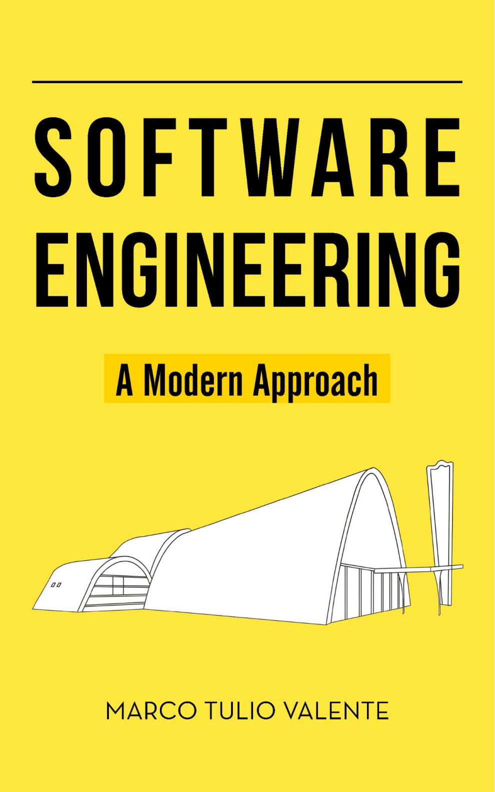


SOFTWARE ENGINEERING

A Modern Approach

MARCO TULIO VALENTE

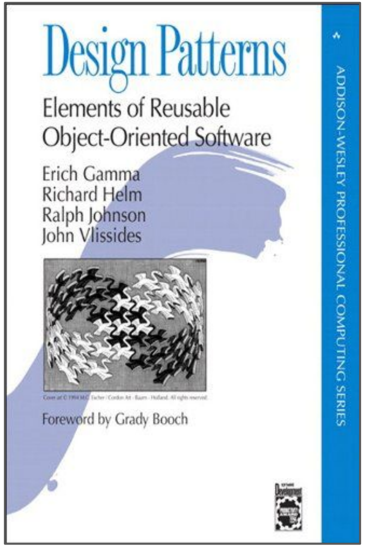
Chapter 6 - Design Patterns

Prof. Marco Tulio Valente

https://softengbook.org

Design Patterns

- Recurrent solutions to design problems faced by developers
- Gang of Four (GoF) book


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

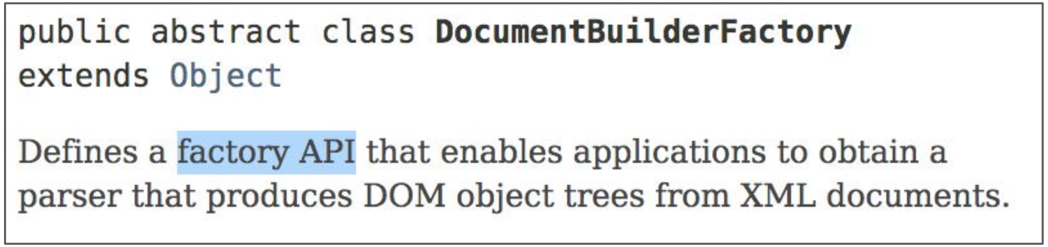
1994

Usage #1: Design Reuse

- Suppose we have a design problem
  - It might exist a pattern that solves this problem
  - Reusing it prevents us from reinventing the wheel

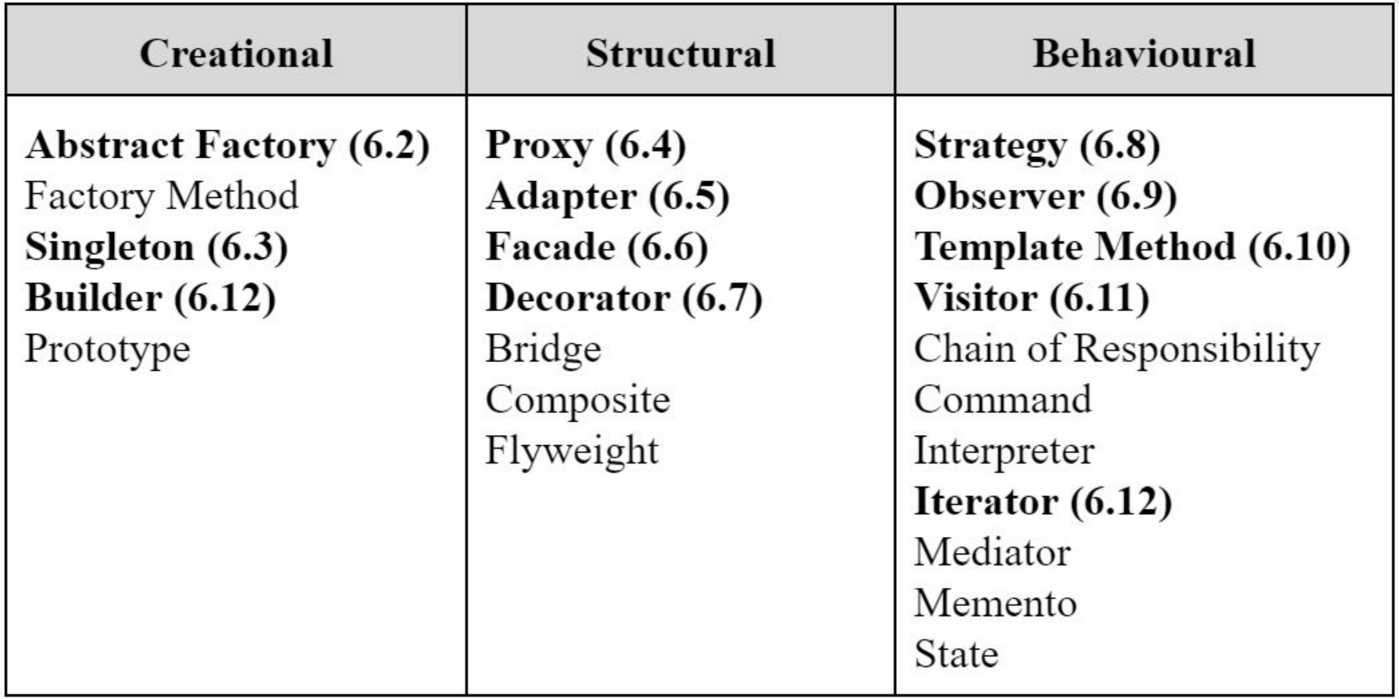
Usage #2: Vocabulary

- Vocabulary for discussions, documentation, etc.

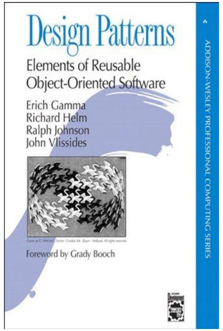


```
public abstract class DocumentBuilderFactory
extends Object
```

Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.



Creational	Structural	Behavioural
Abstract Factory (6.2)	**Proxy (6.4)**	**Strategy (6.8)**
Factory Method	**Adapter (6.5)**	**Observer (6.9)**
Singleton (6.3)	**Facade (6.6)**	**Template Method (6.10)**
Builder (6.12)	**Decorator (6.7)**	**Visitor (6.11)**
Prototype	Bridge	Chain of Responsibility
	Composite	Command
	Flyweight	Interpreter
		Iterator (6.12)
		Mediator
		Memento
		State


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

23 patterns

Structure

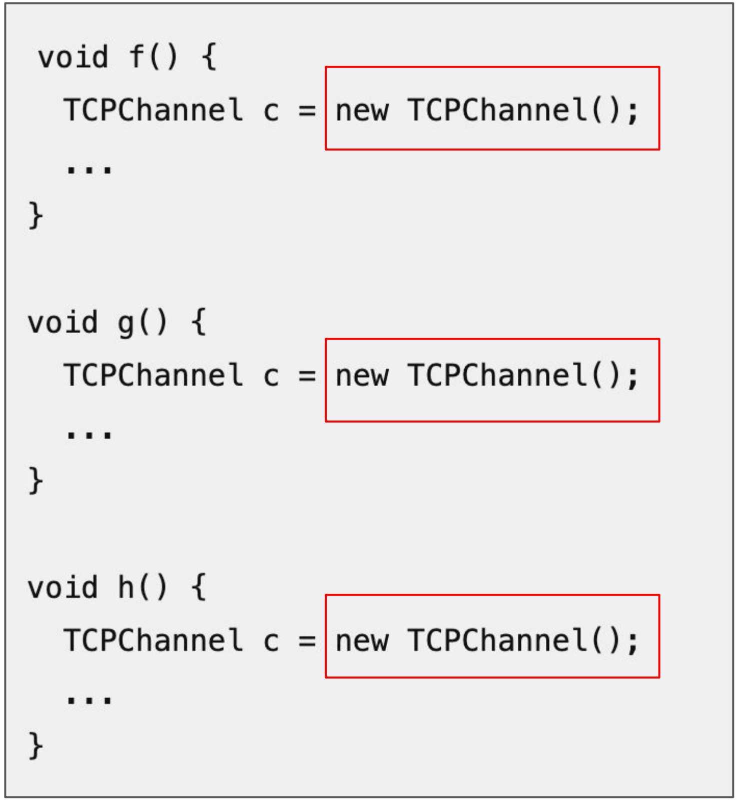
- Context
- Problem
- Solution (using a design pattern)

Important: Design for Change

- Design patterns facilitate future changes in the code
- If the code is unlikely to to change, the use of patterns is an example of overengineering

(1) Factory

Context: System that uses communication channels



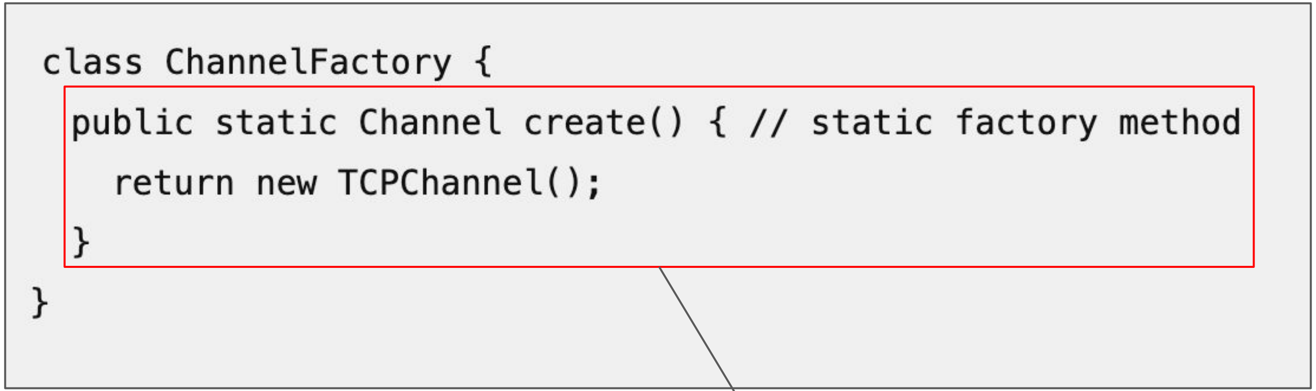
```
 void f() {
   TCPChannel c = new TCPChannel();
   ...
}

void g() {
   TCPChannel c = new TCPChannel();
   ...
}

void h() {
   TCPChannel c = new TCPChannel();
   ...
}
```

Problem

- Clients will need to use UDP instead of TCP
- How to parameterize the **new** calls?
- How to provide a design that facilitates this change?

Solution: Factory Pattern

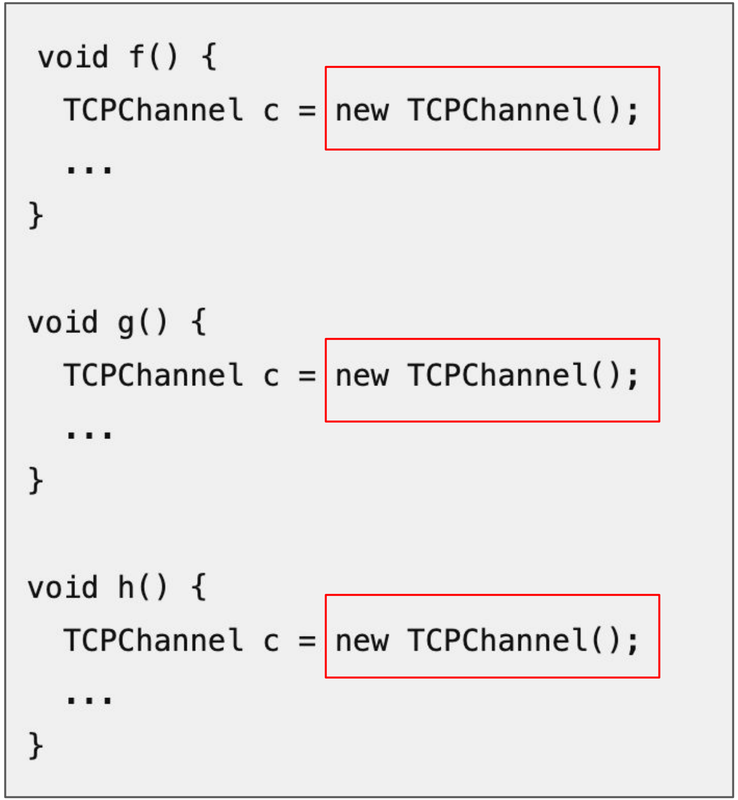
- Factory: method that centralizes the creation of objects



```
class ChannelFactory {
  public static Channel create() { // static factory method
    return new TCPChannel();
  }
}
```

single point of change if we
have to change to UDP

Without a Factory



```
 void f() {
   TCPChannel c = new TCPChannel();
   ...
}

void g() {
   TCPChannel c = new TCPChannel();
   ...
}

void h() {
   TCPChannel c = new TCPChannel();
   ...
}
```

Without a Factory

```
 void f() {
   TCPChannel c = new TCPChannel();
   ...
}

void g() {
   TCPChannel c = new TCPChannel();
   ...
}

void h() {
   TCPChannel c = new TCPChannel();
   ...
}
```

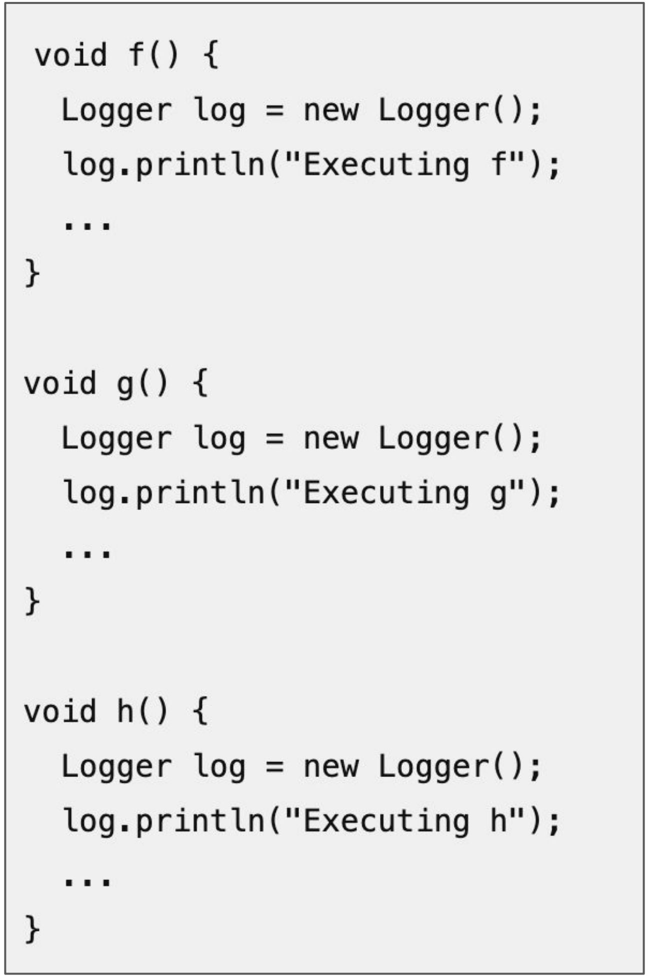
⇒

With a Factory

```
void f() {
   Channel c = ChannelFactory.create();
   ...
}

void g() {
   Channel c = ChannelFactory.create();
   ...
}

void h() {
   Channel c = ChannelFactory.create();
   ...
}
```

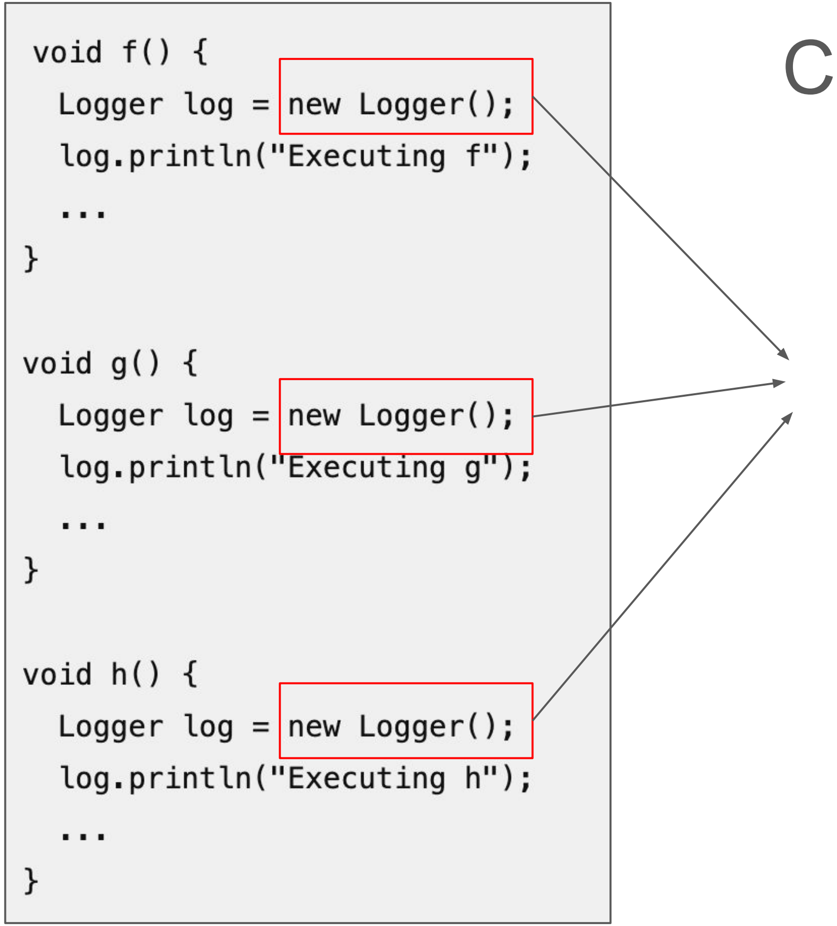
(2) Singleton

Context: Logger class



```
 void f() {
   Logger log = new Logger();
   log.println("Executing f");
   ...
}

void g() {
   Logger log = new Logger();
   log.println("Executing g");
   ...
}

void h() {
   Logger log = new Logger();
   log.println("Executing h");
   ...
}
```

Context: Logger class

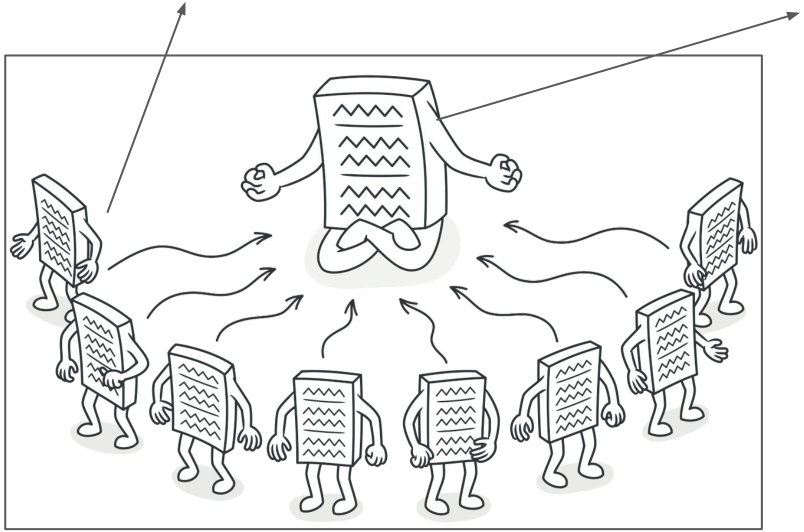


```
void f() {
  Logger log = new Logger();
  log.println("Executing f");
  ...
}

void g() {
  Logger log = new Logger();
  log.println("Executing g");
  ...
}

void h() {
  Logger log = new Logger();
  log.println("Executing h");
  ...
}
```

Problem: Every method uses its own instance of Logger

Problem

- Every operation should be logged in the same file
- How to make clients use the same Logger instance?



https://refactoring.guru/design-patterns/singleton

Solution: Singleton Pattern

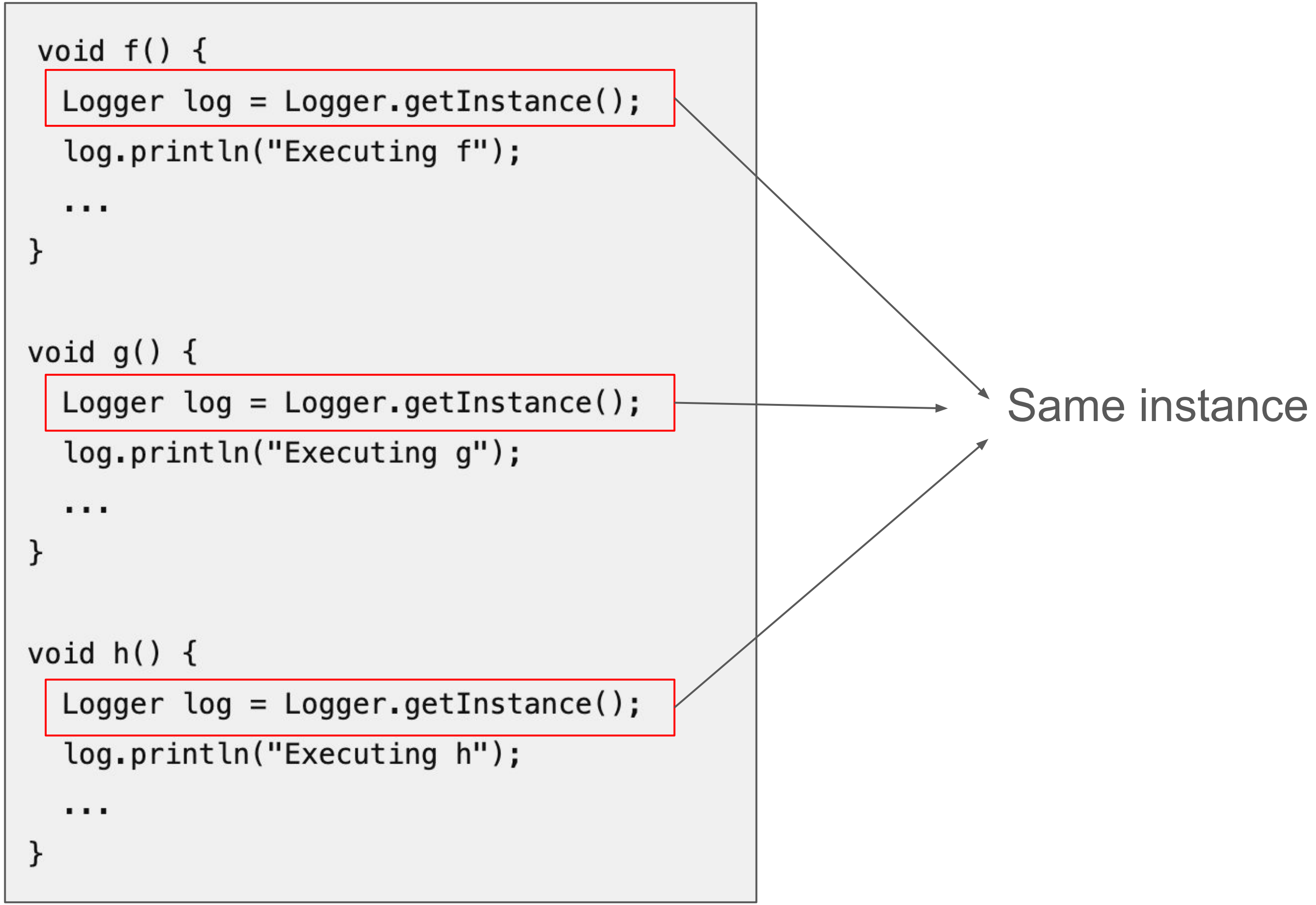
- Transform the Logger class into a Singleton
- Singleton: class that has at most one instance

```
class Logger {

 private Logger() {} // prohibits new Logger() in clients

 private static Logger instance; // single instance

 public static Logger getInstance() {
   if (instance == null)  // 1st time getInstance is called
      instance = new Logger();
   return instance;
 }

 public void println(String msg) {
   // logs msg to console, but it could be to a file
   System.out.println(msg);
 }
}
```

```
class Logger {

  private Logger() {} // prohibits new Logger() in clients

  private static Logger instance; // single instance

  public static Logger getInstance() {
    if (instance == null)  // 1st time getInstance is called
      instance = new Logger();
    return instance;
  }

  public void println(String msg) {
    // logs msg to console, but it could be to a file
    System.out.println(msg);
  }
}
```

```
class Logger {

  private Logger() {} // prohibits new Logger() in clients

  private static Logger instance; // single instance

  public static Logger getInstance() {
1   if (instance == null)  // 1st time getInstance is called
      instance = new Logger();
2   return instance;
  }

  public void println(String msg) {
    // logs msg to console, but it could be to a file
    System.out.println(msg);
  }
}
```

```
class Logger {

  private Logger() {} // prohibits new Logger() in clients

  private static Logger instance; // single instance

  public static Logger getInstance() {
    if (instance == null)  // 1st time getInstance is called
      instance = new Logger();
    return instance;
  }

  public void println(String msg) {
    // logs msg to console, but it could be to a file
    System.out.println(msg);
  }
}
```

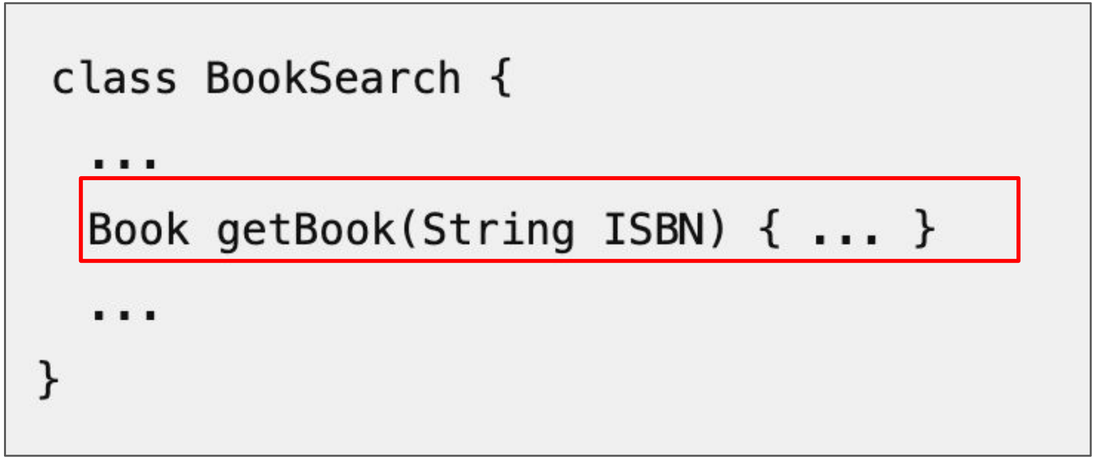


```
void f() {
  Logger log = Logger.getInstance();
  log.println("Executing f");
  ...
}

void g() {
  Logger log = Logger.getInstance();
  log.println("Executing g");
  ...
}

void h() {
  Logger log = Logger.getInstance();
  log.println("Executing h");
  ...
}
```

Same instance

(3) Proxy

Context: Book search function



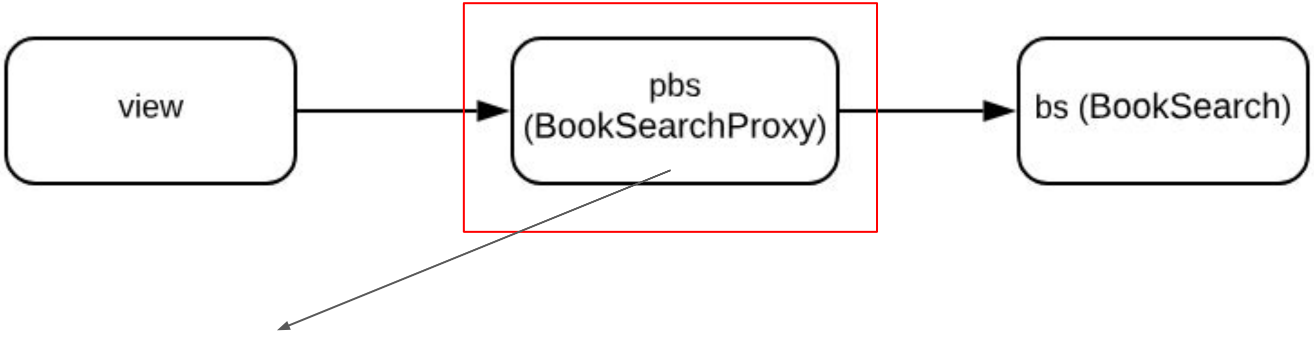
```
class BookSearch {
  ...
  Book getBook(String ISBN) { ... }
  ...
}
```

Problem: use a cache to improve performance

- If "book in cache"
  - return the book immediately
  - otherwise, continue with the search
- But we don't want to change the code of BookSearch
  - It is already working
  - There is a developer who maintains it

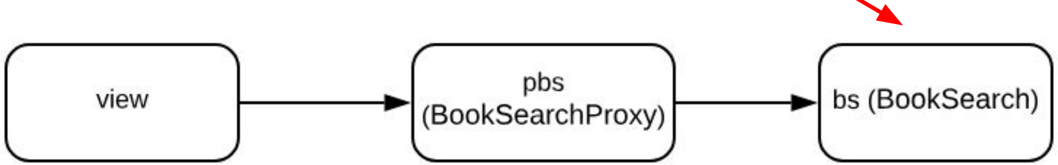
Solution: Proxy Pattern

- Proxy: intermediary object between client and a base object
- Clients no longer speak directly with the base object
- They have to go through the proxy



cache logic

```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```

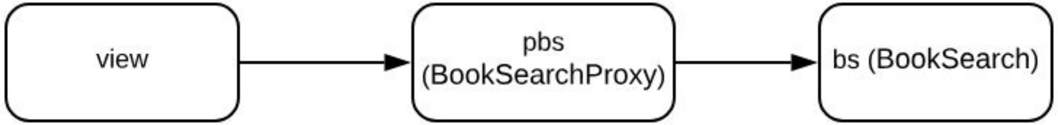


view

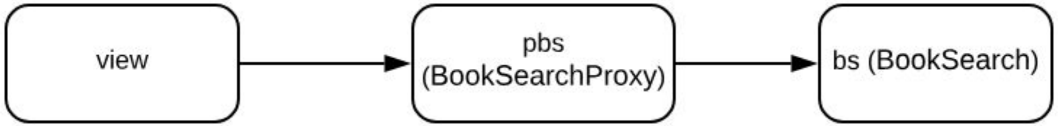
pbs (BookSearchProxy)

bs (BookSearch)

```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```

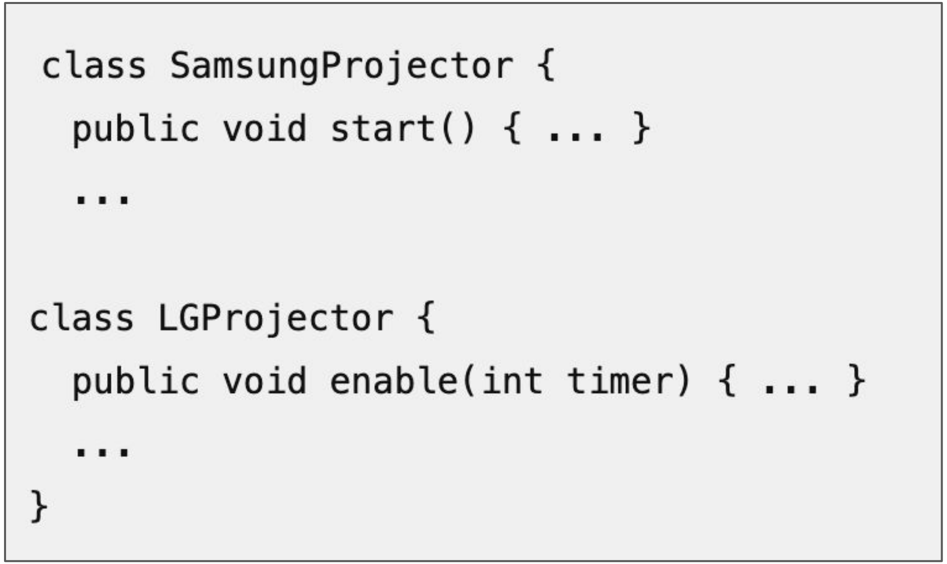


```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```



(4) Adapter

Context: Multimedia Projectors Control System

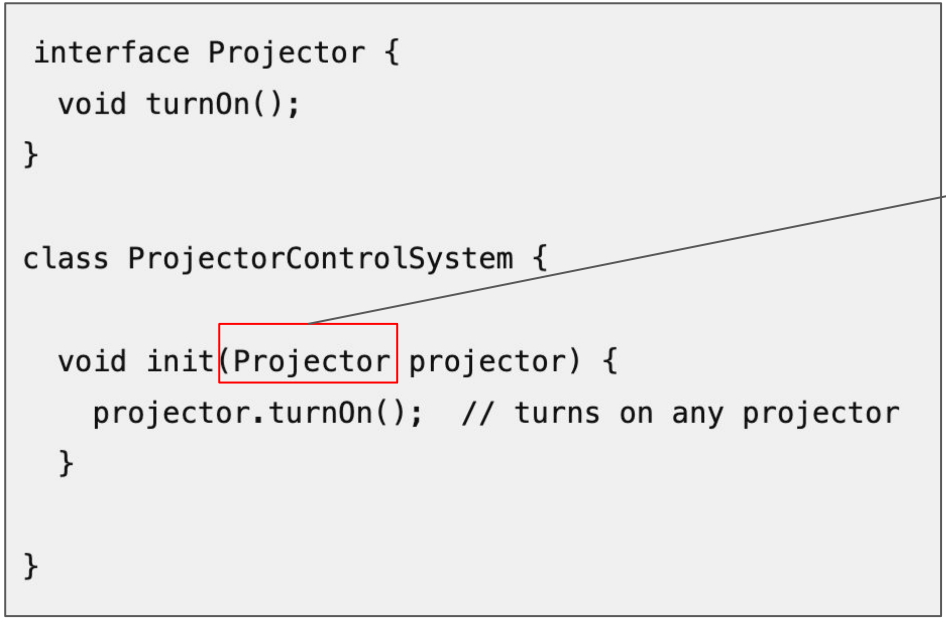


```
class SamsungProjector {
  public void start() { ... }
  ...

class LGProjector {
  public void enable(int timer) { ... }
  ...
}
```

Provided by projector manufacturers
⇒ we can’t edit them!

Problem

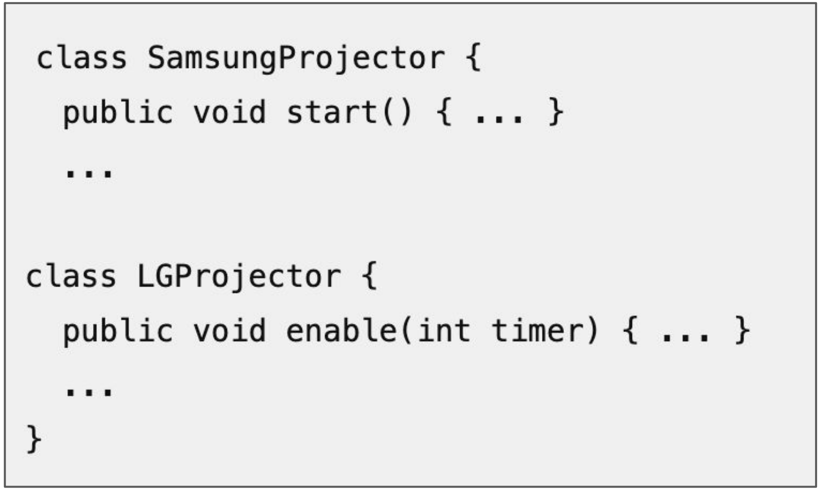
- In the multimedia control system, we would like to use a single `Projector` interface



```
 interface Projector {
   void turnOn();
}

class ProjectorControlSystem {

  void init(Projector projector) {
    projector.turnOn();  // turns on any projector
  }

}
```

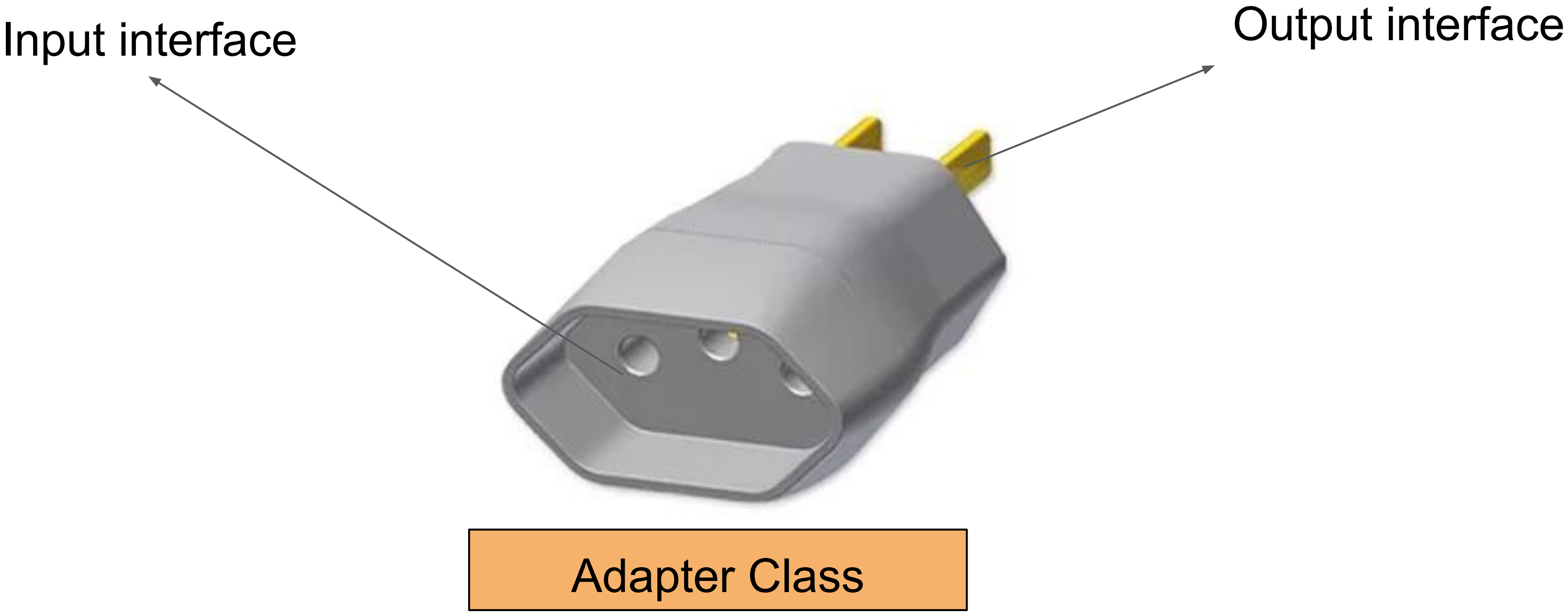
We'd like to use this interface, without worrying about the classes that implement it

Problem



```
 class SamsungProjector {
   public void start() { ... }
   ...

class LGProjector {
   public void enable(int timer) { ... }
   ...
}
```

- Classes (drivers) from manufacturers
- We can't edit them to implement the Projector interface

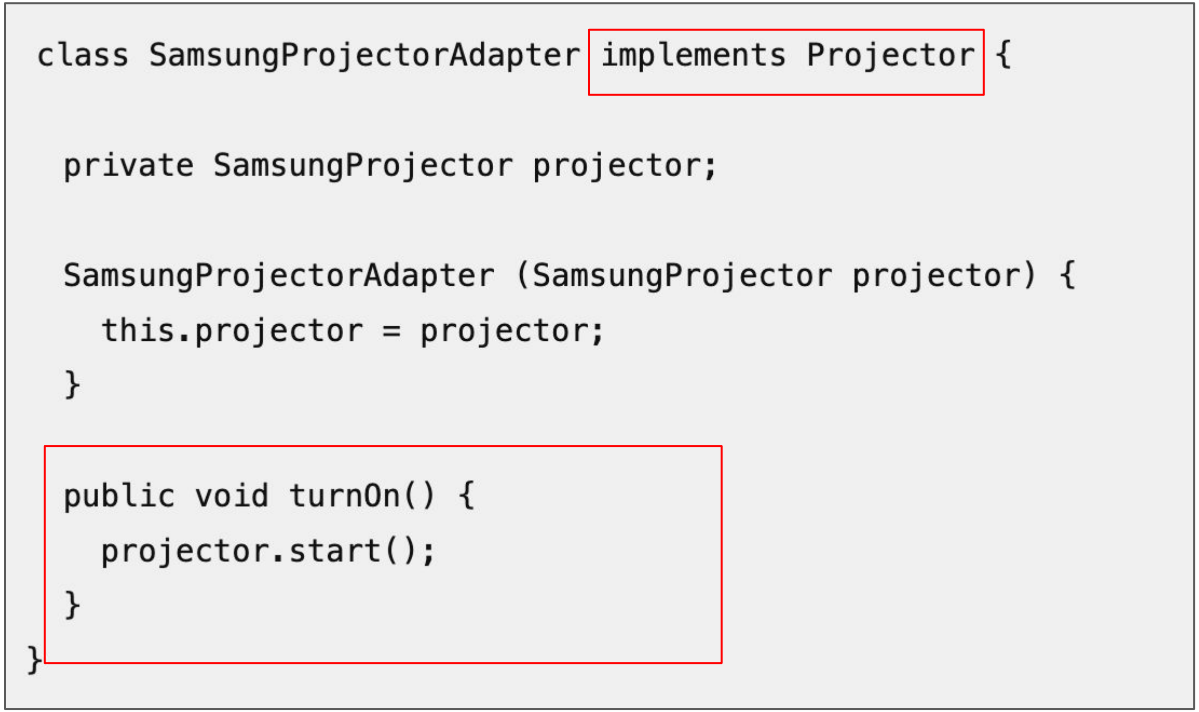


Input interface

Output interface

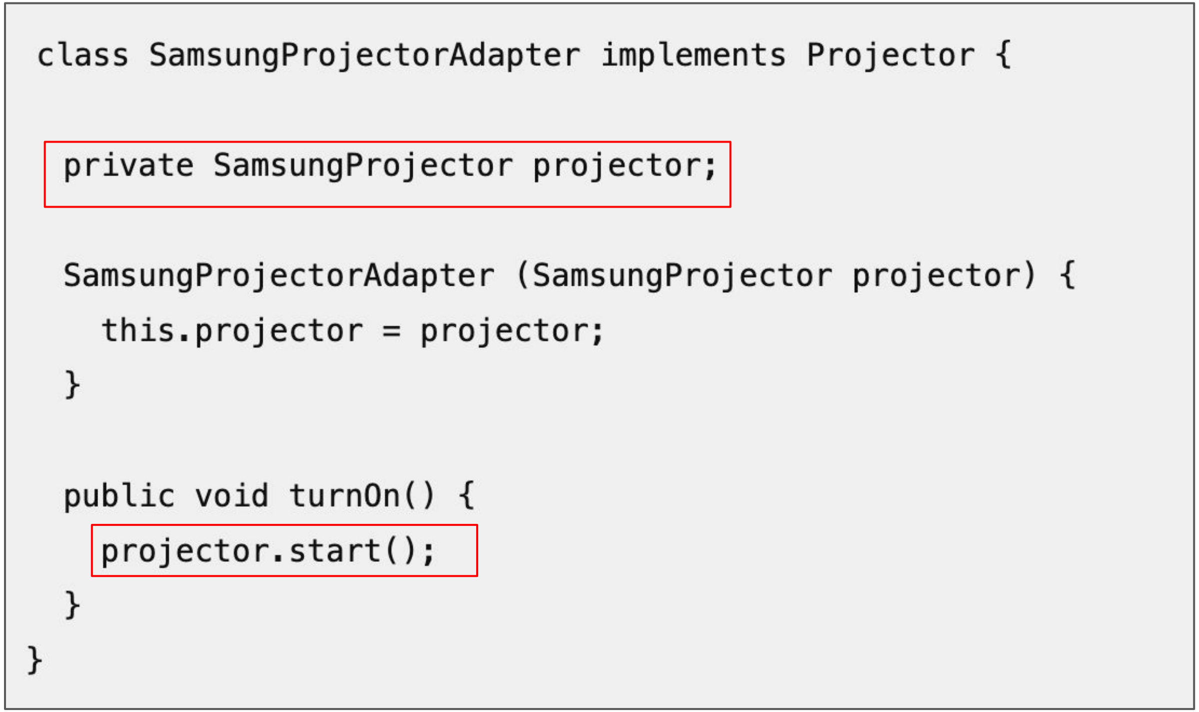
Adapter Class

Solution: Adapter class



```
class SamsungProjectorAdapter implements Projector {

  private SamsungProjector projector;

  SamsungProjectorAdapter (SamsungProjector projector) {
    this.projector = projector;
  }

  public void turnOn() {
    projector.start();
  }
}
```

Solution: Adapter class



```
class SamsungProjectorAdapter implements Projector {

 private SamsungProjector projector;

 SamsungProjectorAdapter (SamsungProjector projector) {
   this.projector = projector;
 }

 public void turnOn() {
   projector.start();
 }
}
```

SamsungProjector

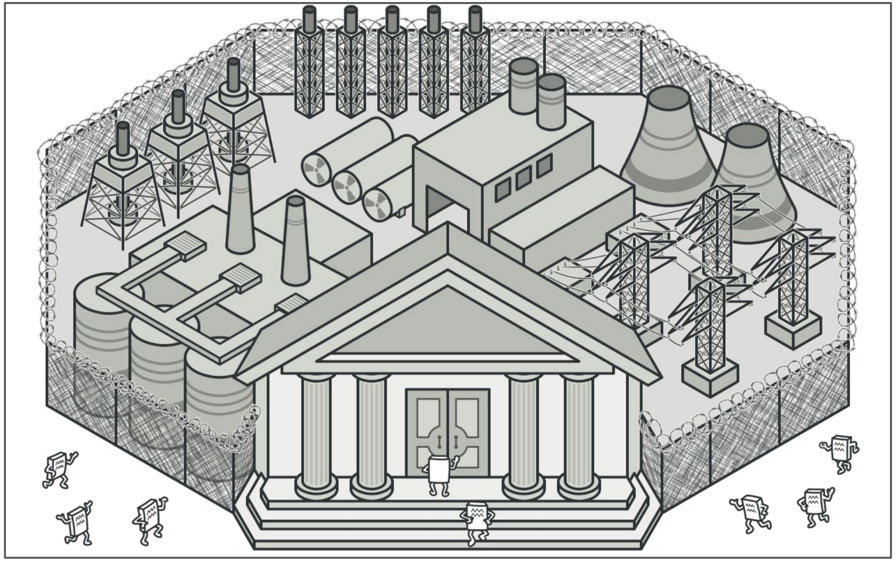
Projector

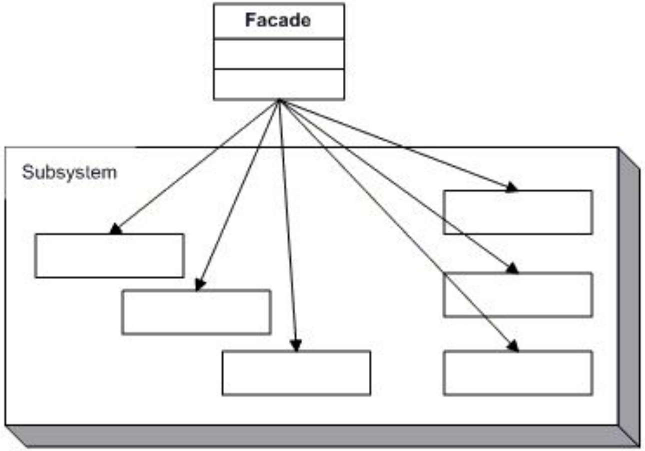
SamsungProjectorAdapter

(5) Facade

Context, Problem & Solution

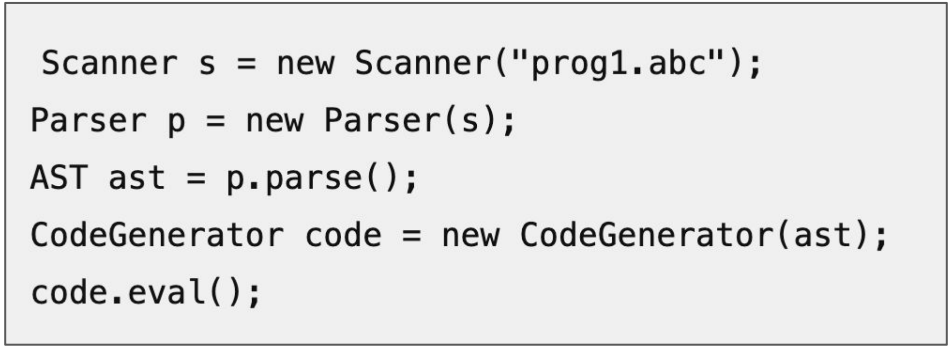
- Context: suppose a module M used by several other modules
- Problem: M's interface is complex
    - Clients are complaining that it's hard to use it
- Solution: create a simpler interface for M, called Facade





https://refactoring.guru/design-patterns/facade

Example: Interpreter



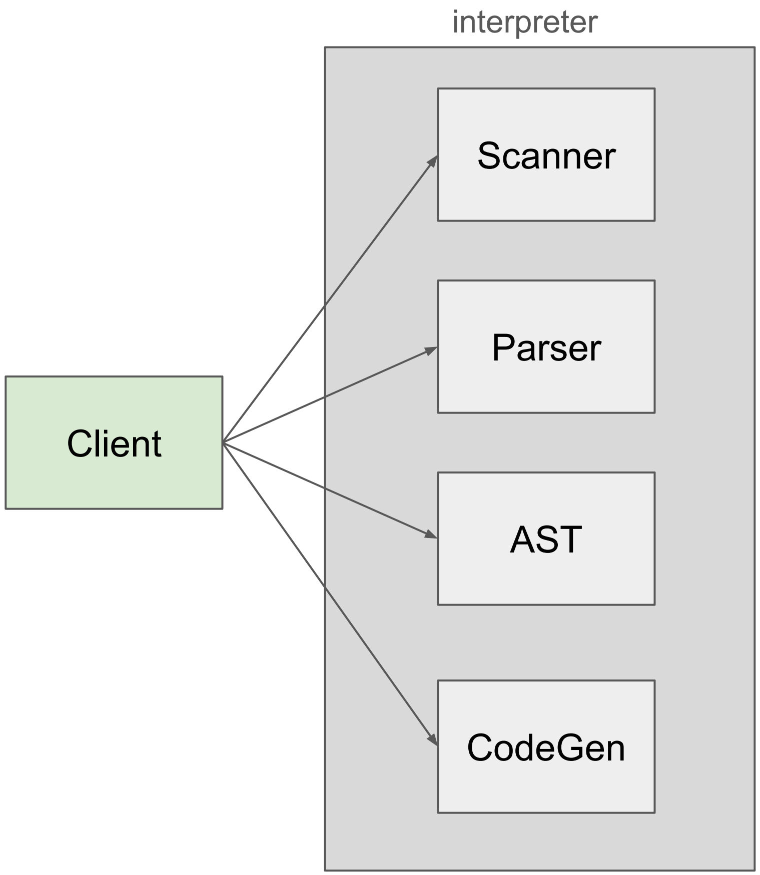
```
 Scanner s = new Scanner("prog1.abc");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

```
Scanner s = new Scanner("prog1.abc");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

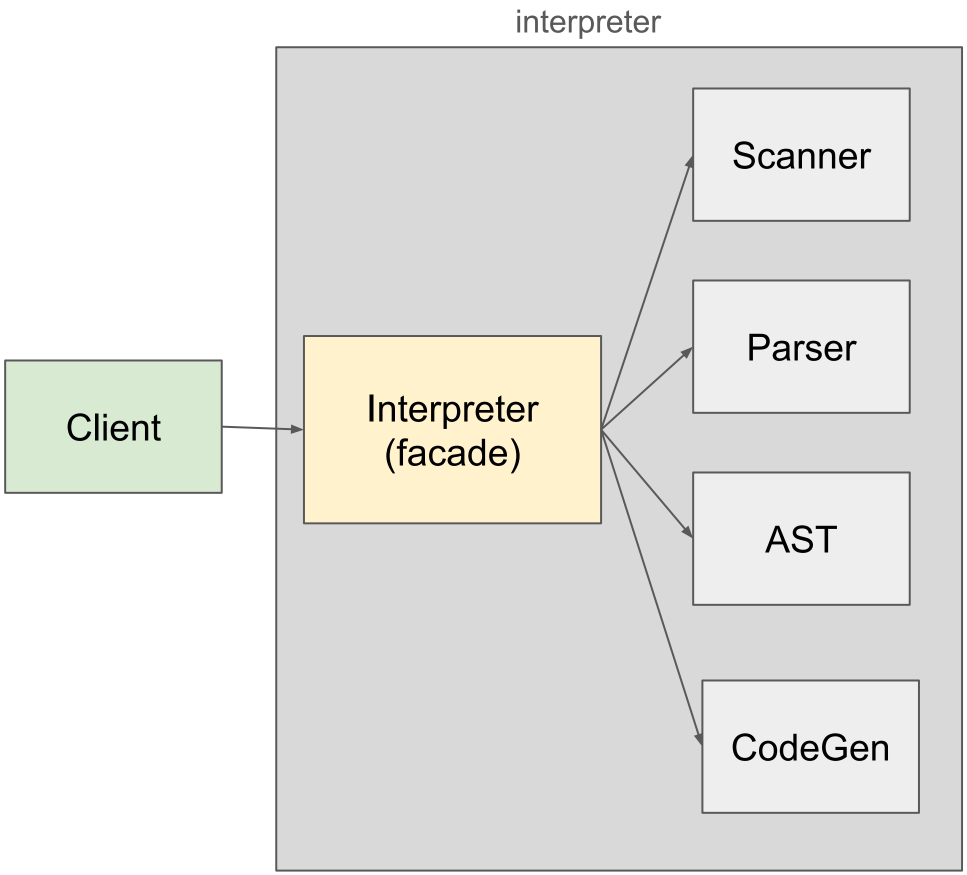
```
new Interpreter("prog1.abc").eval();
```

Facade: very simple interface

Without a Facade



With a Facade



Exercises

1. Singleton is one of the most controversial design patterns. Erich Gamma, one of the GoF authors, has even recommended in an interview that it should be removed from the catalog:

> Erich: When discussing which patterns to drop, we found that we still love them all. (Not really—I'm in favor of dropping Singleton…)

Explain why Singletons can cause problems if not used properly.

2. Why doesn't the Singleton implementation shown in the slides work with concurrent systems (multi-thread)? Describe a bug that can occur when it's used with this type of system.

3. In addition to performance optimization via cache, as commented in the slides, describe three other non-functional requirements that can be implemented in a Proxy.

4. Answer the following questions that correlate design patterns with design properties (cohesion, coupling, etc.) and design principles (SOLID: single responsibility, open/closed, Liskov, interface segregation, and dependency inversion):

(a) Which design property is improved with a proxy?

(b) Which design principle is followed when we use a proxy?

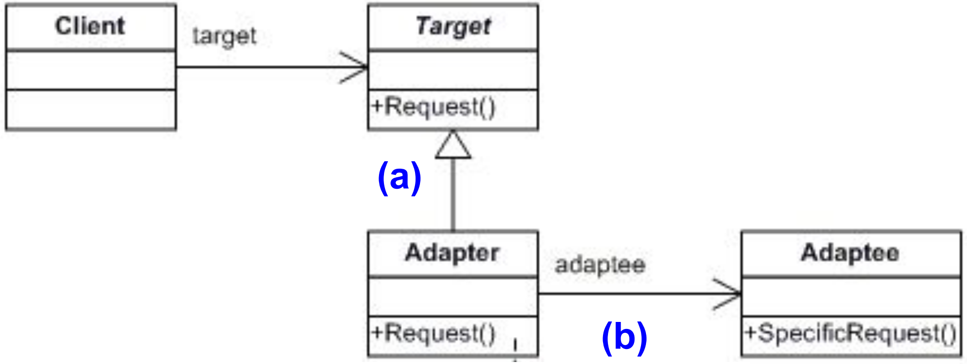
(c) A facade improves which design property?

(d) Adapters are related to which design principle?

(e) If designed incorrectly, facades violate which design principle?

5. The following class diagram illustrates the Adapter pattern.

- In UML, what type of relationship is (a)? And what type is (b)?
- Map `Target`, `Adapter`, and `Adaptee` to the interfaces and classes from the Projector example discussed in the slides.

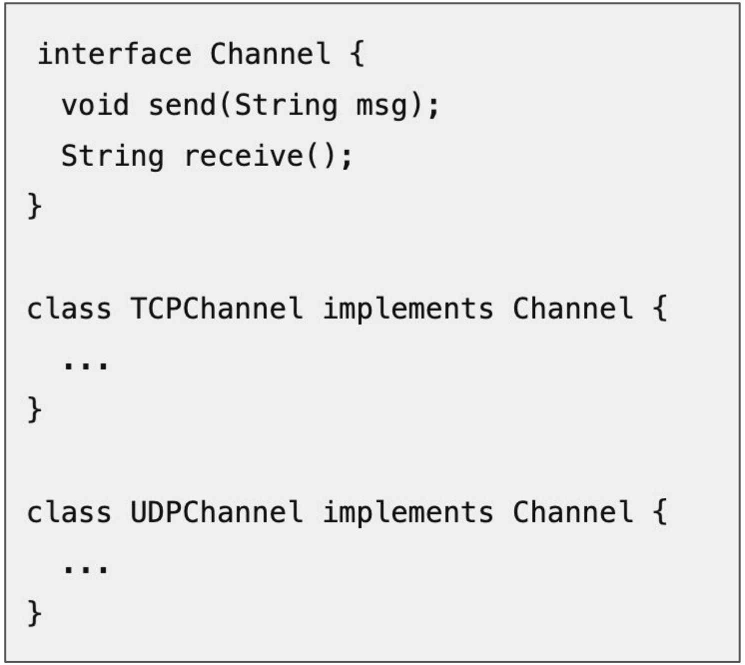


Source: GoF book

(6) Decorator

Context: System that uses communication channels

(Used before to explain the Factory pattern)



```
interface Channel {
  void send(String msg);
  String receive();
}

class TCPChannel implements Channel {
  ...
}

class UDPChannel implements Channel {
  ...
}
```

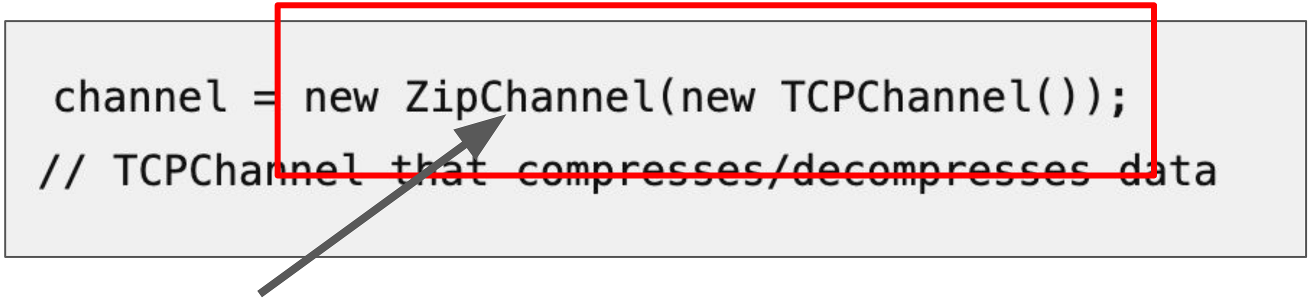
Problem: We need to add extra functionalities to channels

- Default channels (TCP, UDP) are not enough
- We also need channels with:
    - Data compression/decompression
    - Buffers
    - Logging
    - etc

Solution: Decorator Pattern

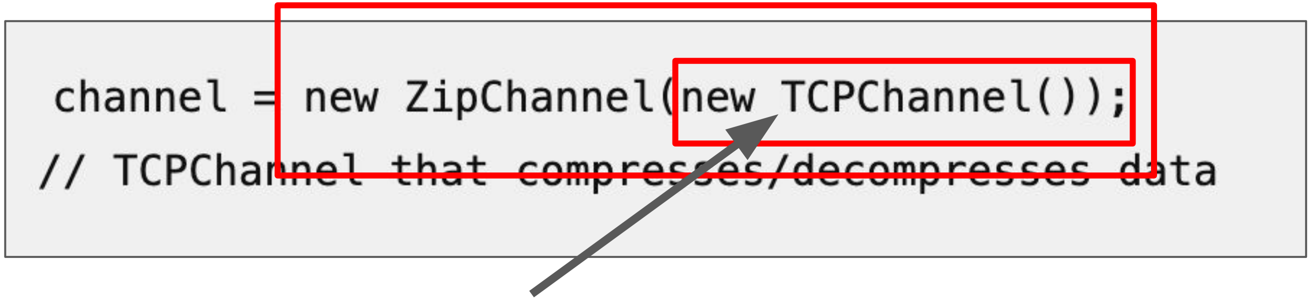
- Solves this problem through composition, instead of inheritance
- Thus, without creating an excessive number of classes

Example



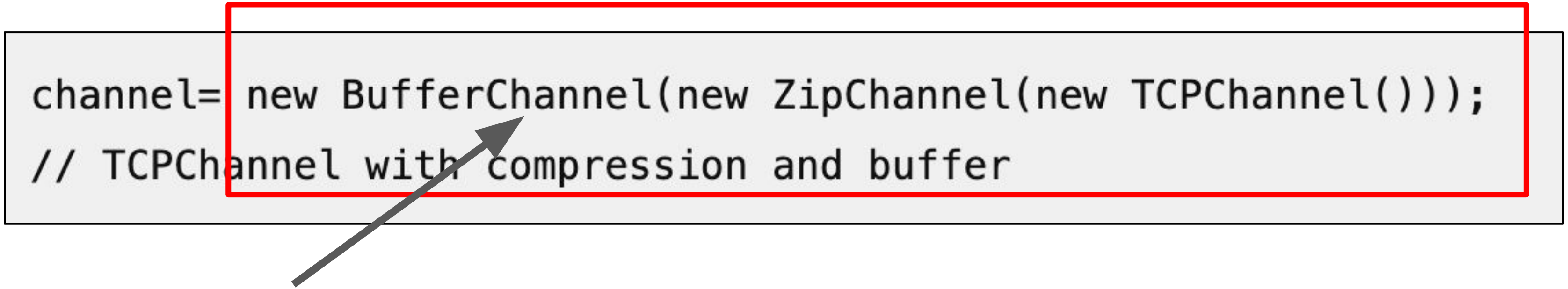
```
channel = new ZipChannel(new TCPChannel());
// TCPChannel that compresses/decompresses data
```

Example



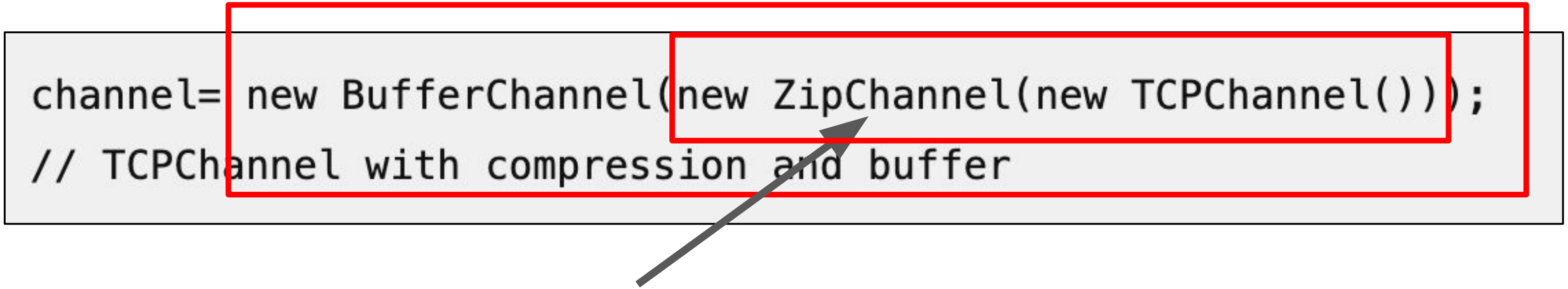
```
channel = new ZipChannel(new TCPChannel());
// TCPChannel that compresses/decompresses data
```

Another example



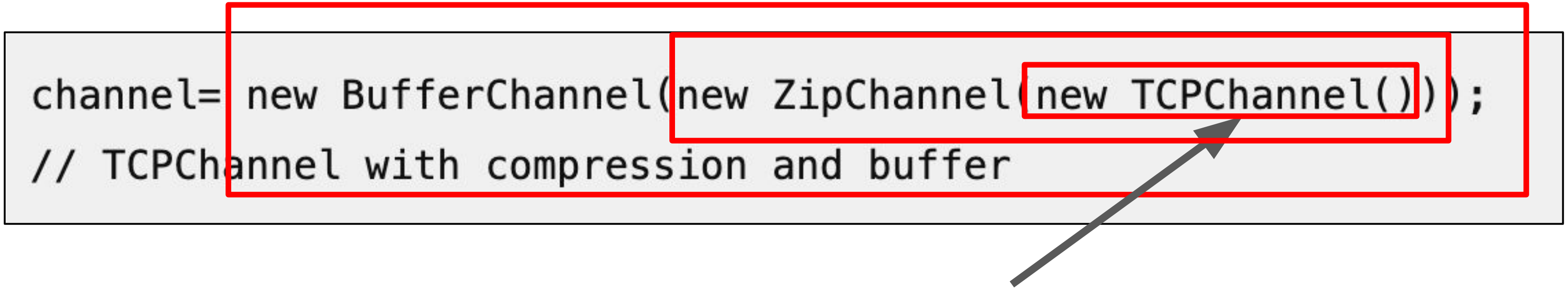
```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

Another example

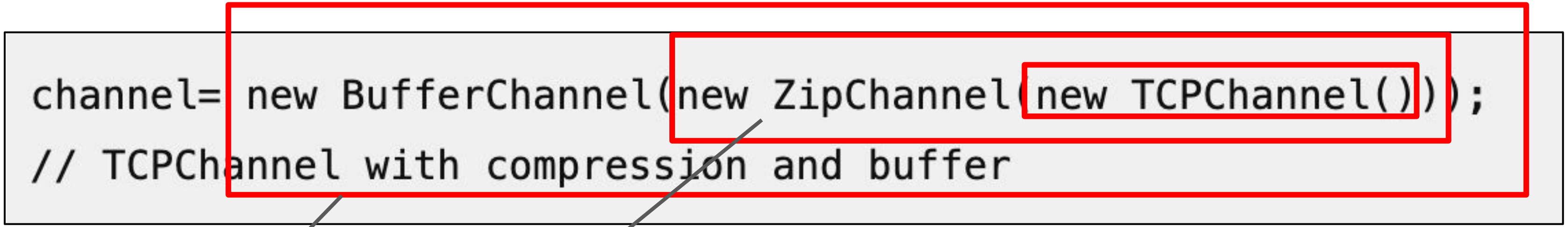


```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

Another example



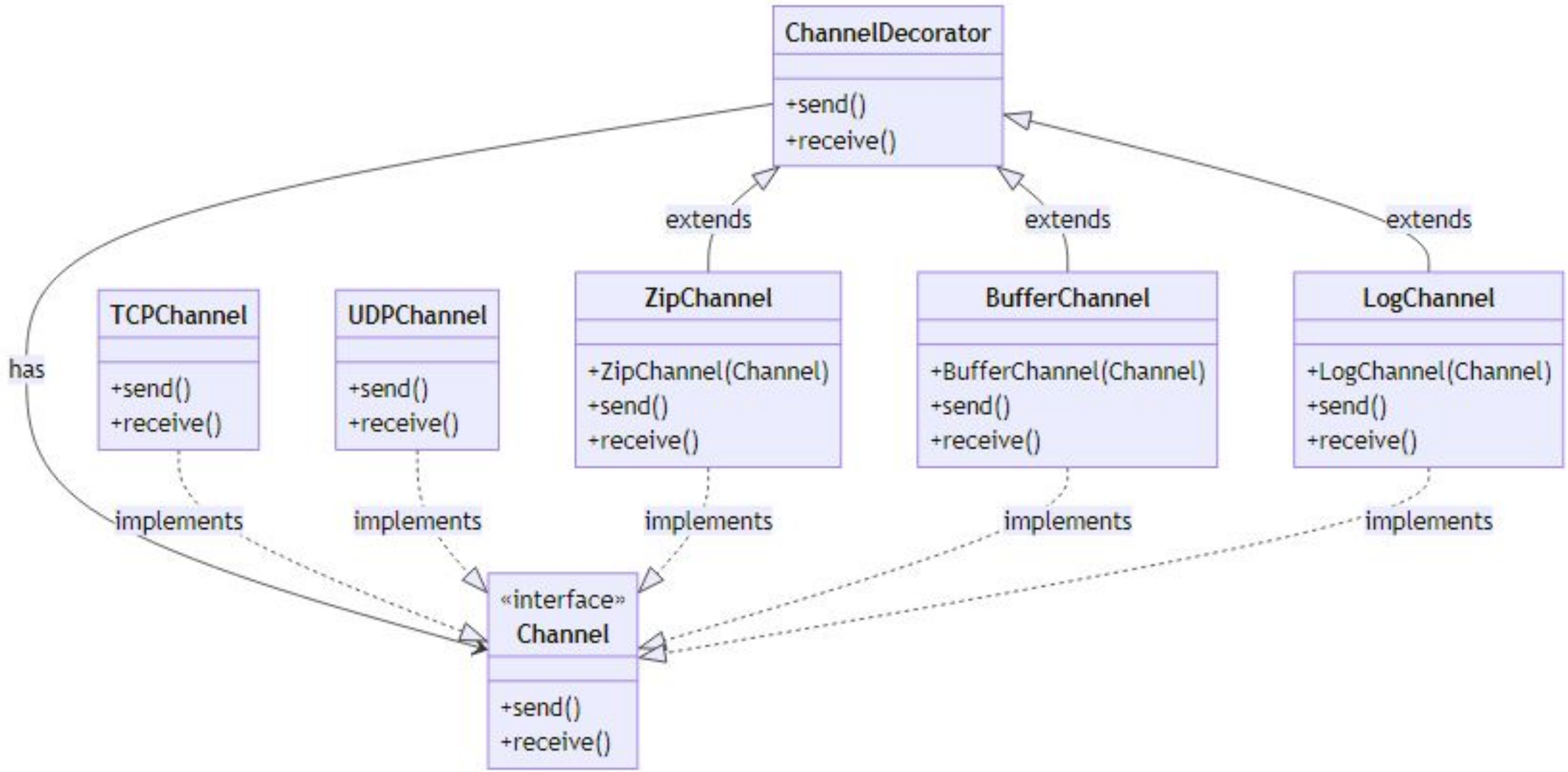
```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```



```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

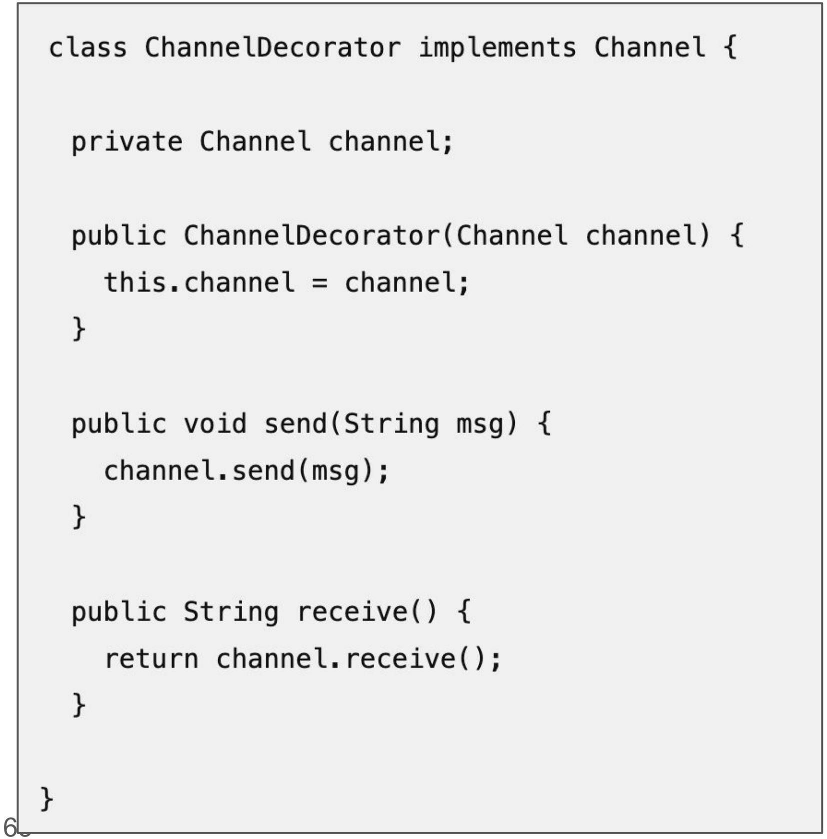
Inside one box, there is another box, which has another box... until reaching the gift. That is, until reaching TCPChannel or UDPChannel.

Decorator Implementation



ChannelDecorator
+send()
+receive()
extends
extends
extends
has
TCPChannel
+send()
+receive()
UDPChannel
+send()
+receive()
ZipChannel
+ZipChannel(Channel)
+send()
+receive()
BufferChannel
+BufferChannel(Channel)
+send()
+receive()
LogChannel
+LogChannel(Channel)
+send()
+receive()
implements
implements
implements
implements
implements
«interface»
Channel
+send()
+receive()

ChannelDecorator



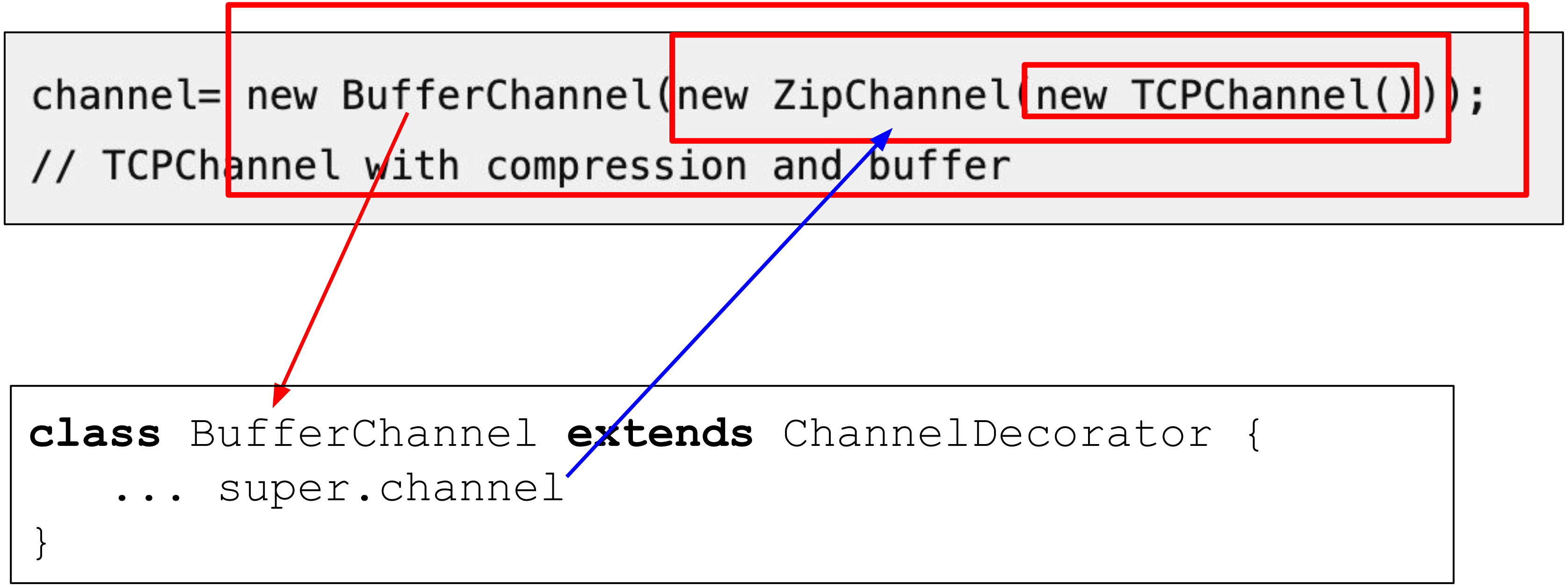
```
class ChannelDecorator implements Channel {

  private Channel channel;

  public ChannelDecorator(Channel channel) {
    this.channel = channel;
  }

  public void send(String msg) {
    channel.send(msg);
  }

  public String receive() {
    return channel.receive();
  }

}
```

Decorators are subclasses of this class

ZipChannel Implementation

```
class ZipChannel extends ChannelDecorator {

  public ZipChannel(Channel c) {
    super(c);
  }

  public void send(String msg) {
    "compress message msg"
    super.send(msg);
  }

  public String receive() {
    String msg = super.receive();
    "decompress message msg"
    return msg;
  }

}
```

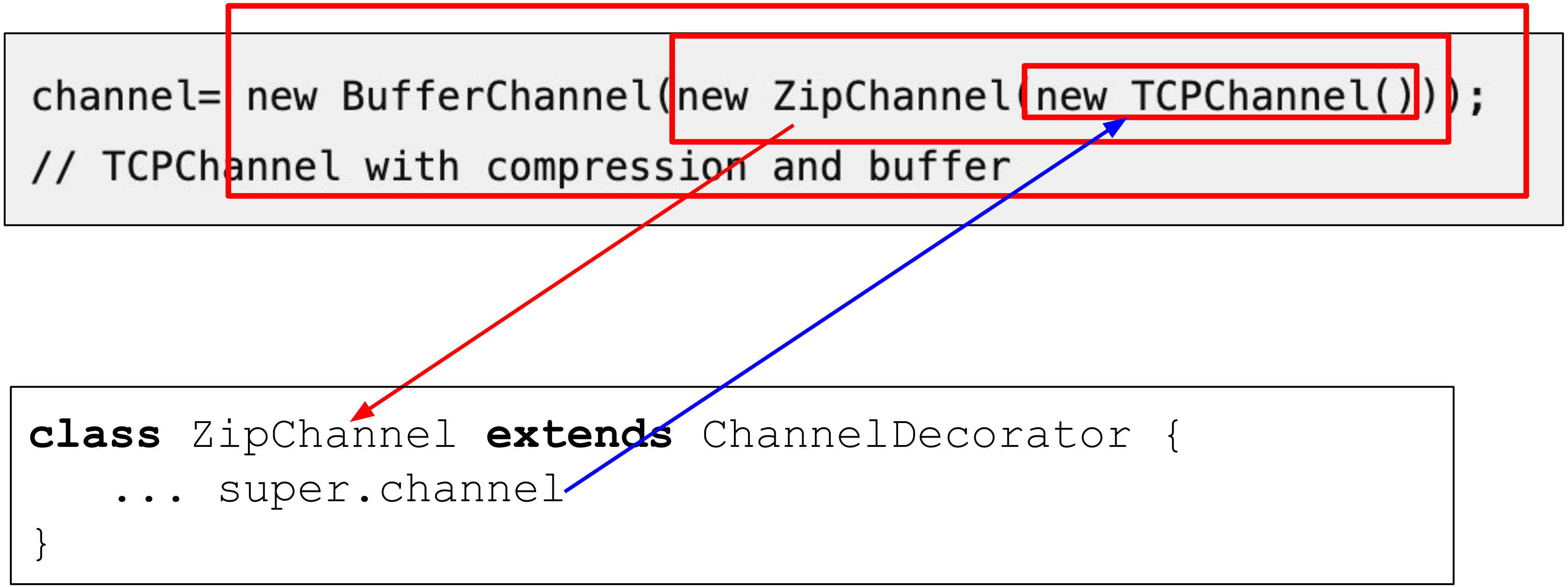
Example



```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

```
class BufferChannel extends ChannelDecorator {
    ... super.channel
}
```

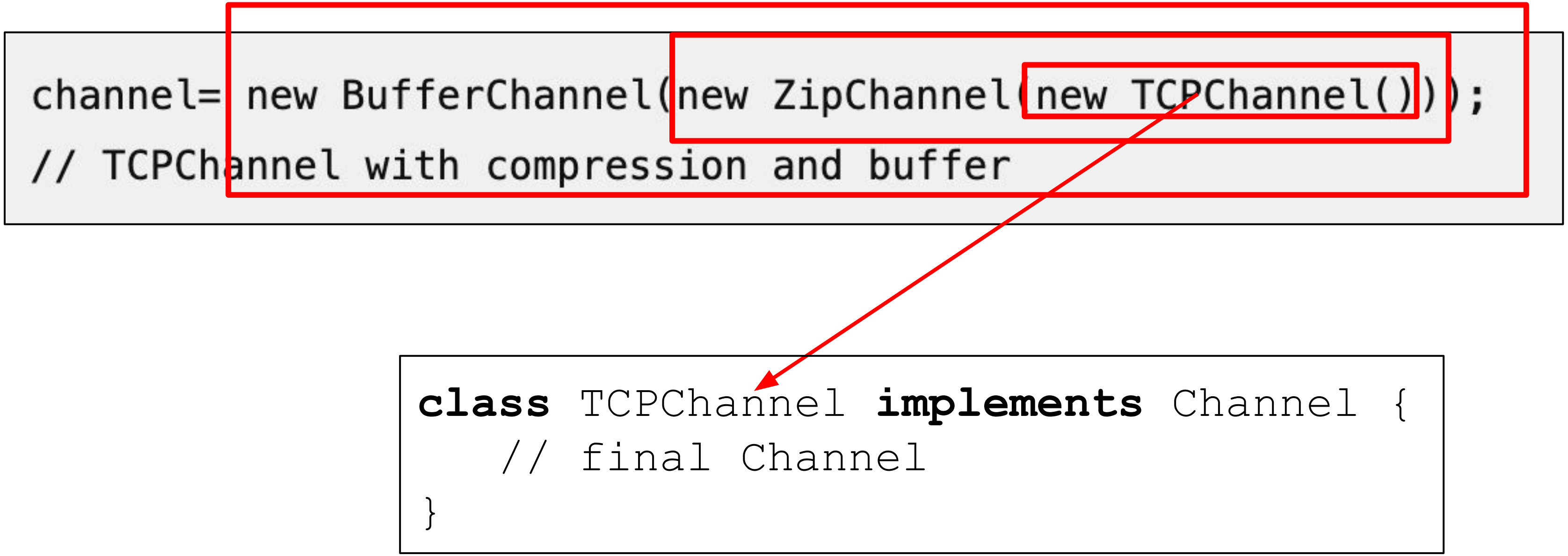
Another example



```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

```
class ZipChannel extends ChannelDecorator {
    ... super.channel
}
```

Another example

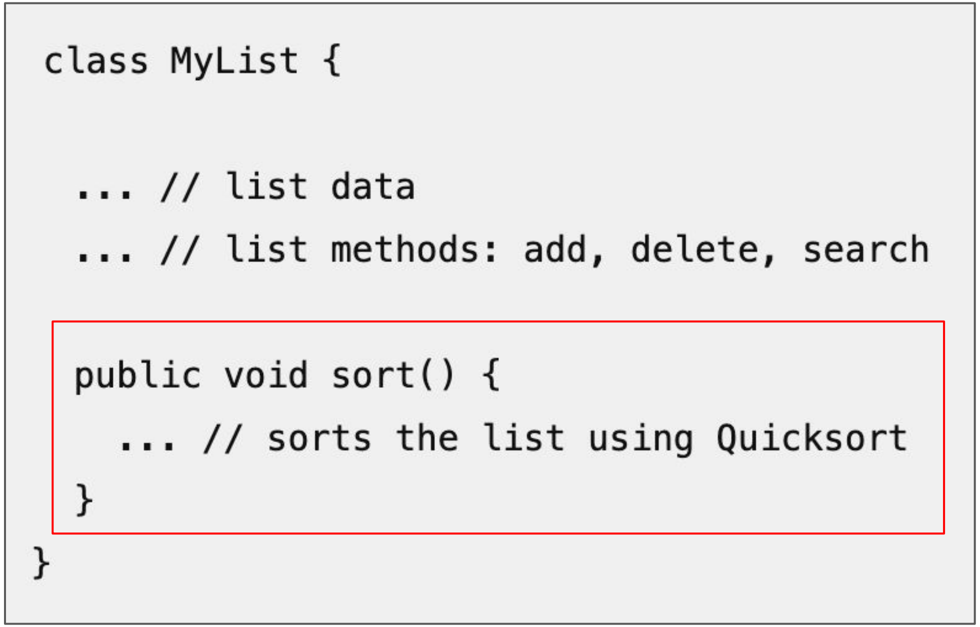


```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```

```
class TCPChannel implements Channel {
    // final Channel
}
```

(7) Strategy

Context: Data Structures Library



```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  public void sort() {
    ... // sorts the list using Quicksort
  }
}
```

Problem

- MyList class is **not** open to extensions
- Example: other sorting algorithm (ShellSort, HeapSort, etc)

Solution: Strategy Pattern

- Goal: parametrize the algorithms used by a class
- Make a class open to new algorithms
- In the example: new sorting algorithms

Step #1: Create a hierarchy of strategies

(strategy = algorithm)

```
abstract class SortStrategy {
  abstract void sort(MyList list);
}

class QuickSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}

class ShellSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}
```

Step #2: Modify MyList to use the strategy hierarchy

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }
}
```

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }
}
```

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }
}
```

(8) Observer

Context: Weather Station System

- Two main classes: Temperature and Thermometer
- Several thermometers: digital, analog, web, mobile, etc
- If the temperature changes, the thermometers should be updated

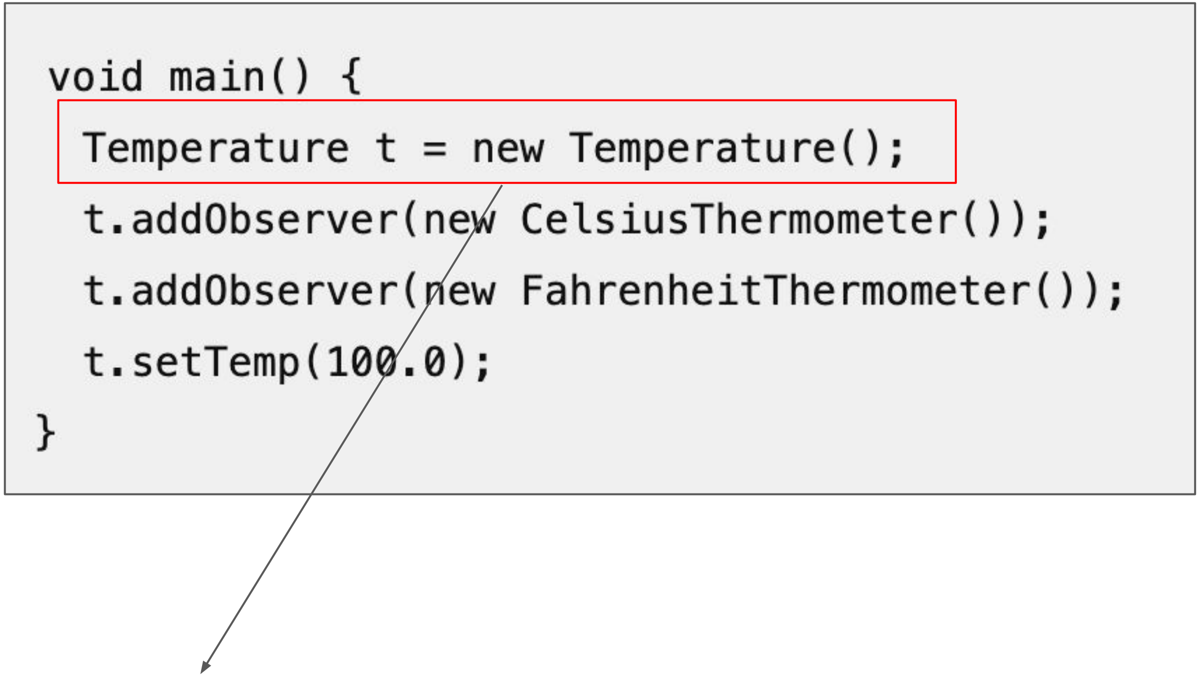
Problem

- We don't want to couple Temperature to Thermometers
- Reasons:
  - Make domain class (model) independent of view classes (or UI classes)
  - Make it easy to add a new type of thermometer

Solution: Observer Pattern

- Implements a one-to-many relationship between:
  - Subject (Temperature)
  - Observers (Thermometers)
- When the Subject changes, Observers are notified
- Subject doesn't know the concrete type of its Observers

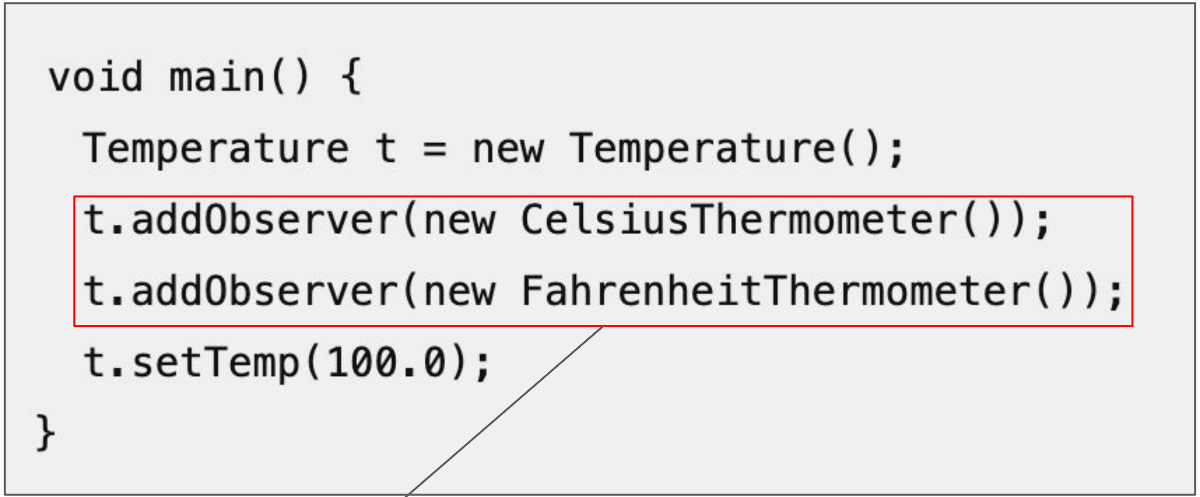
Example



```
void main() {
  Temperature t = new Temperature();
  t.addObserver(new CelsiusThermometer());
  t.addObserver(new FahrenheitThermometer());
  t.setTemp(100.0);
}
```

Subject

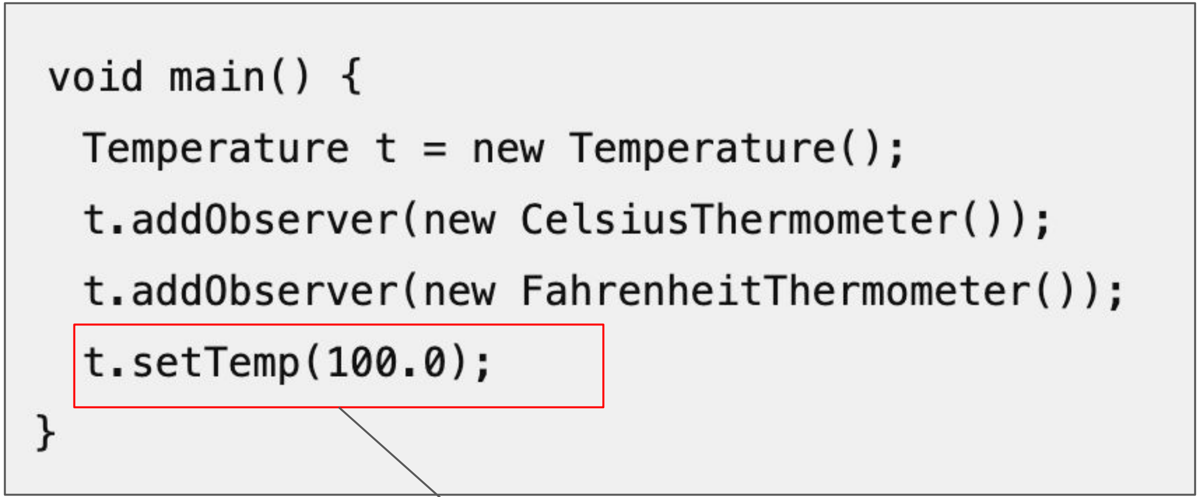
Example



```
void main() {
  Temperature t = new Temperature();
  t.addObserver(new CelsiusThermometer());
  t.addObserver(new FahrenheitThermometer());
  t.setTemp(100.0);
}
```

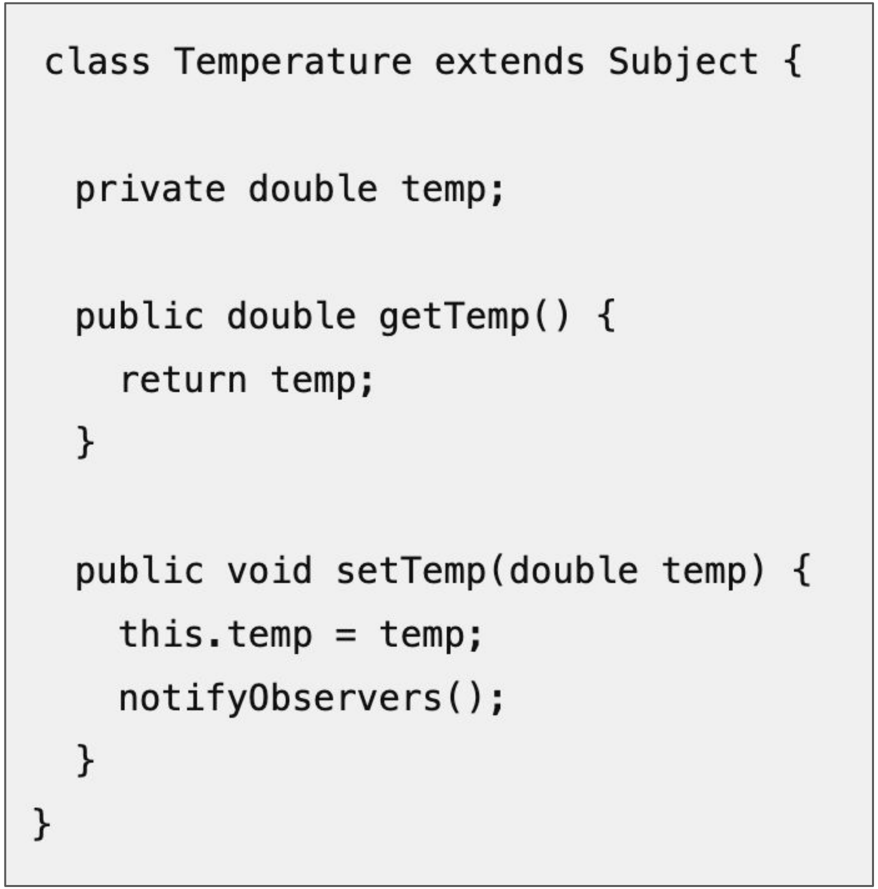
Two observers

Example

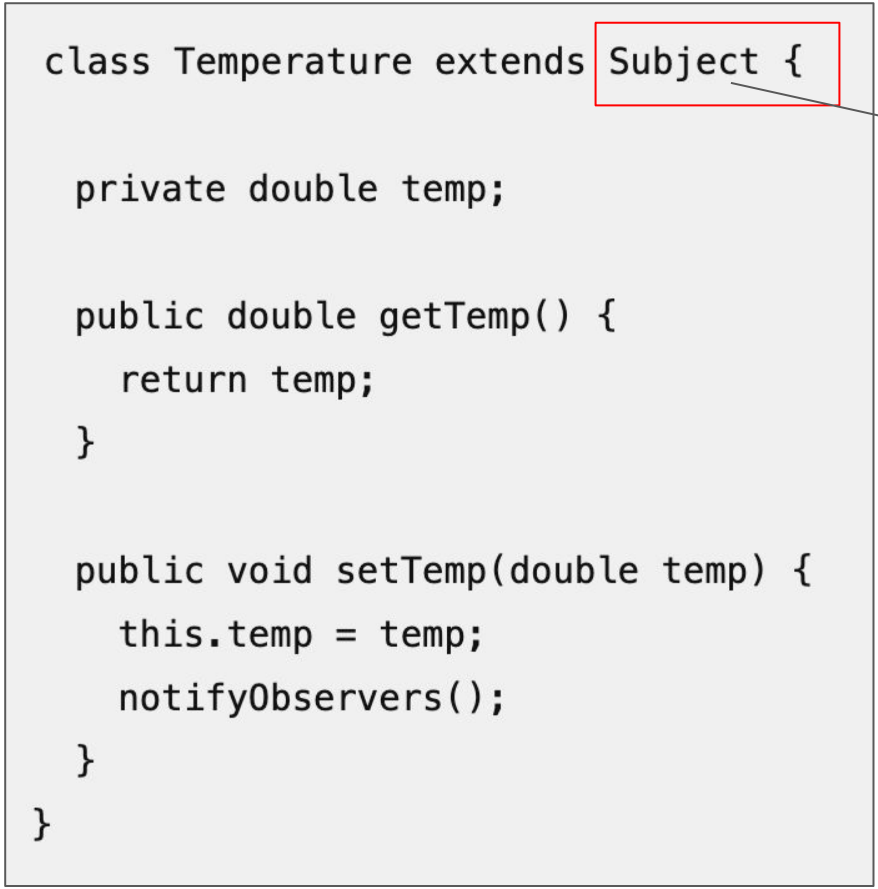


```
void main() {
  Temperature t = new Temperature();
  t.addObserver(new CelsiusThermometer());
  t.addObserver(new FahrenheitThermometer());
  t.setTemp(100.0);
}
```

Notifies observers

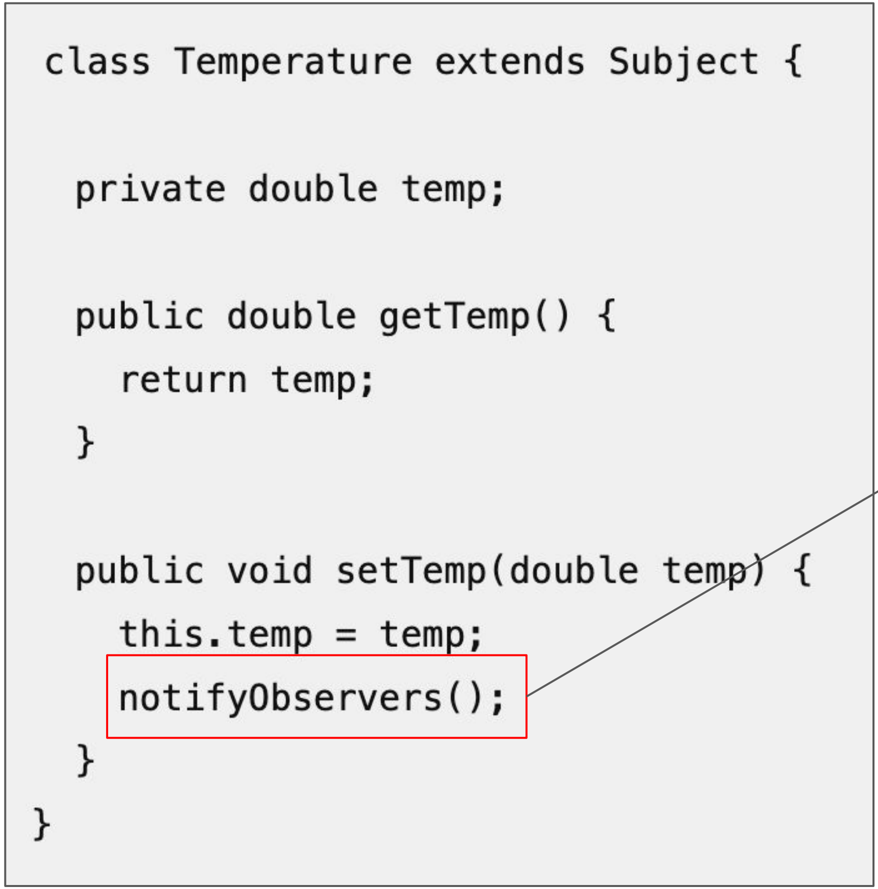


```
class Temperature extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

  public void setTemp(double temp) {
    this.temp = temp;
    notifyObservers();
  }
}
```



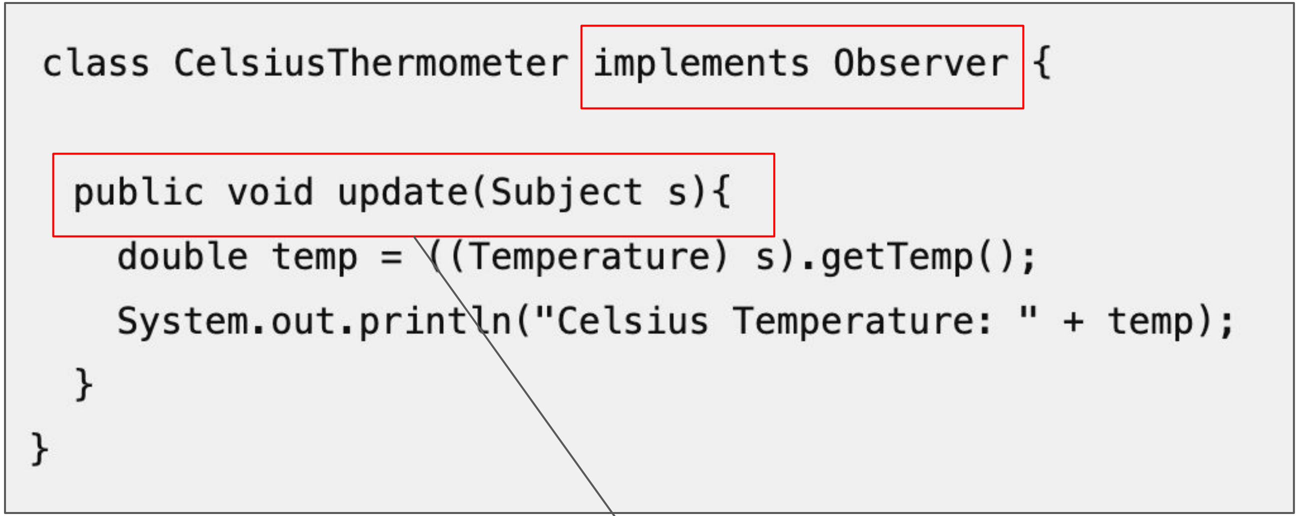
```
class Temperature extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

  public void setTemp(double temp) {
    this.temp = temp;
    notifyObservers();
  }
}
```

Class that implements addObservers and notifyObservers



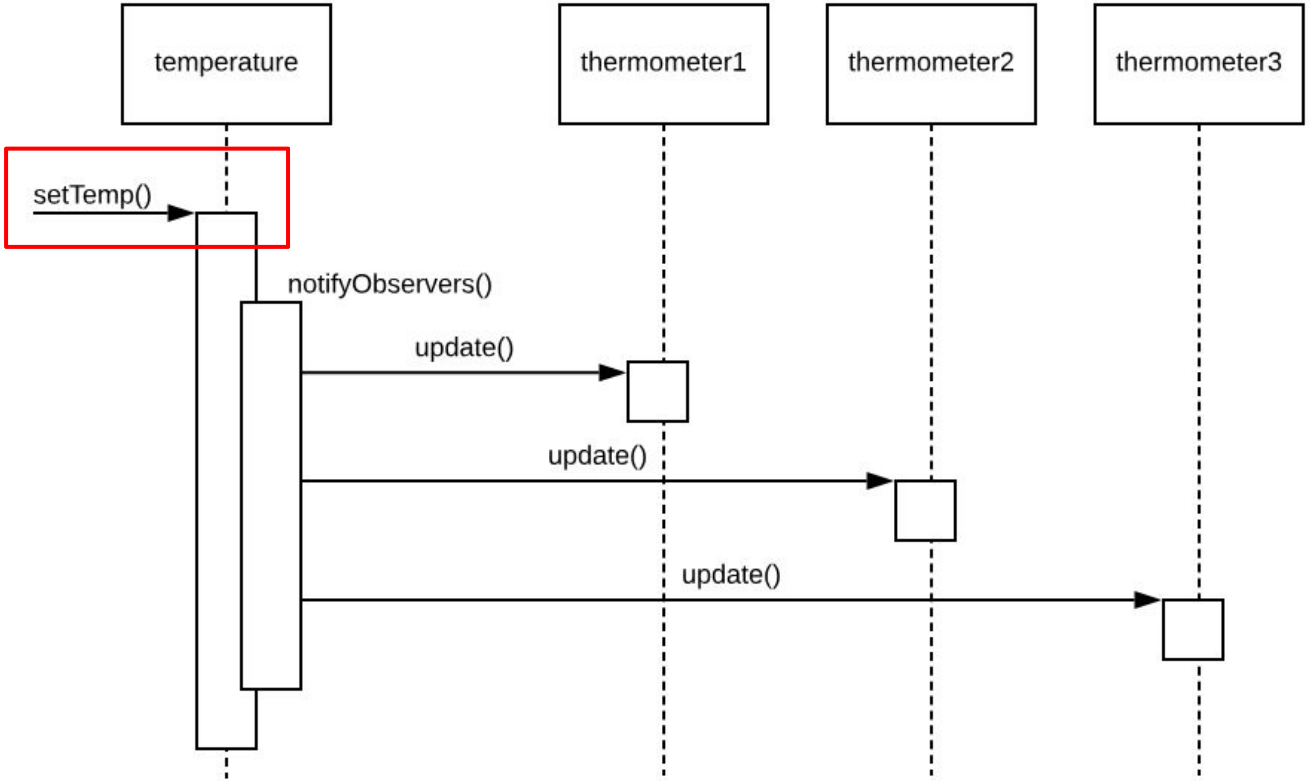
```
class Temperature extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

  public void setTemp(double temp) {
    this.temp = temp;
    notifyObservers();
  }
}
```

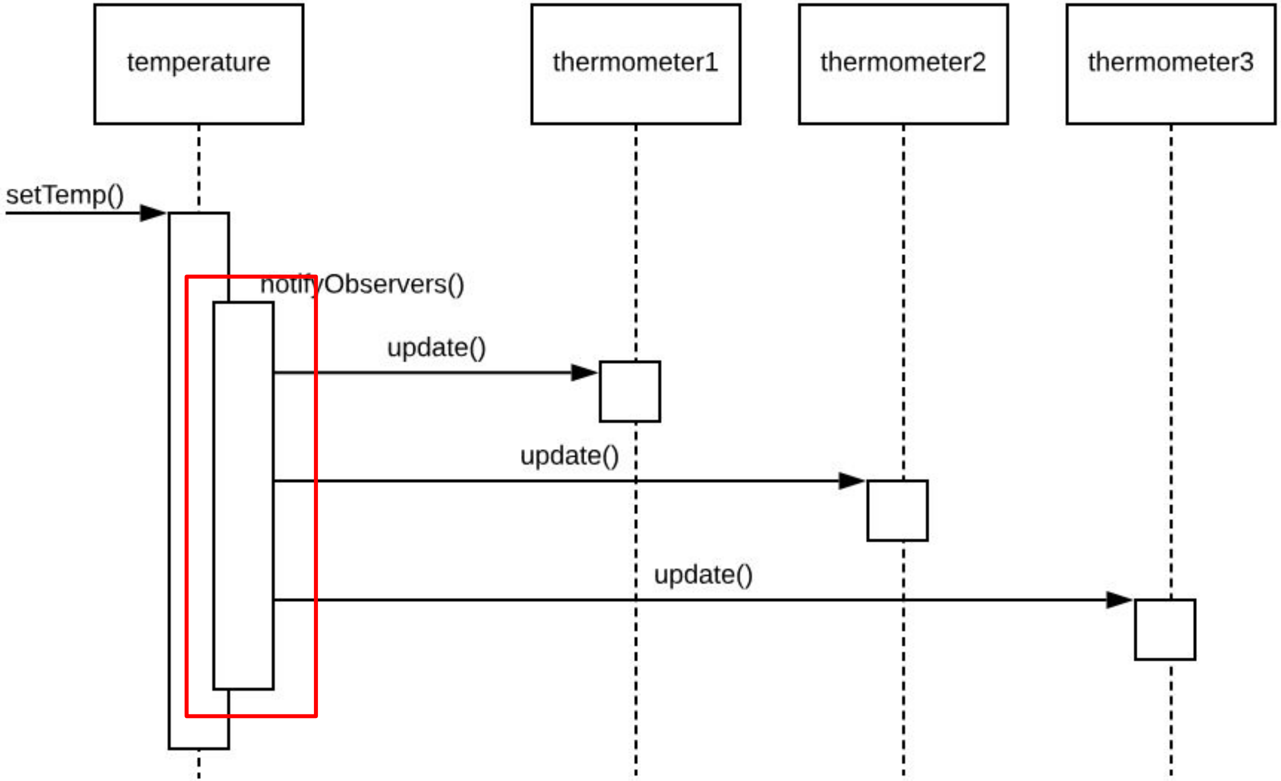
Notifies all observers (thermometers) added to this temperature

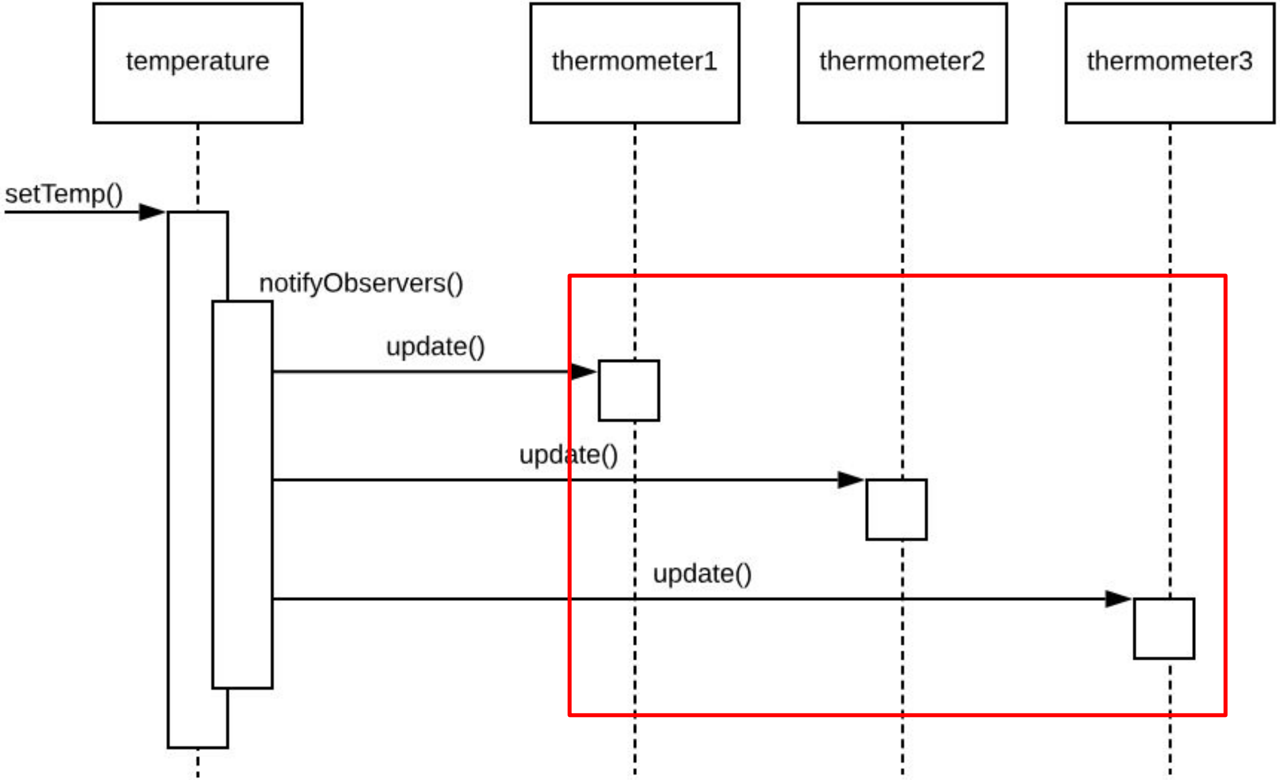


```
class CelsiusThermometer implements Observer {

  public void update(Subject s){
    double temp = ((Temperature) s).getTemp();
    System.out.println("Celsius Temperature: " + temp);
  }
}
```

Observers must implement this method. Calling notifyObservers (in Temperature) results in the execution of update of each observer.

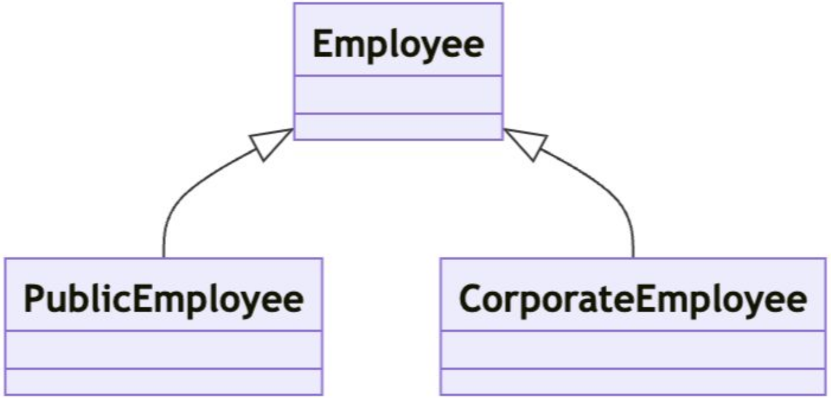

temperature
thermometer1
thermometer2
thermometer3
setTemp()
notifyObservers()
update()
update()
update()


temperature
thermometer1
thermometer2
thermometer3
setTemp()
notifyObservers()
update()
update()
update()


temperature
thermometer1
thermometer2
thermometer3
setTemp()
notifyObservers()
update()
update()
update()

(9) Template Method

Context: Payroll System

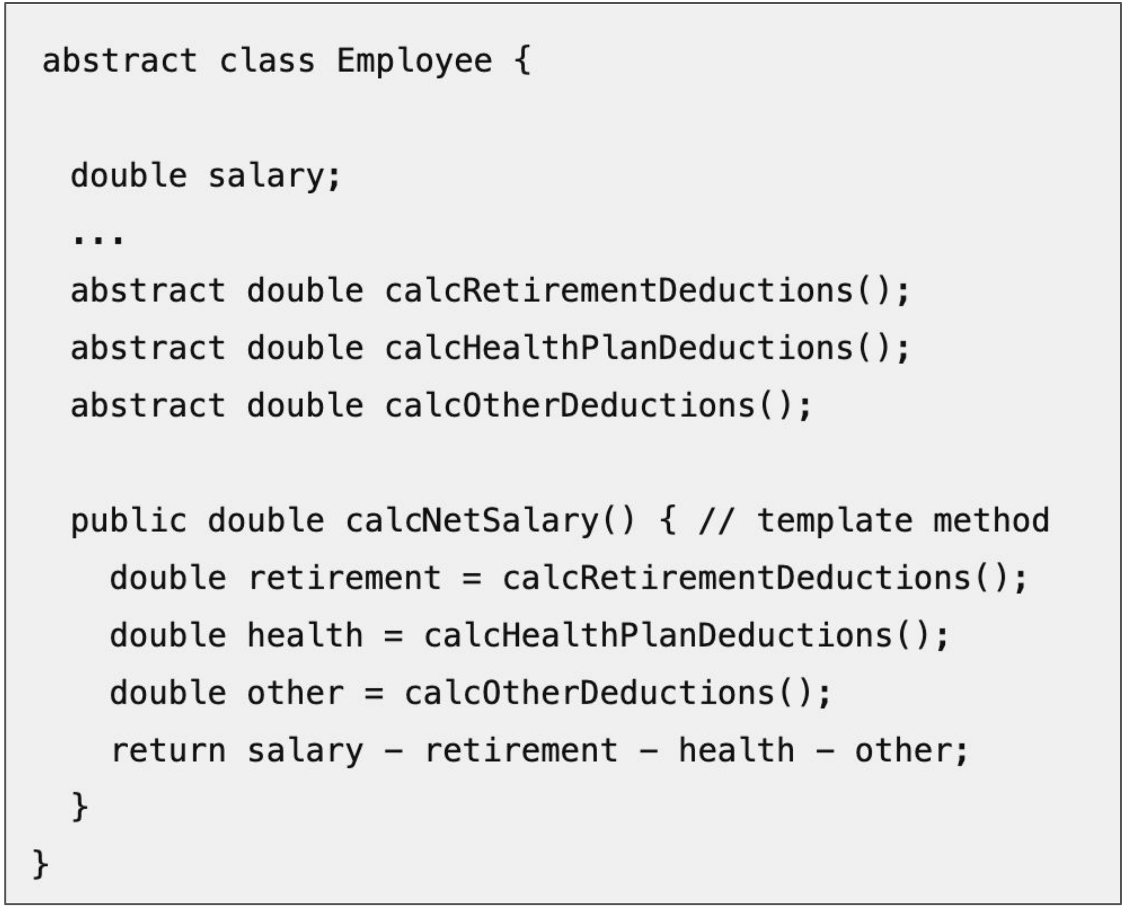


Problem

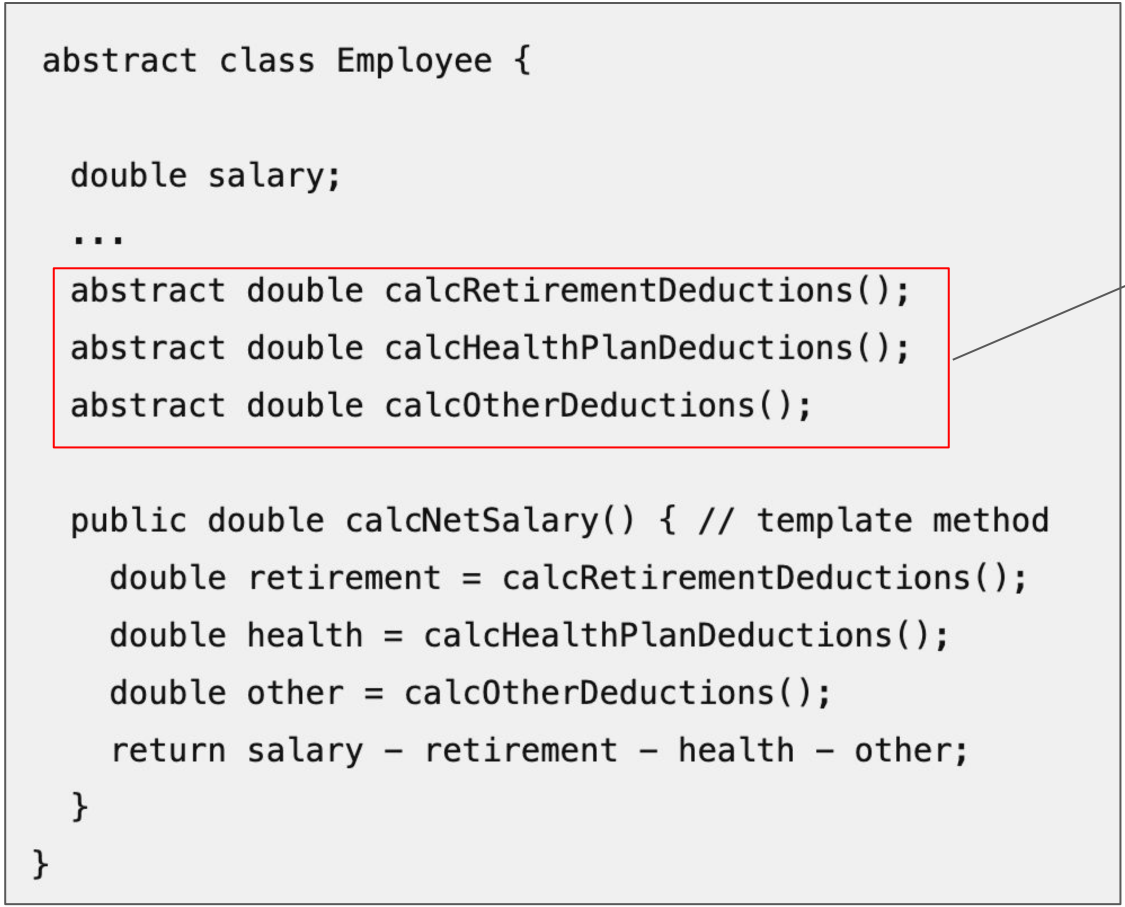
- Calculating salaries:
    - Similar steps for public and corporate employees
    - But, there are details that are different
- In the parent class (Employee) we want to define the main workflow (or template method) for calculating salaries
- And leave to the subclasses refining these steps

Solution: Template Method Pattern

- Implements the skeleton of an algorithm in a base class
- But leaves some steps (abstract methods) to subclasses
- Subclasses can customize an algorithm, but without changing its core behavior



```
abstract class Employee {

  double salary;
  ...
  abstract double calcRetirementDeductions();
  abstract double calcHealthPlanDeductions();
  abstract double calcOtherDeductions();

  public double calcNetSalary() { // template method
    double retirement = calcRetirementDeductions();
    double health = calcHealthPlanDeductions();
    double other = calcOtherDeductions();
    return salary - retirement - health - other;
  }
}
```



```
abstract class Employee {

  double salary;
  ...
  abstract double calcRetirementDeductions();
  abstract double calcHealthPlanDeductions();
  abstract double calcOtherDeductions();

  public double calcNetSalary() { // template method
    double retirement = calcRetirementDeductions();
    double health = calcHealthPlanDeductions();
    double other = calcOtherDeductions();
    return salary - retirement - health - other;
  }
}
```

Implemented by the subclasses

```
abstract class Employee {

  double salary;
  ...
  abstract double calcRetirementDeductions();
  abstract double calcHealthPlanDeductions();
  abstract double calcOtherDeductions();

  public double calcNetSalary() { // template method
    double retirement = calcRetirementDeductions();
    double health = calcHealthPlanDeductions();
    double other = calcOtherDeductions();
    return salary - retirement - health - other;
  }
}
```

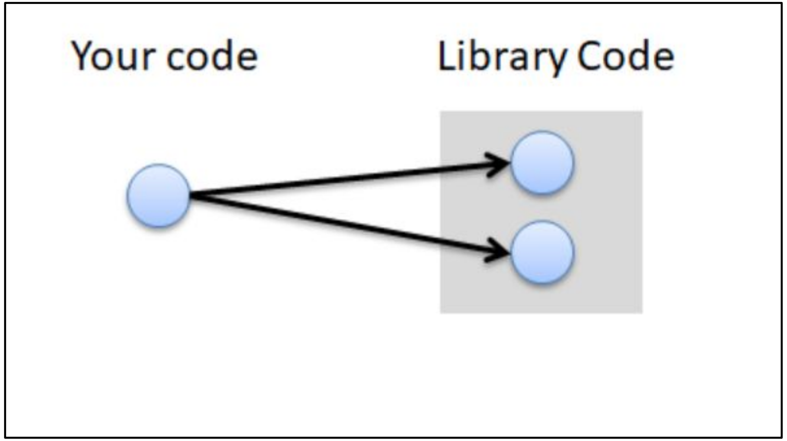
```
abstract class Employee {

  double salary;
  ...
  abstract double calcRetirementDeductions();
  abstract double calcHealthPlanDeductions();
  abstract double calcOtherDeductions();

  public double calcNetSalary() { // template method
    double retirement = calcRetirementDeductions();
    double health = calcHealthPlanDeductions();
    double other = calcOtherDeductions();
    return salary - retirement - health - other;
  }
}
```

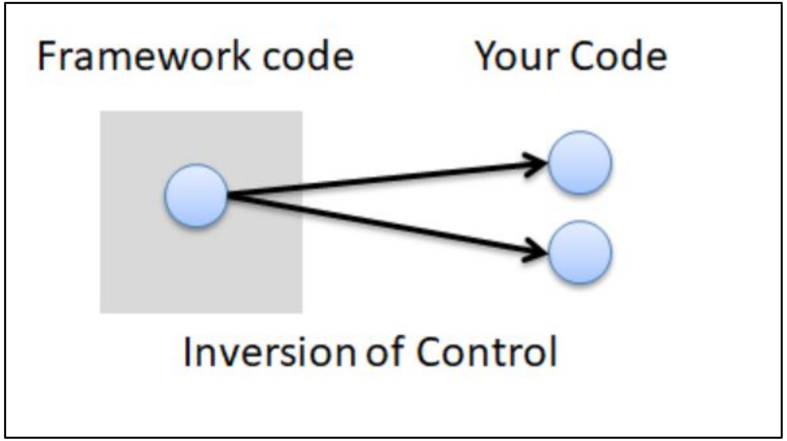
Main steps (or template, or model) for calculating net salaries

Inversion of Control

- Template Method is used to implement Inversion of Control, mainly in frameworks
- Framework: defines the "template" of a system
    - Clients can parameterize some steps
    - Framework calls the code defined by the clients
    - Thus, the term inversion of control

Frameworks vs Libraries

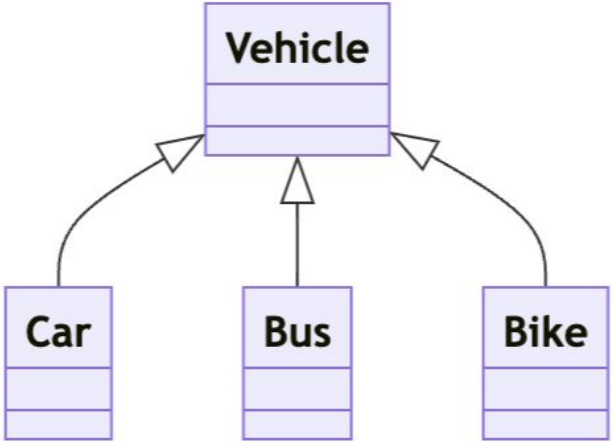




Source: https://github.com/prmr/SoftwareDesign/blob/master/modules/Module-06.md

(10) Visitor

Context: System with Vehicle and subclasses



System also has a polymorphic list of Vehicle

```
List<Vehicle> parkedVehicleList = new ArrayList<Vehicle>();

list.add(new Car(..));

list.add(new Bus(..));

...
```

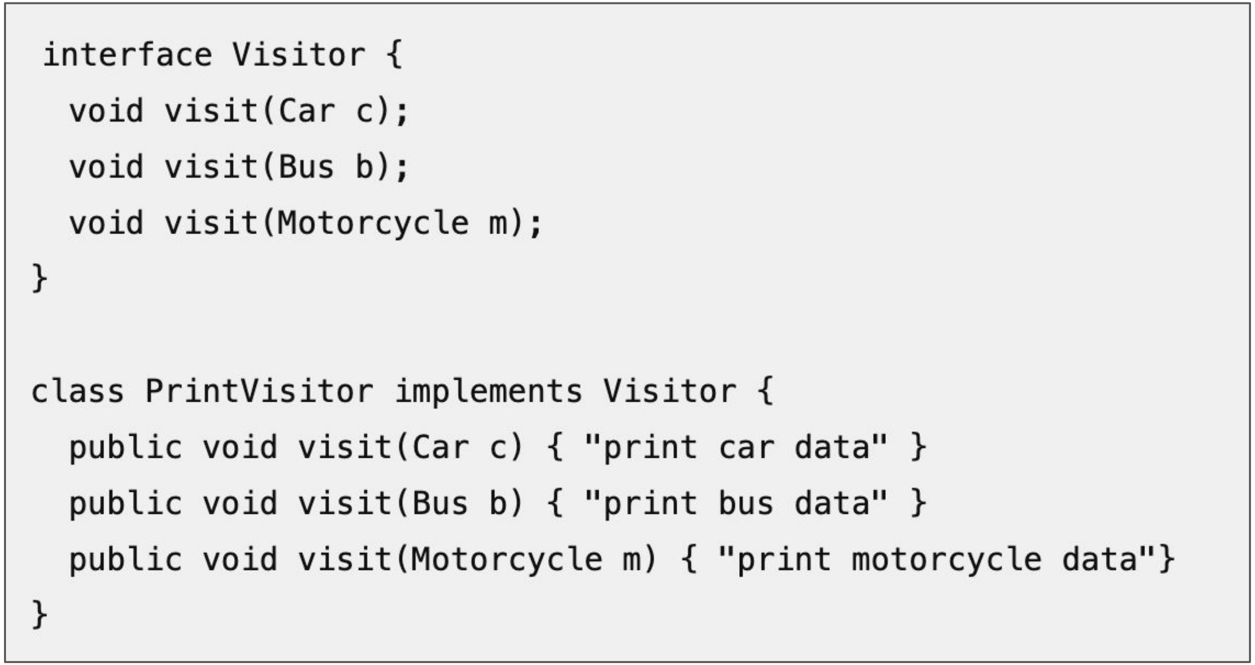
Problem

- We have to perform certain operations with these vehicles
- Example:
    - Print vehicle data
    - Save vehicle data to a database
    - Send a message to vehicle owners
    - Compute vehicle taxes
    - etc

Problem

- We also want to follow the Open/Closed principle:
    - Keep Vehicle and subclasses closed to changes
    - But open to extensions
    - Extensions: operations performed on the vehicles

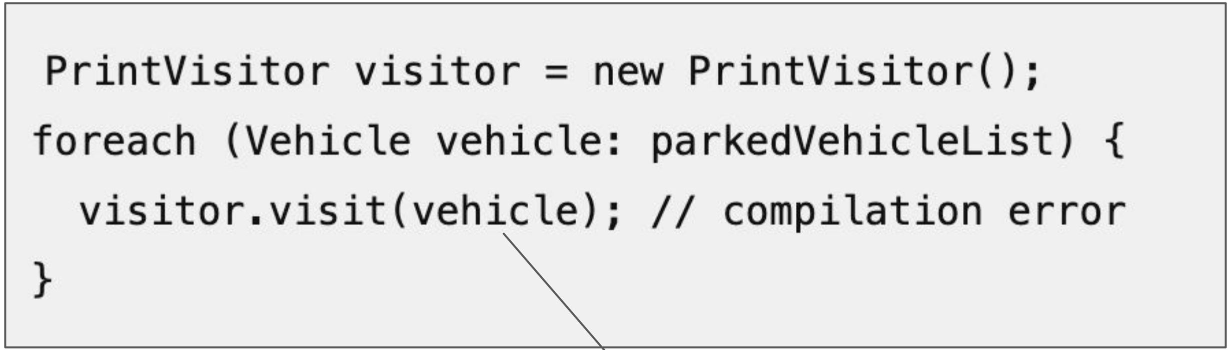
First Alternative



```
 interface Visitor {
  void visit(Car c);
  void visit(Bus b);
  void visit(Motorcycle m);
}

class PrintVisitor implements Visitor {
  public void visit(Car c) { "print car data" }
  public void visit(Bus b) { "print bus data" }
  public void visit(Motorcycle m) { "print motorcycle data"}
}
```

The interface was created because tomorrow we might have other Visitors (e.g., save data to a JSON file).

First Alternative

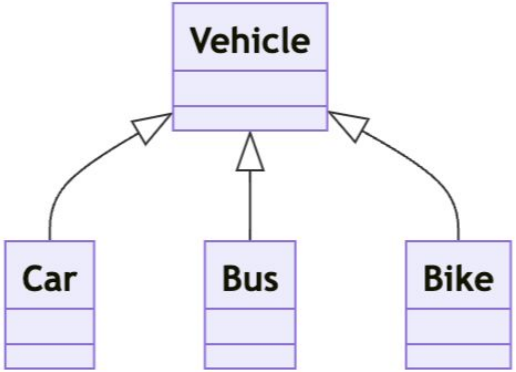


```
 PrintVisitor visitor = new PrintVisitor();
foreach (Vehicle vehicle: parkedVehicleList) {
  visitor.visit(vehicle); // compilation error
}
```

In Java and similar languages, the compiler does not know the dynamic type of "vehicle".
Therefore, it does not know which method of PrintVisitor should be called.

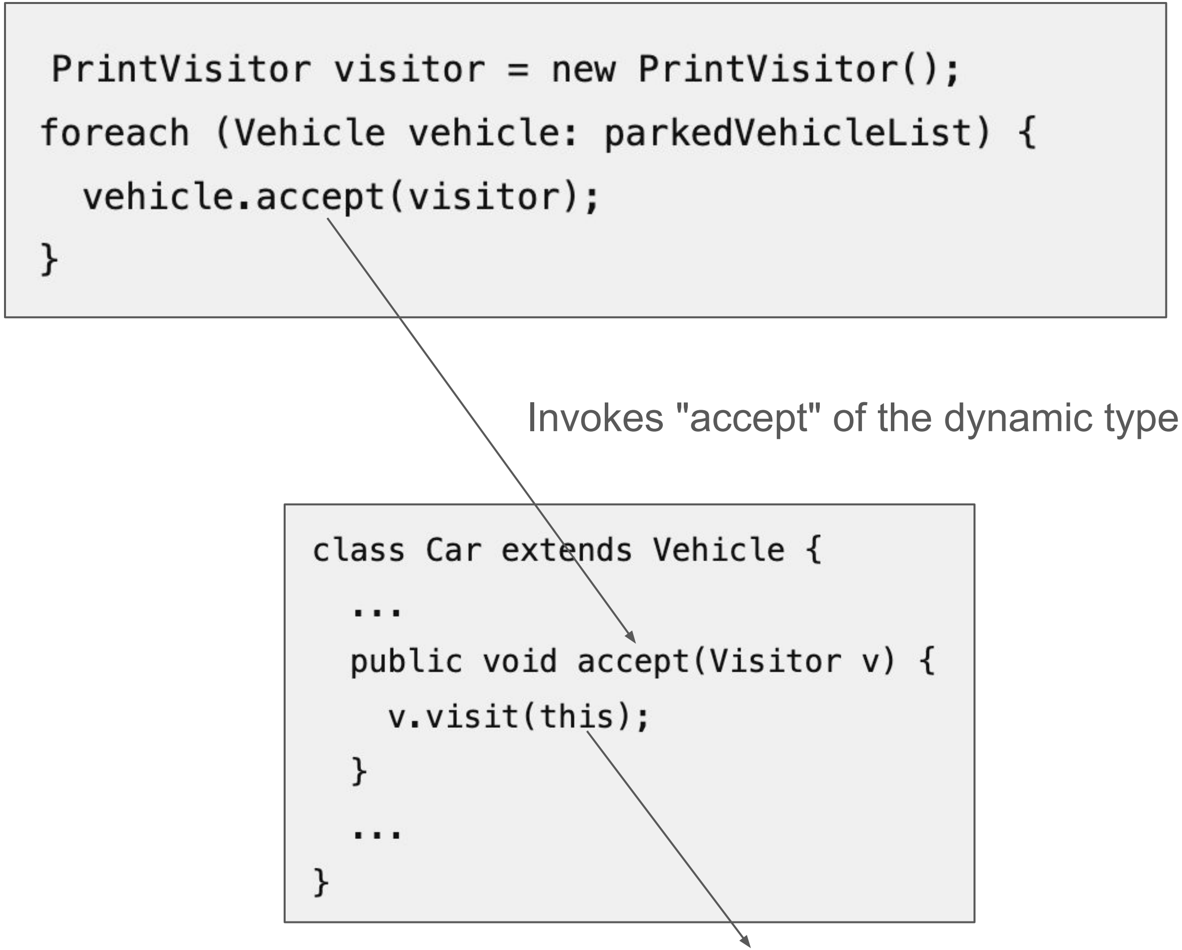
Solution: Visitor Pattern

- Enables adding a generic operation to a family of classes without modifying their code



The idea is to "open" these classes to the execution of a generic operation

```
abstract class Vehicle {
  abstract public void accept(Visitor v);
}

class Car extends Vehicle {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}

class Bus extends Vehicle {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}
```



```
PrintVisitor visitor = new PrintVisitor();
foreach (Vehicle vehicle: parkedVehicleList) {
  vehicle.accept(visitor);
}
```

Invokes "accept" of the dynamic type of the vehicle. Suppose it is a Car.

```
class Car extends Vehicle {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}
```

Type of "this" is known statically (Car)

Visitor: advantages & disadvantages

Advantage #1

- Visitors facilitate adding any method to a class hierarchy
  - A second Visitor may exist with different operations
  - Example: calculating vehicle taxes

Disadvantage #1

- Suppose we need to add a new class to the hierarchy:
    - For example, a Truck class
    - We will have to update all Visitors with a new method: visit(Truck).

Disadvantage #2

- Visitors can break information hiding:
  - Vehicle might have to implement public methods exposing its internal state
  - Just to allow the access by the Visitors

When is it not worth using design patterns?

When is it not worth using design patterns?

- Design Patterns ⇔ Design for change
- If the likelihood of changes is zero, we don't need design patterns

Trade-off: Design for Change vs Complexity

- To enable design for change, design patterns complicate the design
- Example: they require the creation of additional classes

Example: Strategy

Solution without a design pattern

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  public void sort() {
    ... // sorts the list using Quicksort
  }
}
```

1 class

Solution without a design pattern

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  public void sort() {
    ... // sorts the list using Quicksort
  }
}
```

1 class

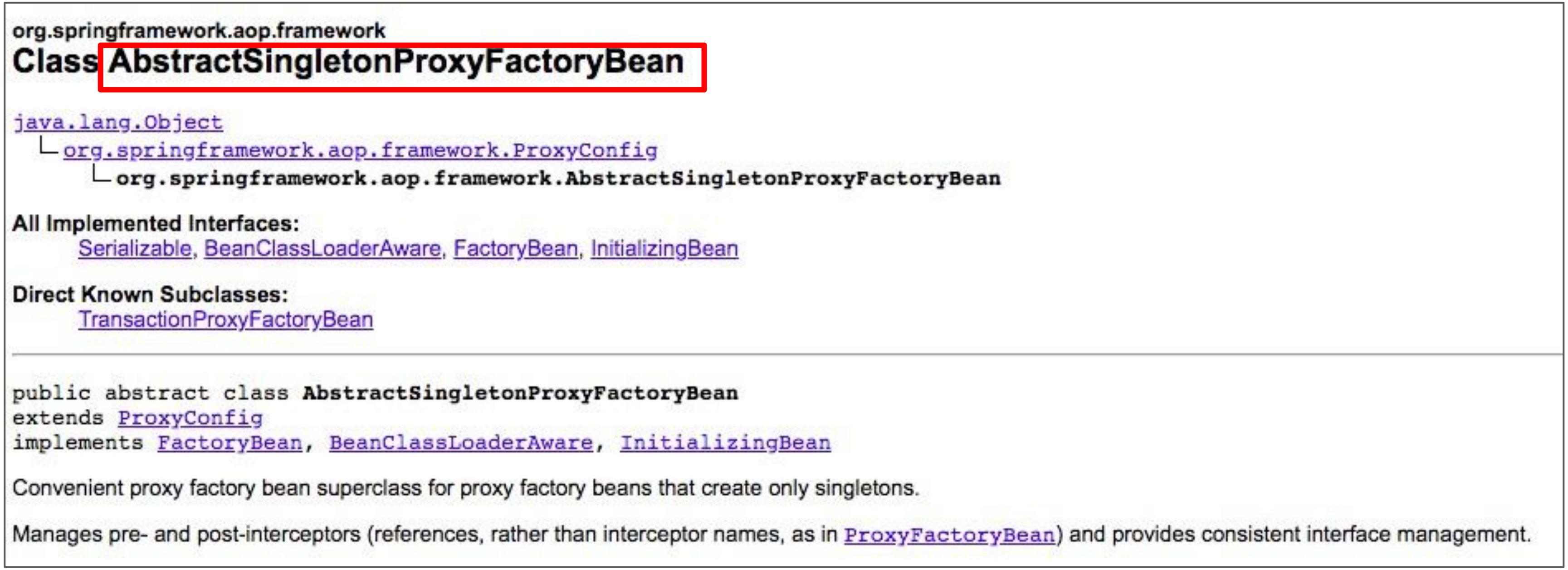
Solution with a design pattern

```
class MyList {

  ... // list data
  ... // list methods: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }
}
```

```
abstract class SortStrategy {
  abstract void sort(MyList list);
}

class QuickSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}

class ShellSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}
```

1 class (with more code)
+1 abstract class
+2 classes with strategies

Patternitis: a (likely) example



org.springframework.aop.framework

Class AbstractSingletonProxyFactoryBean

```
java.lang.Object
  └ org.springframework.aop.framework.ProxyConfig
      └ org.springframework.aop.framework.AbstractSingletonProxyFactoryBean
```

All Implemented Interfaces:

Serializable, BeanClassLoaderAware, FactoryBean, InitializingBean

Direct Known Subclasses:

TransactionProxyFactoryBean

```
public abstract class AbstractSingletonProxyFactoryBean
extends ProxyConfig
implements FactoryBean, BeanClassLoaderAware, InitializingBean
```

Convenient proxy factory bean superclass for proxy factory beans that create only singletons.

Manages pre- and post-interceptors (references, rather than interceptor names, as in `ProxyFactoryBean`) and provides consistent interface management.

End