Topic:
Design Patterns in Software Engineering

-Deeksha Agrawal

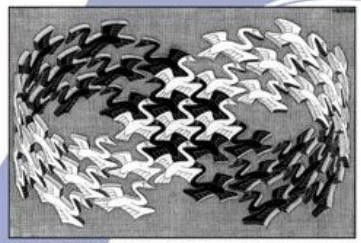
Design Patterns

What are Design Patterns?

Design Patterns

Elements of Reusable Object-Oriented Software

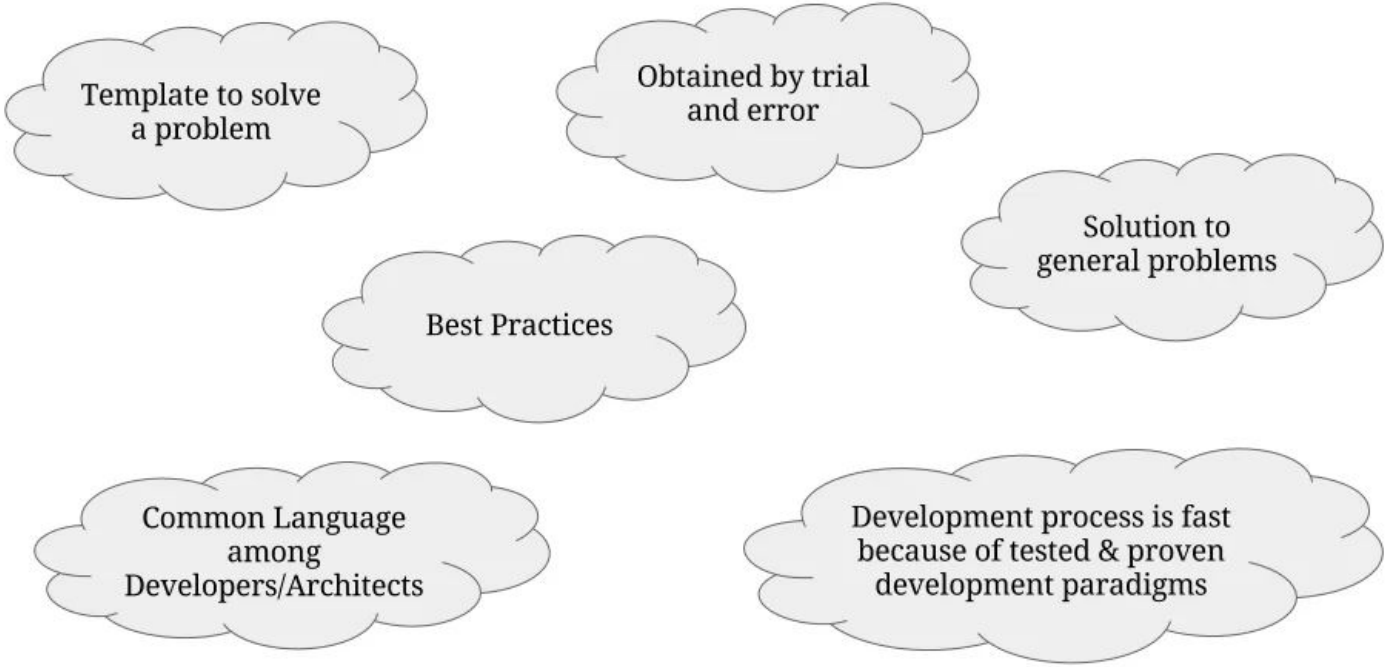
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



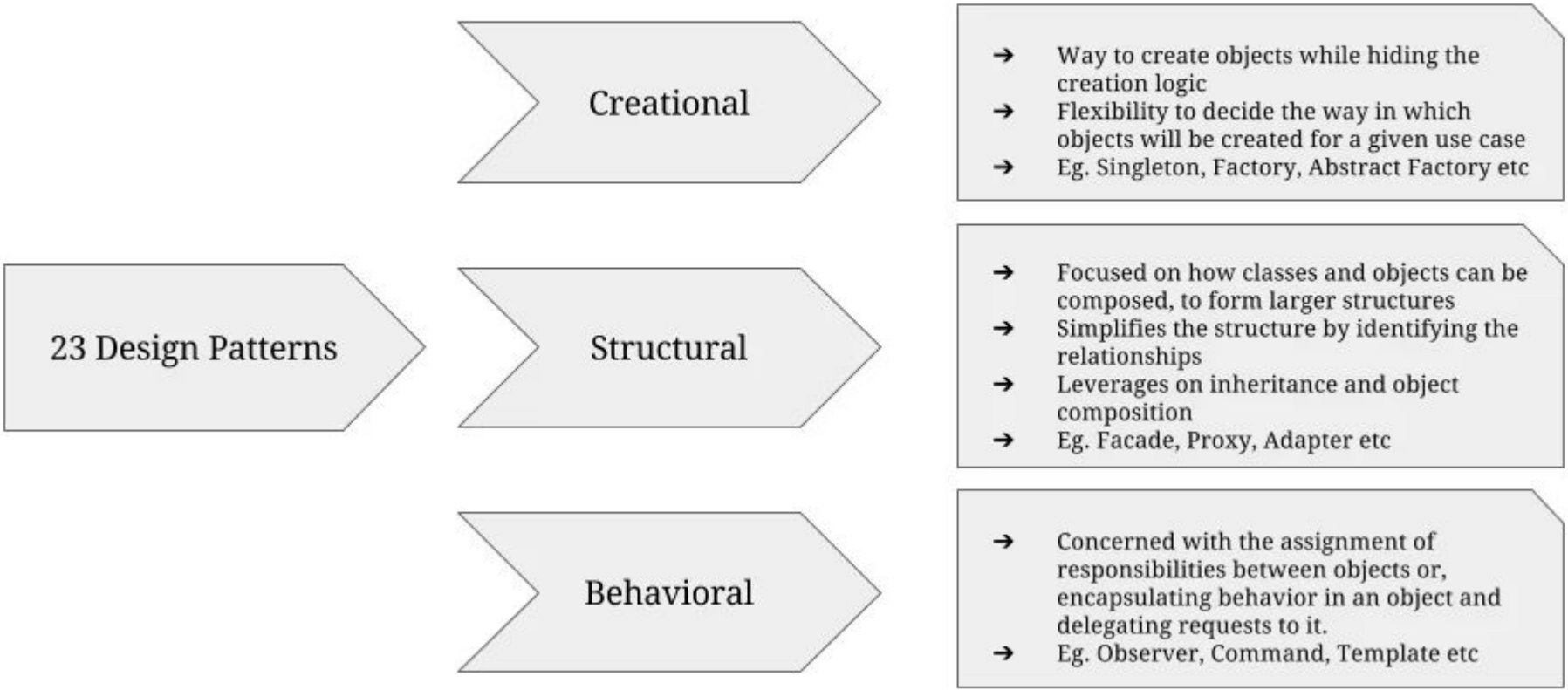
Foreword by Grady Booch

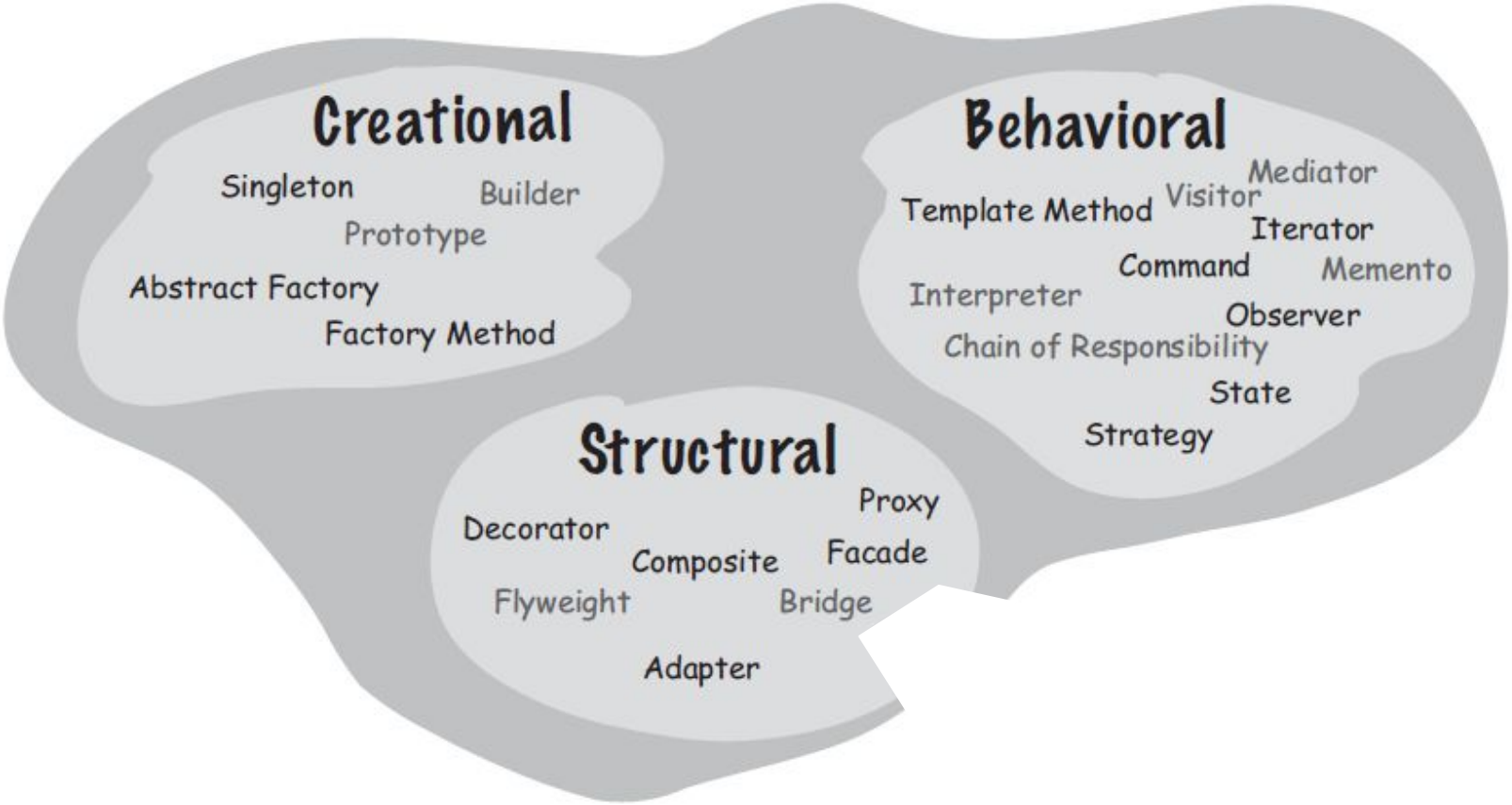
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

WHY?


Template to solve a problem
Obtained by trial and error
Solution to general problems
Best Practices
Common Language among Developers/Architects
Development process is fast because of tested & proven development paradigms

Types of Design Patterns



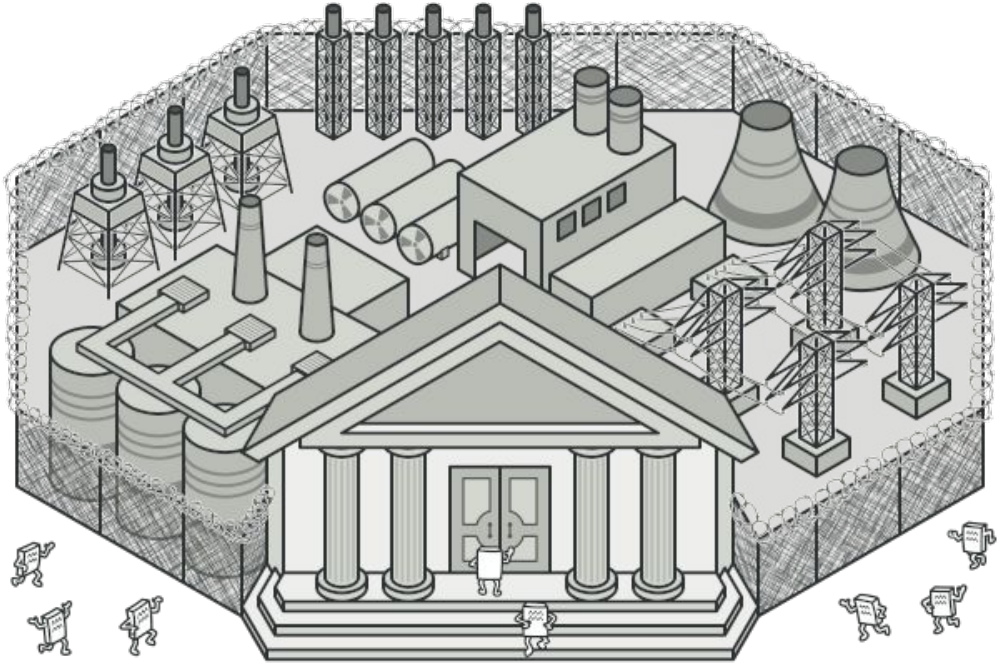

Creational
Singleton
Builder
Prototype
Abstract Factory
Factory Method
Behavioral
Mediator
Visitor
Template Method
Iterator
Command
Memento
Interpreter
Observer
Chain of Responsibility
State
Strategy
Structural
Proxy
Decorator
Composite
Facade
Flyweight
Bridge
Adapter

Structural Design Patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

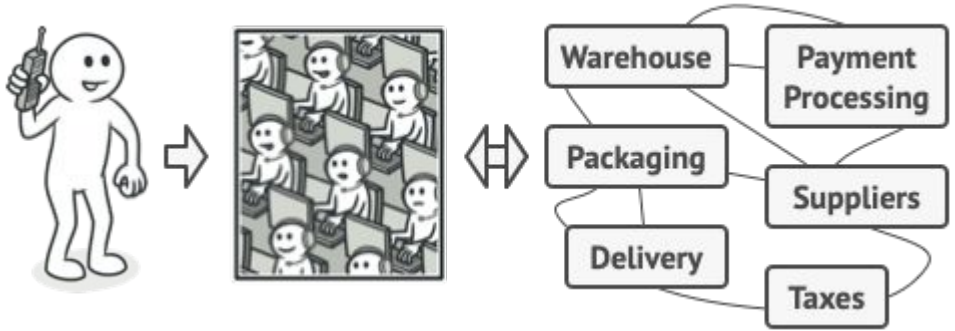
Facade Pattern

- provides a simplified interface to a library, a framework, or any other complex set of classes.

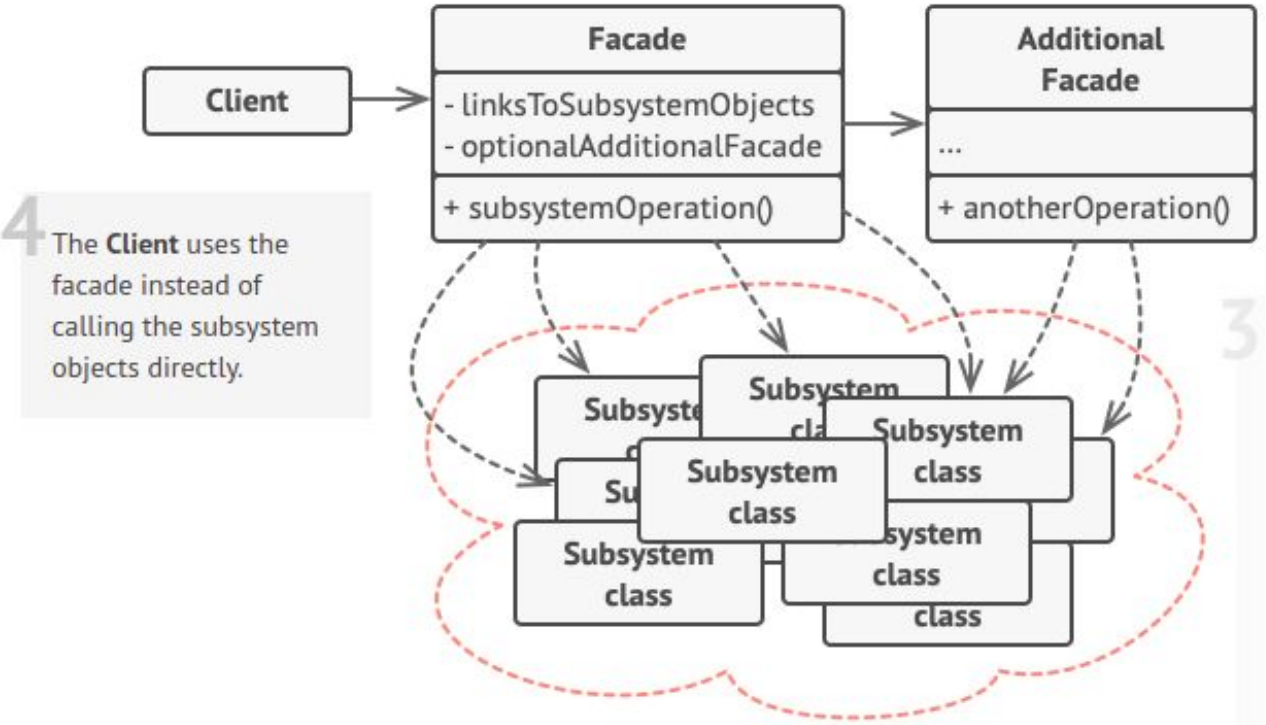


Facade Pattern - Problem

- the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

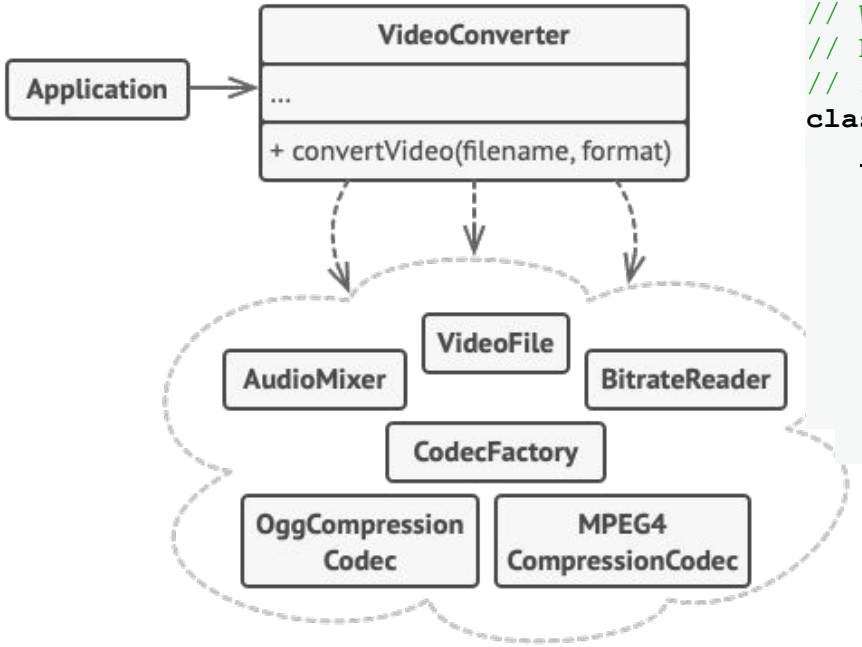


Facade Pattern - Solution



The Facade provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

Facade Pattern - Solution

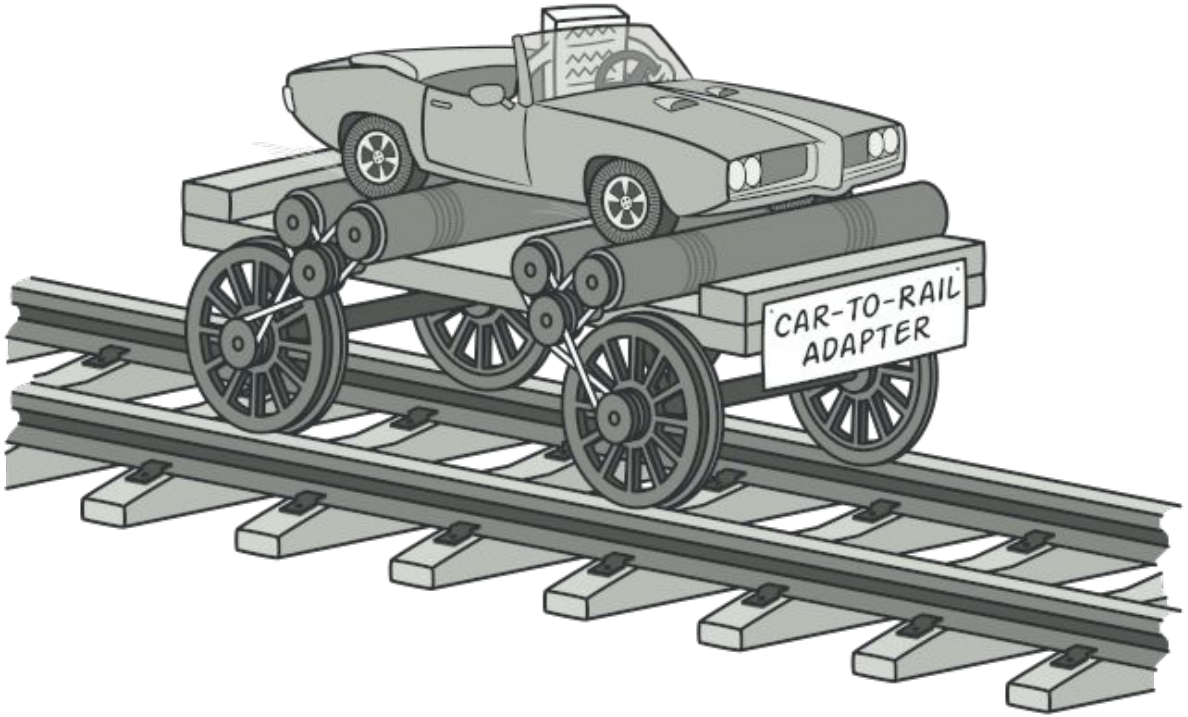



```
// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
    method convertVideo(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = new CodecFactory.extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
            return new File(result)


class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("catvideo.ogg", "mp4")
        mp4.save()
```

simplifies interaction with a complex video conversion framework.

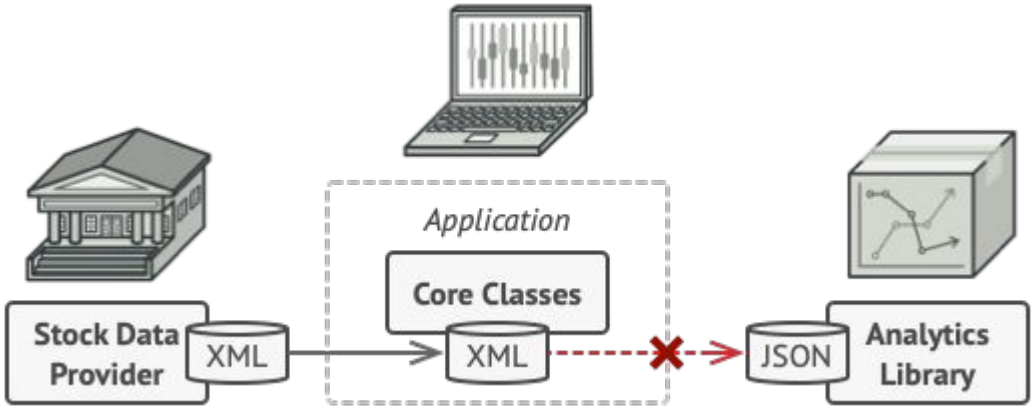
Adapter Pattern

- allows objects with incompatible interfaces to collaborate.



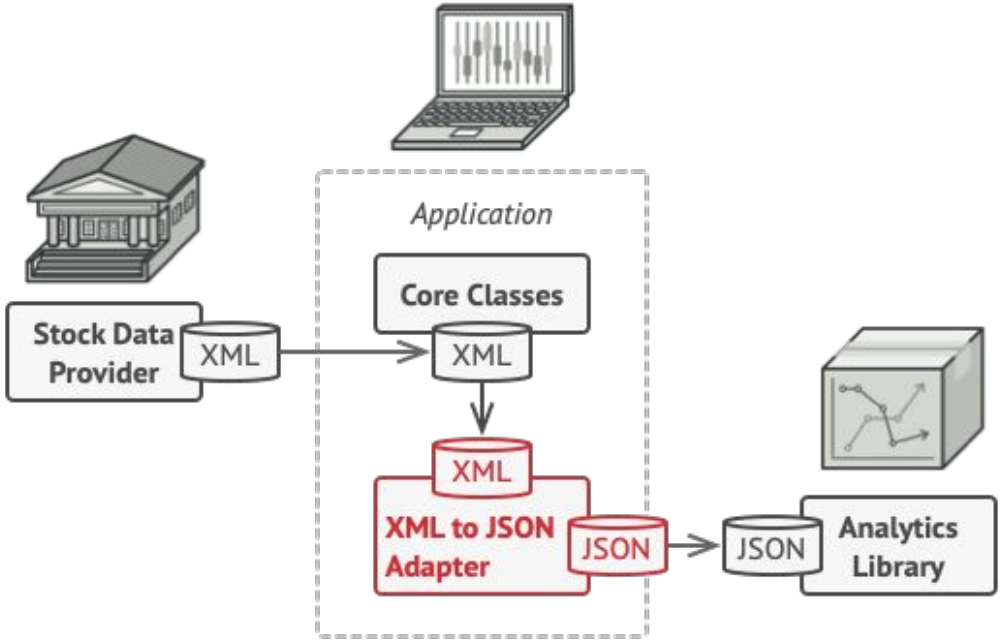
Adapter Pattern - Problem

- can’t use the analytics library “as is” because it expects the data in a format that’s incompatible with our stock market monitoring app.



Adapter Pattern - Solution

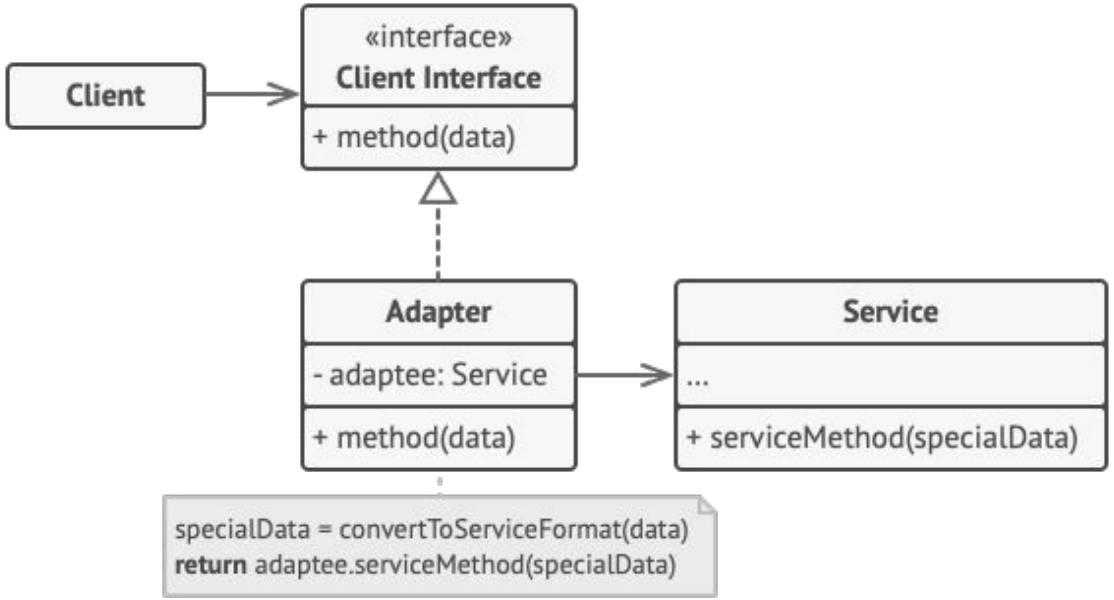
- Adapter - a special object that converts the interface of one object so that another object can understand it.



The adapter gets an interface, compatible with one of the existing objects.
Using this interface, the existing object can safely call the adapter's methods.
Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Adapter Pattern - Implementation Structure

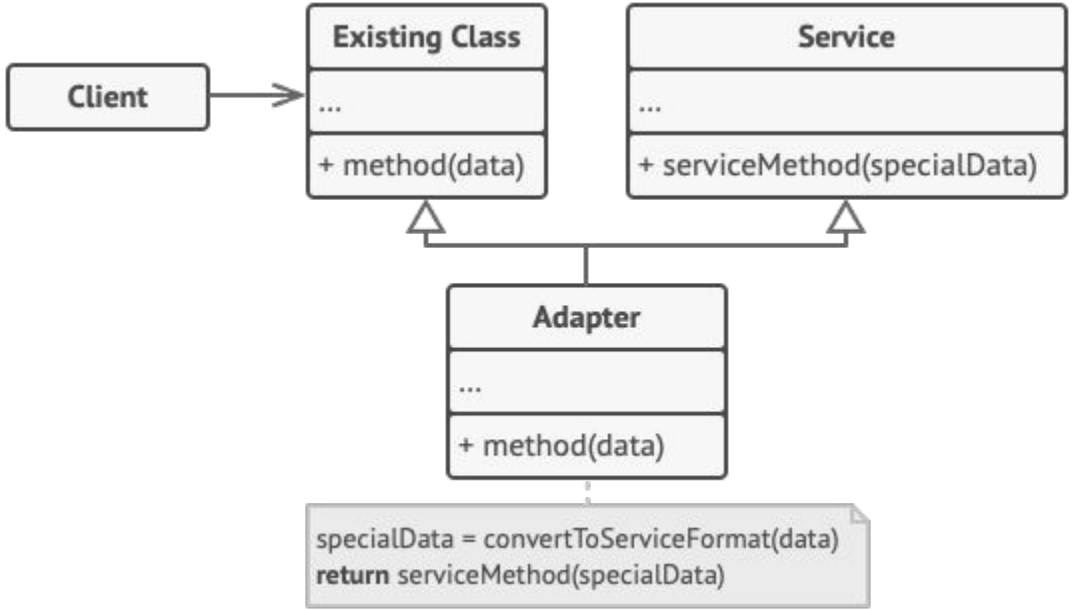
- Object Adapter - the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.



The Adapter is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

Adapter Pattern - Implementation Structure

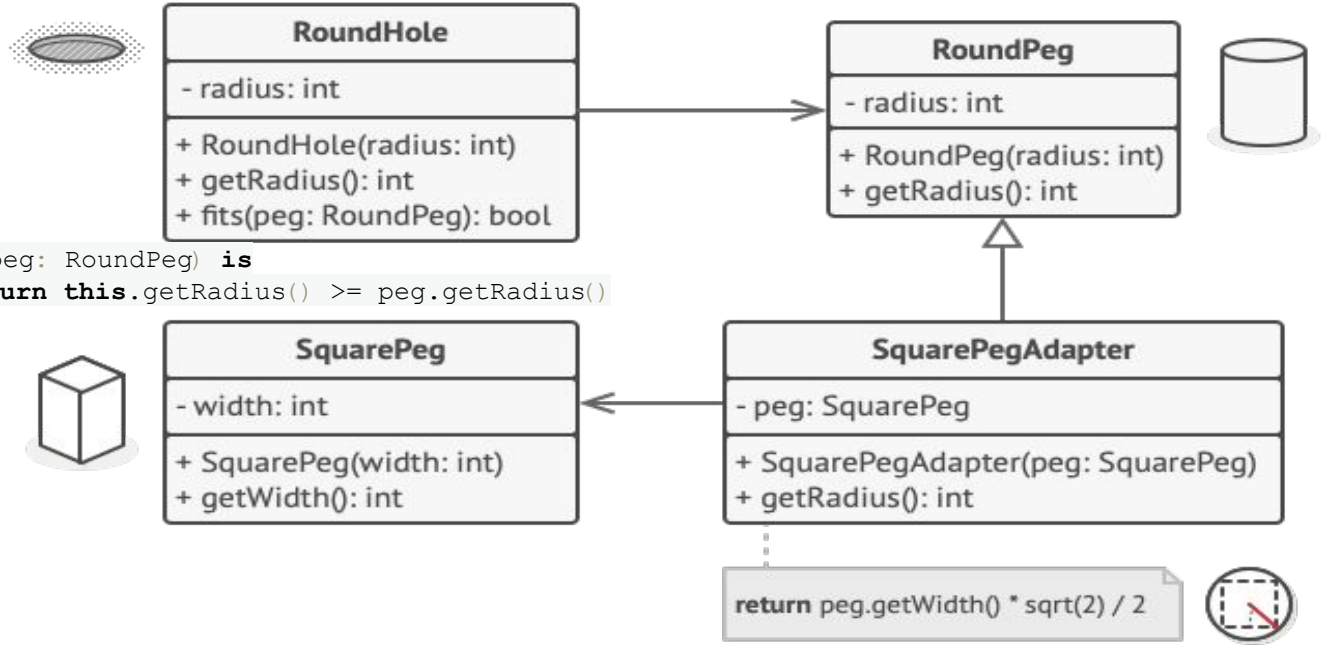
- Class adapter-This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time.



The Class Adapter doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Adapter Pattern - Example

- The Adapter pretends to be a round peg, with the radius of the smallest circle that can accommodate the square peg




```
method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()
```

Adapter Pattern - Example

- The Adapter pretends to be a round peg, with the radius of the smallest circle that can accommodate the square peg

```
class RoundHole is
    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()
```

```
// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true
```

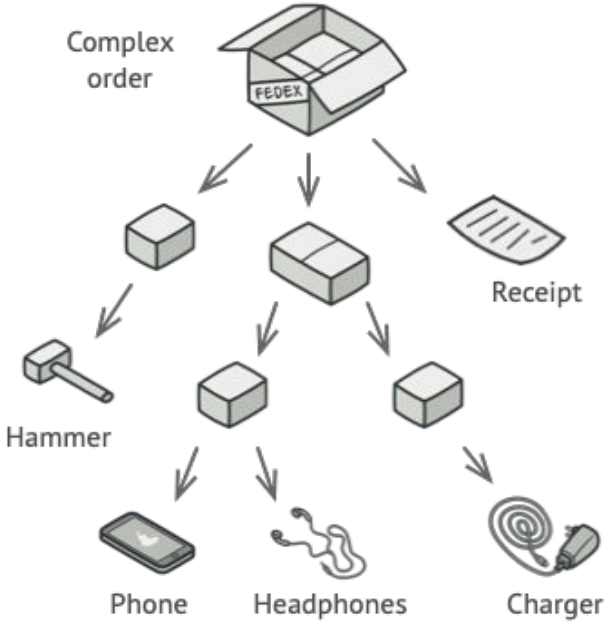
```
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg
```

```
small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)
```

```
small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

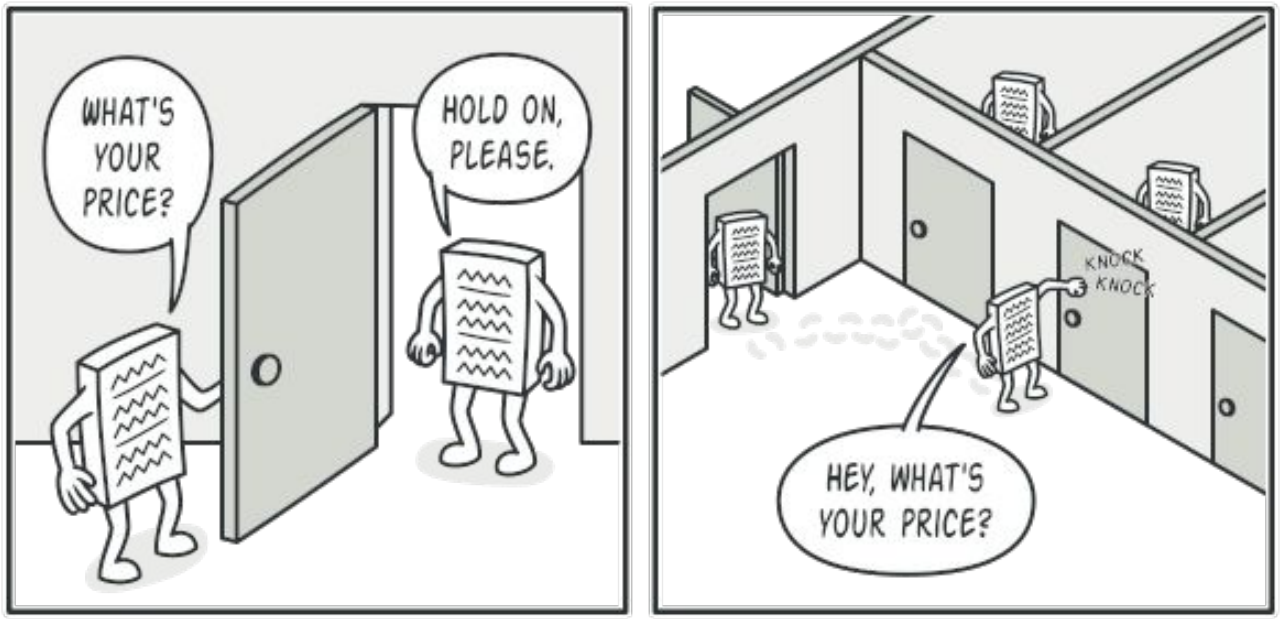
Composite Pattern - Problem

- Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

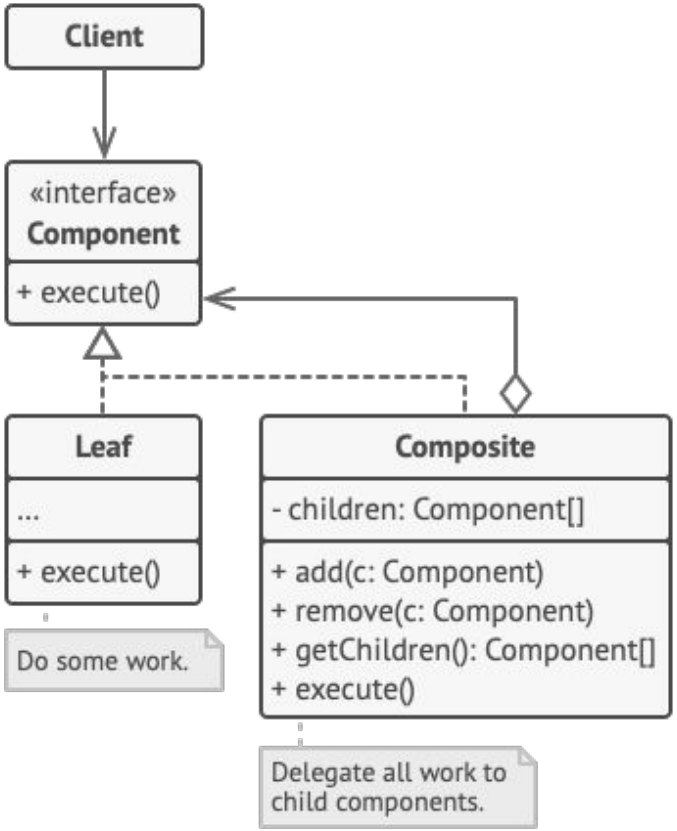


Composite Pattern - Solution

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

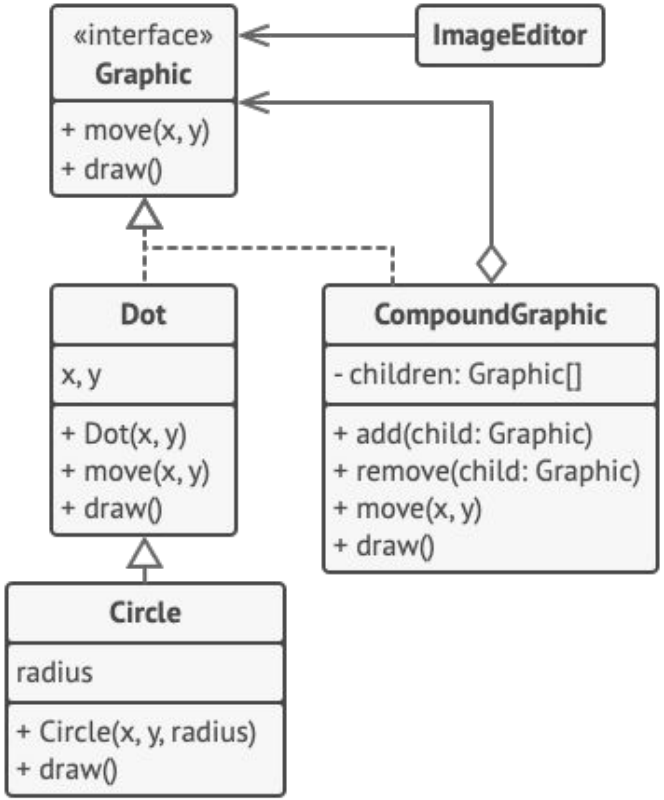


Composite Pattern - Solution



Composite Pattern - Example

The Composite pattern lets you implement stacking of geometric shapes in a graphical editor.



A compound shape has the same methods as a simple shape. However, instead of doing something on its own, a compound shape passes the request recursively to all its children and "sums up" the result.

Composite Pattern - Example

```
class ImageEditor is
    field all: Graphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combine selected components into one complex composite component.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // All components will be drawn.
        all.draw()
```

Behavioral Design Patterns

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

Strategy Pattern

- lets one define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

- The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

Strategy Pattern - Problem

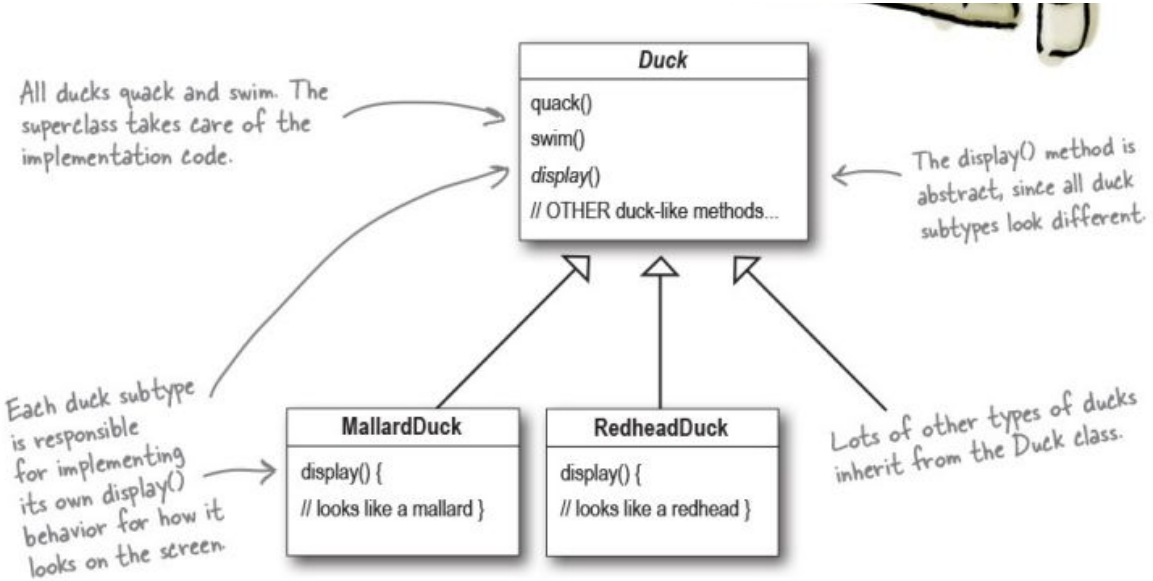
Amazing Game Company

Imagine that you work for a company that makes a highly successful game *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created on Duck superclass from which all other duck types inherit.

Example taken from Book : Head First Design Patterns

Strategy Pattern - Problem

- simulation game, SimUDuck . The game can show a large variety of duck species swimming and making quacking sounds



Strategy Pattern - Problem

Amazing Game Company

To stay competitive, the executives decide that flying ducks is just what SimUDuck needs to blow away the competitors!

You know what to do because you are a true OO genius. We will simply add a new method that will allow ducks to fly!

Strategy Pattern - Problem

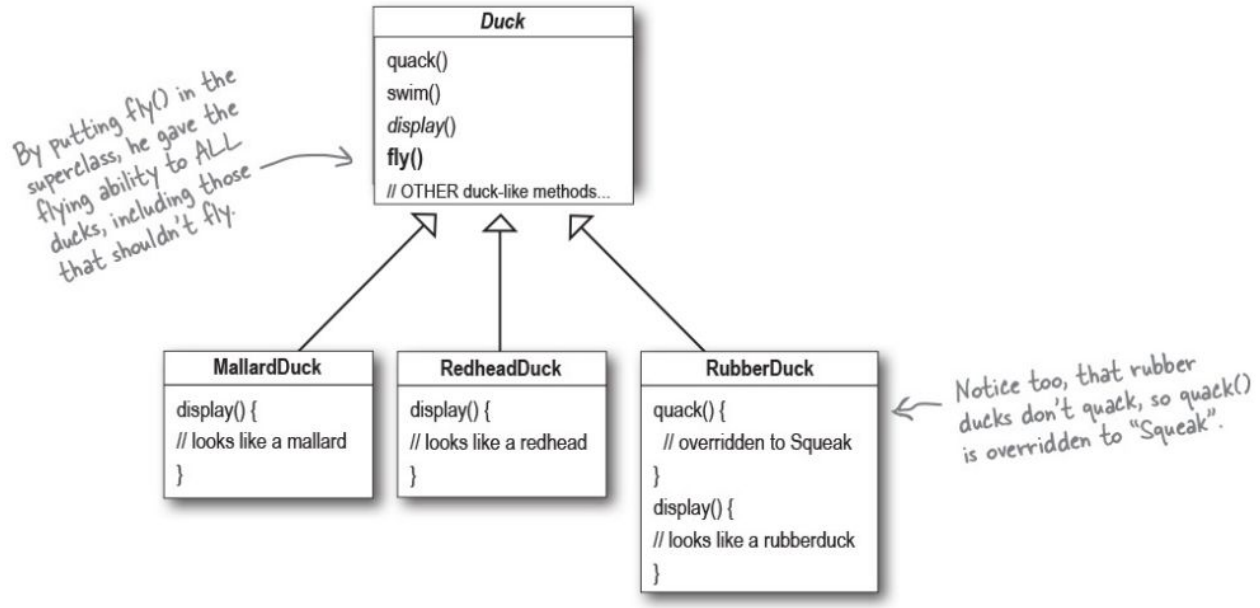
Amazing Game Company
BIG PROBLEM!

The executives contact you from a shareholders meeting. They just gave a demo of the game and their were **rubber ducks** flying all over the screen!

What happened?

Strategy Pattern - Problem

- When we have a lot of similar classes that only differ in the way they execute some behavior.



Strategy Pattern - Problem

Well, what can we do to make this work?

Strategy Pattern - Problem

Well, what can we do to make this work?

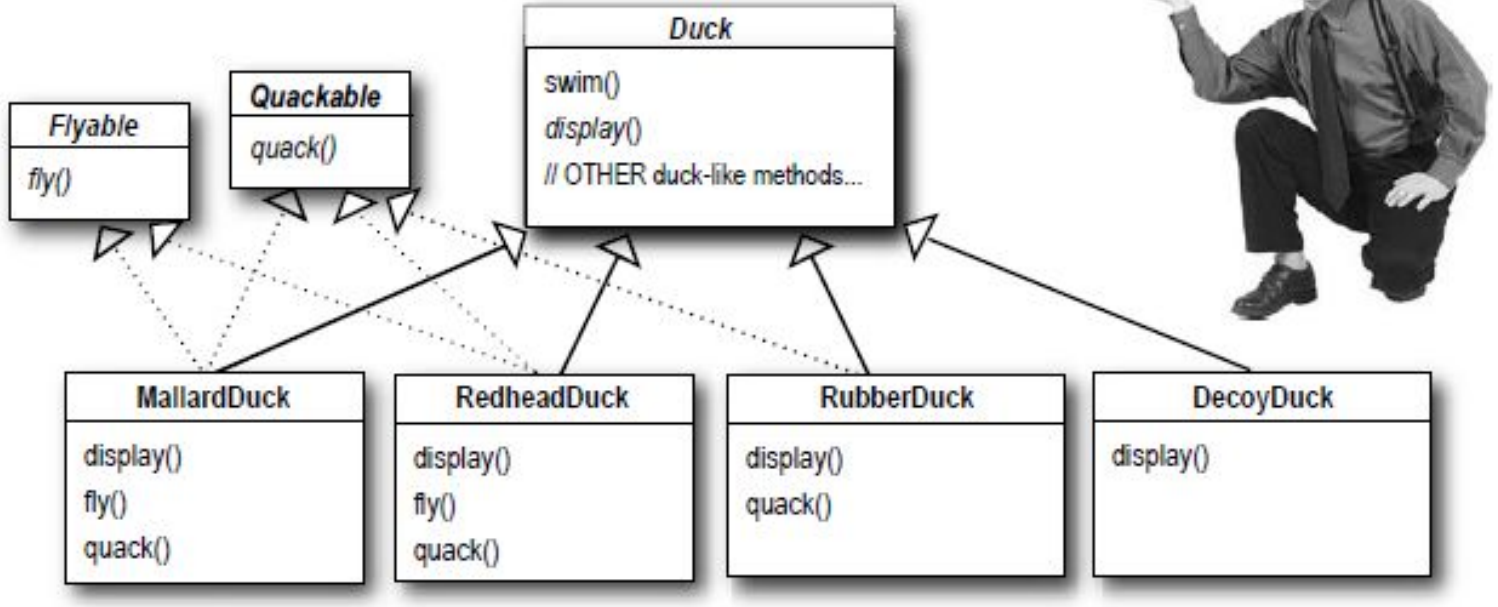
Sure, we can override the fly method to do nothing in our RubberDuck class.

Strategy Pattern - Problem

But, what if we add a DecoyDuck?

```
class DecoyDuck extends Duck("decoyduck") {
  def display() = name + " float like a piece of wood!"
  def quack() = ""
  def swim() = ""
  def fly() = ""
}
```

SimUDuck With Interfaces/Traits! - Perfect?

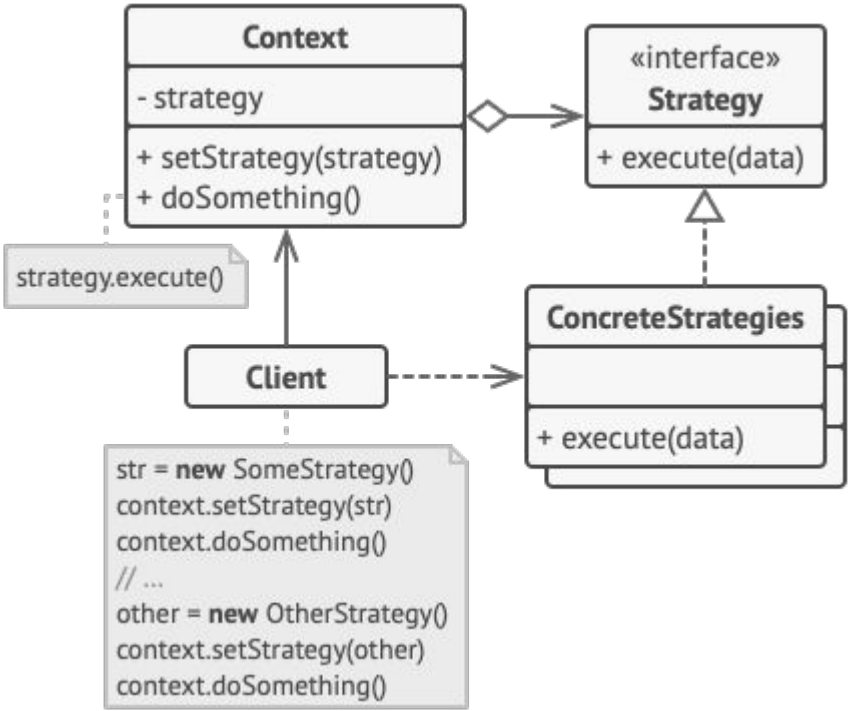


Code Duplication

Wouldn't it be cool if there was a way to build software so that when we need to change it, we could do so with the least possible impact on existing code?

Well, what can we do to make this work?

Strategy Pattern - Structure

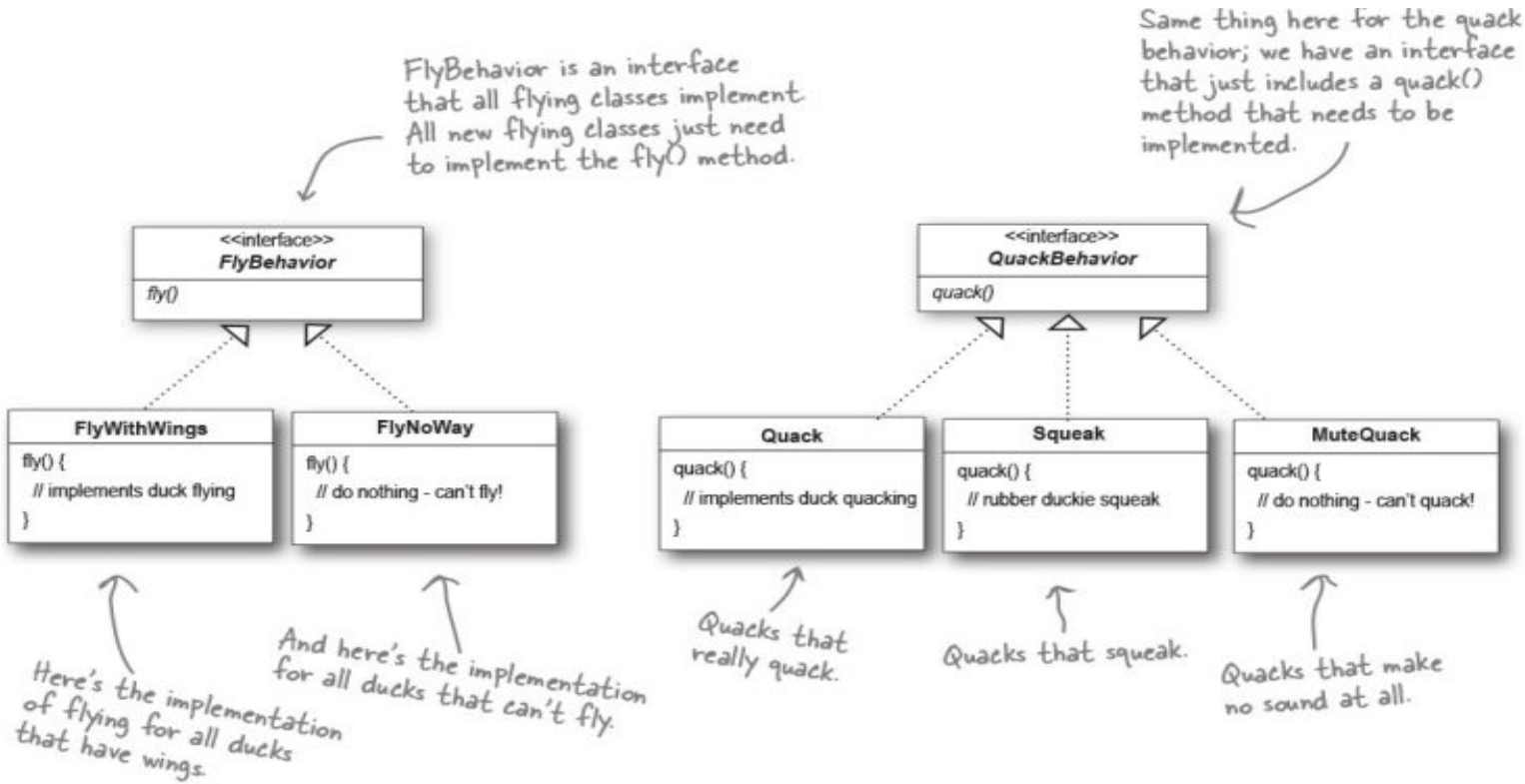


Design Principle - Strategy Pattern Solution

Identify the aspects of your application that vary and separate them from what stays the same.
Take what varies and "encapsulate" it so it won't affect the rest of your code.

Result: Fewer unintended consequences from code changes and more flexibility in your systems!

Strategy Pattern - Solution



Design Principle - Strategy Pattern Solution

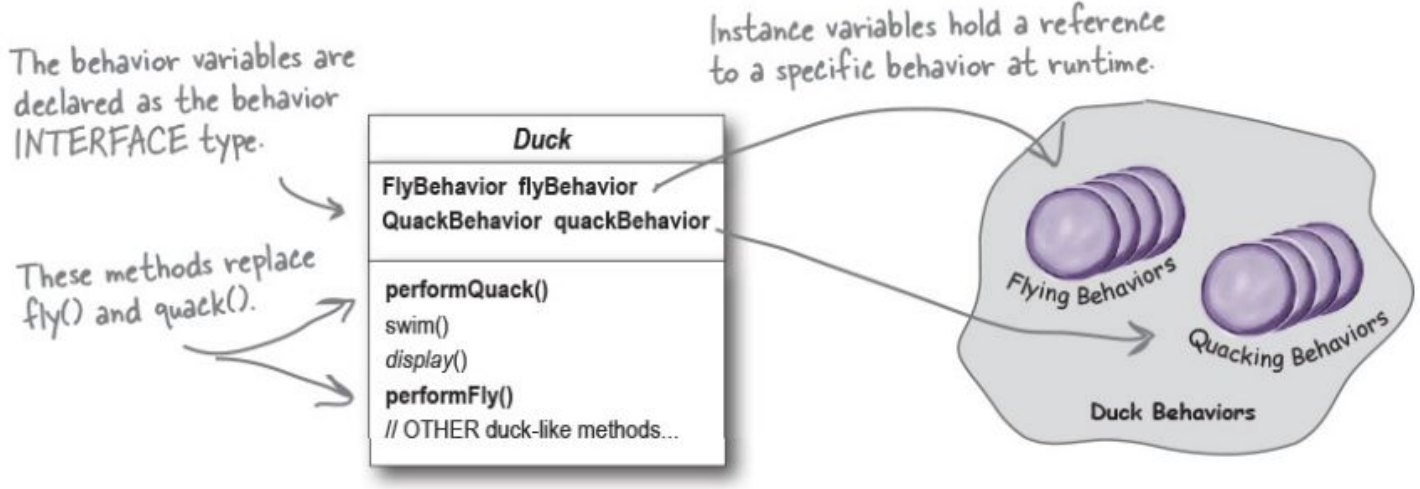
Inheritance forms an "is-a" relationship between classes. In our fixed version of the SimUDuck hierarchy we had…
MallardDuck IS-A Duck IS-A Flyable IS-A Quackable
RubberDuck IS-A Duck IS-A Quackable
DecoyDuck IS-A Duck

Design Principle - Strategy Pattern Solution

Favor composition over Inheritance!

Strategy Pattern - Solution

Separating what changes from what stays the same.



Strategy Pattern - Solution

```
public class MiniDuckSimulator {
    public static void main(String[] a
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }
```

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

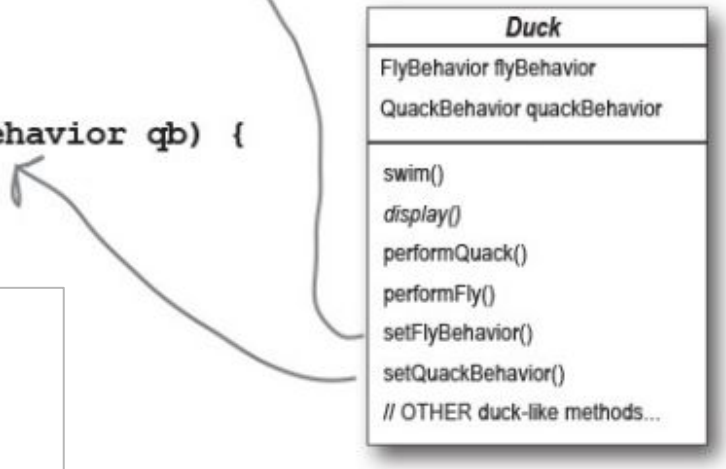
This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e., calls quack() on the duck's inherited quackBehavior reference).

Then we do the same thing with MallardDuck's inherited performFly() method.

Strategy Pattern - Solution

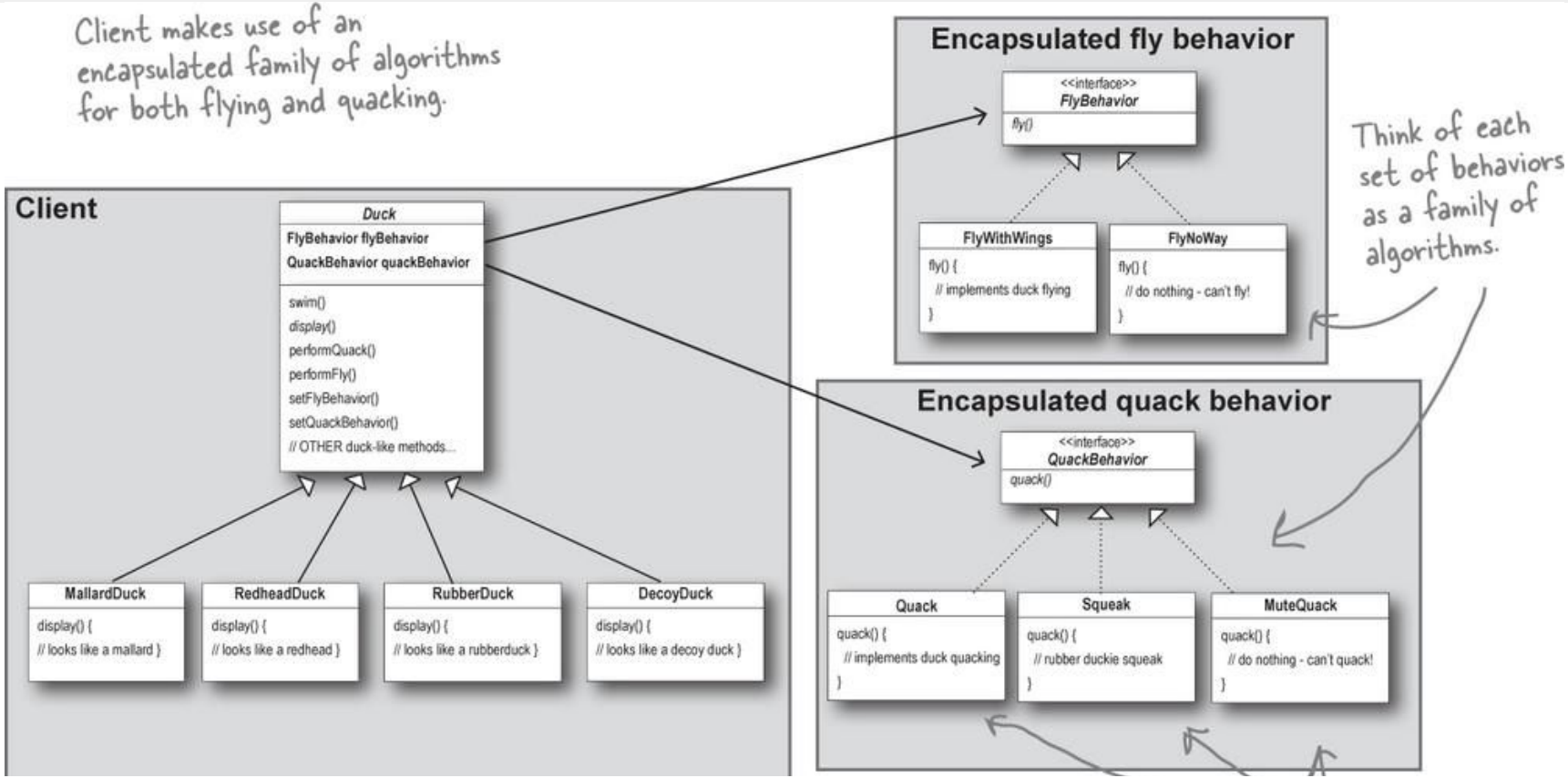
When we want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```




```
Duck dynamicMallard = new MallardDuck();
   dynamicMallard.SetFly(new FlyWithWings());
   dynamicMallard.Fly();
   dynamicMallard.SetQuack(new Quack());
   dynamicMallard.Quack();
```

Strategy - Summary



SimUDuck Application: New Idea!

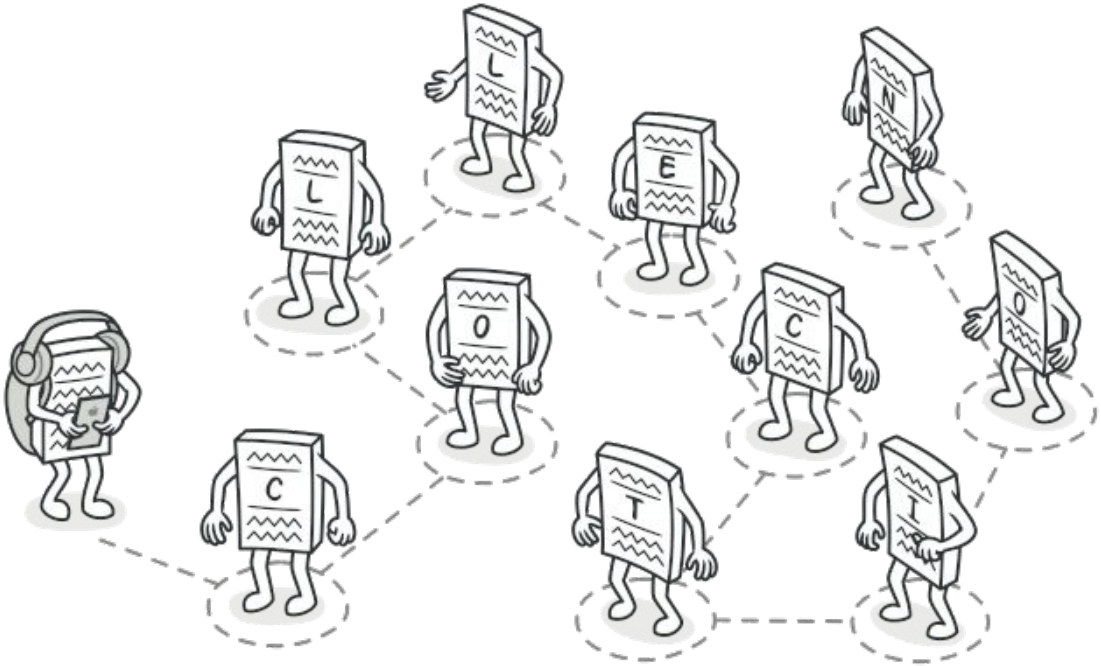
Amazing Game Company

Now, the executives want to add "magic" to the game. That is, they want ducks to be able to change the behavior of other ducks they come in contact with. They want ducks to be able to teach other ducks new tricks.

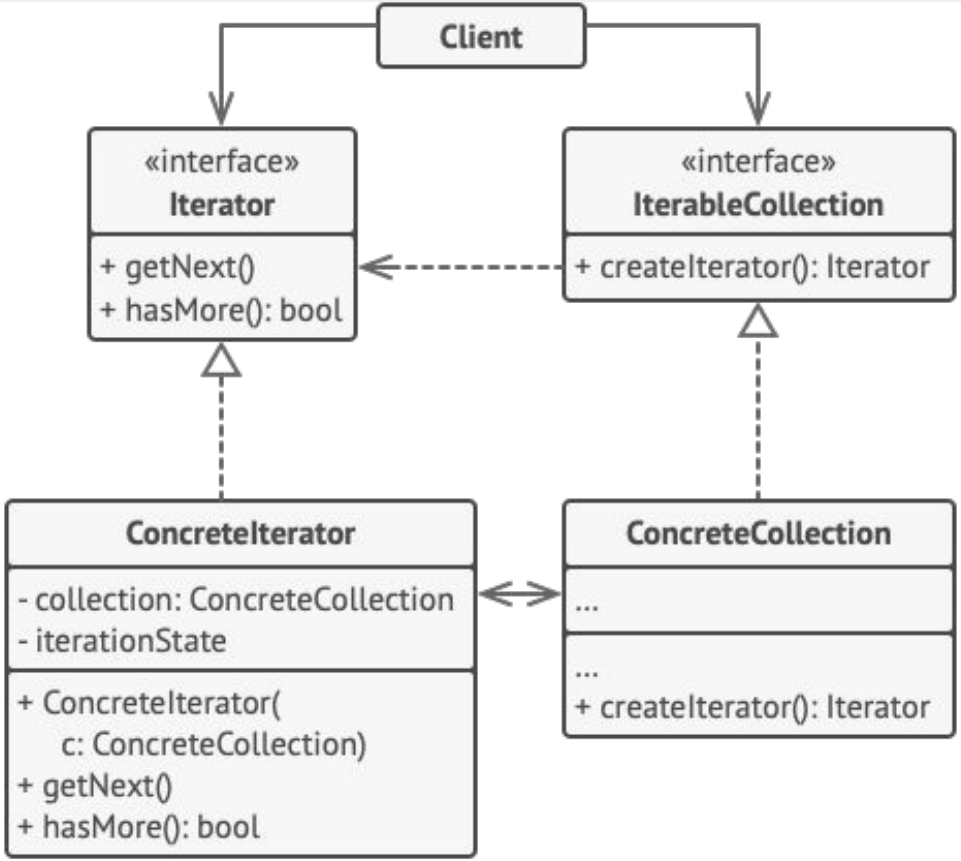
Can we teach a rubber duck to fly?

Iterator Pattern - Problem

- Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation



Iterator Pattern - Solution Template



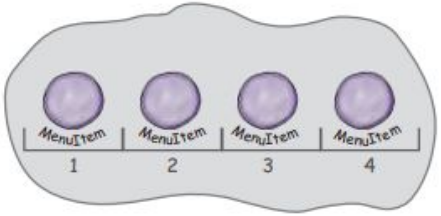
Iterator Pattern - Example

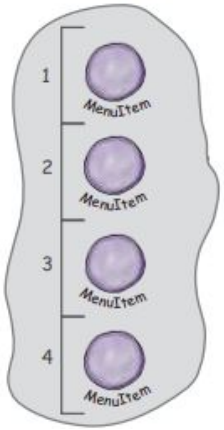
What did we do?

We wanted to give the Waitress an easy way to iterate over menu items...

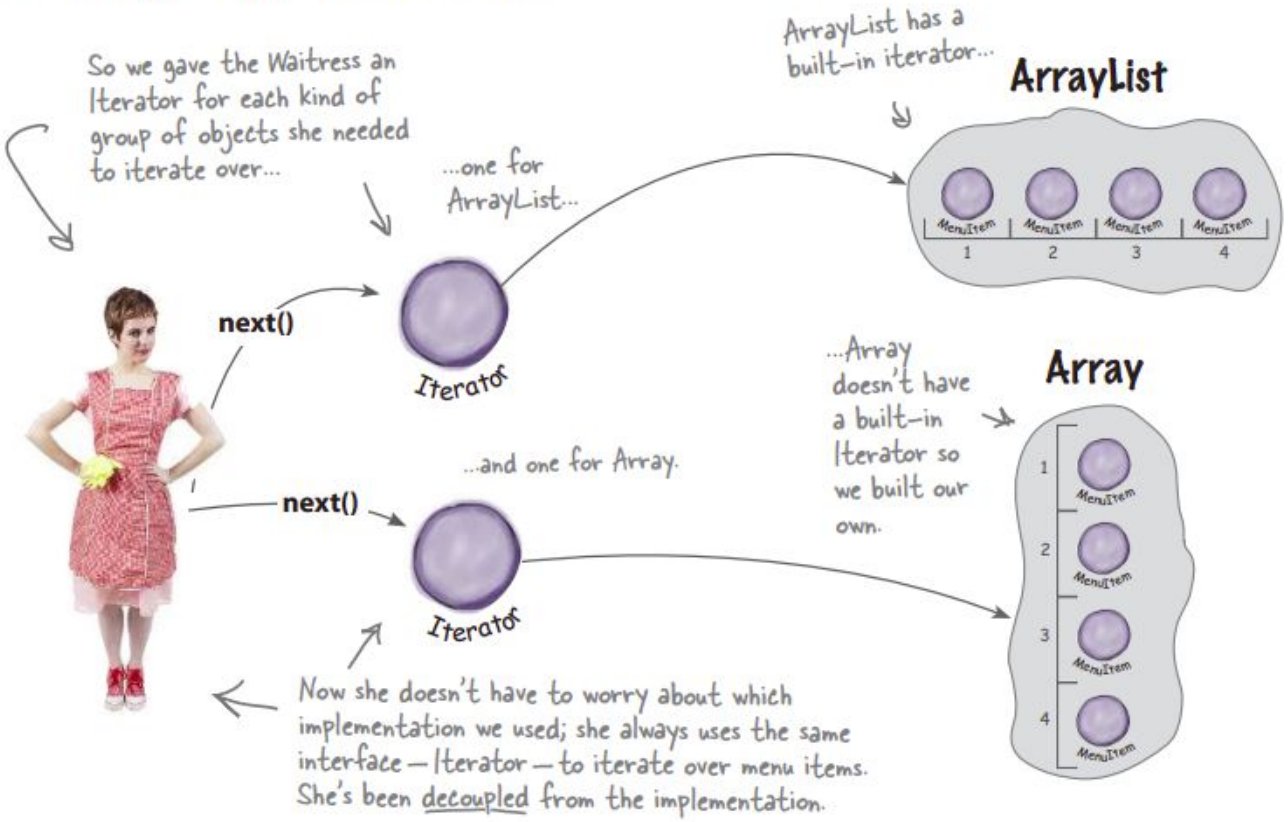
...and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.

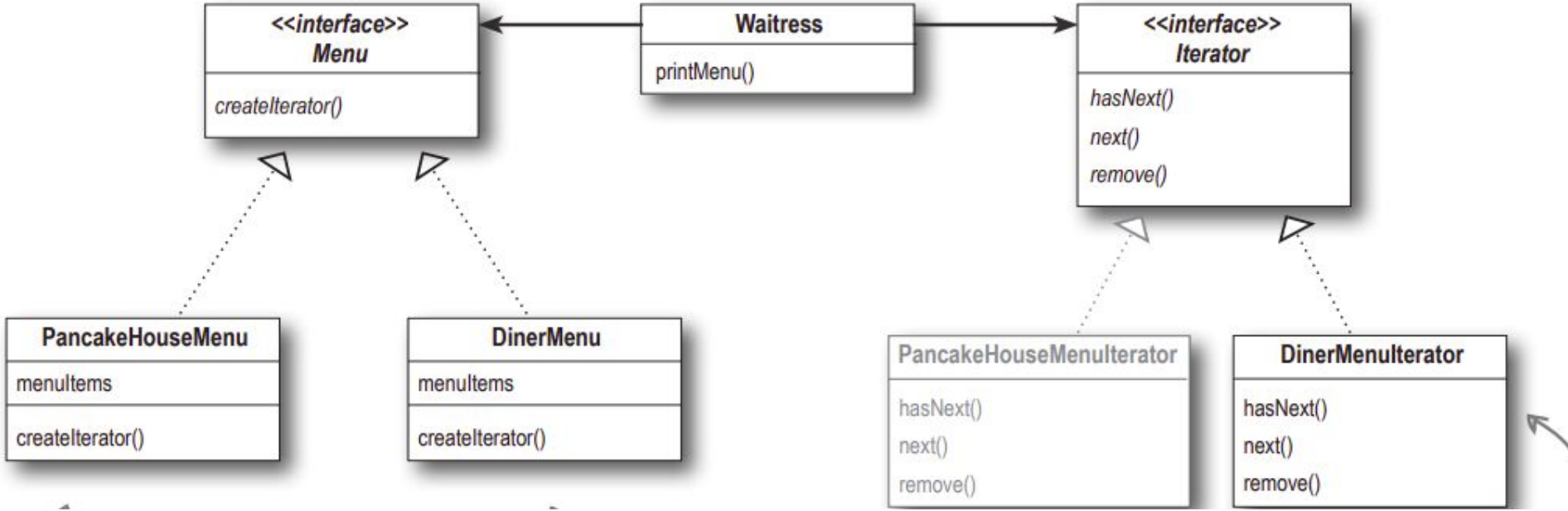




Iterator Pattern - Example



Iterator Pattern - Example



Iterator Pattern - Example

```
public void printMenu() {
    Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
    Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);
    System.out.println("\nLUNCH");
    printMenu(dinerIterator);
    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}

private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
        MenuItem menuItem = iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
```

We're using the cafe's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

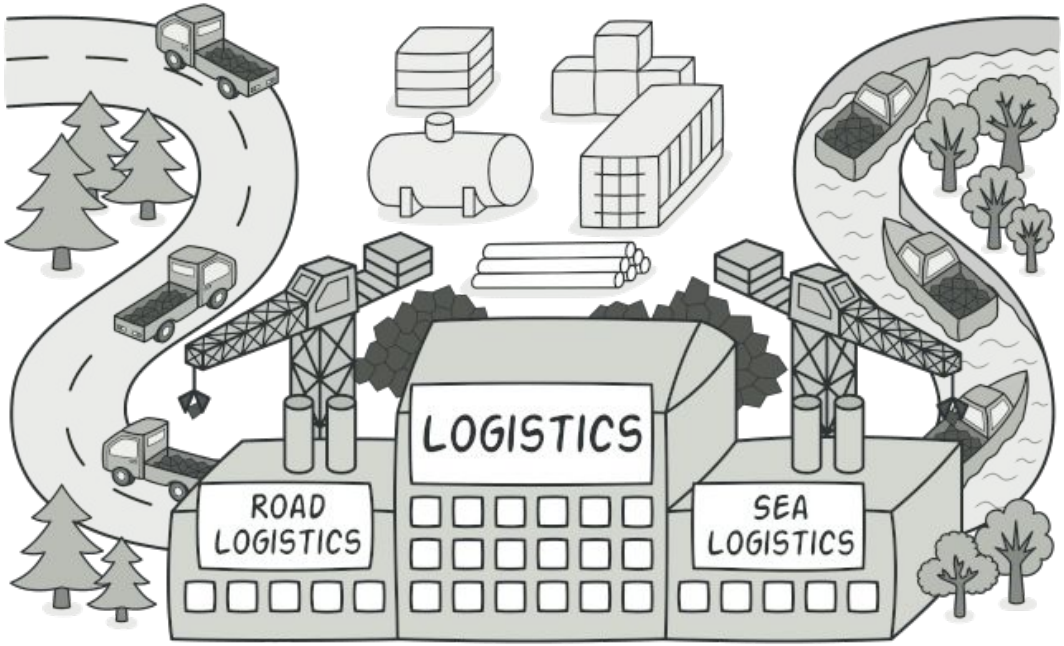
Nothing changes here.

Creational Design Patterns

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory Pattern - Problem

- It is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



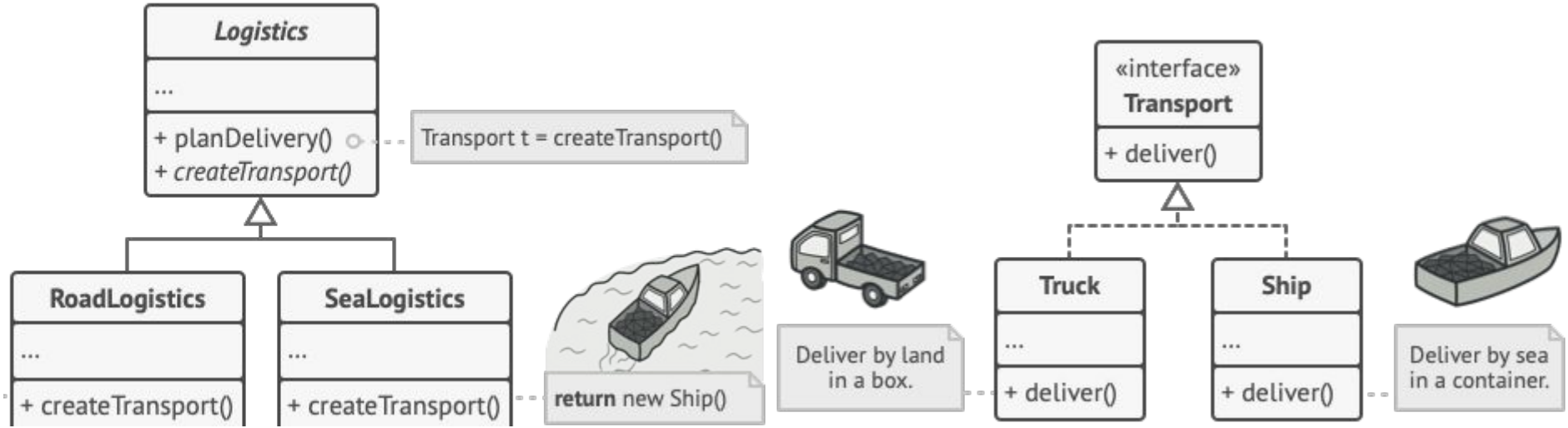
Factory Pattern - Problem

- Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

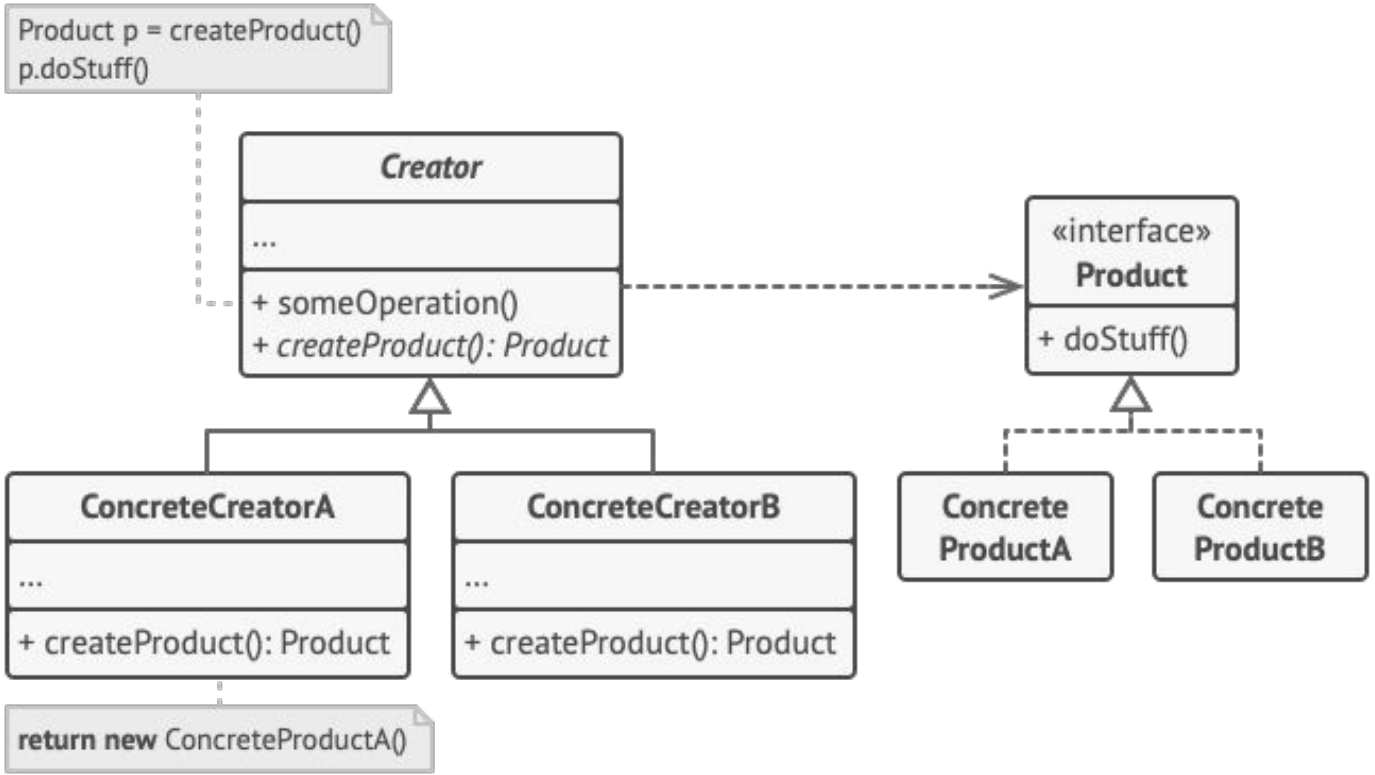
- Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.

Factory Pattern - Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.The Factory Method separates product construction code from the code that actually uses the product.

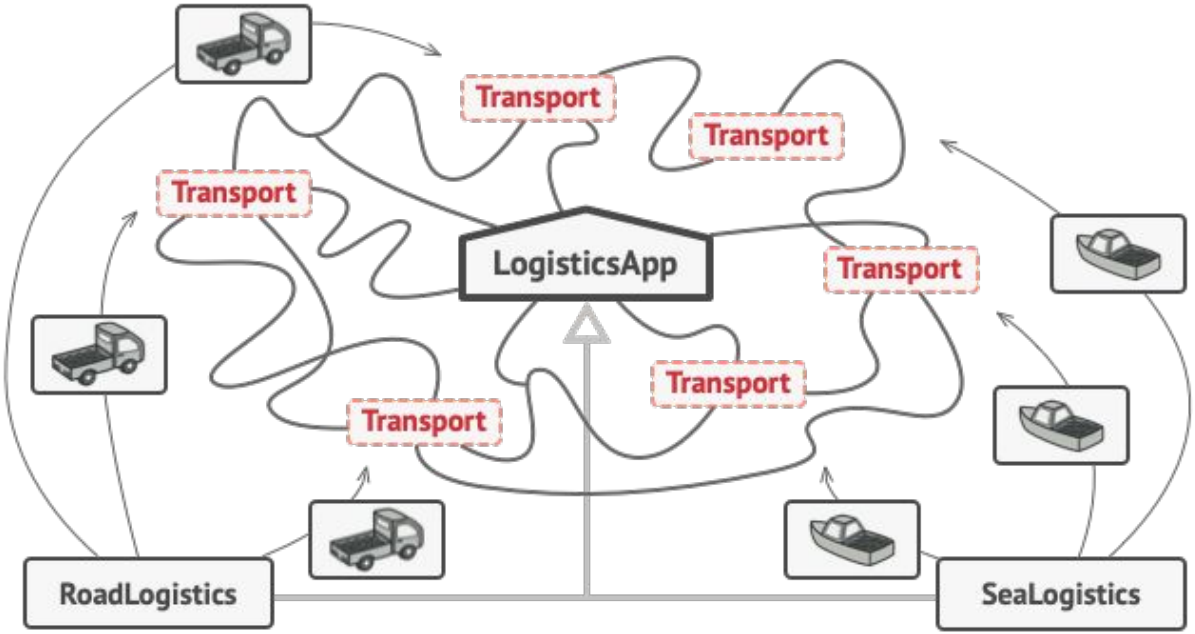


Factory Pattern - Solution



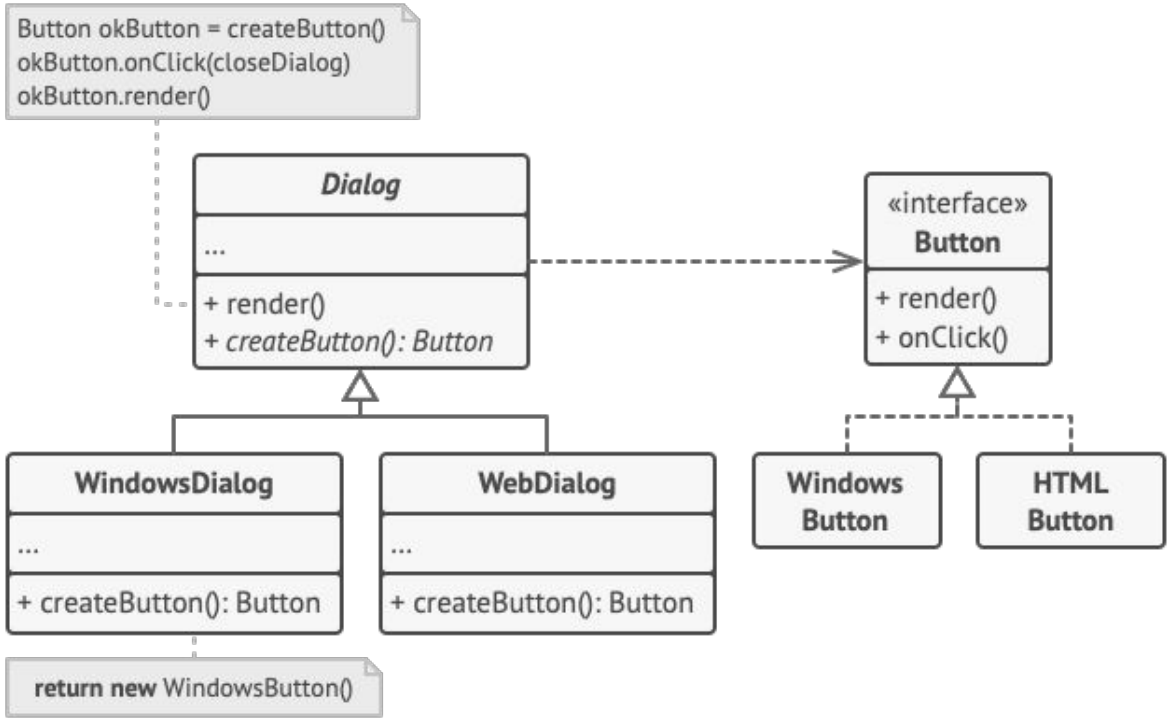
Factory Pattern - Solution

- The code that uses the factory method knows that all transport objects are supposed to have the deliver method, but exactly how it works isn't important to the client.



Factory Pattern - Example

- Used for creating cross-platform UI elements without coupling the client code to concrete UI classes.



Factory Pattern - Example

- Used for creating cross-platform UI elements without coupling the client code to concrete UI classes.

```
class Application is
    field dialog: Dialog

    // The application picks a creator's type depending on the
    // current configuration or environment settings.
    method initialize() is
        config = readApplicationConfigFile ()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown OS.")


    method main() is
        this.initialize()
          dialog.render ()
```

Factory Pattern - Applicability

- Use the Factory Method when you don’t know beforehand the exact types and dependencies of the objects your code should work with.

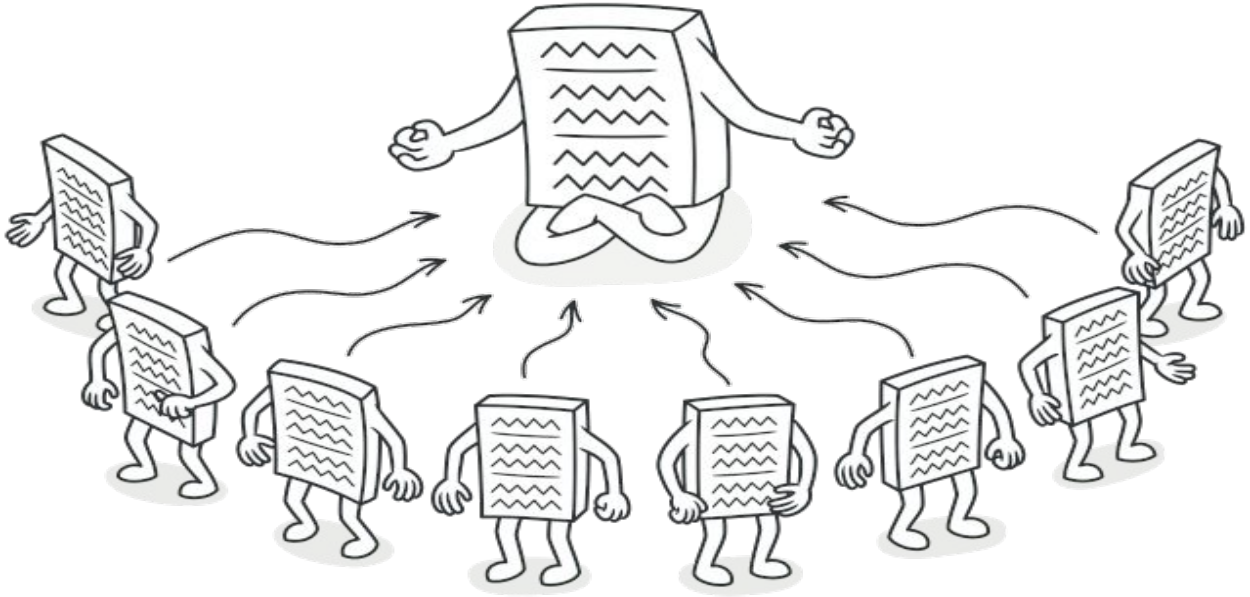
- Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.

- Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

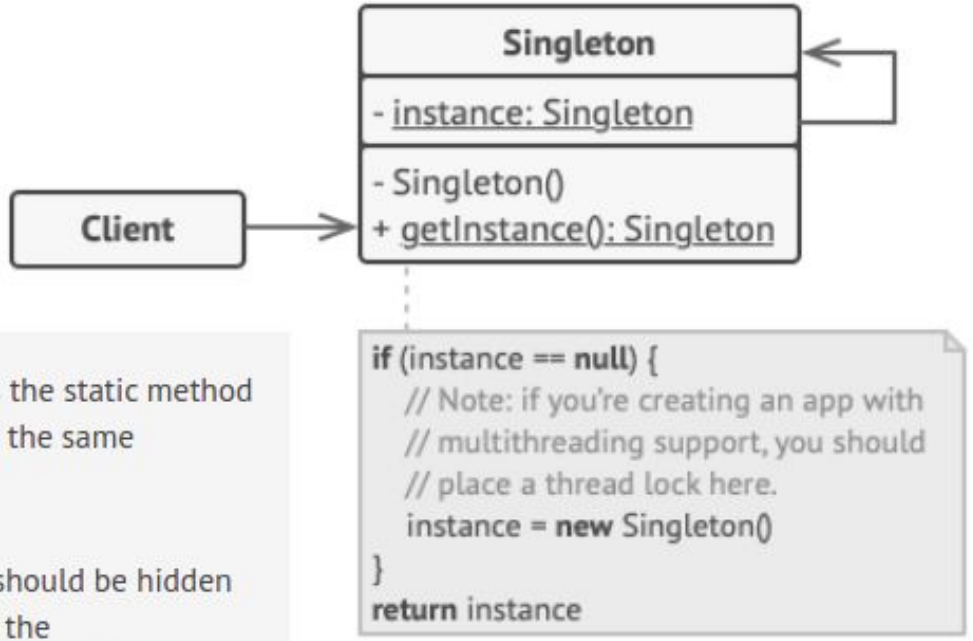
.

Singleton Pattern - Problem

- Lets one ensure that a class has only one instance, while providing a global access point to this instance.



Singleton Pattern - Solution



1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

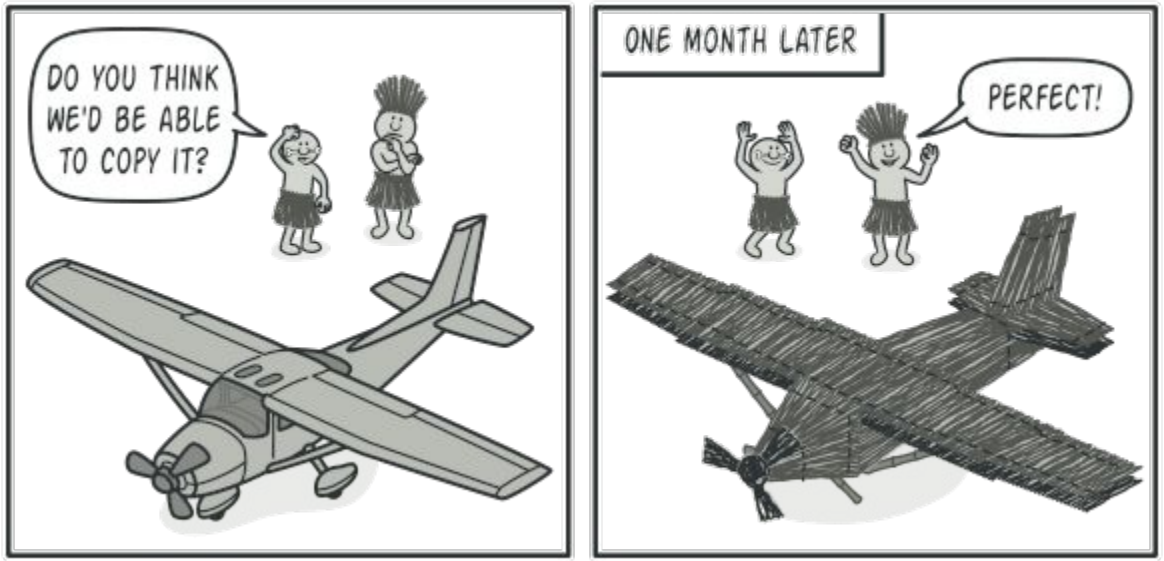
The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

Singleton Pattern - Example

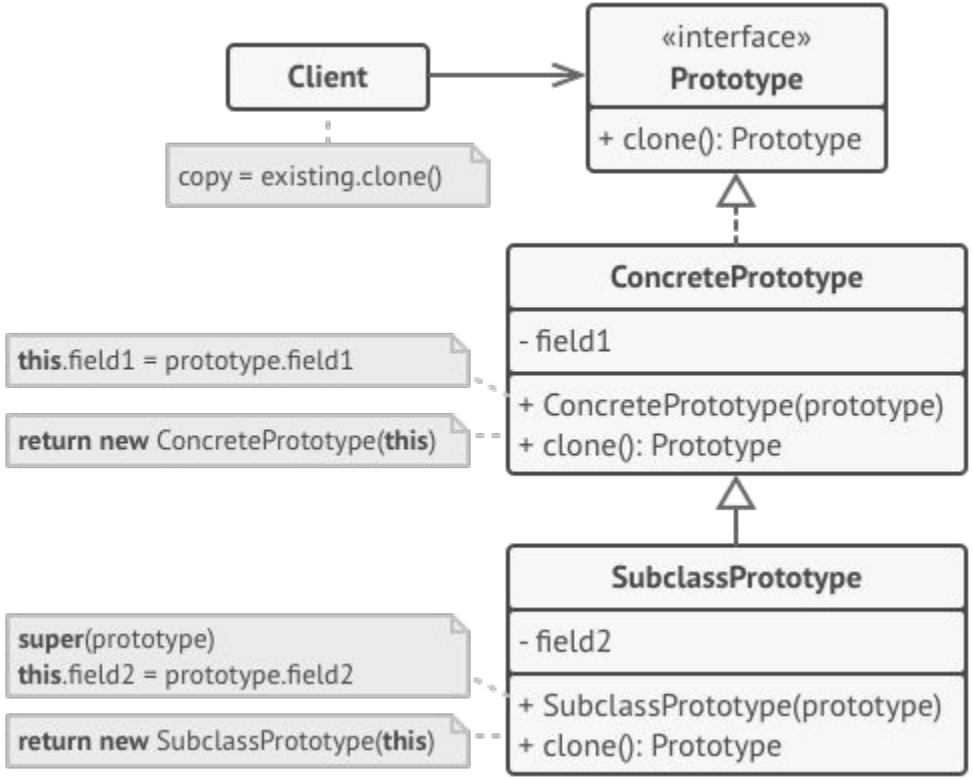
```
class Database is
    private static field instance: Database
    private constructor Database() is
        // Some initialization code, such as the actual connection to a database server.

    // The static method that controls access to the singleton instance.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance
    // Finally, any singleton should define some business logic which can be executed on its
instance.
    public method query(sql) is

class Application is
    method main() is
        Database foo = Database.getInstance()
         foo.query("SELECT ...")
```

Prototype Pattern - Problem

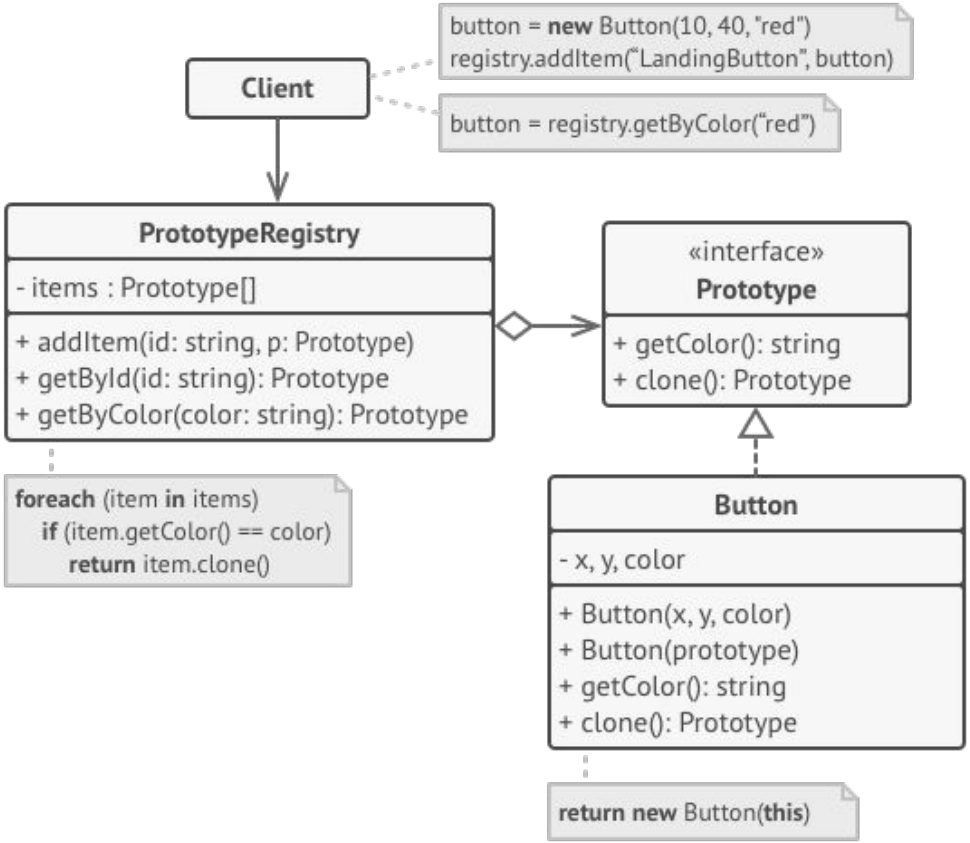
- Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.



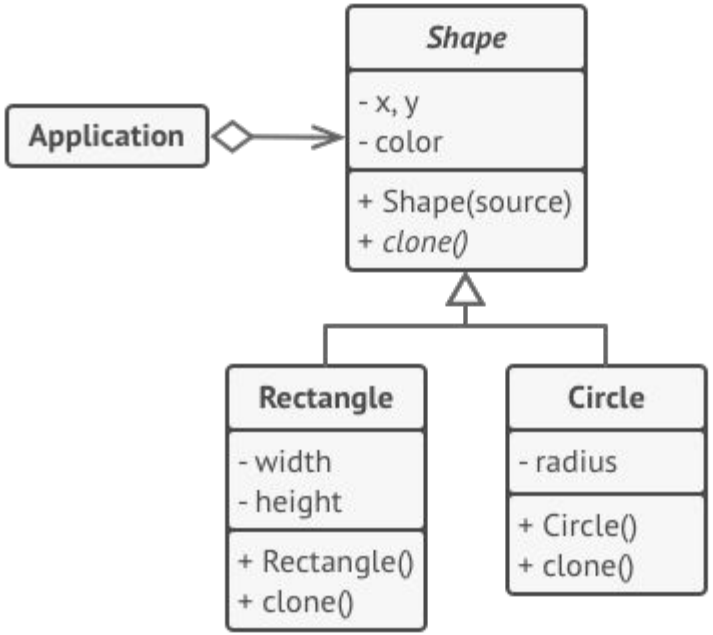
Prototype Pattern - Solution



Prototype Pattern - Solution

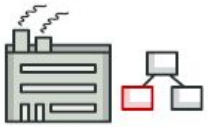


Prototype Pattern - Example



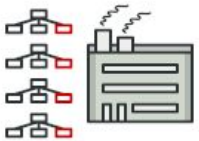

```
class Application is
    field shapes: array of Shape

        method businessLogic() is
                Array shapesCopy =  new Array of Shapes.

                    foreach (s in shapes) do
                shapesCopy.add (s.clone())
```

Other Creational Patterns



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

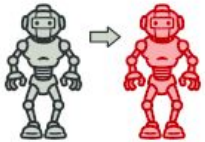


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

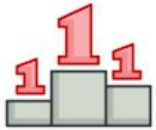
Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

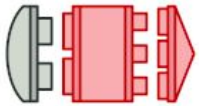
Lets you copy existing objects without making your code dependent on their classes.



Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Other Structural Patterns

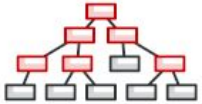


Adapter

Allows objects with incompatible interfaces to collaborate.

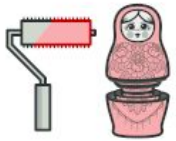
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

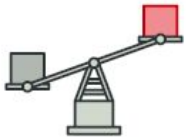


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



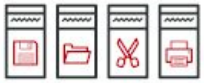
Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Other Behavioural Patterns

Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

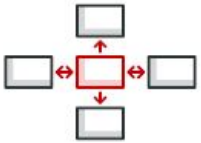


Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

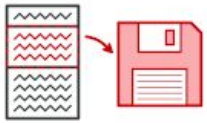
Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Criticism of Design Patterns

THANK YOU !