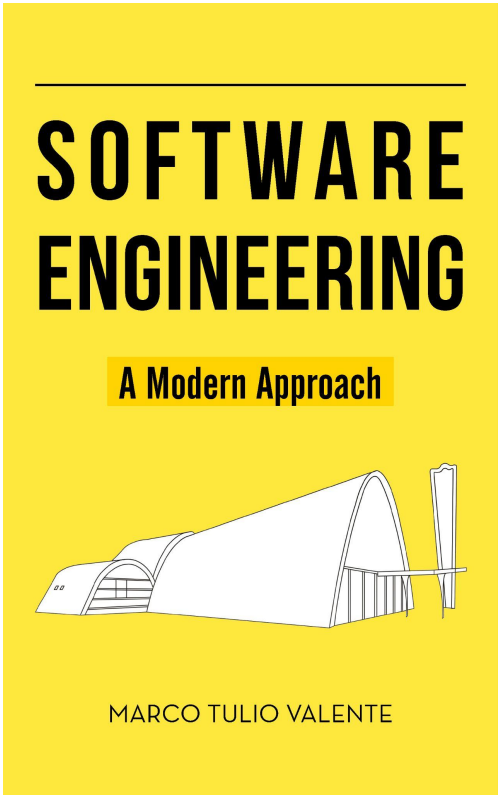


Chapter 6 - Design Patterns

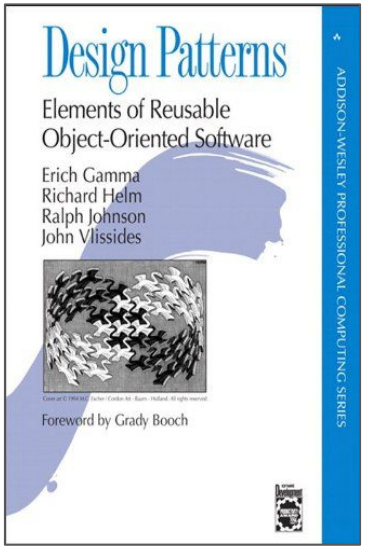
Prof. Marco Tulio Valente

https://softengbook.org

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

Design Patterns

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



1994

Usage #1: Design Reuse

- Suppose we have a design problem
- There may exist a design pattern that solves this problem
- Reusing this pattern helps us avoid reinventing the wheel

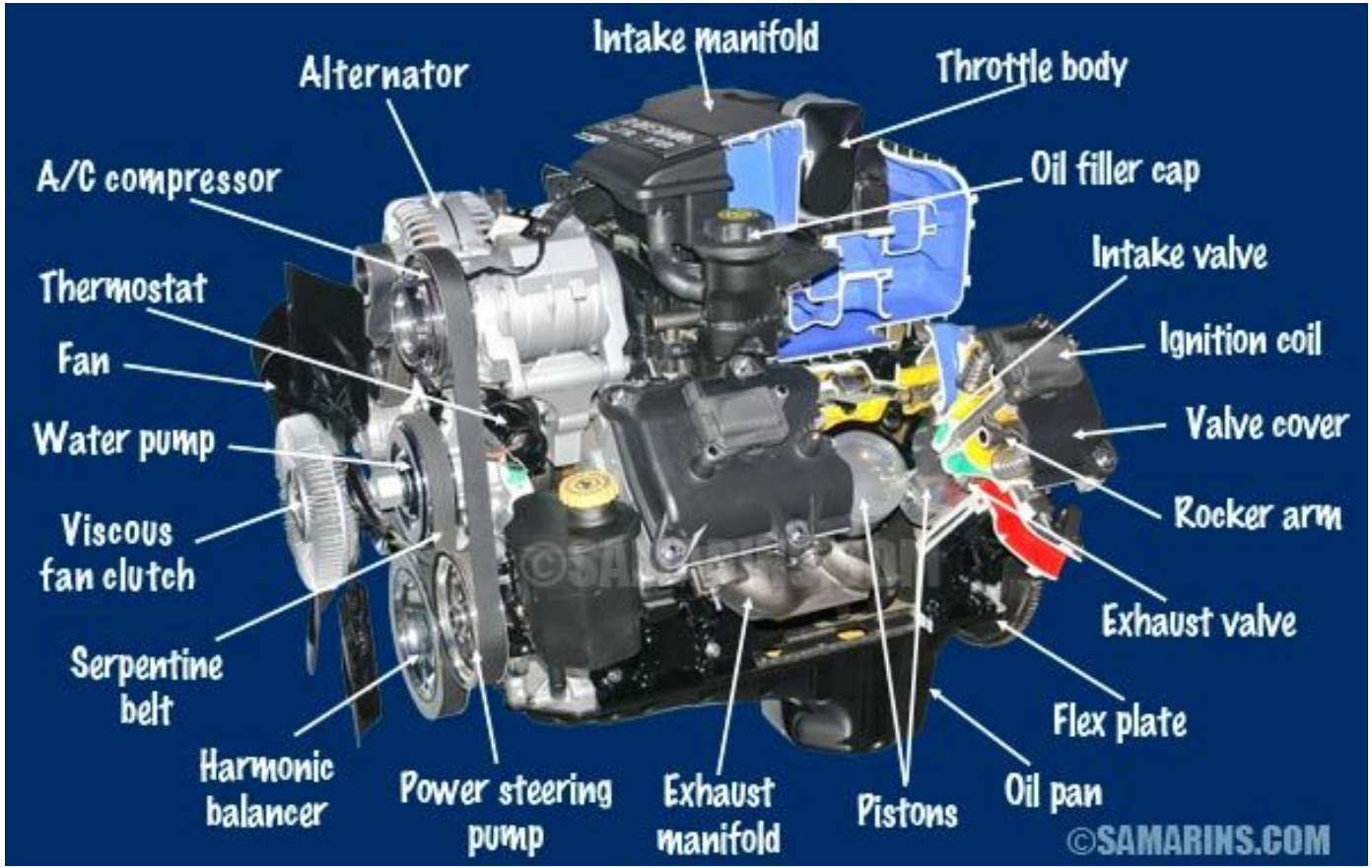
Usage #2: Vocabulary

- Vocabulary for discussions, documentation, etc.

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

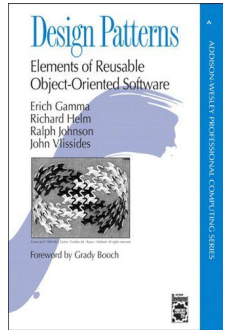
Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.

What components make up an engine?



Similarly, which components (or patterns) can I reuse in my software design?

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



23 patterns

Structure

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

Important: Design for Change

- Design patterns simplify future changes in the code
- If the code is unlikely to change, implementing patterns might represent overengineering

(1) Factory

Context: System with communication channels

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

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

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

Problem

- Applications will need to change from TCP to UDP
- How should we design the system to accommodate this change?
- How can we parameterize the communication protocol?

Solution: Factory Pattern

- Factory: method that encapsulates 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();
  ...
}
```

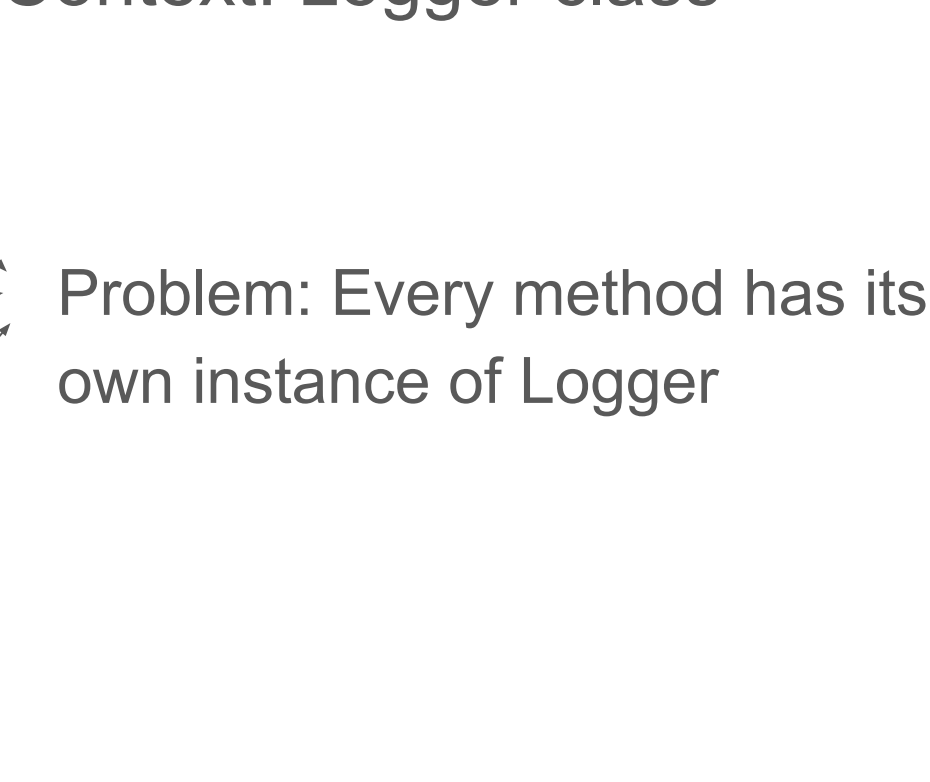
(2) Singleton

```
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

Context: Logger class

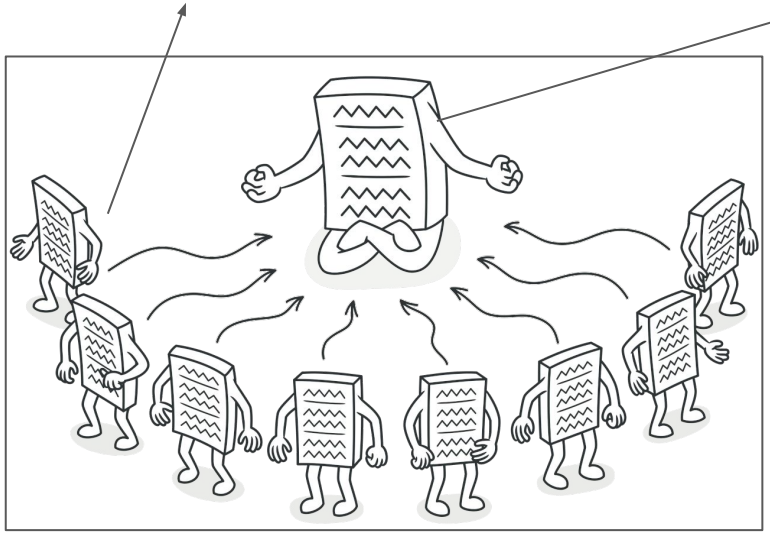
```
void f() {
  Logger log = new Logger();
  log.println("Executing f");
  ...
}

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

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



Problem: Every method has its own instance of Logger

Problem

- All operations should be logged in the same file
- How can we ensure clients use the same Logger instance?"



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

Solution: Singleton Pattern

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

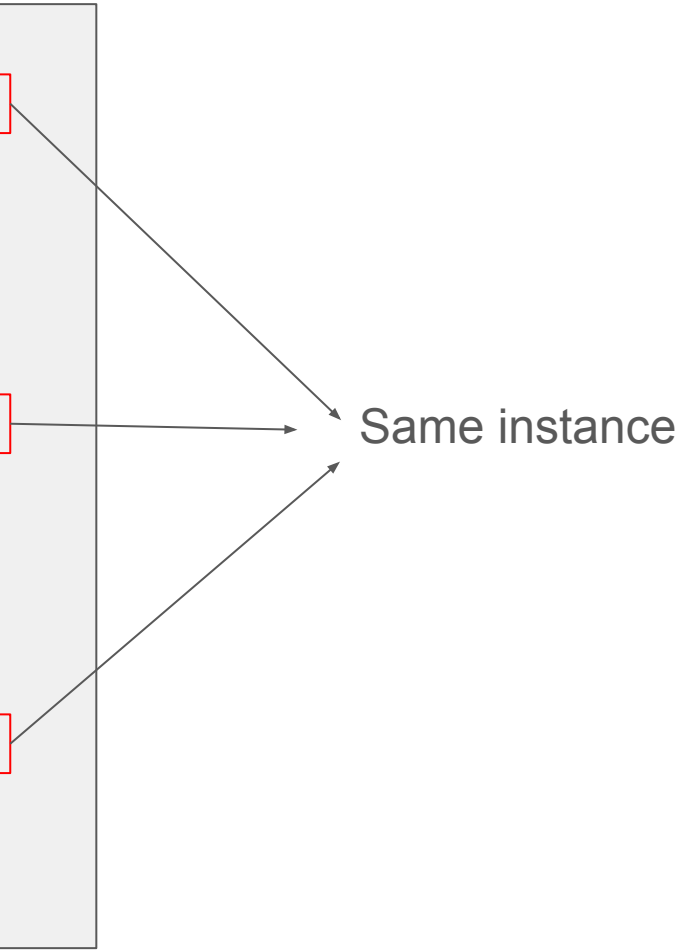
```
class Logger {

  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");
  ...
}
```



(3) Proxy

Context: Book search function

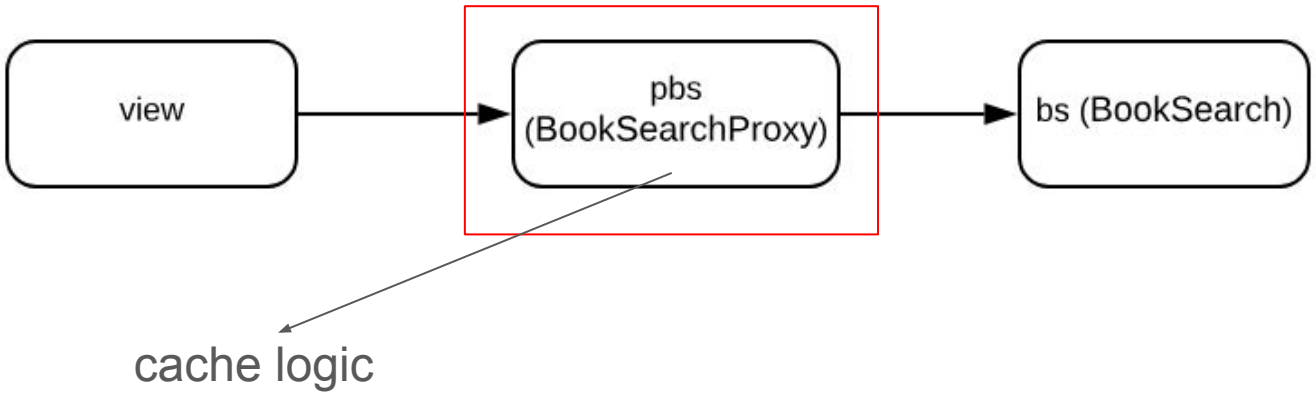
```
class BookSearch {
  ...
  Book getBook(String ISBN) { ... }
  ...
}
```

Problem: using a cache to improve performance

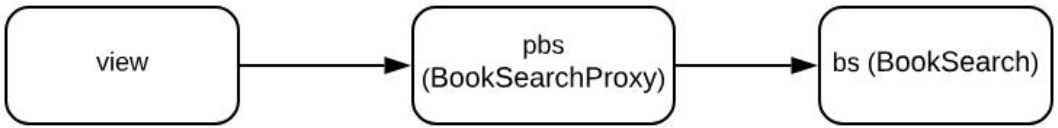
- If "book in cache"
  - return the book immediately
  - otherwise, continue with the search
- We want to avoid modifying BookSearch because:
  - It is already tested and working
  - Another developer maintains it

Solution: Proxy Pattern

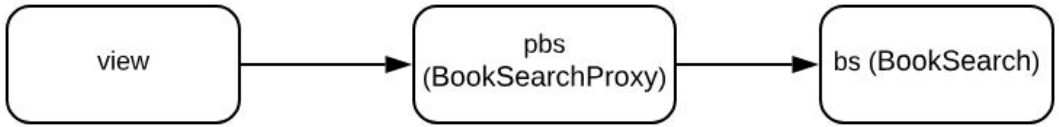
- Proxy: intermediary object between client and a base object
- Clients communicate directly with the proxy instead of the base object; the proxy forwards requests to the base object



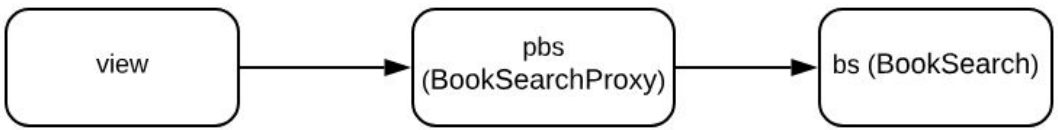
```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```



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



```
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) { ... }
  ...
}
```

Drivers are provided by projector manufacturers

⇒ therefore we can't edit them!

Problem

- In the multimedia control system, we need to use a single Projector interface

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

class ProjectorControlSystem {

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

}
```

Goal: Use this interface while abstracting away the implementation classes

Problem

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

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

- Vendor-provided classes (drivers) from manufacturers
- We cannot modify 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

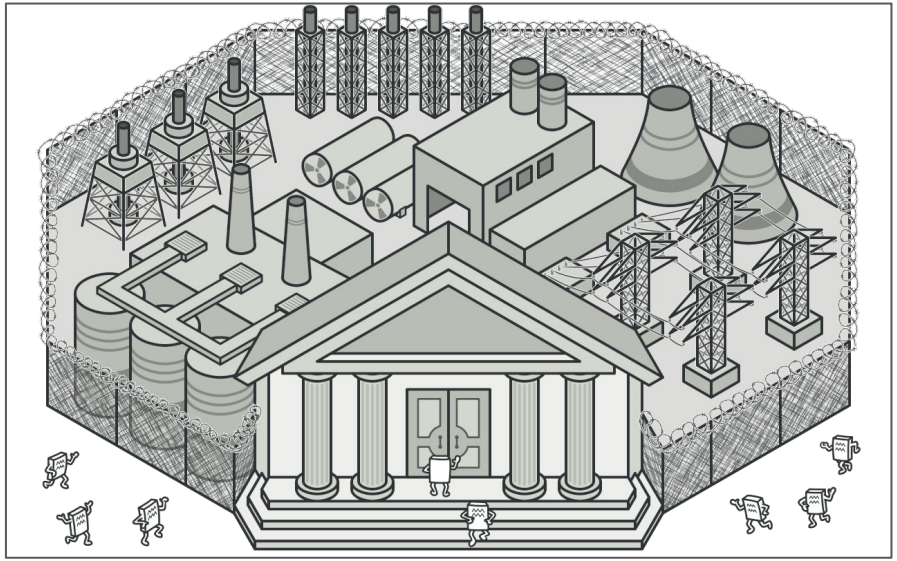
Projector

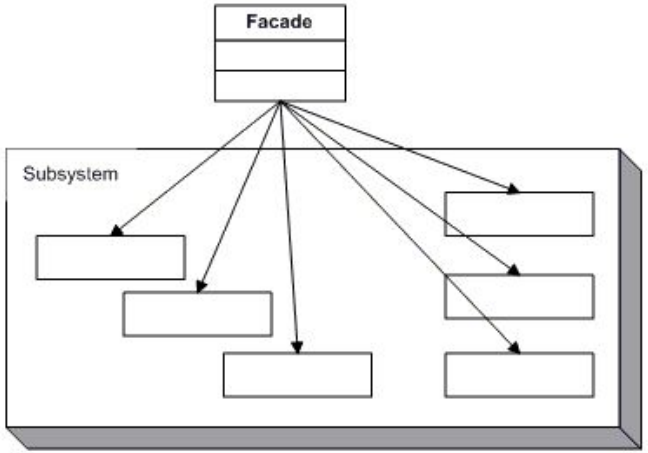
SamsungProjectorAdapter

(5) Facade

Context, Problem & Solution

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





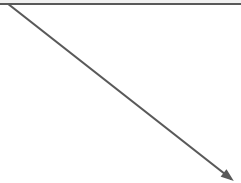
https://refactoring.guru/design-patterns/facade

Example: Interpreter

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

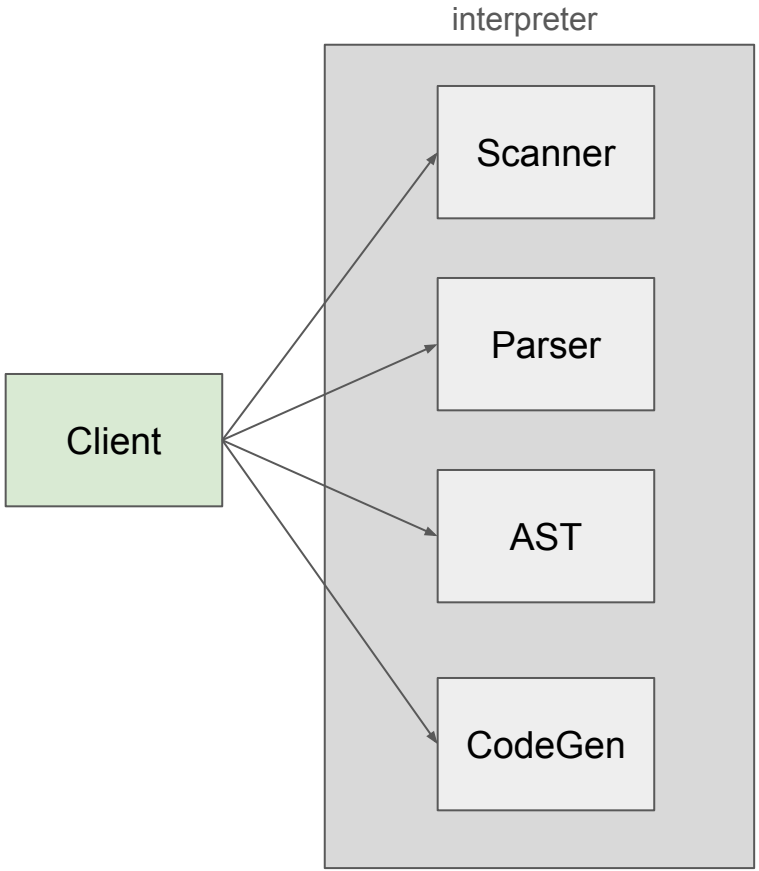
```
Scanner s = new Scanner("prog1.abc");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

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

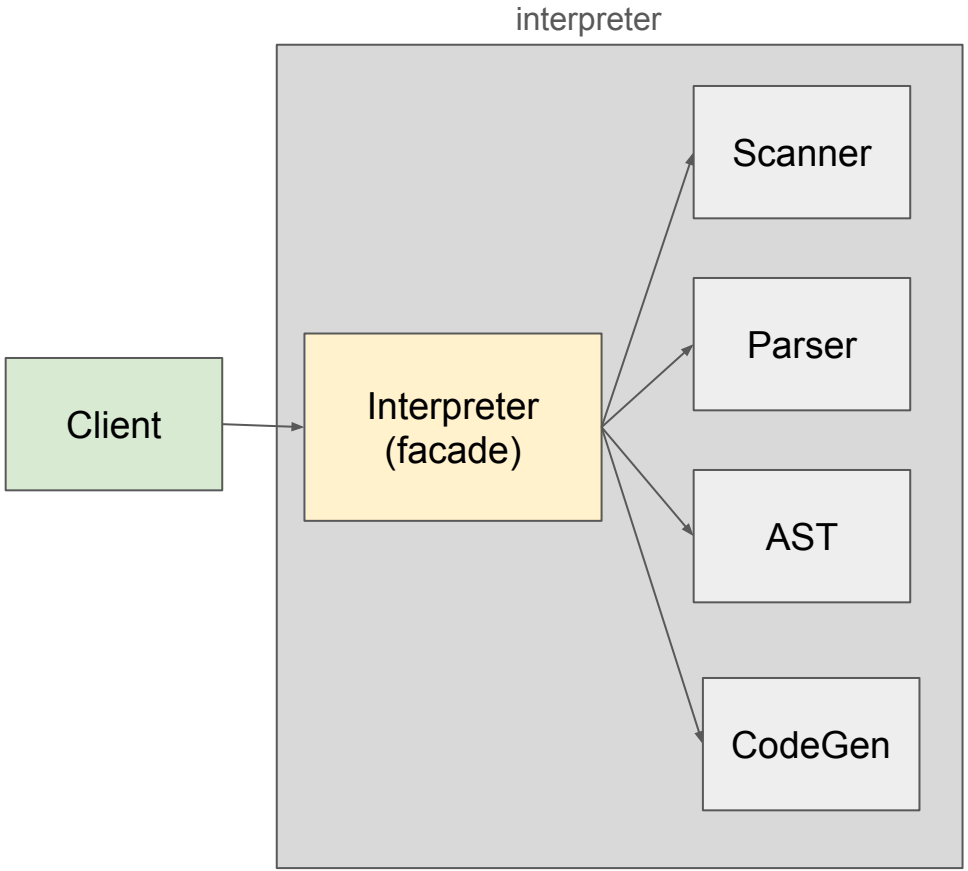


Facade: very simple interface

Without a Facade



With a Facade



Exercises

1. Singleton is one of the most controversial design patterns. Erich Gamma, one of the GoF book's authors, has stated 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 isn't the Singleton implementation shown in the slides compatible with concurrent systems (multi-threaded applications)? Provide an example of a bug that can occur in this type of system.

3. Beyond performance optimization via caching, as discussed in the slides, identify three other non-functional requirements that can be implemented using a Proxy.

4. Answer the following questions about the relationship between design patterns, design properties (cohesion, coupling, etc.), and the SOLID principles:

(a) Which design property does a Proxy pattern improve?

(b) Which SOLID principle is followed when using a Proxy?

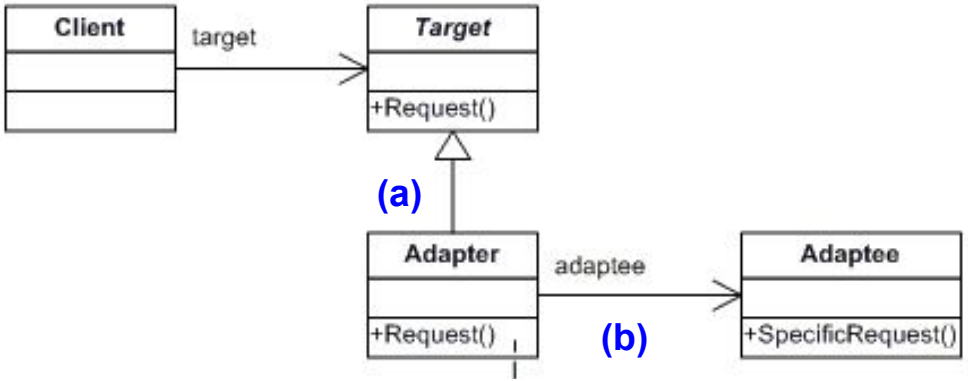
(c) Which design property does a Facade pattern improve?

(d) The Adapter pattern follows which SOLID principle?

(e) When implemented incorrectly, which SOLID principle might a Facade pattern violate?

5. Analyze this class diagram that illustrates the Adapter pattern:

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



Source: GoF book

(6) Decorator

Context: System that uses communication channels

(previously used 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 additional functionalities to channels

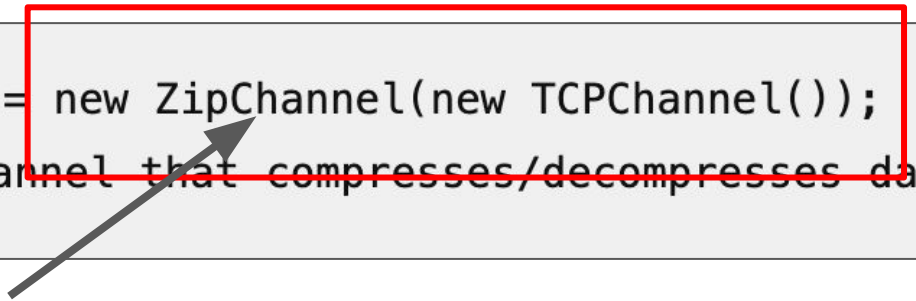
- Basic channels (TCP, UDP) are not sufficient
- We also need channels with:
  - Data compression/decompression
  - Buffering capabilities
  - Logging
  - etc

Solution: Decorator Pattern

- Address this problem through composition, rather than inheritance
- Thus, avoids creating an excessive number of classes

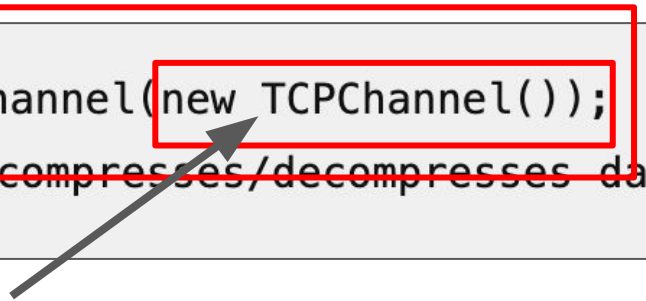
Example

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



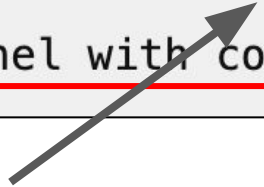
Example

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



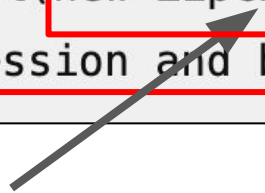
Another example

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



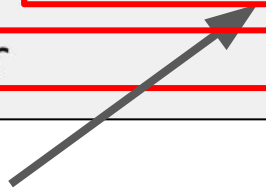
Another example

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



Another example

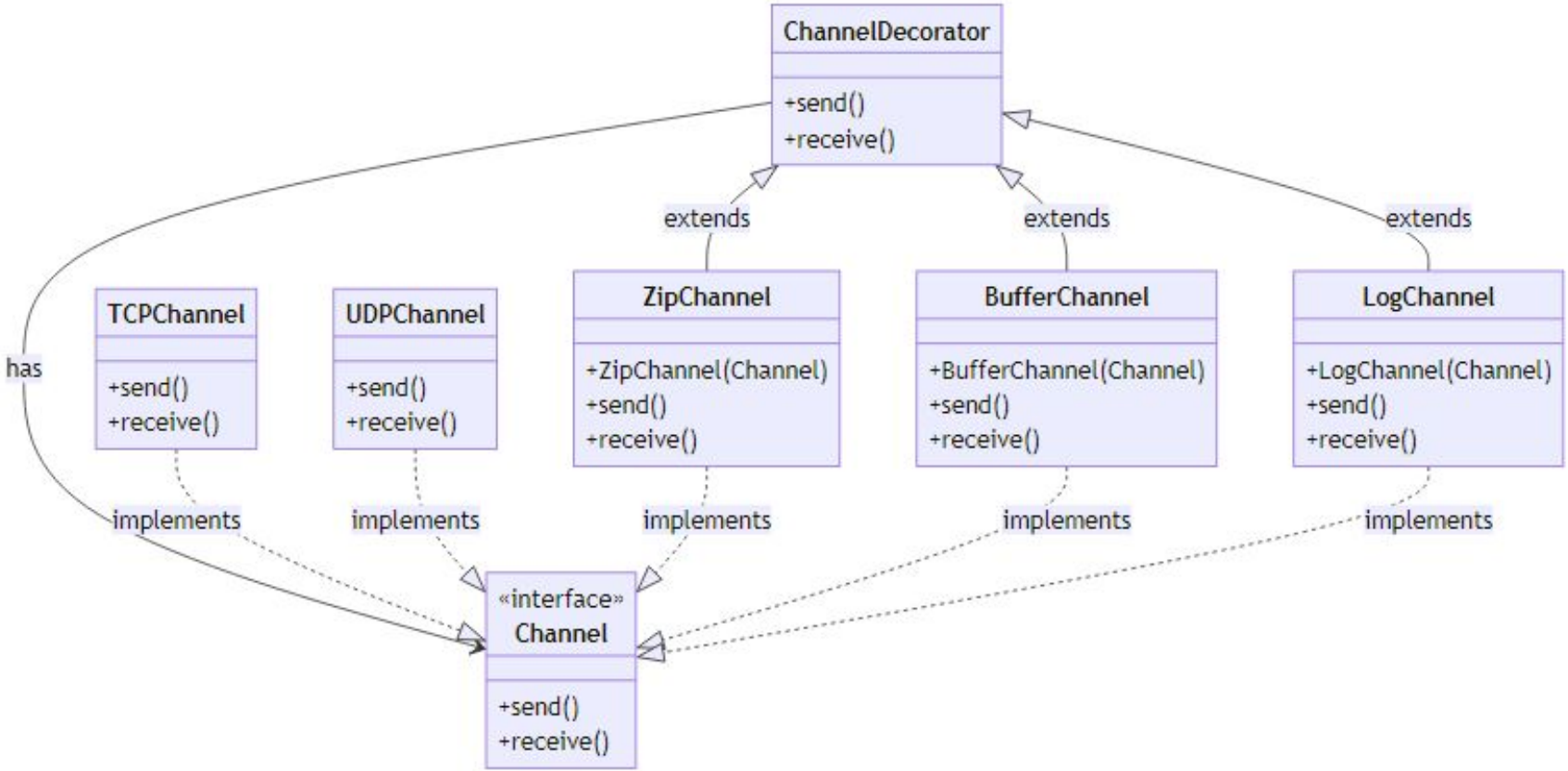
```
channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel with compression and buffer
```



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

Inside one box, there is another box, which has another box... until finally reaching the gift—similar to how our decorators wrap around 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 for extensions
- Example: using other sorting algorithms (ShellSort, HeapSort, etc)

Solution: Strategy Pattern

- Goal: parametrize the algorithms used by a class
- Enable a class to be open to new algorithms
- In our example: implementing new sorting algorithms

Step #1: Create a Strategy hierarchy
(where each strategy represents an 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 class to use a Strategy

```
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
- When the temperature changes, all thermometers should be updated

Problem

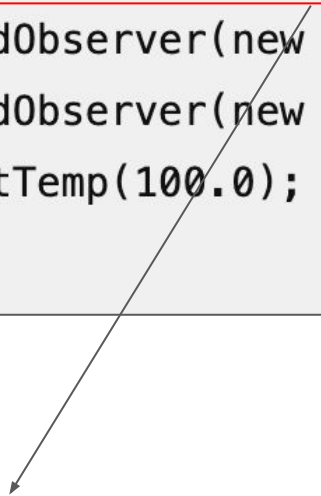
- We don't want to couple Temperature to Thermometers
- Reasons:
  - Maintain domain class (model) independent from view classes (UI classes)
  - Enable easy addition of new thermometer types

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

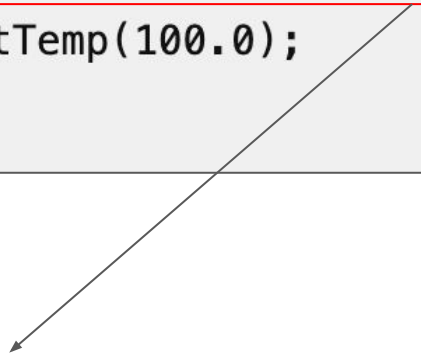
```
void main() {
  Temperature t = new Temperature();
  t.addObserver(new CelsiusThermometer());
  t.addObserver(new FahrenheitThermometer());
  t.setTemp(100.0);
}
```



Subject

Example

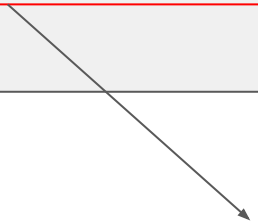
```
void main() {
  Temperature t = new Temperature();
  t.addObserver(new CelsiusThermometer());
  t.addObserver(new FahrenheitThermometer());
  t.setTemp(100.0);
}
```



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();
  }
}
```

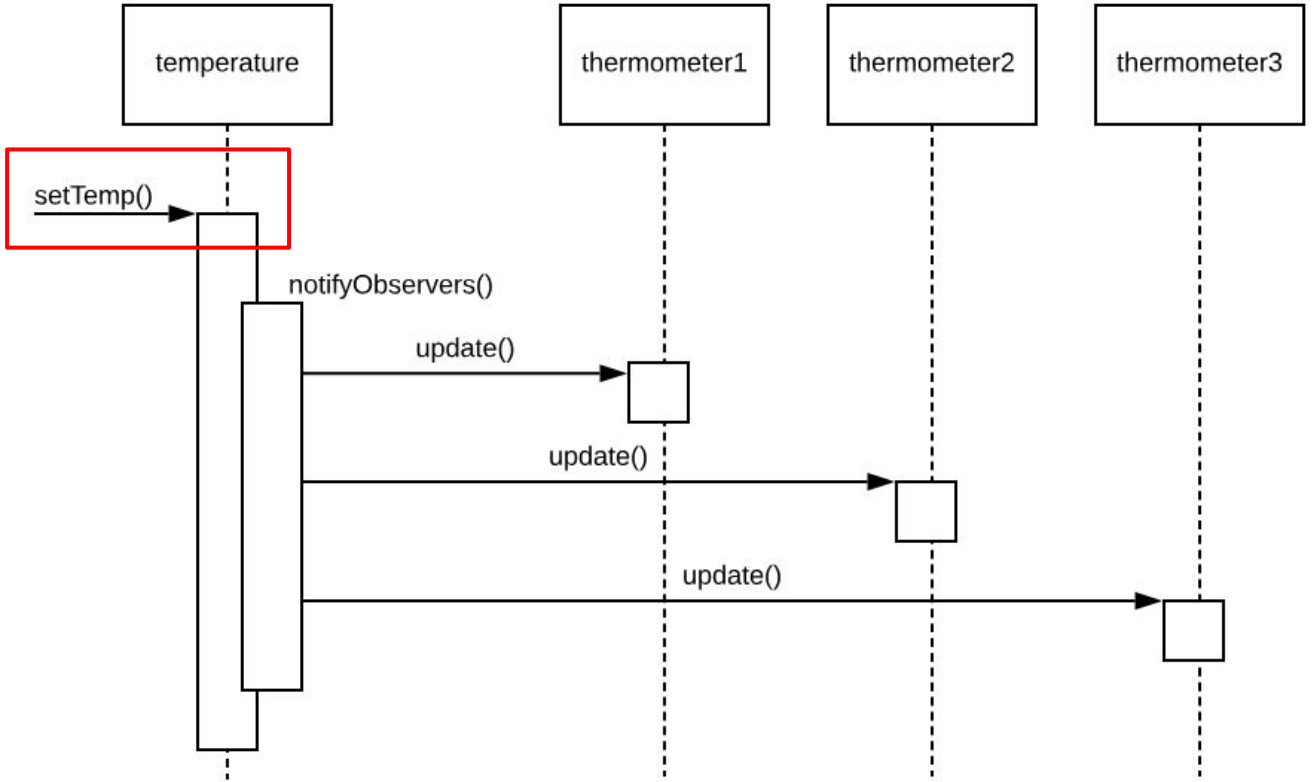
Class that implements addObservers and notifyObservers

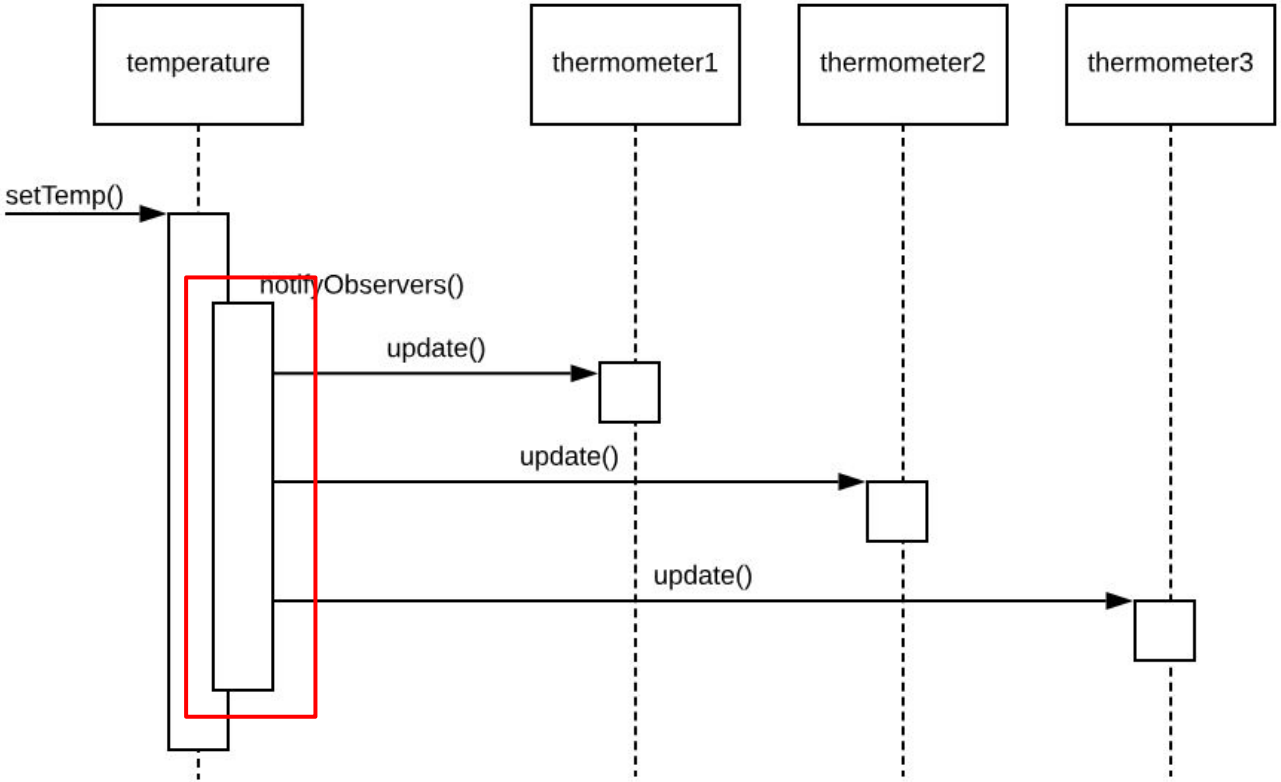
```
class Temperature extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

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

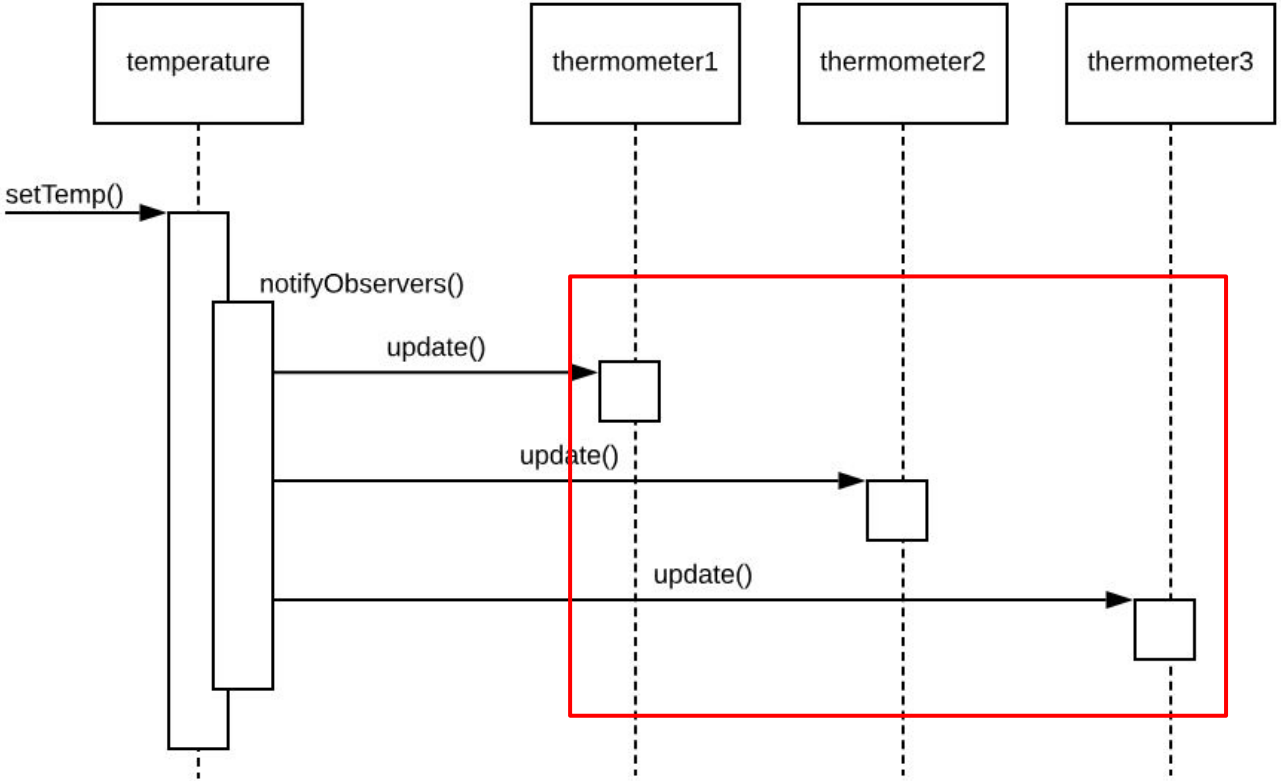
Notifies all registered observers (thermometers) when this temperature value changes

```
class CelsiusThermometer implements Observer {

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

All observers must implement this method. When Temperature's notifyObservers method is called, it triggers the update() execution of each registered observer.

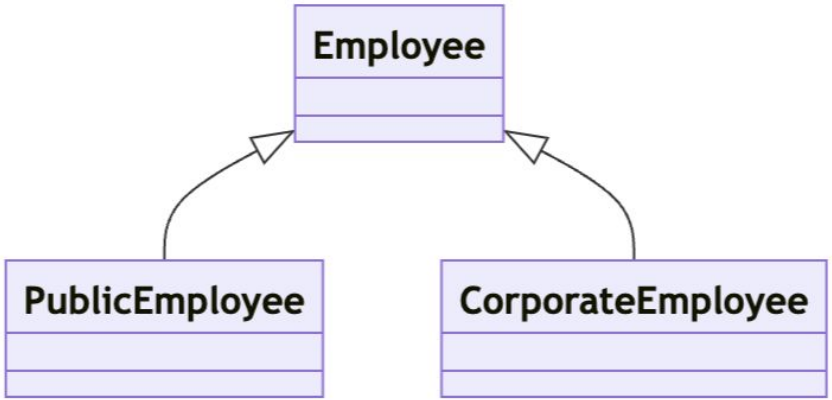

temperature
thermometer1
thermometer2
thermometer3
setTemp()
notifyObservers()
update()
update()
update()


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

- Salary calculation process: Similar steps for public and corporate employees
- However, there are specific differences in implementation
- In the parent class (Employee), we want to define the main workflow (template method pattern) for calculating salaries
- Subclasses will implement the specific details of 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 specific steps of the algorithm, while preserving 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;
  }
}
```

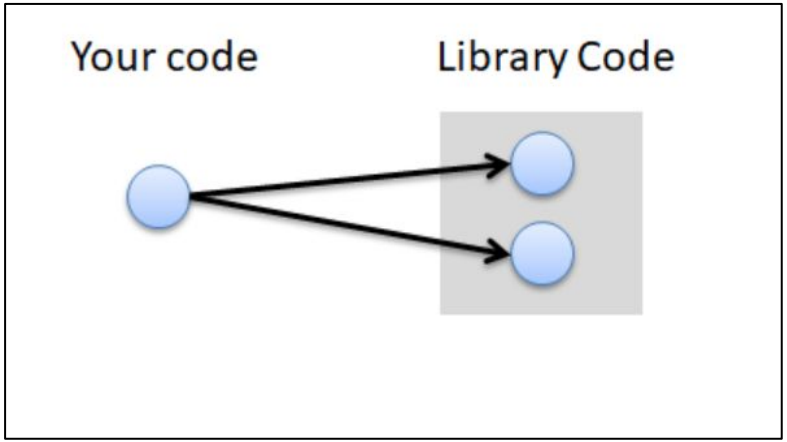
```
abstract class Employee {

  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;
  }
}
```

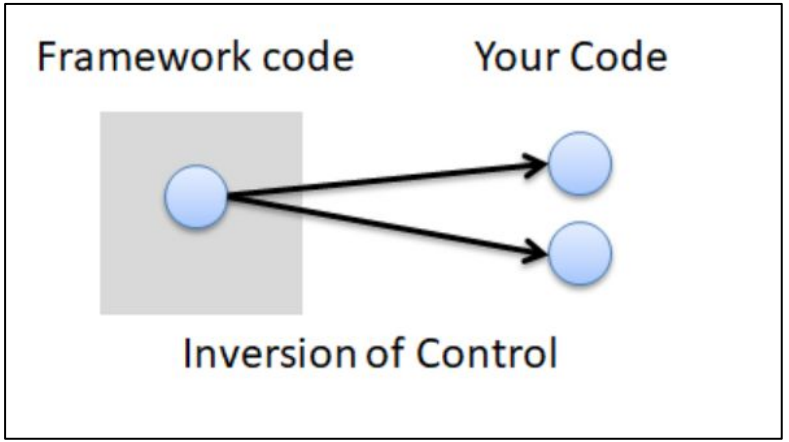
Template method for net salary calculation

Inversion of Control

- Template Method enables Inversion of Control, particularly in frameworks
- Framework: defines the "template" of a system
  - Clients can customize specific steps
  - Framework executes the code defined by the clients
  - Thus, the term inversion of control

Frameworks vs Libraries

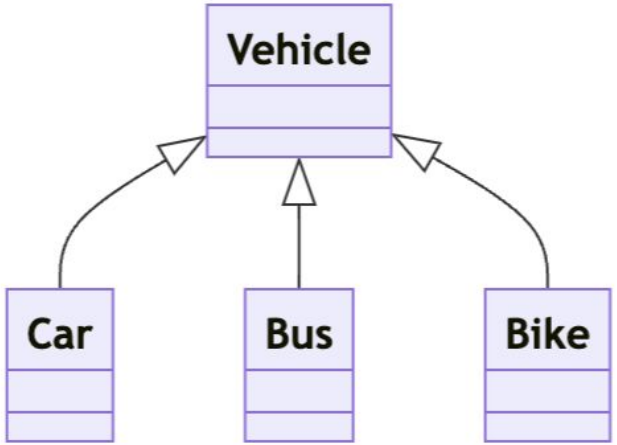




Source: https://github.com/prmr/SoftwareDesign/blob/master/modules/Module-06.md

(10) Visitor

Context: System with Vehicle and subclasses



The system has a polymorphic list of Vehicle objects

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

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

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

...
```

Problem

- We need to perform certain operations on all vehicles
- Examples:
  - Print vehicle data
  - Save vehicle data to the database
  - Send notifications to vehicle owners
  - Compute vehicle taxes
  - etc

Problem

- We also want to follow the Open/Closed principle
- Keep Vehicle and its subclasses closed for modification, while remaining open for extensions
- Extensions: new operations performed on 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"}
}
```

An interface is used to enable future Visitor implementations (e.g., implementing a JSON file storage visitor)

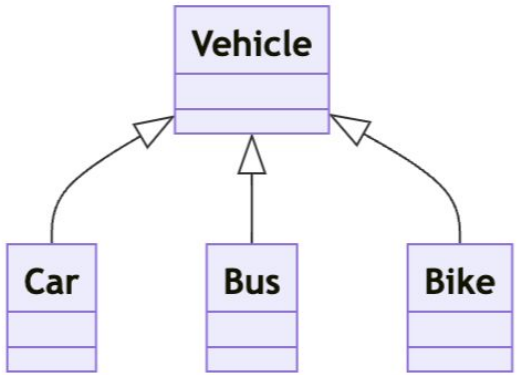
First Alternative

```
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 "visit" method of the PrintVisitor should be executed.

Solution: Visitor Pattern

- Enables the addition of new operations to a class hierarchy without modifying existing 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);
  }
  ...
}
```

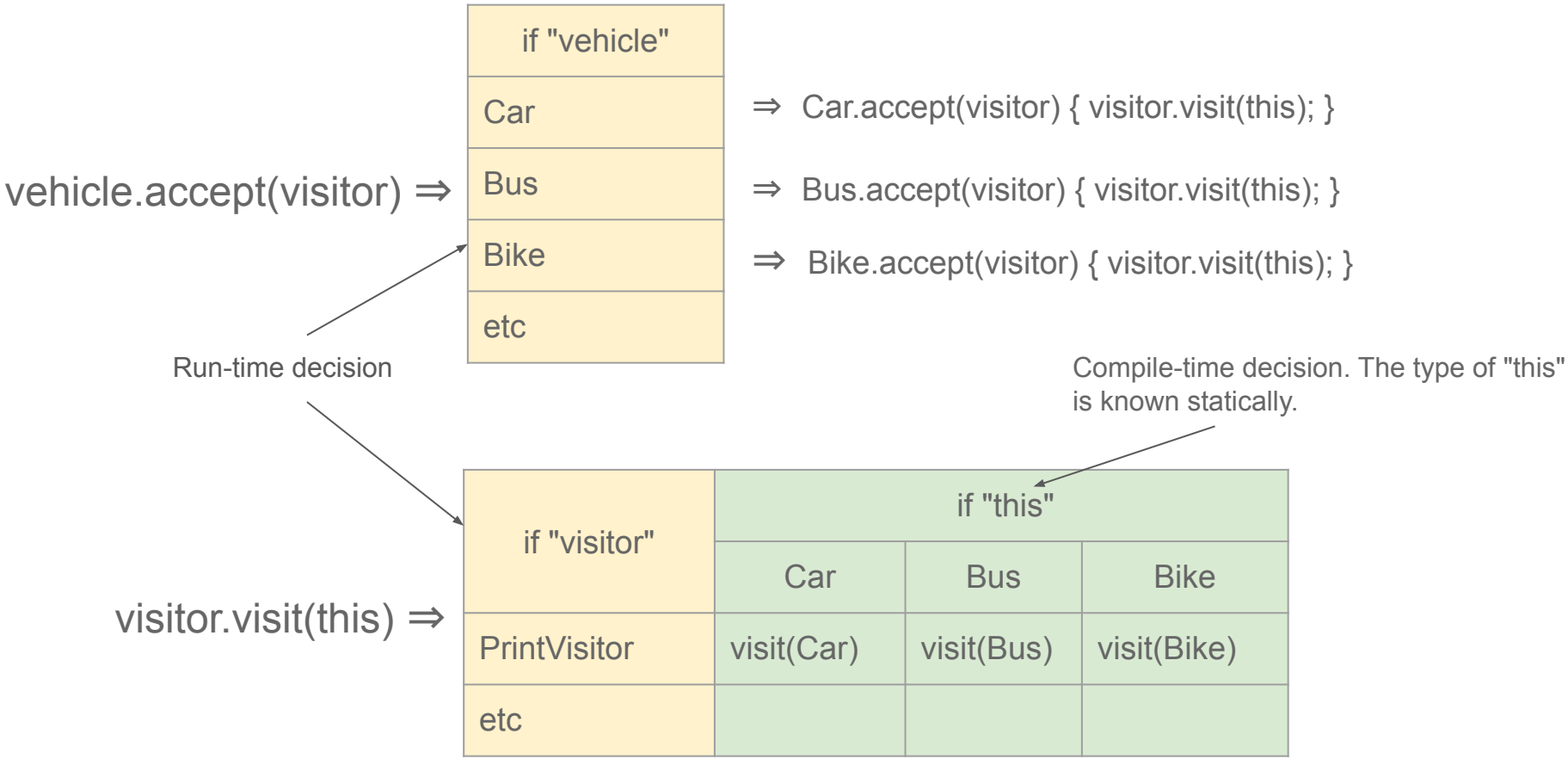
```
PrintVisitor visitor = new PrintVisitor();
foreach (Vehicle vehicle: parkedVehicleList) {
  vehicle.accept(visitor);
}
```

Invokes the "accept" method based on 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)

Method dispatch consists of two dynamic steps and one static step



Visitor: advantages & disadvantages

Advantage #1

- Visitor facilitates adding new operations to a class hierarchy
- We can also define multiple visitors
- Example: A TaxVisitor for calculating vehicle taxes

Disadvantage #1

- Suppose we need to add a new class to the hierarchy:
  - Such as a Truck class
  - Each existing Visitor must be modified to include a new method: visit(Truck)

Disadvantage #2

- Visitors may compromise information hiding
- Example: Vehicle classes often need to implement public methods that expose internal state
- These methods exist solely to allow access by Visitors

When should you avoid design patterns?

When should you avoid design patterns?

- Design Patterns ⇔ Design for change
- When the likelihood of changes is zero, using design patterns is unnecessary

Trade-off: Design for Change vs Complexity

- Design patterns increase complexity when enabling design for change
- For example: they often require creating 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

Implementation without a design pattern

```
class MyList {

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

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

1 class

Implementation using 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