated with it are:

- ✓ Price changes in toppings will lead to alteration in the existing code.
- ✓ New toppings will force us to add new methods and alter getCost() method in superclass.
- ✓ For some pizzas, some toppings may not be appropriate yet the subclass inherits them.

This design violates one of the most popular design principle – **The Open-Closed Principle**



Decorator Pattern

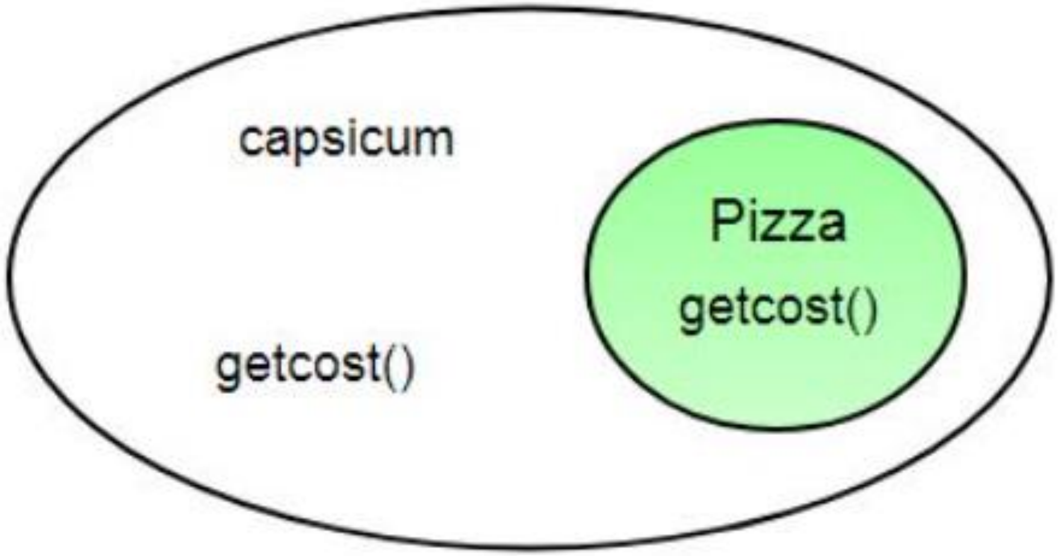
The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Pattern

The solution:

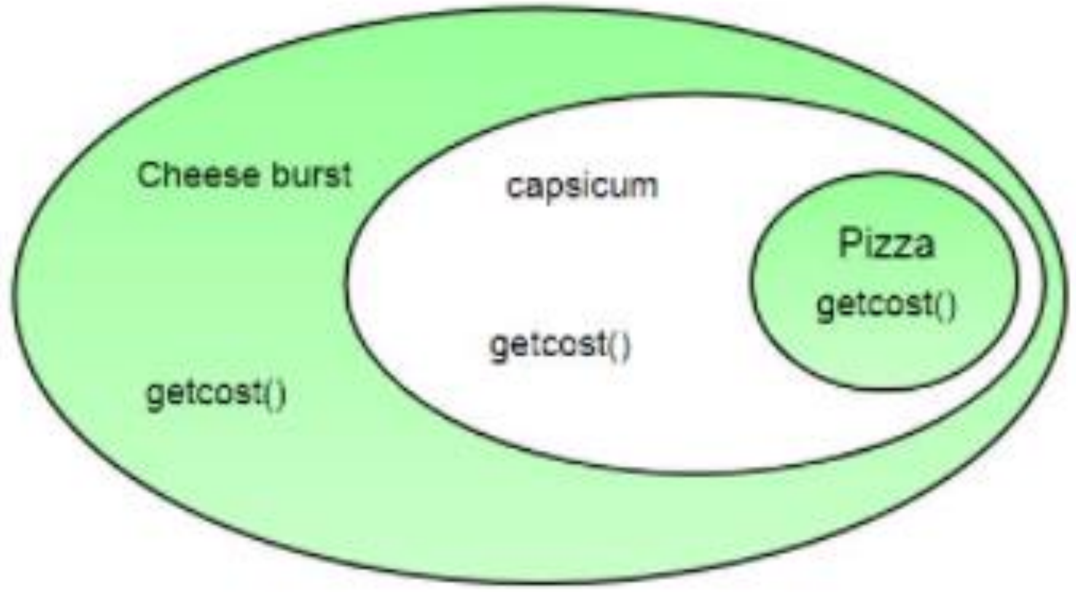
- ✓ Previous designs are not a good one
- ✓ Now, we will take a pizza and “decorate” it with toppings at runtime

1. Take a pizza object.
2. “Decorate” it with a Capsicum object.



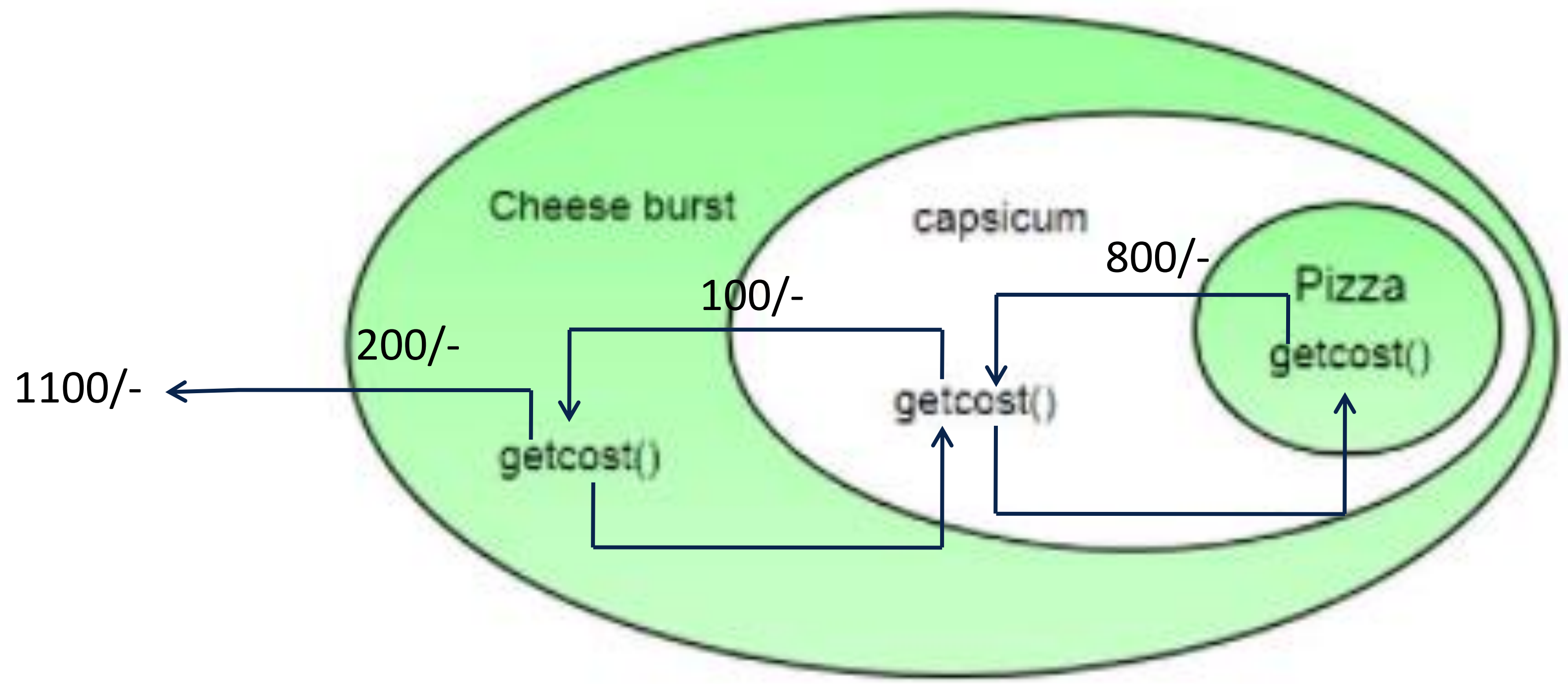
Decorator Pattern

3. “Decorate” it with a CheeseBurst object.



Decorator Pattern

4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.



✓Now we get a pizza with cheeseburst and capsicum toppings.
✓Visualize the “decorator” objects like wrappers.

DECORATOR PATTERN

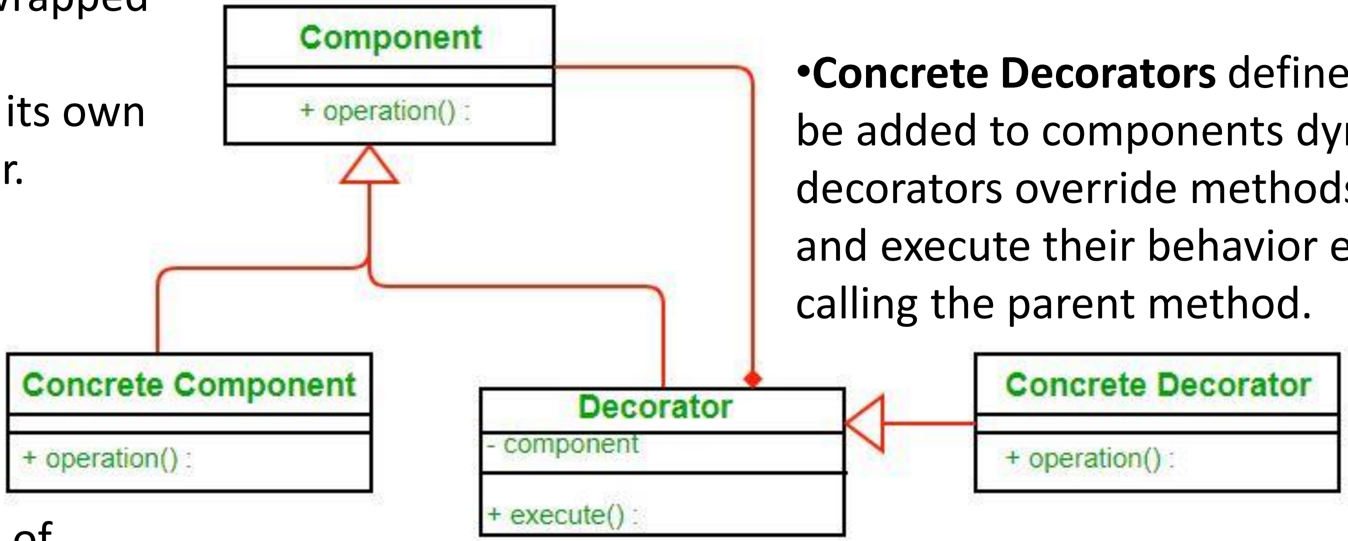
Here are some of the properties of decorators:

- ✓ Decorators have the same super type as the object they decorate.
- ✓ You can use multiple decorators to wrap an object.
- ✓ We can decorate objects at runtime.

DECORATOR PATTERN

Structure:

•The **Component** declares the common interface for both wrappers and wrapped objects.
•Each component can be used on its own or may be wrapped by a decorator.



•**Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

•**Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
•The ConcreteComponent is the object we are going to dynamically decorate.

•The **Decorator** class has a field for referencing a wrapped object The decorator delegates all operations to the wrapped object.
•Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).

DECORATOR PATTERN

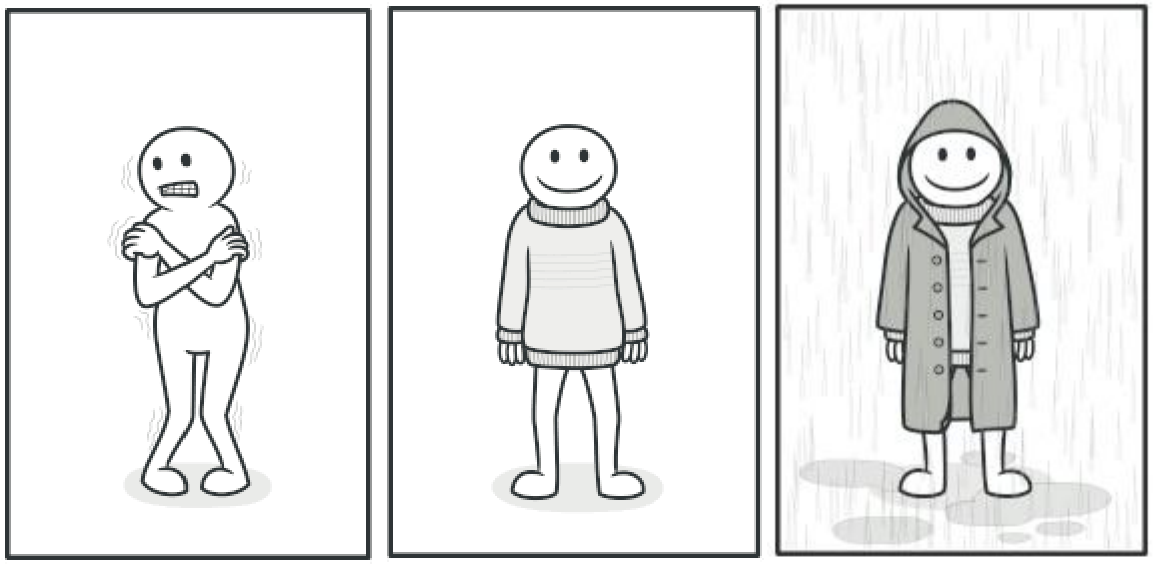
Advantages:

- ✓ The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.

- ✓ The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.

Decorator Pattern

Real-World Analogy

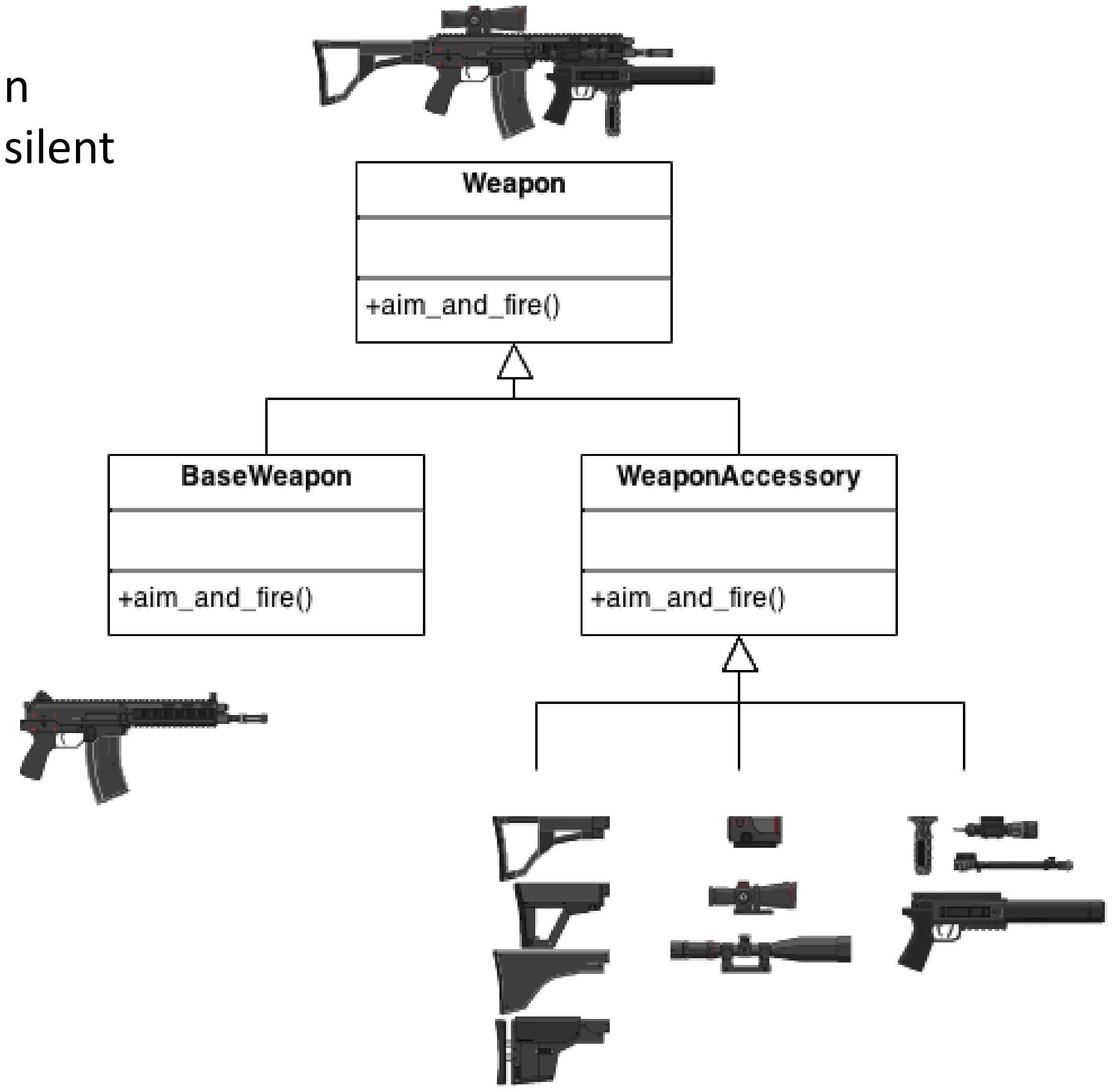


- ✓ Wearing clothes can be an example of using decorators.
- ✓ When you’re cold, you wrap yourself in a sweater.
- ✓ If you’re still cold with a sweater, you can wear a jacket on top.
- ✓ If it’s raining, you can put on a raincoat.
- ✓ All of these garments “extend” your basic behavior but aren’t part of you, and you can easily take off any piece of clothing whenever you don’t need it.

Decorator Pattern

More scenario:

- ✓ assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.



SEARCH THE ANSWER

What is the difference between decorator and strategy?
When should we use each of them?

Additional Quiz

Rules:

1. Each team will get one topic to present.
2. Each member should be ready to present . Any one of you can be asked to present the topic.
3. If you do well , whole team will get good marks. If you failed to perform your team will not get good marks. So, each of you should be well prepared.
4. Each presentation will be evaluated by me . (70% marks)
5. Each team will evaluate other teams (30% marks)

SEC 2

Factory	Singleton	Adapter	Bridge	Façade	Proxy
200041201	200041214	200041223	200041231	200041246	200041260
200041202	200041217	200041224	200041234	200041253	200041261
200041203	200041218	200041225	200041235	200041254	200041262
200041204	200041219	200041227	200041239	200041255	200041263
200041205	200041221	200041228	200041240	200041256	200041264
			200041242	200041257	190041250

SEC 1

Factory	Singleton	Adapter	Bridge	Façade	Proxy
200041101	200041114	200041123	200041132	200041143	200041156
200041105	200041116	200041125	200041134	200041144	200041157
200041106	200041119	200041126	200041138	200041147	200041158
200041108	200041120	200041127	200041139	200041150	200041161
200041109	200041121	200041130	200041141	200041153	200041162
200041113	200041122	200041131	200041142	200041155	200041163 200041164