Structural Patterns

Intro

Adapter

Façade

Structural Patterns

- How classes and objects are composed to form larger structures
  - How to form larger structures from individual parts, generally of different classes
  - How to glue different pieces of a system together in a flexible and extensible fashion.
  - How to recast pieces that don't fit (but that you need to use) into pieces that do fit.
- Helps to guarantee that when one of the parts changes, the entire structure does not need to change.

Structural design patterns

- Similar to *data structures, but*
  - structural design patterns also specify the *methods* that connect objects, not merely the references between them.
  - A structural design pattern also describes *how data moves* through the pattern whereas data structures only describe how data is arranged in the structure.

Structural Patterns

- Adapter
  - To convert interface of one class to another, so that unrelated or incompatible classes can work together
- Bridge
  - To decouple abstraction from implementation, so that they can vary independently
- Composite
  - To treat elements of a tree structure, both individual and composite elements uniformly
- Decorator
  - To add responsibilities to objects dynamically
- Façade
  - To provide unified interface to a set of subsystems for ease of use
- Flyweight
  - To support large number of fine-grained objects efficiently
- Proxy
  - To provide a placeholder for an object to control access to it.

ADAPTER

Motivating example:

- We are developing an e-commerce platform that initially uses the payment gateway, called PayFast, for processing payments.
- Unfortunately, our code is tightly coupled with the PayFast API.
- Now, we want to expand our platform to support another payment gateway, QuickPay, without changing the existing codebase that interacts with PayFast.

Motivating example

- **Problem:** PayFast and QuickPay APIs have different interfaces.
  - PayFast::pay() but QuickPay::qpay()
- The existing codebase is designed to work with PayFast, and directly integrating QuickPay requires significant changes.
  - Maybe even introduce conditionals all over the place.

- **General problem:** How to make unrelated or incompatible classes work together?

Adapter

- **Intent**
  - **Convert interface** of one class to another, so that unrelated or incompatible classes can work together
  - A.k.a Wrapper
- **Applicability**
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that collaborates with unrelated or unforeseen classes, i.e. don’t have compatible interfaces.

Adapter

- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
  - Incompatible method signatures

In short: When you've got *this*, and you need *that*, *Adapter* solves the problem.

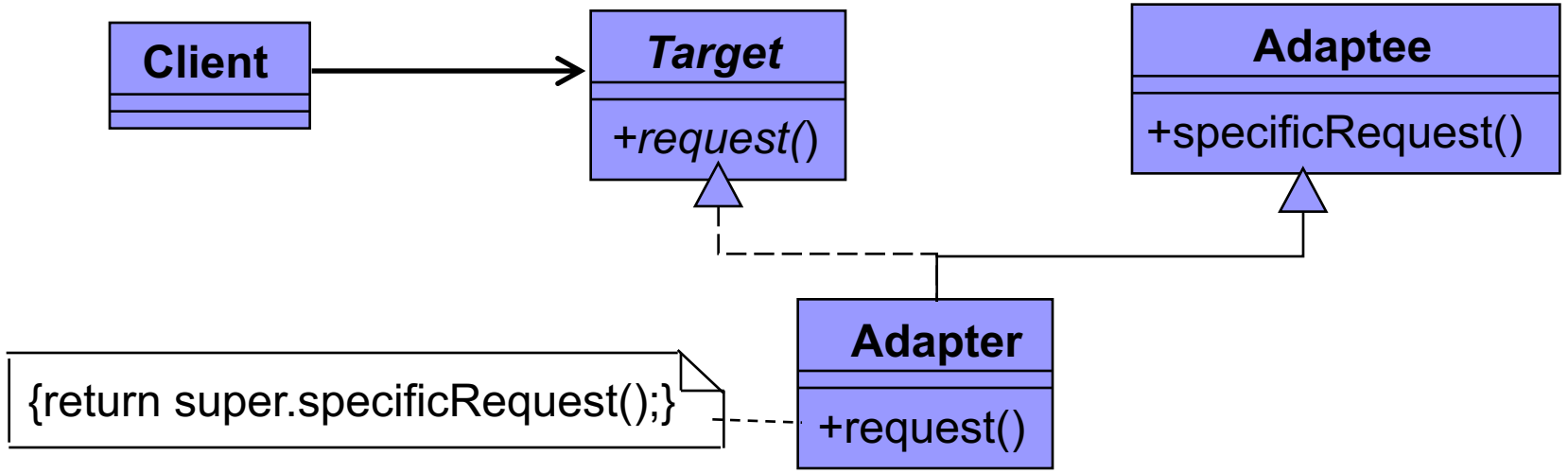
Defining problem

```
class Incompatible {
    public void g() {…}
    public void h() {…}
}
interface WhatIWant {
    void f();
}

Mycode(){
    WhatIwant target=…;
    target.f();
}
```

- Convert interface of one class to another, so that unrelated or incompatible classes can work together

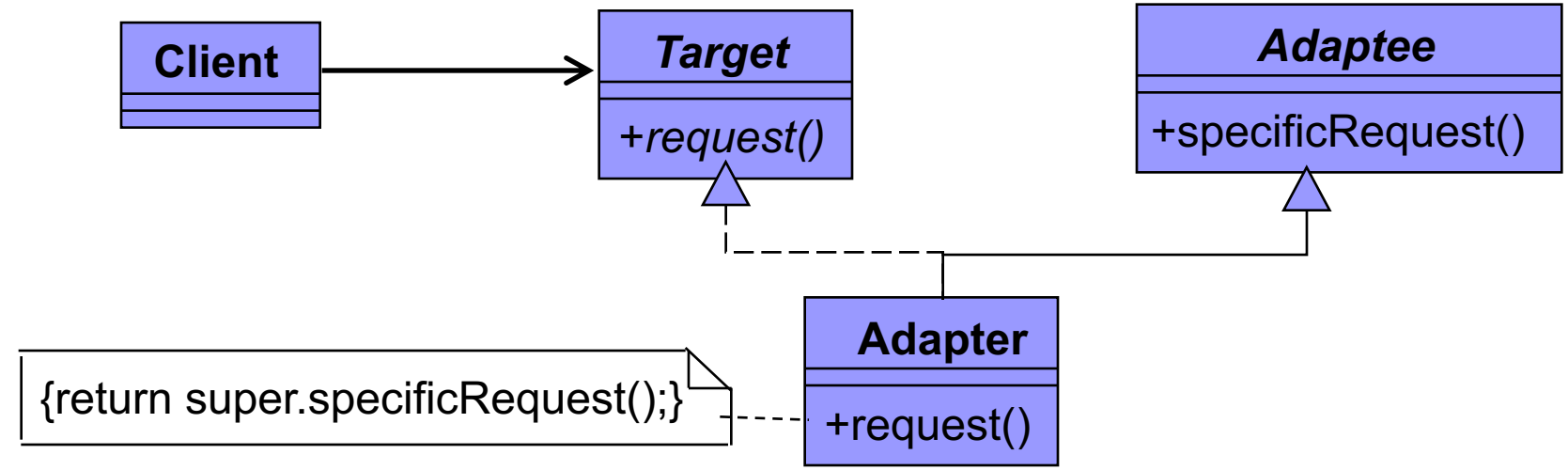
Adapter-Structure



Class Adapter pattern

Adapter-Structure

Java

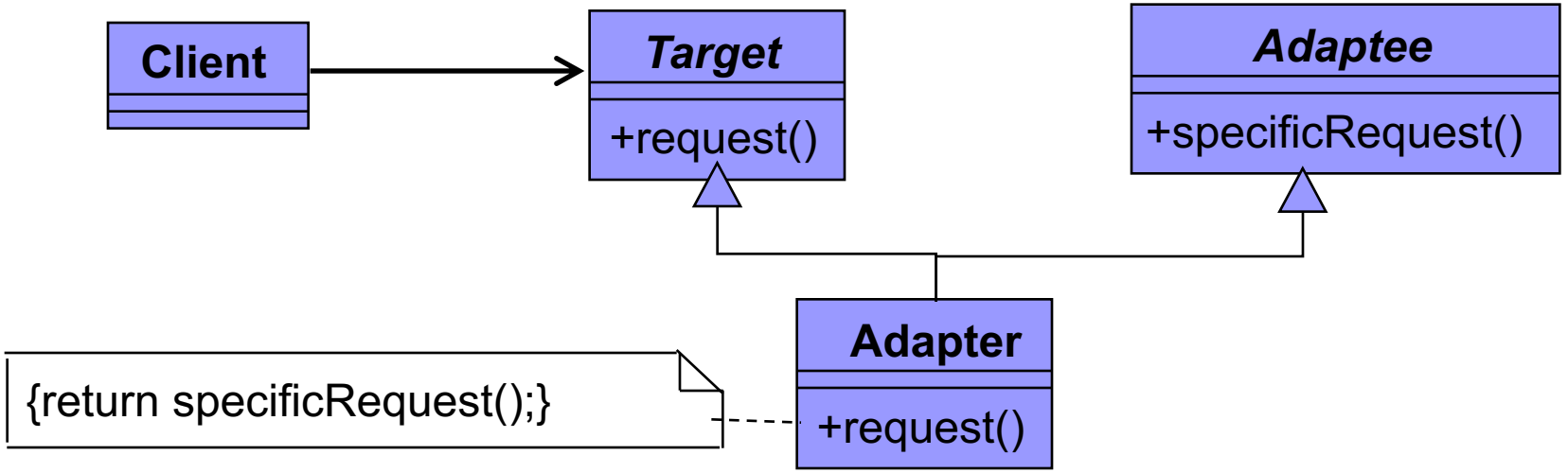



```
class Adaptee {     //Adaptee
   public void g() {…}
   public void h() {…}
}
 interface WhatIWant {//Target
   public void f();
}
```

```
class Adapter
        extends Adaptee
        implements WhatIWant{
    void f(){
        super.g();
        super.h();}
}
```

Adapter-Structure

C++

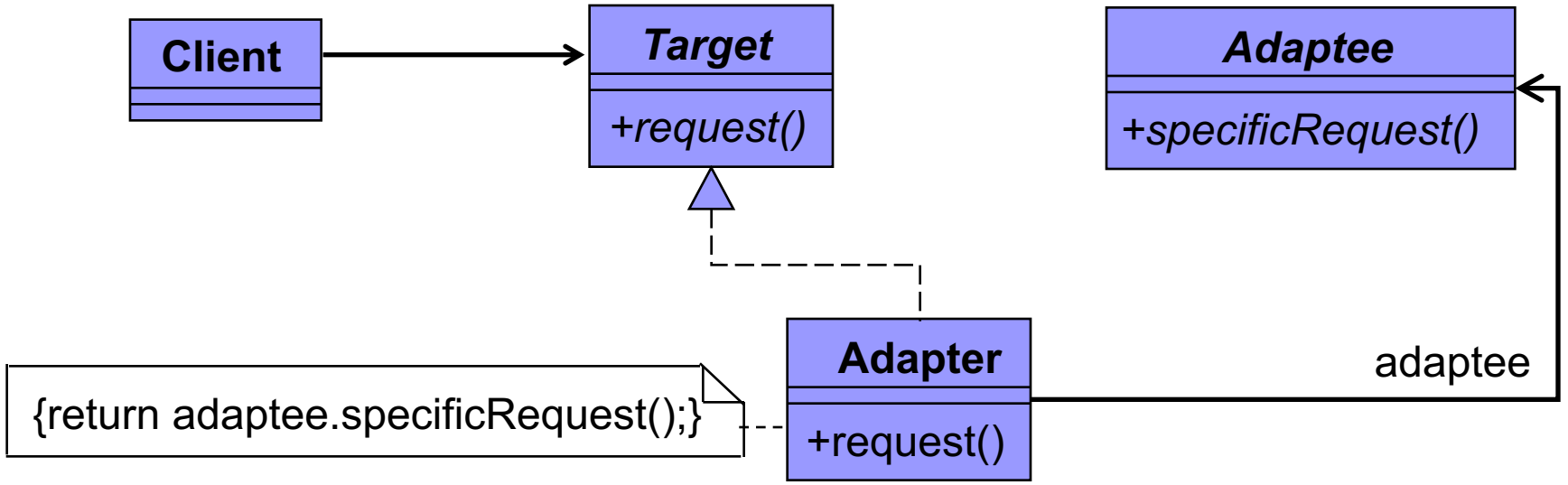



```
class Adaptee { //Adaptee
   public: void g() {….}
        void h() {…..}
};
class WhatIWant {//Target
   public:
        virtual void f()=0;
};
```

```
class
Adapter: public Adaptee,
         private WhatIHave {
    public:
       void f()
       {
         g();h();
        }
}
```

Adapter-Structure

Object Adapter pattern

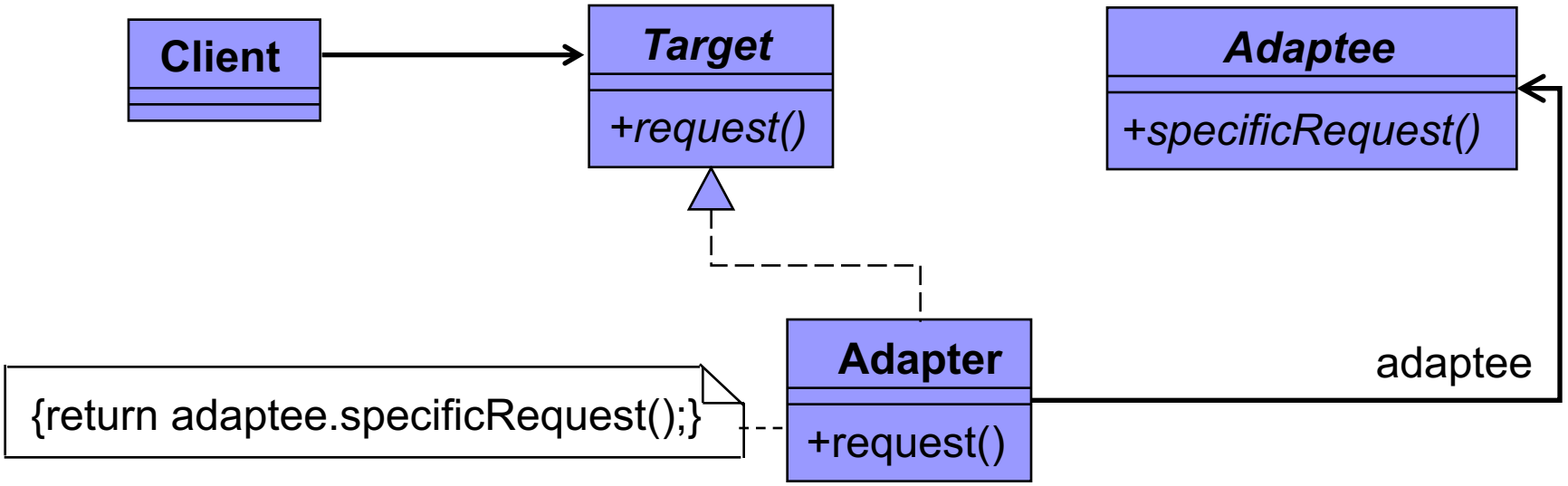


JAVA

Adapter-Structure

```
class Adaptee {    //Adaptee
   public void g() {…}
   public void h() {…}
}
 interface WhatIWant {   //Target
   public void f();
}
```

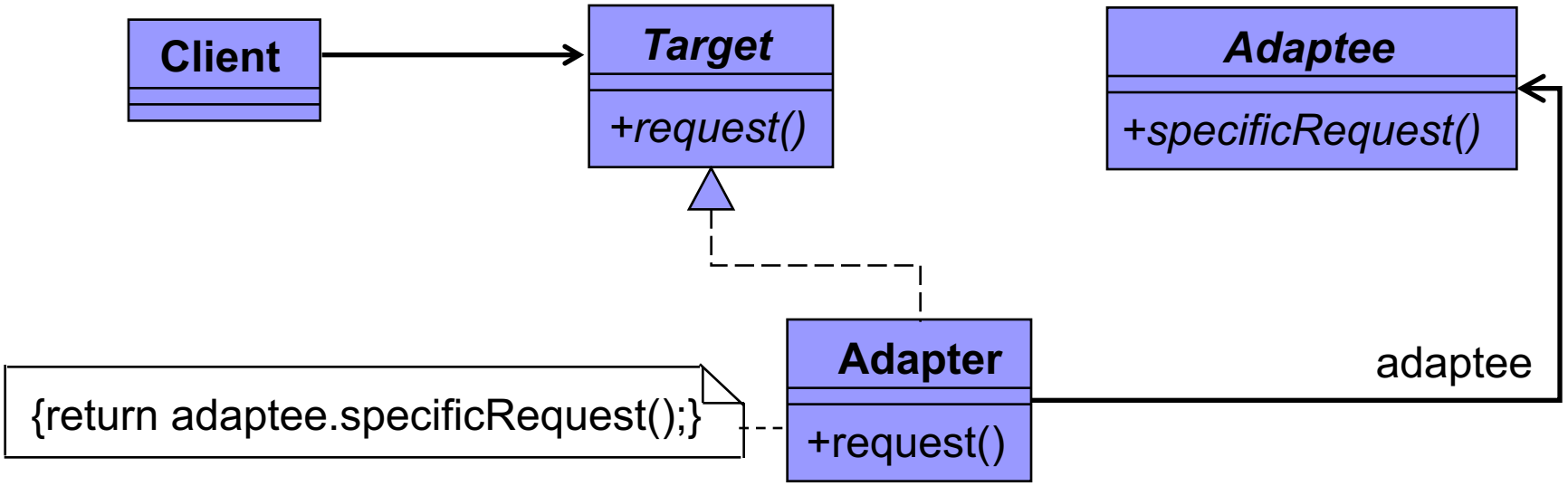
```
class Adapter
    implements WhatIWant{
    private Adaptee adaptee;
    public void f(){
            adaptee.g();
            adaptee.h();}
}
```



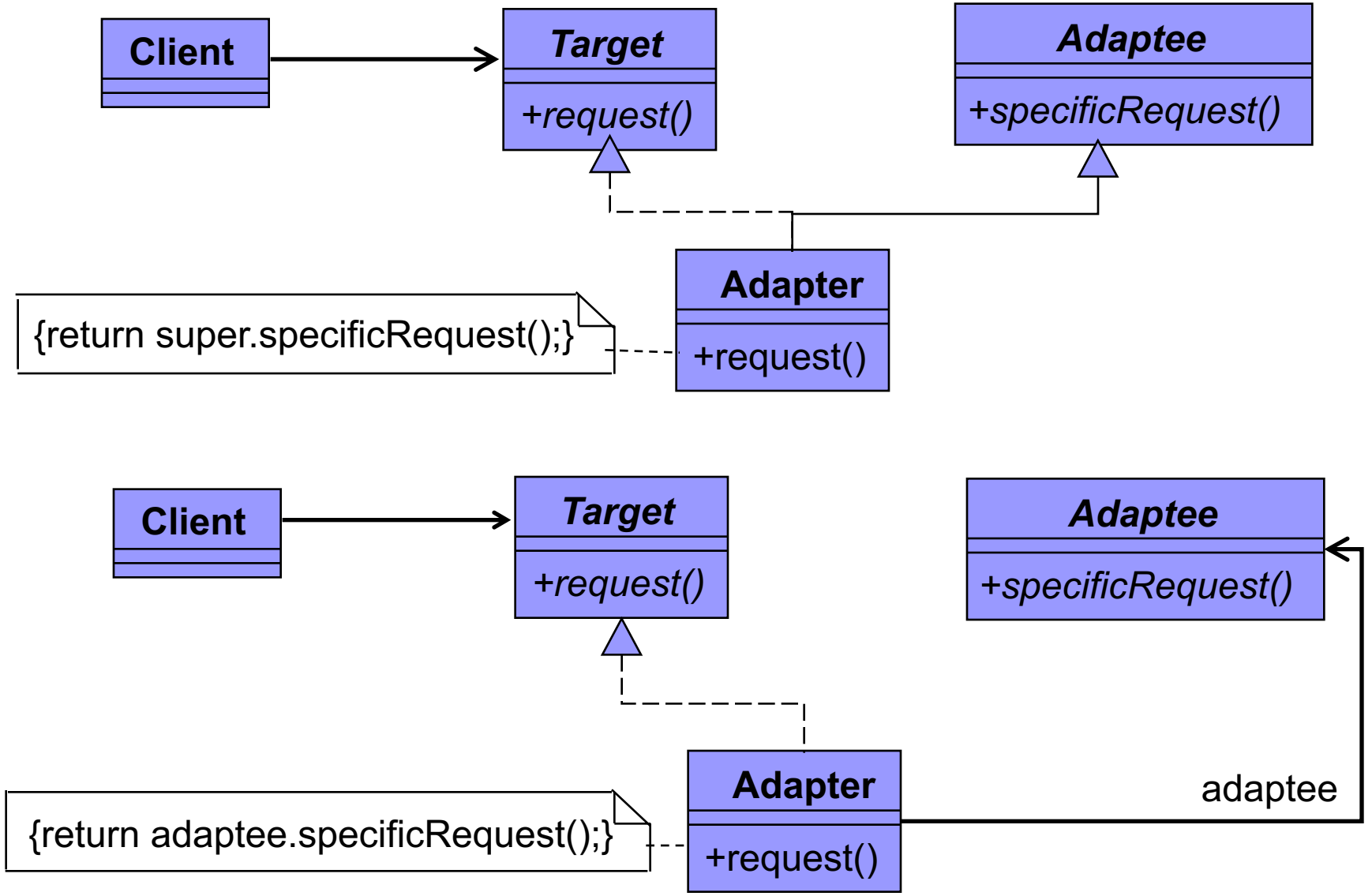
C++ Adapter-Structure

```
class Adaptee {    //Adaptee
   public:  void g() {…}
            void h() {…}
};
 class WhatIWant {   //Target
   public: virtual void f()=0;
};
```

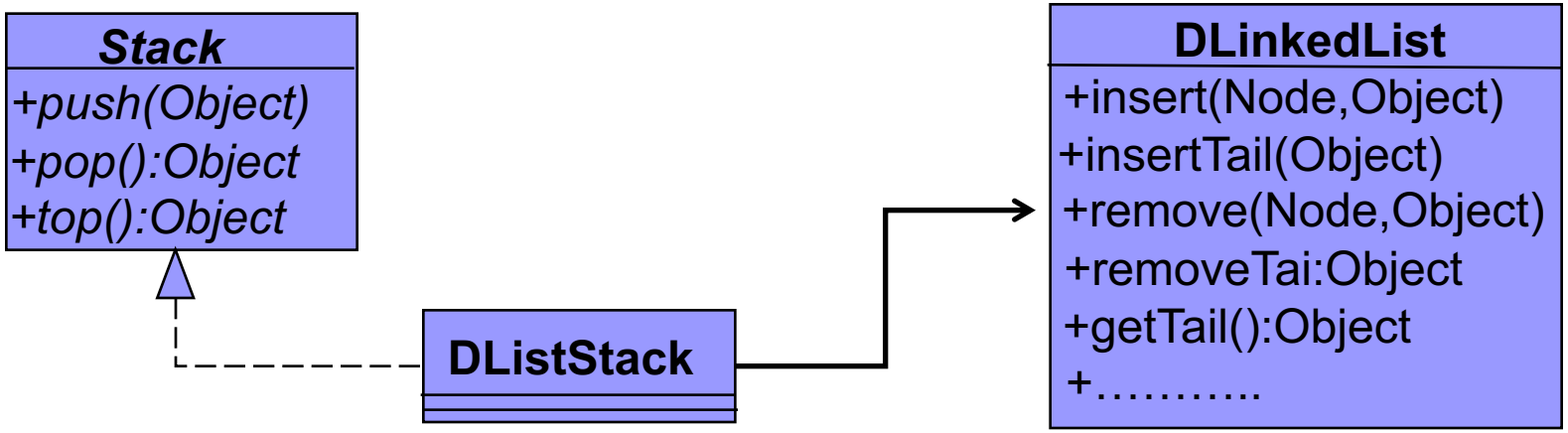
```
class Adapter: public WhatIWant{
    private:
        Adaptee* adaptee;
    public:
        void f(){ adaptee.g();
                  adaptee.h();}
};
```



Adapter-Structure



Example



- Client program uses a Stack.
- There is a double linked list implementation
- Adapt the linked list to a Stack interface

```
class DListStack implements Stack {
  private DLinkedList dlist;
  public DListStack(){dlist = new DLinkedList(); }
  public void push(Object o){dlist.insertTail (o);}
  public Object pop(){return dlist.removeTail ();}
  public Object top(){return dlist.getTail ();}}
```

Example: Adapters in STL (C++)

- **std:stack** adapts *deque, list, vector*
- **std:queue** adapts *deque* and *list*
- **std:priority_queue** adapts *vector* and *list*

```
template<typename T, typename Container = std::deque<T>>
class stack;
```

- They convert one interface into another interface clients expect
  - i.e. implement priority_queue operations using vector operations via delegation.

```
template<typename T, typename Container = std::vector<T>,
         typename Compare = std::less<typename Container::value_type>>
class priority_queue;
```

Possible uses of adapter

- We cannot change the library interface(code), since we may not have its source code.
  - Even if we did have the source code, we probably should not change the library for each domain-specific application.
- Sometimes a toolkit or class library cannot be used because its interface is incompatible with the interface required by an application.
- Want to create a reusable class that cooperates with unrelated classes with incompatible interfaces.

Example

- I have legacy Java CGI web server programs written using a java library .
- Servlets provide functionality similar to CGI programs, but are considerably more efficient.

- CGIVariables class in the old library stores all CGI environment variables in a hash table and allows access to them via a `get(String evName)` method.
- The servlet library has an HttpServletRequest class which has a `getX()` method for each CGI environment variable.
- We want to use servlets. Should we rewrite all of our existing Java CGI programs??

Soln: Let's minimize the recoding

- CGIAdapter class with the same interface (a get() method) as the original CGIVariables class, but puts a wrapper around the HttpServletRequest class.
  - Change CGIVariables to CGIAdapter class in existing code
  - but the form of the get() method invocations need not change.

Implementation issues

- C++ adapters with multiple inheritance
  - Inherit publicly form Target but privately from Adaptee
  - So that the Adapter would be a subtype of Target but not of Adaptee.
- How much adaptation?
  - Simple interface conversion that just changes operation names and order of arguments
  - Totally different set of operations

Implementation issues

- two-way adapter
  - Useful if you need to use both the old and the new interface
    - when some components use the old, some use the new
  - A two-way adapter supports both the Target and the Adaptee interface.
  - It allows an adapted object (Adapter) to appear as an Adaptee object or a Target object
  - Two-way adapter conforms to both of the adapted classes and can work in either system.

2 –way adapter example

- We want an adapter that acts as

  a SquarePeg or a RoundPeg

- public interface IRoundPeg {
  public void insertIntoHole(String msg); }
- public interface ISquarePeg {
  public void insert(String str);}

- In C++ multiple inheritance will do the job

C++

```
class IRoundPeg {
    public: virtual void insertIntoHole(const String& msg)=0;
};
class ISquarePeg {
      public: virtual void insert(const String& str) =0;
};
class PegAdapter: public ISquarePeg, public IRoundPeg {
 public:
        void insert(const String& str) override{
                insertIntoHole(str);}
        void insertIntoHole (const String& msg) override {
                insert(msg);}
};
```

JAVA

```
public class PegAdapter
                implements ISquarePeg, IRoundPeg {
   private RoundPeg roundPeg;
   private SquarePeg squarePeg;
  public PegAdapter(RoundPeg pegR, SquarePeg pegS ) {
        this.roundPeg = pegR;
        this.squarePeg = pegS;
  }
   public void insert(String str) {
       roundPeg.insertIntoHole(str);
  }
   public void insertIntoHole(String msg){
         squarePeg.insert(msg);}
}
```

Client code example

```
public class TestPegs {
 public static void main(String args[]) {
   RoundPeg roundPeg = new RoundPeg(); // Create some pegs.
   SquarePeg squarePeg = new SquarePeg();

 // Create a two-way adapter and do an insert with it.
   ISquarePeg roundToSquare = new PegAdapter(roundPeg);
    roundToSquare.insert("Inserting round peg...");

// Create a two-way adapter and do an insert with it.
   IRoundPeg squareToRound = new PegAdapter(squarePeg);
   squareToRound.insertIntoHole("Inserting square peg...");
}
```

Adapter-Consequences

- Class Adapter
  - Lets Adapter to overwrite some adaptee behavior
  - Introduces only one object, no additional indirection needed to get to the adaptee
  - Won’t work when we want to adapt a class and all its subclasses
- Object Adapter
  - A single Adapter can work with many adaptees
  - Harder to overwrite some adaptee behavior

Adapter -Known uses

- Adapters in Java IO
  - java.io.InputStreamReader adapts java.io.InputStream to have a correct java.io.Reader interface
  - java.io.OutputStreamWriter adapts OutputStream to a Writer interface
- STL adaptors in C++:
  - Container adaptors include stack, queue, priority queue
  - Iterator adaptors include reverse iterators, std::back_inserter() iterators
  - Function adaptors include negators and binders

Adapter

- Intent
  - Convert interface of one class to another, so that unrelated or incompatible classes can work together
- Applicability
  - You want to use an existing class and its interface does not match the one you need
  - You want to create a reusable class that collaborates with unrelated or unforeseen classes, i.e. don’t have compatible interfaces.
  - *(object adapter only)* You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing each one. An object adapter can adapt the interface of its parent class.

FACADE

Façade

- **“If something is ugly, hide it inside an object.”**

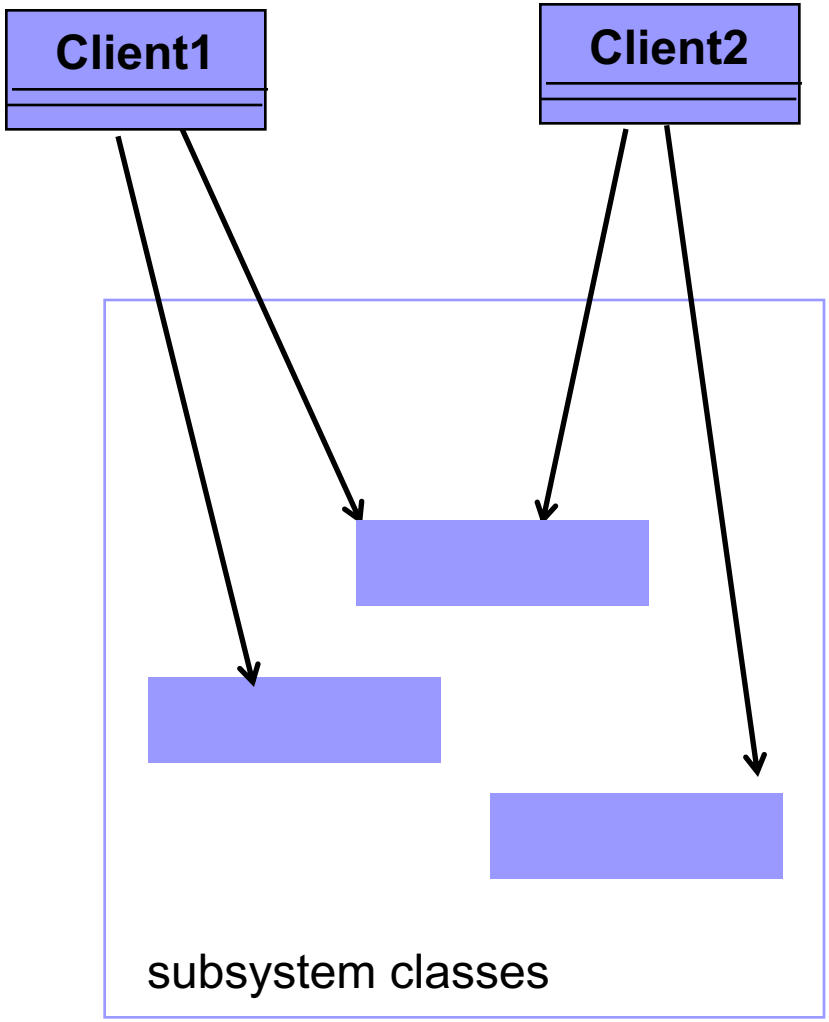
- If you have a rather confusing collection of classes and interactions
  - that the client programmer doesn’t really need to see,
- Create an interface that is useful for the client programmer and that only presents what’s necessary.
- Only clients needing more customizability will need to look beyond the facade

Façade

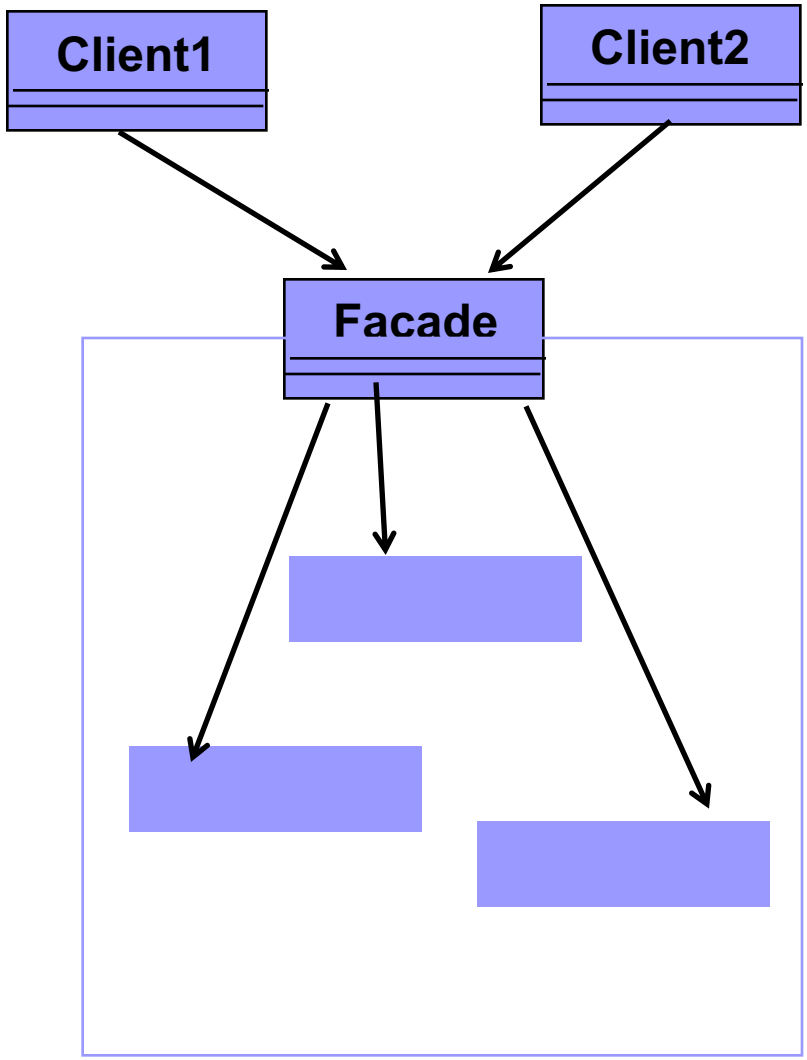
- Intent
  - Provide **unified** interface to a set of interfaces of a subsystem. Defines higher-level interface that makes the subsystem easier to use
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - When there are many dependencies between clients and the implementation classes
  - You want to layer your subsystem

Façade Applicability

- You want to provide a simple interface to a complex subsystem
  - Good enough for most clients
- When there are many dependencies between clients and the implementation classes of an abstraction
  - Decouple clients from subsystem via façade
  - Subsystem portability and independence
- You want to layer your subsystem
  - Each Façade will be entry point of a subsystem
  - If subsystems are dependent, they'll communicate through their facades -> less coupling

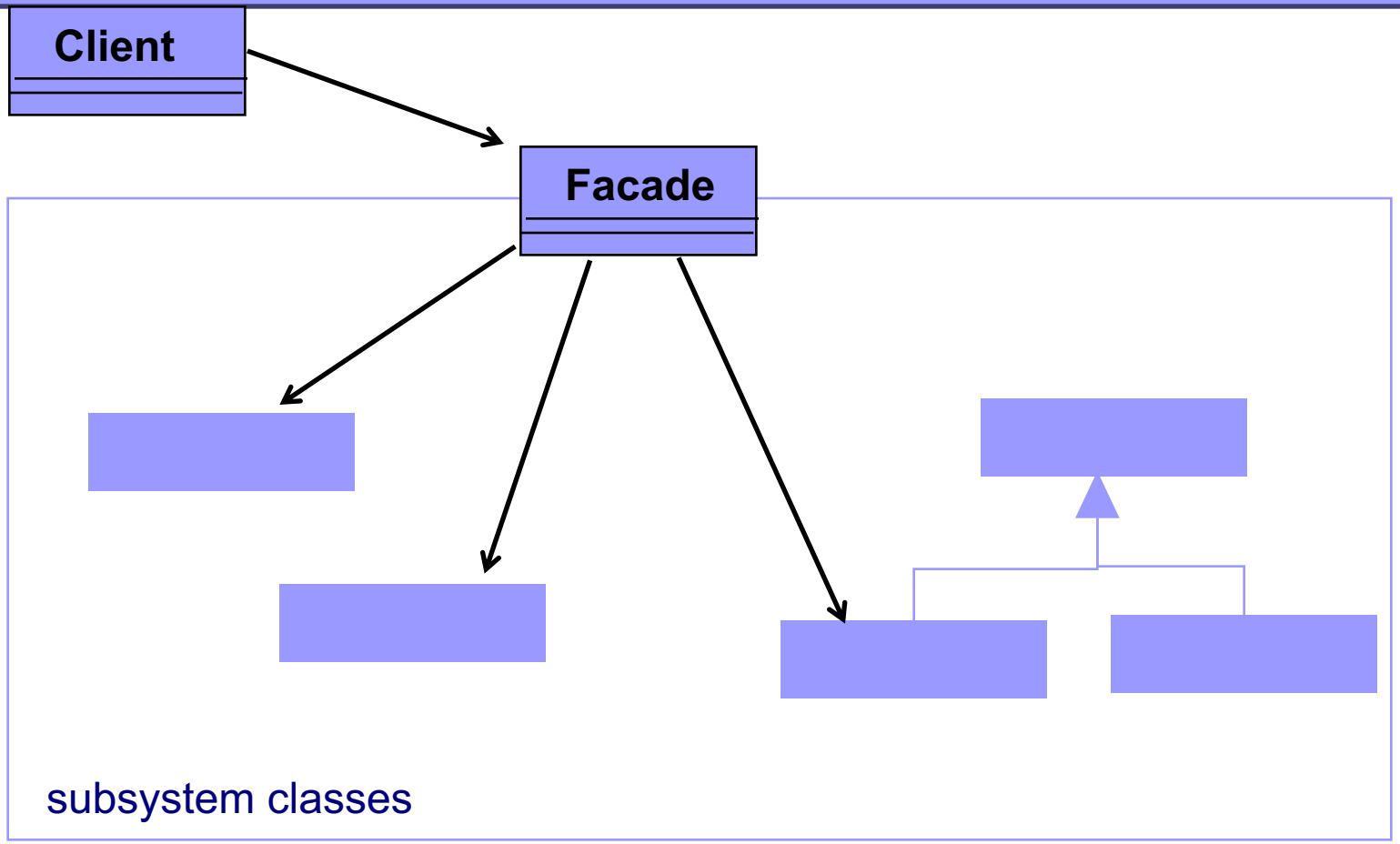


Too much coupling to low level classes



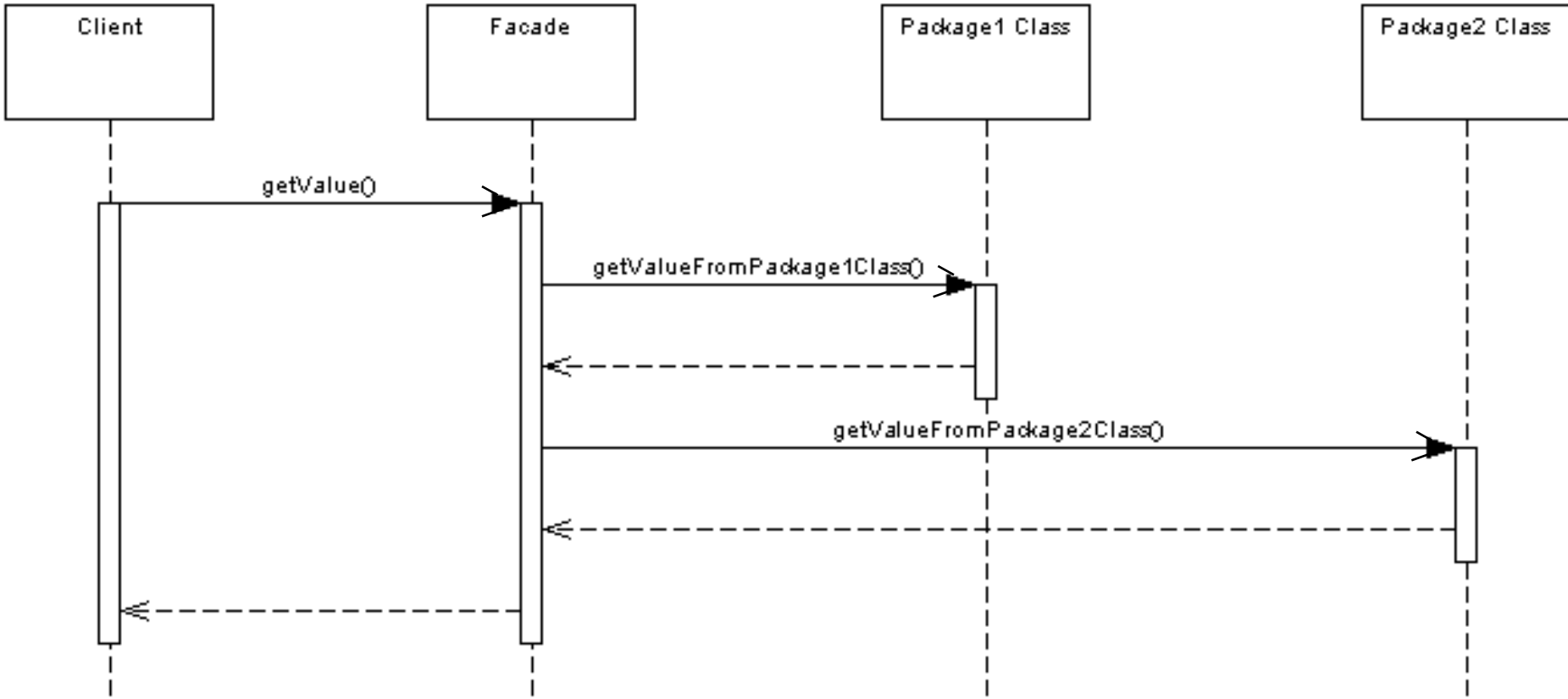
Coupling reduced

Façade -Structure

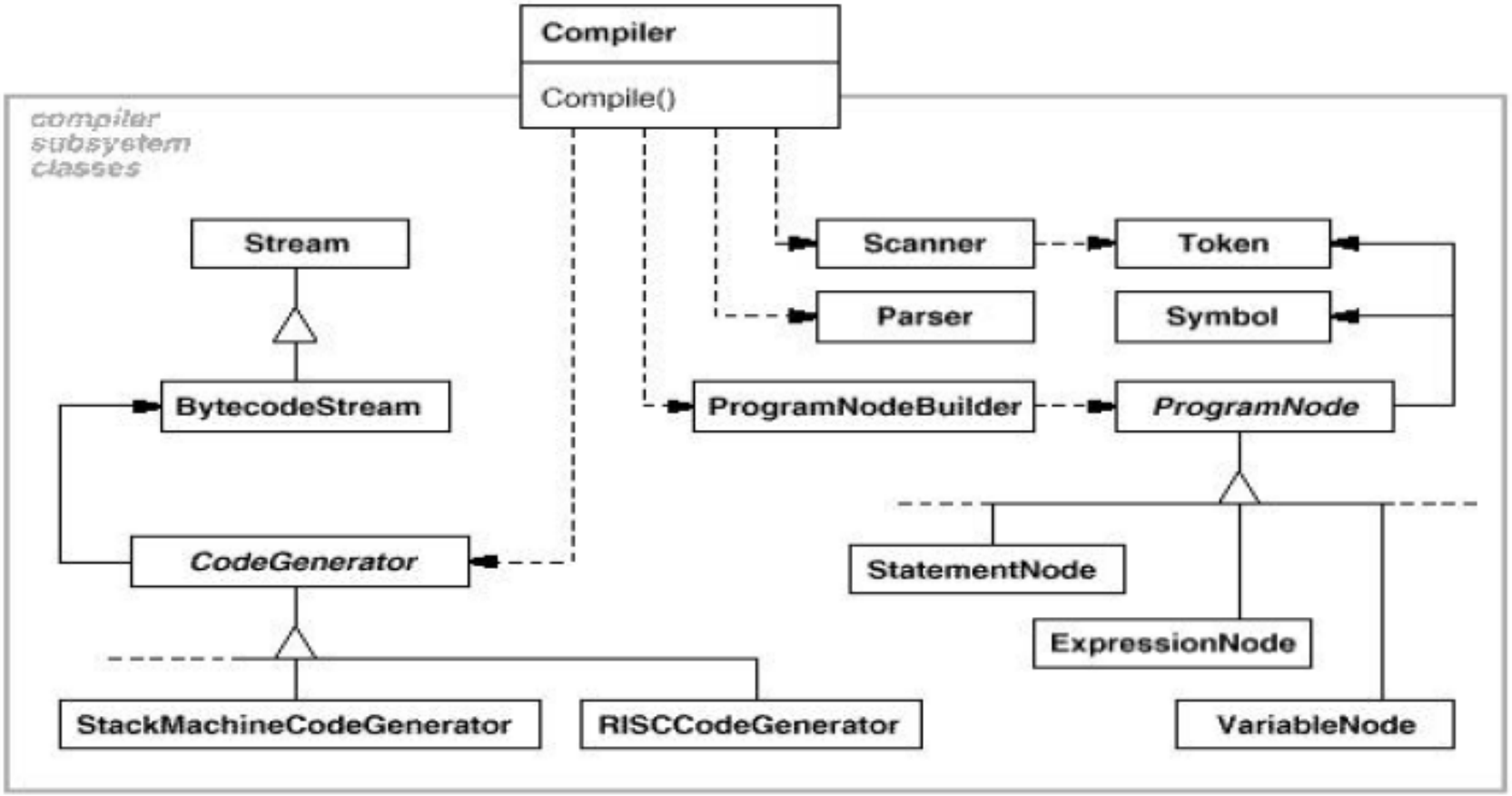


For example, the java.net.URL is a façade.
This class provides access to the contents of URLs and hides many classes from its clients

How façade works

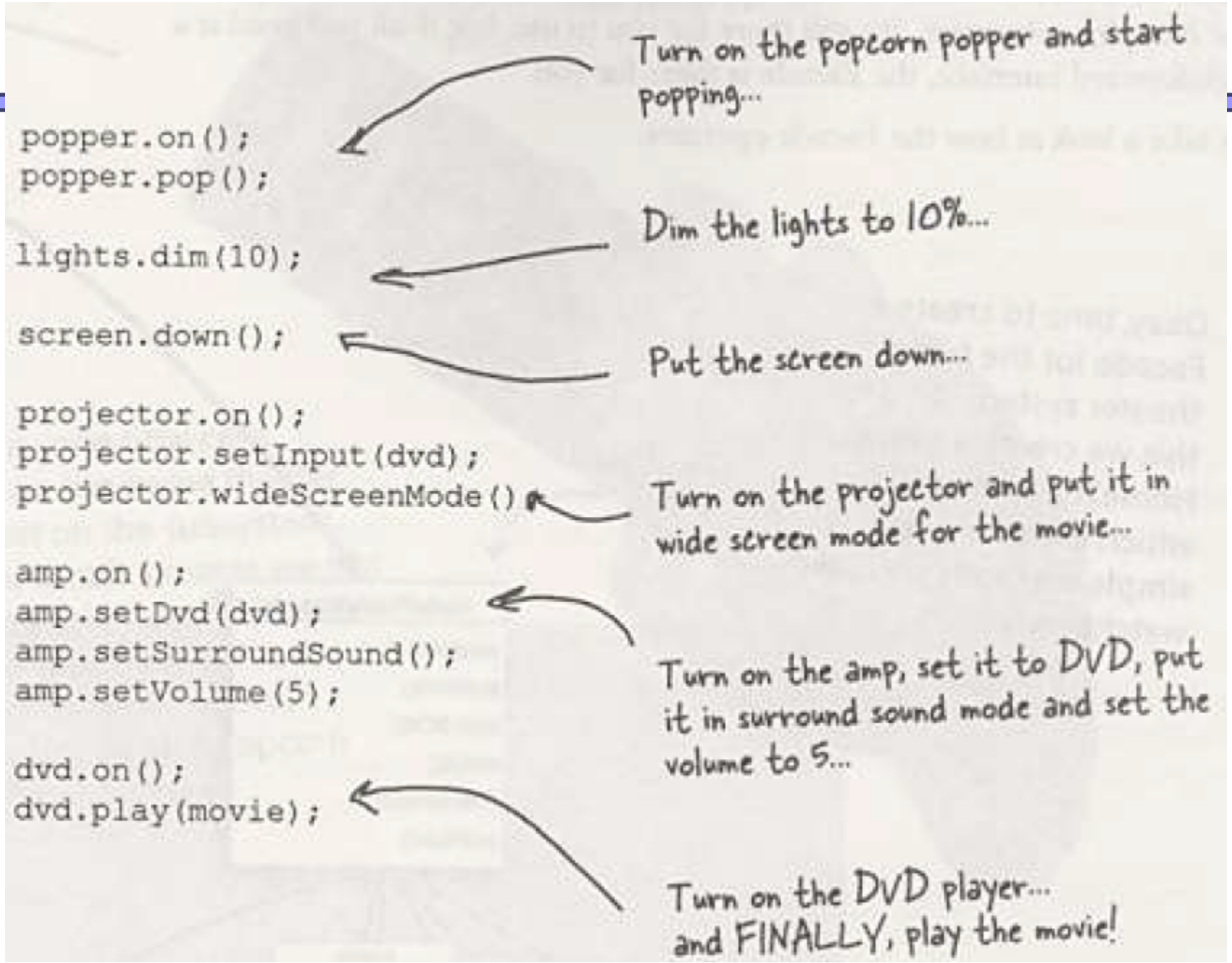


Example from the GOF book



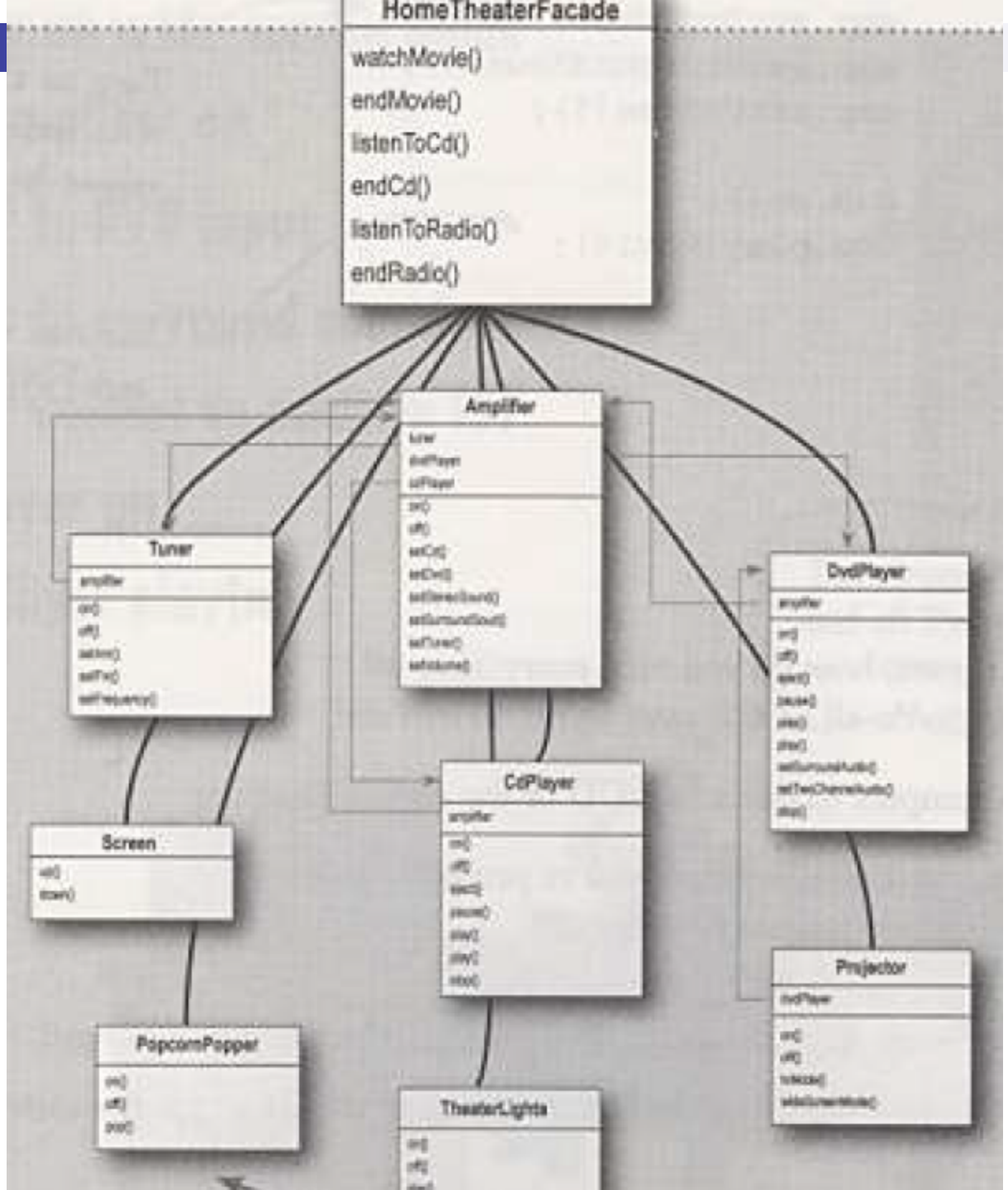
Compiler Façade offers a single, simple interface to the compiler subsystem. It makes life easier for most programmers without hiding lower-level functionality from the few programmers that need it.

Watching a movie : head first design patterns



Movie system setup

- When movie is over, how to turn everything off?
- Do all these steps in reverse?
- Listen to cd or the radio?
- If you want to upgrade your system, you need to have to learn the new procedure


HomeTheaterFacade
watchMovie()
endMovie()
listenToCd()
endCd()
listenToRadio()
endRadio()
Amplifier
Tuner
DvdPlayer
CdPlayer
Screen
Projector
PopcornPopper
TheaterLights

Movie System Facade

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

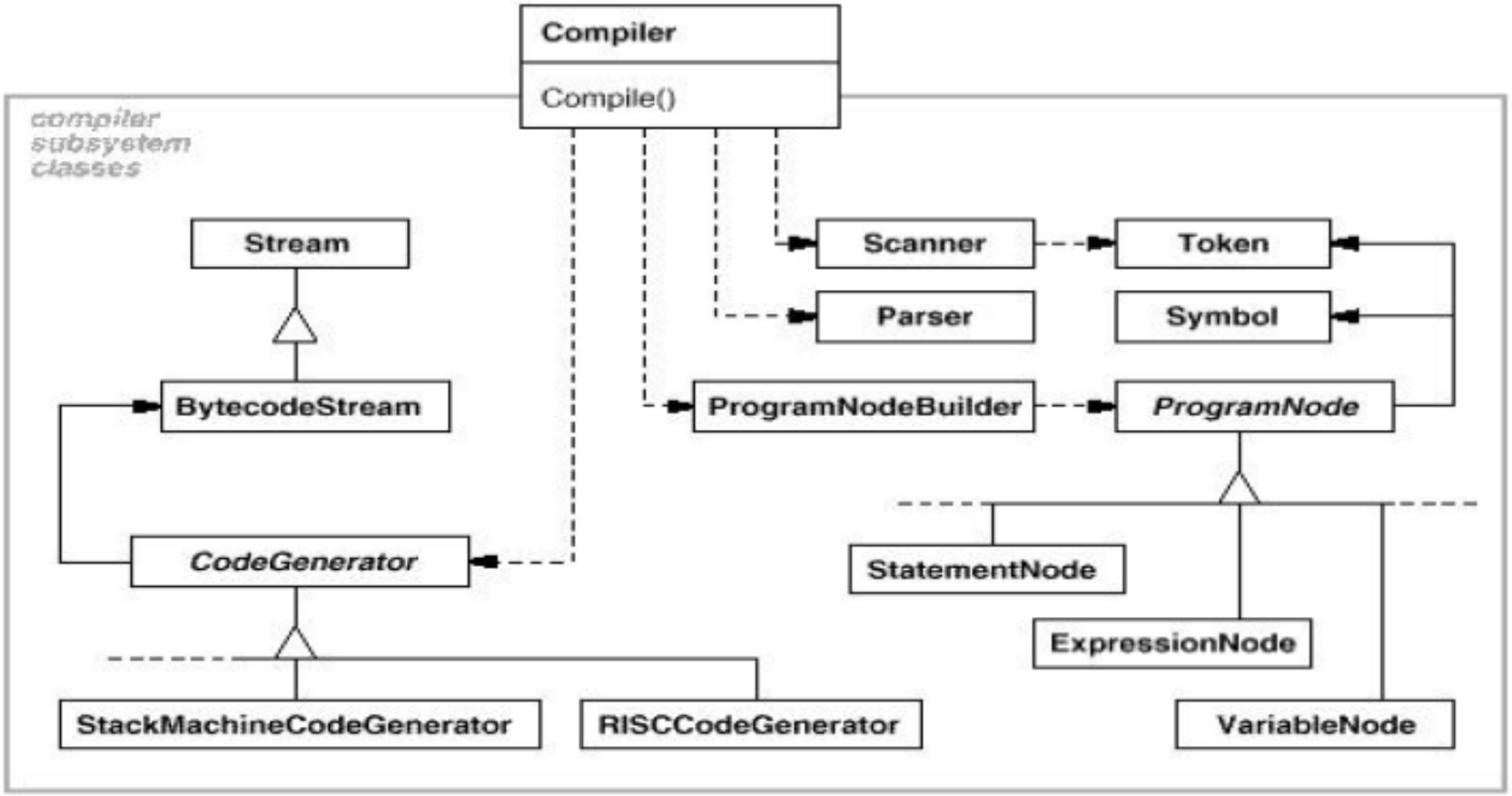
Client code

```
public class Driver{
 public static void main(String[] args){
    //instantiate the individual components e.g. tuner

    HomeTheatherFacade homeTheater=
            new HomeTeatherFacade(amp,, tuner, dvd,cd,
                projector, popper, screen, lights);

    homeTheather.watchMovie ("mymovie");
    homeTheather.endMovie ();
  }
}
```

Example from the GOF book



Compiler Façade offers a single, simple interface to the compiler subsystem. It makes life easier for most programmers without hiding lower-level functionality from the few programmers that need it.

C++ implementation

```
class Scanner {
    public:….  Scanner(istream&);
};
class Parser {…
  virtual void Parse(Scanner&,
          ProgramNodeBuilder&);
};
class ProgramNodeBuilder {
    public: ……
   ProgramNode* GetRootNode();
virtual ProgramNode* NewAssign
(…..) const;
};
class CodeGenerator {…}
class ProgramNode {…
  virtual void
  Traverse(CodeGenerator&);
};
```

```
class Compiler {
    public:
     Compiler();
     void Compiler::Compile (istream&
input, BytecodeStream& output) {
        Scanner scanner(input);
        ProgramNodeBuilder builder;
        Parser parser;

        parser.Parse(scanner, builder);

        RISCCodeGenerator
generator(output);
        ProgramNode* parseTree =
                builder.GetRootNode();
        parseTree->Traverse(generator);
    }
```

Implementation issues

- Façade is coupled with the subsystem below
  - If subsystem never changes, it is ok
  - Else, changes in subsystem may broke your facade
- Solution1: Façade Interface or Abstract Façade
  - Concrete classes make the actual coupling with the subsystem classes
- Solution 2: configure Façade with different subsystem objects (dependency injection)

Façade –Consequences

- A simple default view of the subsystem that is good enough for most clients
- Shields clients from subsystem components
  - Clients will deal with less number of objects, much easier
- Promotes subsystem independence and **portability**
  - Helps layering a system
- Reduces compilation dependencies in large systems
- Promotes low coupling between subsystem and its clients
  - Change subsystem without affecting the clients

- Facade does not add any functionality,

  it just simplifies interfaces

- It does not prevent sophisticated clients from accessing the underlying classes

Adapter vs Facade

- The Intent !

- *Adapter* ***alters*** the interface so that it matches the one the client expects

- *Façade* ***simplifies*** the interface of a **subsystem**