CptS 487
Software Design and Architecture

Lesson 24

Design Patterns 8:

Façade & Bridge

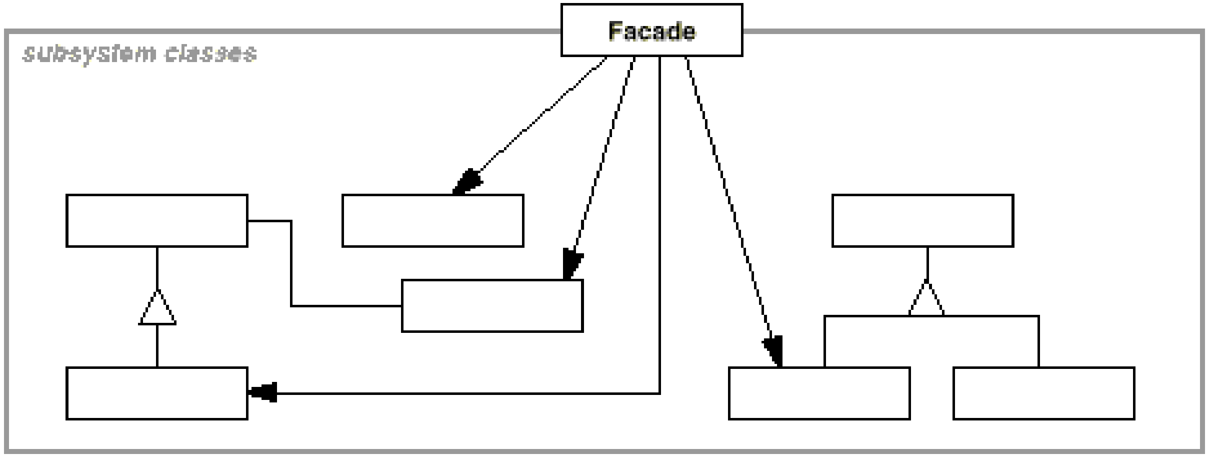
WASHINGTON STATE UNIVERSITY

World Class. Face to Face.

Instructors:

2. Facade

- Structure



- Participants:
  - Facade: Provides a simplified API to interact with subsystems.
    - knows which subsystems are responsible for a request
    - delegates the client requests to appropriate subsystem objects.
  - Subsystem classes:
    - Implement subsystem functionality
    - Handle work assigned by the facade object
    - Have no knowledge of the facade object; that is they keep no references to it

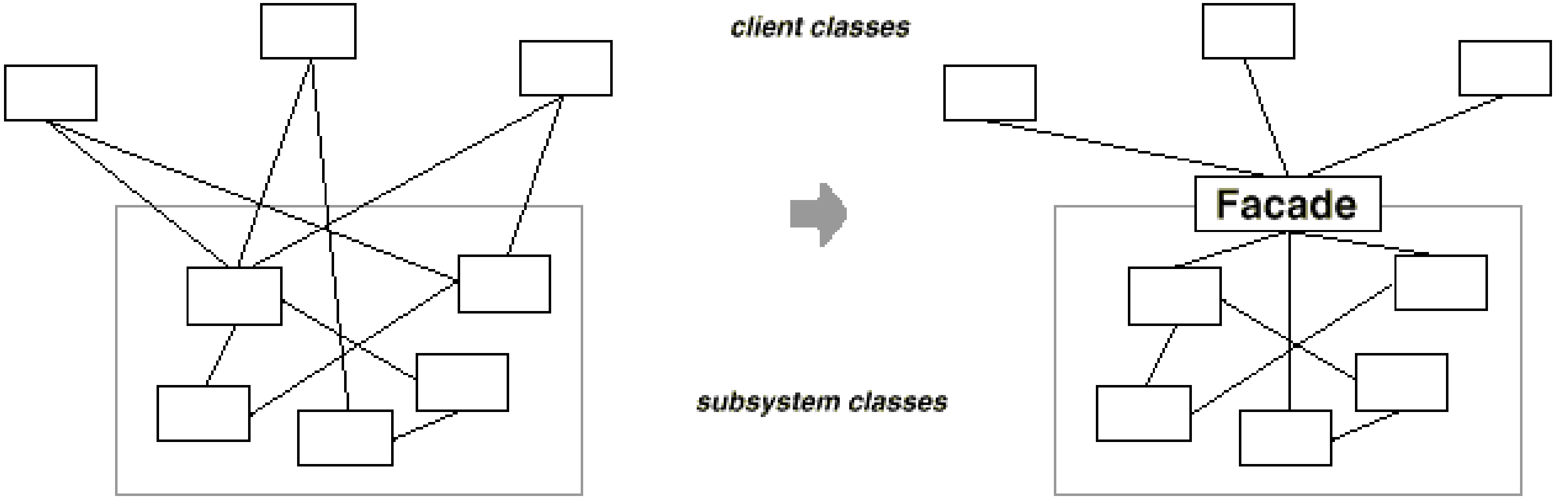
2. Facade (Façade) (Object structural pattern)

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

- Motivation
  - Structuring a system into subsystems helps reduce complexity
  - Subsystems are:
    - groups of classes, or
    - groups of classes and other subsystems
  - The interface exposed by the classes in a subsystem (or set of subsystems) can become quite complex
  - One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

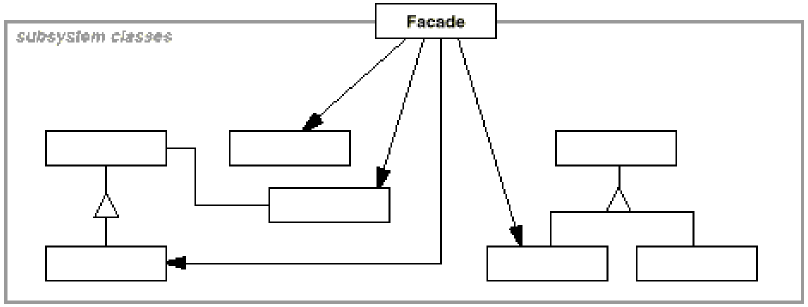
2. Facade

- Motivation



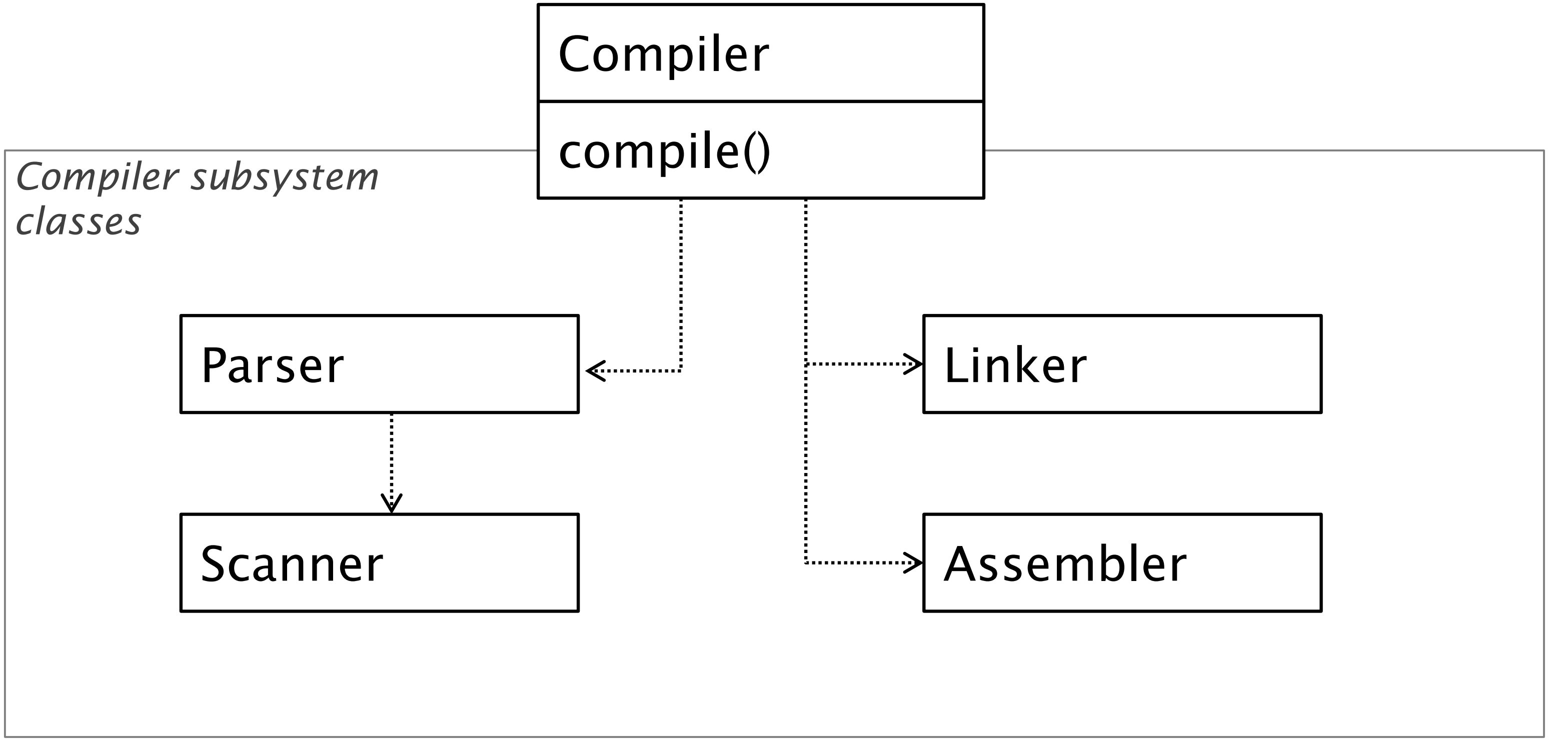
- Applicability
  - Use the Facade pattern:
    - To provide a simple interface to a complex subsystem. This interface is good enough for most clients; more sophisticated clients can look beyond the facade.
    - To decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

2. Facade



- Consequences
  - Benefits
    - It hides the implementation of the subsystem from clients, making the subsystem easier to use
    - It promotes weak coupling between the subsystem and its clients. This allows you to change the classes in the subsystem without affecting the clients.
    - It reduces compilation dependencies in large software systems
    - It simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others
    - It does not prevent sophisticated clients from accessing the underlying classes
    - Note that Facade does not add any functionality, it just simplifies interfaces
  - Liabilities
    - It does not prevent clients from accessing the underlying classes!

Facade Pattern Example - Compiler Subsystem



Facade Pattern Example - Compiler Subsystem

```
public class Scanner {
  public void scan(String sourceFile) {
    System.out.println("Started scanning " + sourceFile);
  }
}
```

`Scanner` **class**

```
public class Parser {

  private Scanner scanner = new Scanner();

  public void parse(String sourceFile) {
    scanner.scan(sourceFile);
    System.out.println("Started parsing " + sourceFile);
  }
}
```

`Parser` **class**

Facade Pattern Example - Compiler Subsystem

```
public class Assembler {

  public String assemble(String sourceFile) {
    sourceFile = sourceFile.toLowerCase().replaceAll(".asm",
                                                     ".obj");
    System.out.println("Translated to binary object code " +
                                               sourceFile);
    return sourceFile;
  }
}
```

Assembler class

```
public class Linker {

  public String link(String sourceFile) {
    sourceFile = sourceFile.toLowerCase().replaceAll(".obj",
                                                     ".exe");
    System.out.println("Linked to executable " + sourceFile);
    return sourceFile;
  }
}
```

Linker class

Facade Pattern Example - Compiler Subsystem

```
public class Compiler {

  Parser parser = new Parser();
  Assembler assembler = new Assembler();
  Linker linker = new Linker();

  public void compile(String sourceFile) {
    String orgSrcFile = sourceFile;
    parser.parse(sourceFile);
    sourceFile = compileInternal(sourceFile);
    sourceFile = assembler.assemble(sourceFile);
    sourceFile = linker.link(sourceFile);

    System.out.println();
    System.out.println("Successfully compiled " + orgSrcFile);
    System.out.println("Final executable is " + sourceFile);
  }

  private String compileInternal(String sourceFile) {
    sourceFile = sourceFile.toLowerCase().replaceAll(".cpp", ".asm");
    System.out.println("Compiled to assembly " + sourceFile);
    return sourceFile;
  }
}
```

- `Compiler` class is the facade that puts all pieces together
- `Compiler` provides a simple interface for compiling source and generating code
- Knows which subsystem classes are responsible for a request
- Delegates client requests to appropriate subsystem objects

Facade Pattern Example - Compiler Subsystem

Test program to show Facade usage:

```
public class Test {

public static void main(String[] args){

    /**
*/
    Compiler compiler = new Compiler();
    compiler.compile("C:\\random.cpp");
  }

}
```

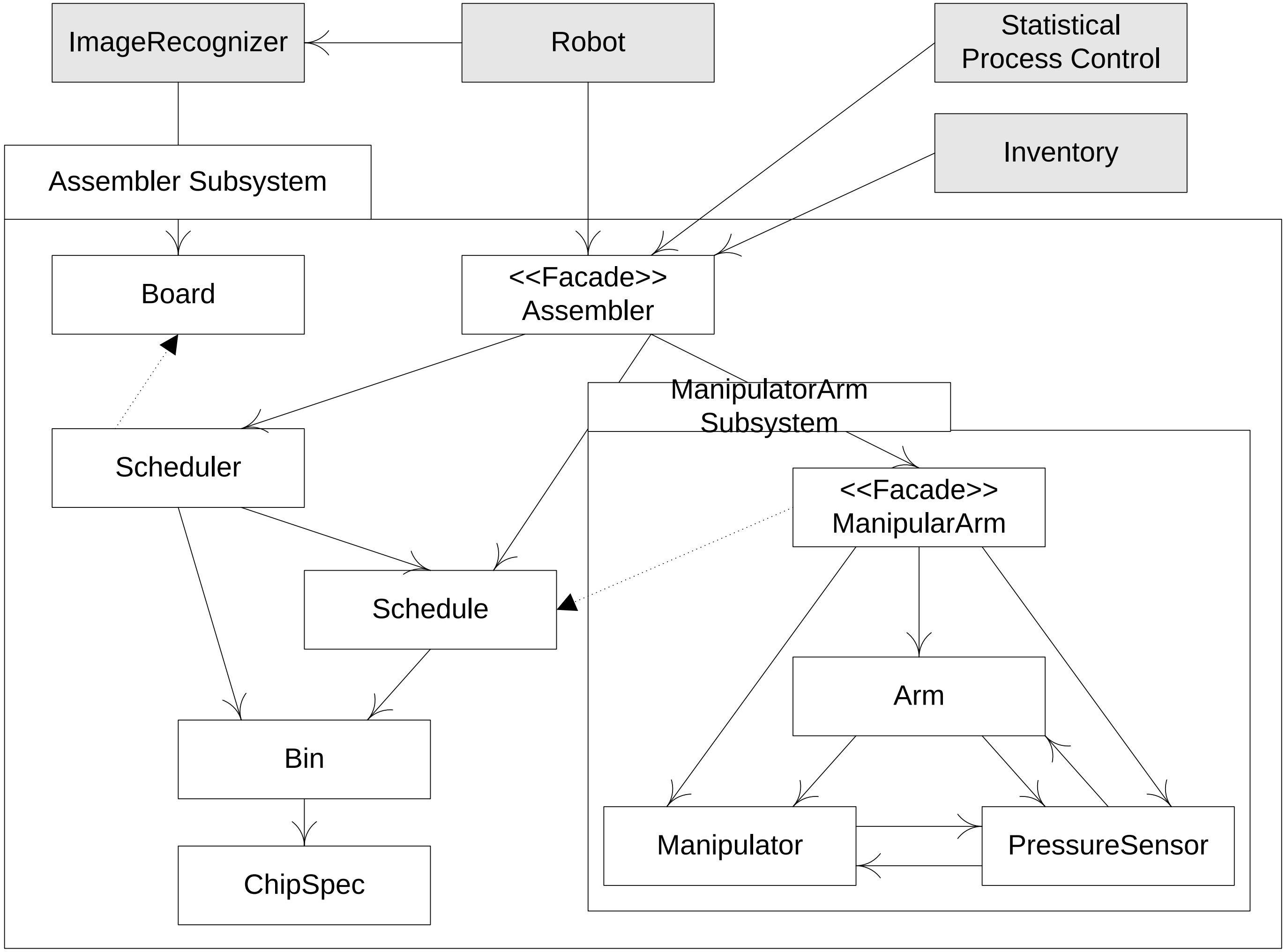
- The client deals with Compiler which generates the final executable.
- He doesn't need to know how Scanner, Parser, Assembler and Linker interact.

It gives the following output:

```
Started scanning C:\random.cpp
Started parsing C:\random.cpp
Compiled to assembly c:\random.asm
Translated to binary object code c:\random.obj
Linked to executable c:\random.exe

Successfully compiled C:\random.cpp
Final executable is c:\random.exe
```

Another Facade Pattern Example



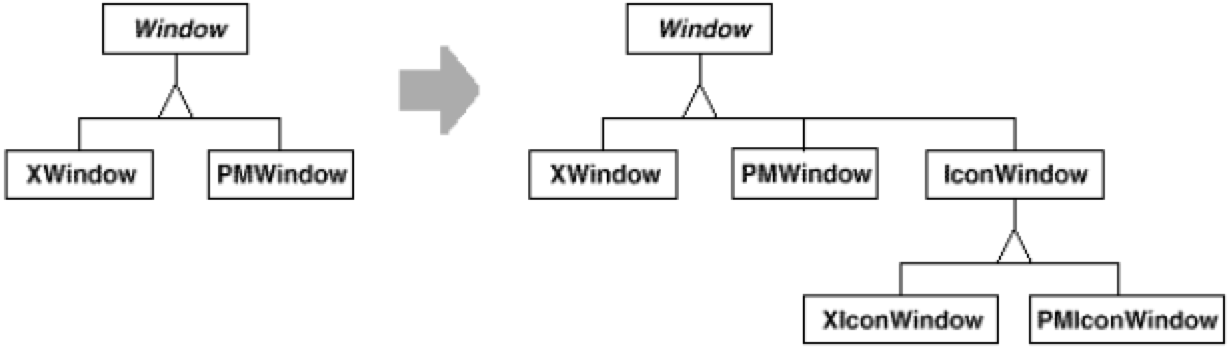
This example is taken from ValTech Design Patters tutorial

4. Bridge (Object structural pattern)

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently

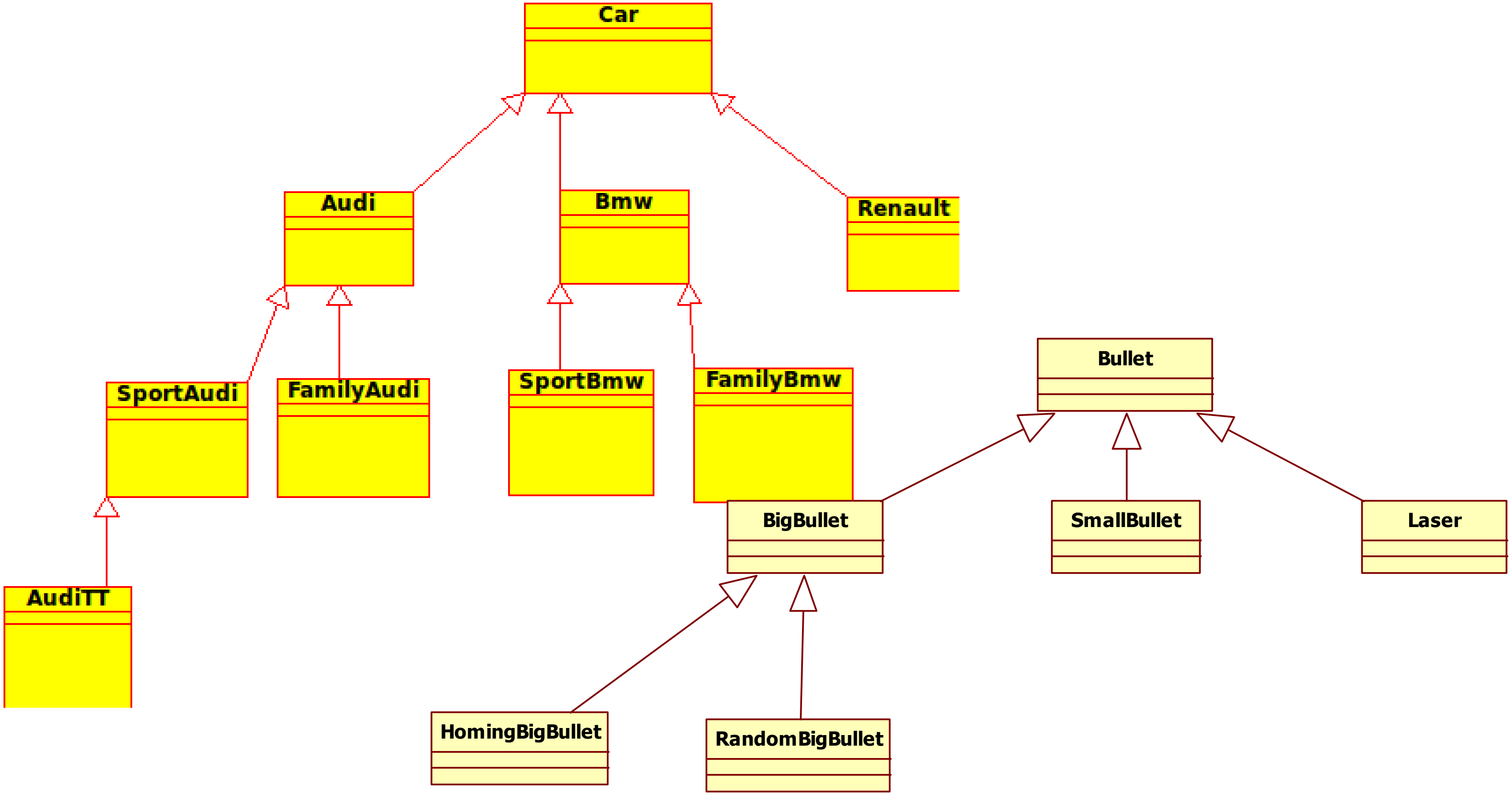
- Also Known As
  - Handle/Body

- Motivation



4. Bridge (Object structural pattern)

- Motivation



4. Bridge (Object structural pattern)

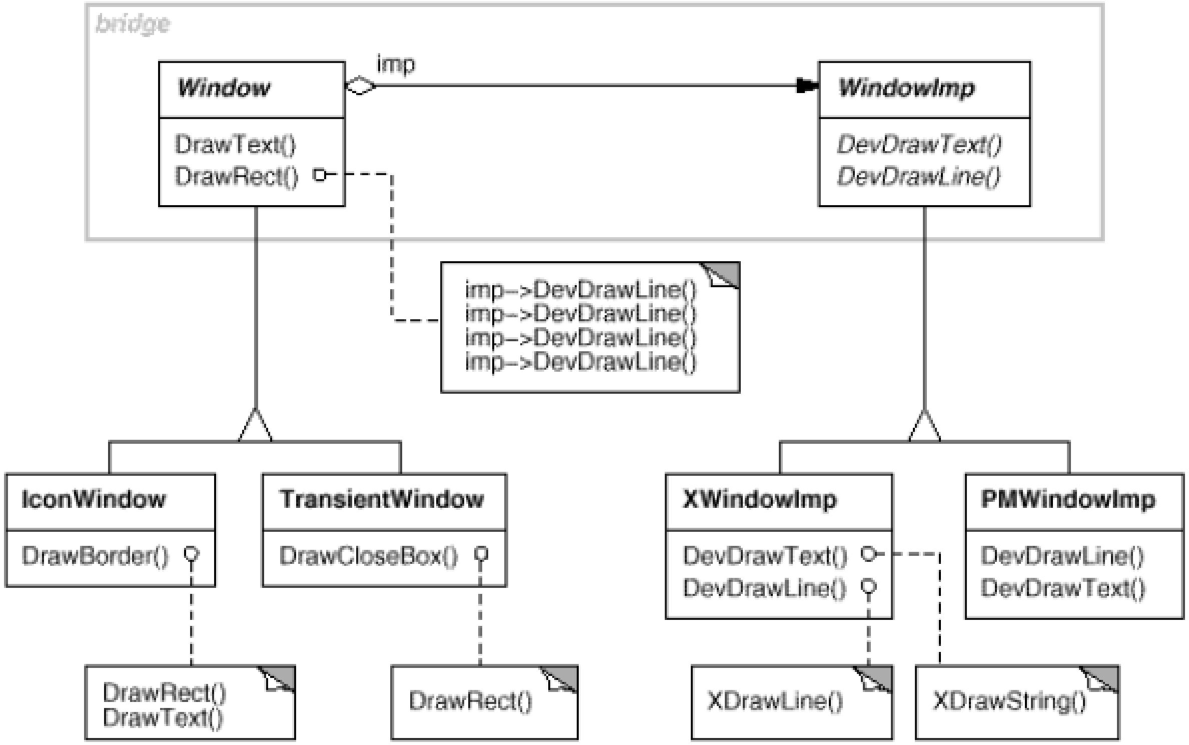
- Motivation
  - It’s inconvenient to extend the abstraction to cover different kinds of implementations.
  - It makes client code platform/implementation dependent.

4. Bridge (Object structural pattern)

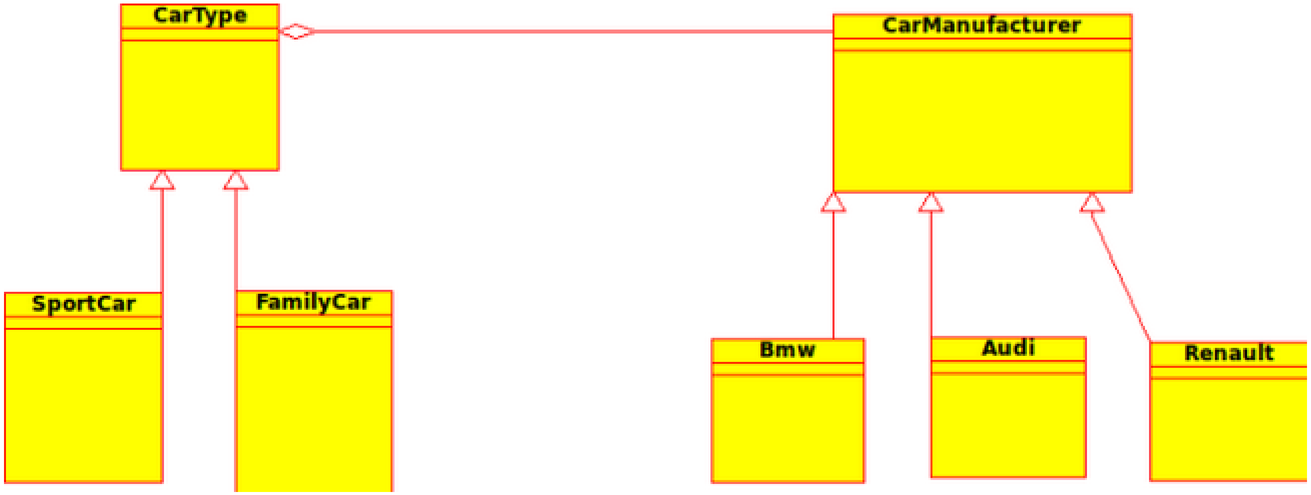
- Motivation

  — It’s inconvenient to extend the abstraction to cover different kinds of implementations.

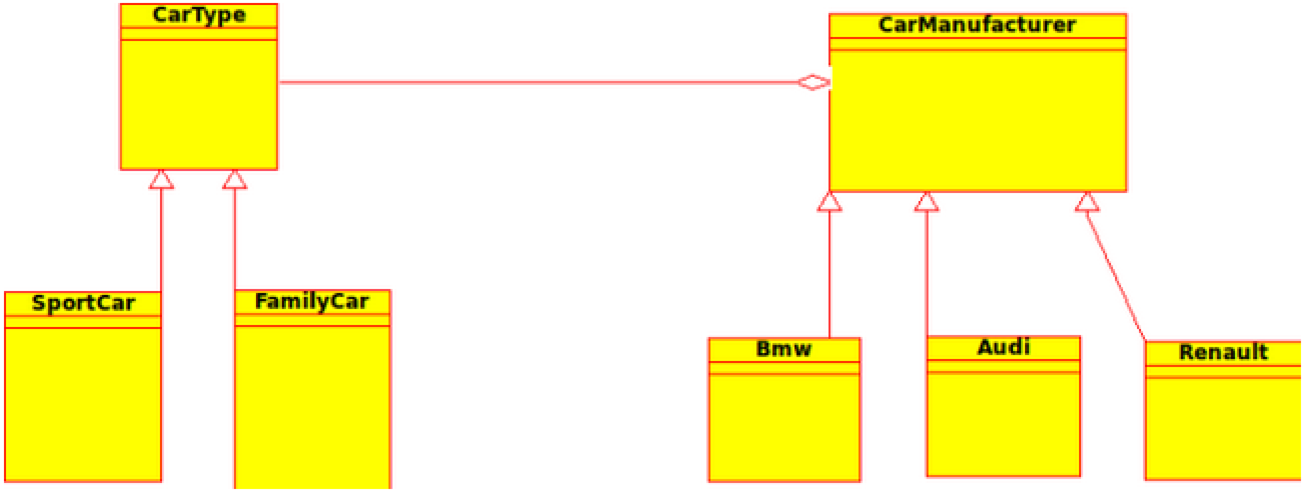
  — It m



pendent.


CarType
CarManufacturer
SportCar
FamilyCar
Bmw
Audi
Renault

Or

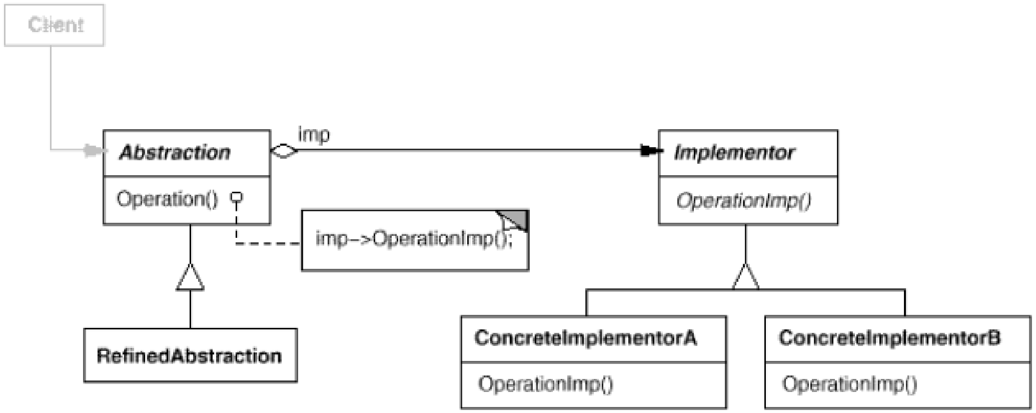

CarType
CarManufacturer
SportCar
FamilyCar
Bmw
Audi
Renault

4. Bridge (Object structural pattern)

- Applicability
  - Avoid a permanent biding between an abstraction and its implementation. For example, when the implementation must be selected or switched at run-time.
  - Both the abstractions and their implementations should be extensible by subclassing.
  - Changes in the implementation of an abstraction should not impact clients, i.e., the clients' code should not have to be recompiled.
  - (C++) Hide the implementation of an abstraction completely from clients.
  - The class inheritance hierarchy becomes ugly.
  - Share an implementation among multiple objects, and hide this fact from clients.

4. Bridge (Object structural pattern)

- Structure

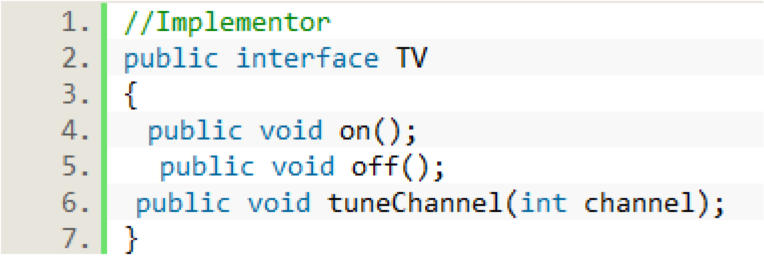


- Participants
  - Abstraction
    - Defines the abstraction's interface. Maintains a reference to an object of type Implementor
  - RefinedAbstraction
    - Extends the interface defined by Abstraction
  - Implementor
    - Defines the interface for implementation classes.
  - ConcreteImplementor
    - Implements the Implementor interface.

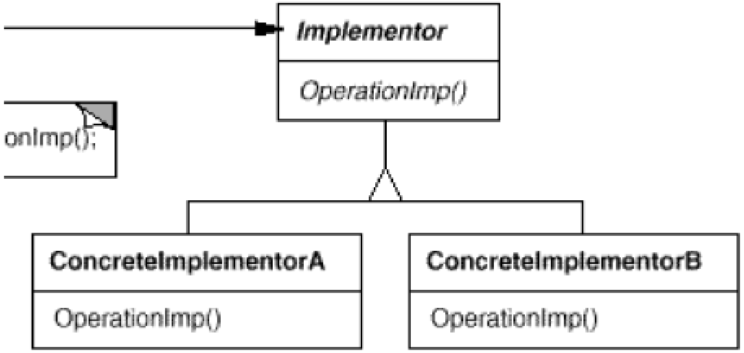
4. Bridge (Object structural pattern)

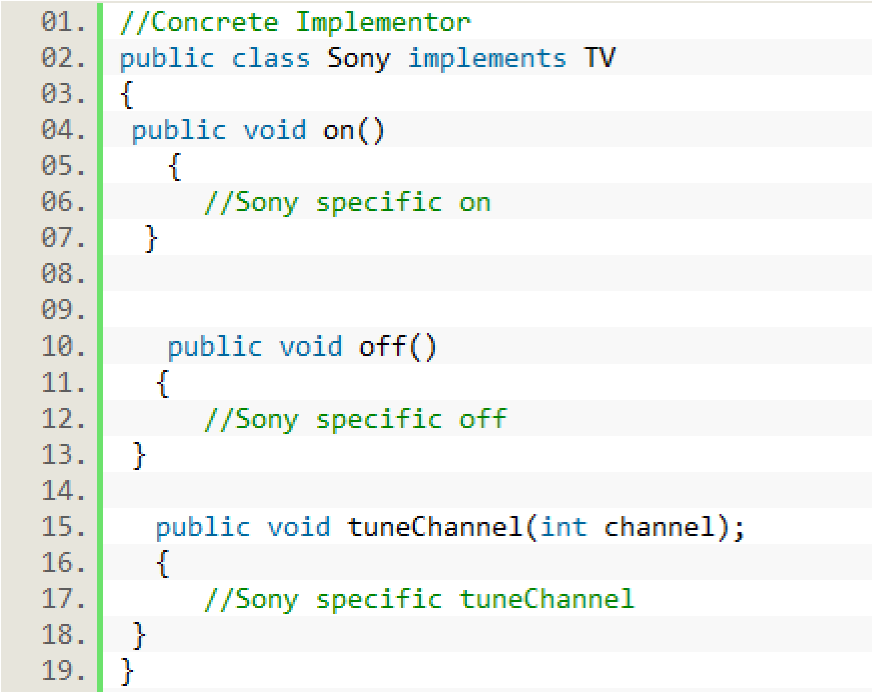
- Consequences
  - Benefits:
    - Decoupling interface and implementation.
    - Improved extensibility.
    - Hiding implementation details from clients.
  - Liabilities:
    - Increase complexity while providing flexibility
    - Possible performance issues

Bridge Example: TV and Remote



```java
//Implementor
public interface TV
{
  public void on();
   public void off();
 public void tuneChannel(int channel);
}
```

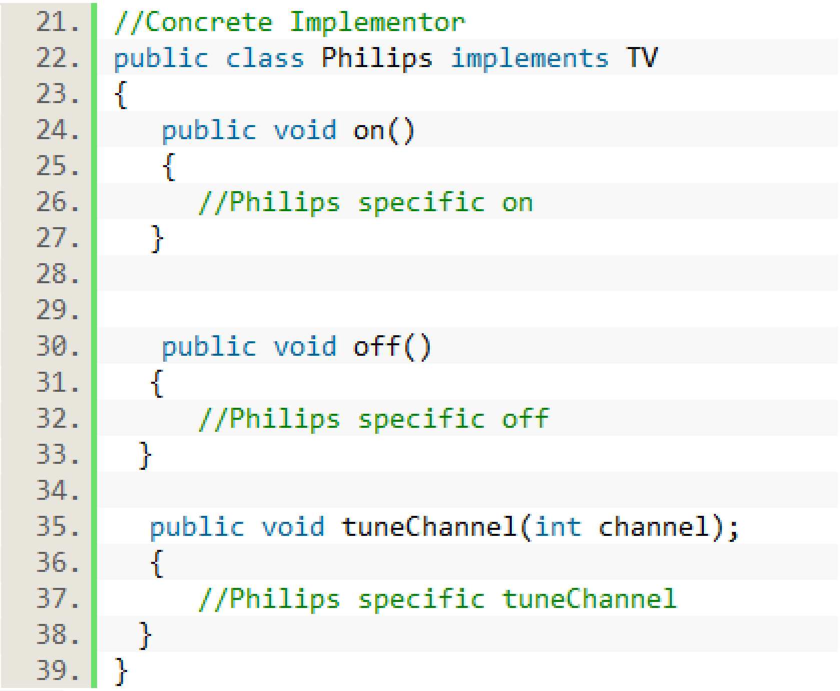




```java
//Concrete Implementor
public class Sony implements TV
{
 public void on()
    {
      //Sony specific on
   }


    public void off()
   {
      //Sony specific off
  }

   public void tuneChannel(int channel);
   {
      //Sony specific tuneChannel
  }
}
```

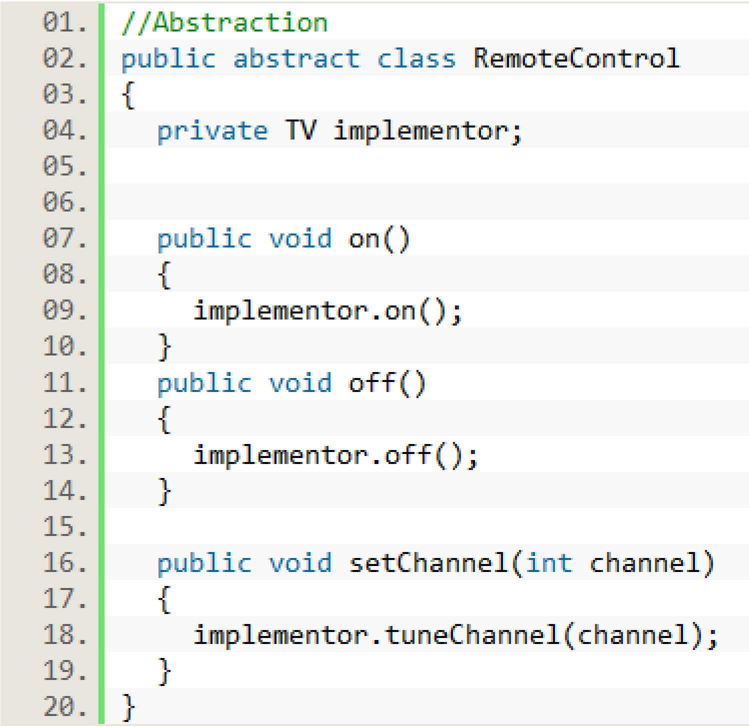


```java
//Concrete Implementor
public class Philips implements TV
{
    public void on()
    {
      //Philips specific on
   }


    public void off()
   {
      //Philips specific off
  }

   public void tuneChannel(int channel);
   {
      //Philips specific tuneChannel
  }
}
```

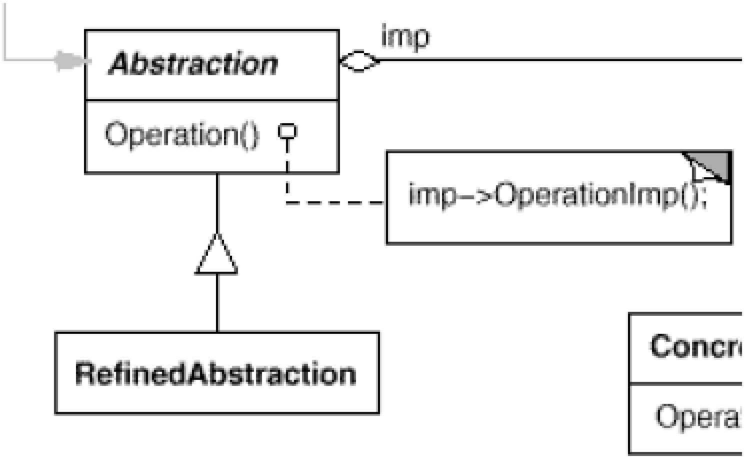
Bridge Example: TV and Remote

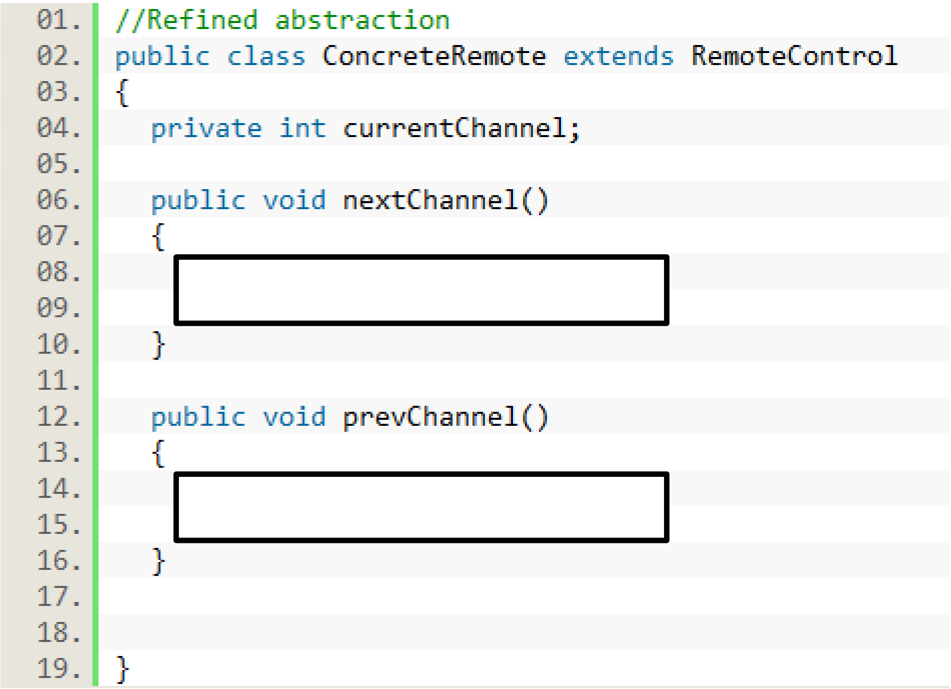


```
//Abstraction
public abstract class RemoteControl
{
  private TV implementor;


  public void on()
  {
    implementor.on();
  }
  public void off()
  {
    implementor.off();
  }

  public void setChannel(int channel)
  {
    implementor.tuneChannel(channel);
  }
}
```





```
//Refined abstraction
public class ConcreteRemote extends RemoteControl
{
  private int currentChannel;

  public void nextChannel()
  {


  }

  public void prevChannel()
  {


  }



}
```