PREVIOUSLY ON ASD…

- Software architecture and architectural design patterns
- A good architecture won't work without a good implementation

ADVANCED SOFTWARE DESIGN
LECTURES 7
DESIGN PATTERNS

Kiko Fernandez

OVERVIEW

What are design patterns

What are some design patterns

Applying some patterns

DESIGN PATTERNS

DESIGN PATTERNS

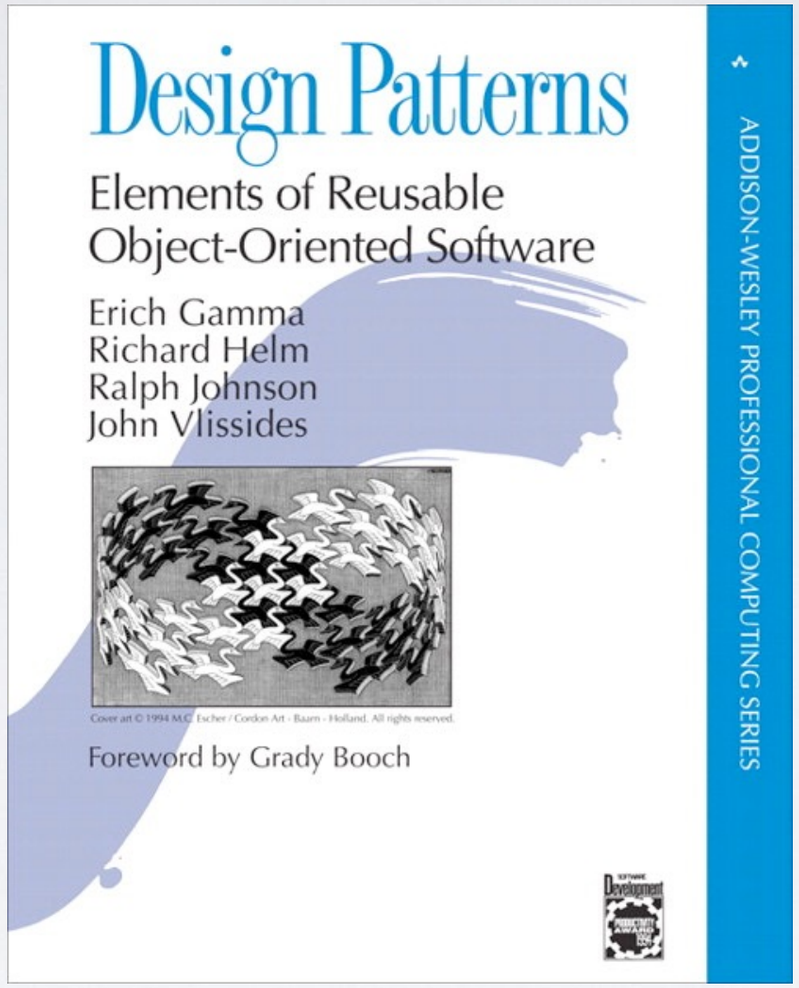
Well-designed solution to common problem.

Collected into catalogues (called pattern languages)

Follow stylised presentation format

Have accompanying code!

THE GANG OF FOUR



The original book that started it all.
Applies Christopher Anderson's architectural design patterns to software.

ESSENTIALS OF A DESIGN PATTERN

A **pattern name** by which we can call it
– let's us talk about design at a higher level of abstraction.

The **problem** to which the pattern applies.

The **solution**, consisting of elements which make up the design, their relationships, responsibilities and collaborations.

The **consequences**, namely the results and trade-offs of applying the pattern.

DESIGN PROBLEMS

DESIGN PROBLEMS

- **Creational patterns**: These design patterns are all about class instantiation

- **Structural patterns**: These design patterns are all about Class and Object composition

- **Behavioural patterns**: Behavioural patterns are those patterns that are most specifically concerned with communication between objects.

ABSTRACT FACTORY, FACTORY METHOD, BUILDER

Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

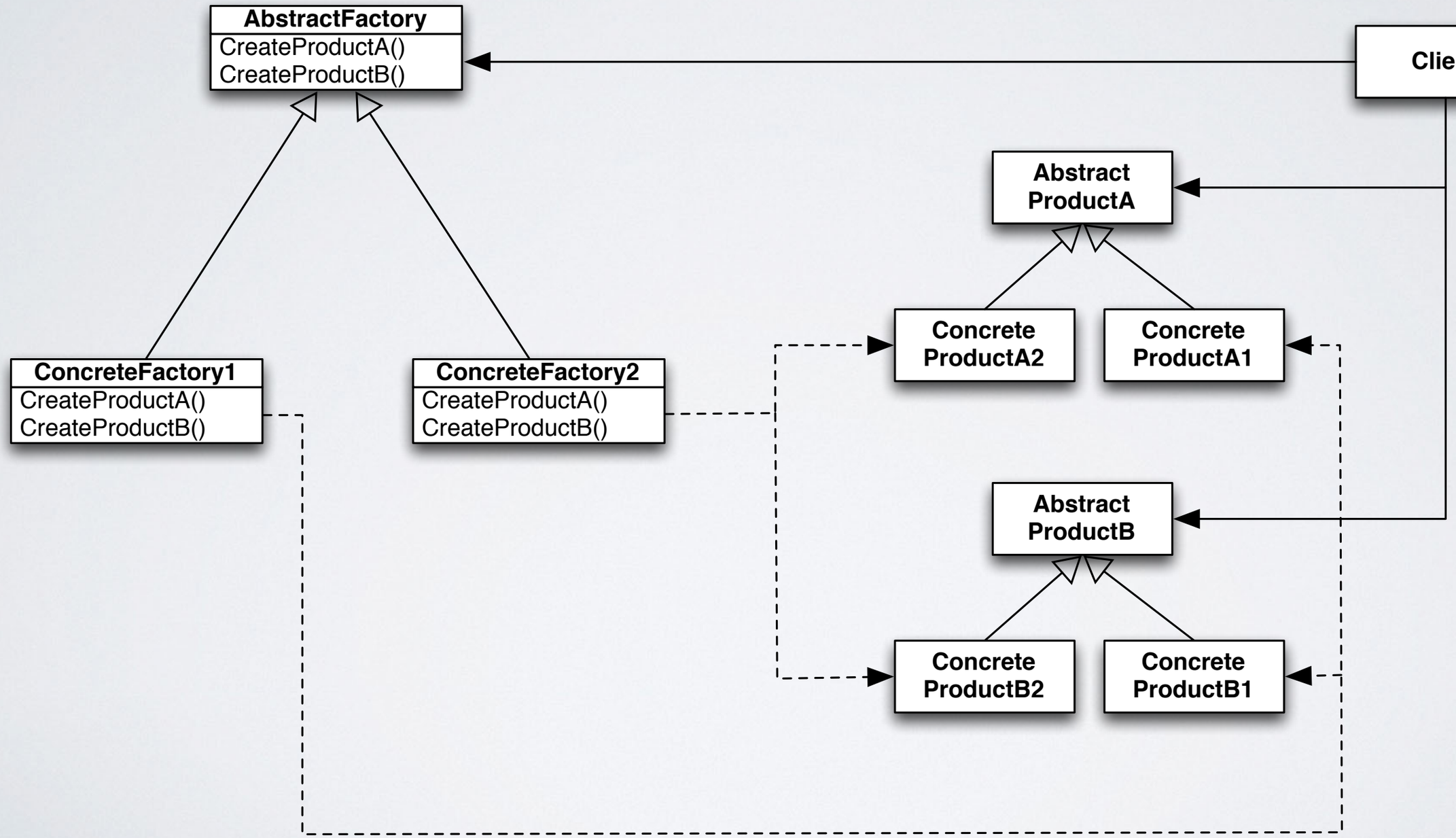
Factory Method – Define an interface for creating an object, but let subclasses decide which class to instantiate.

Builder – separate the construction of a complex objects from its representation so that the same construction process can create different representations.

ABSTRACT FACTORY

Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

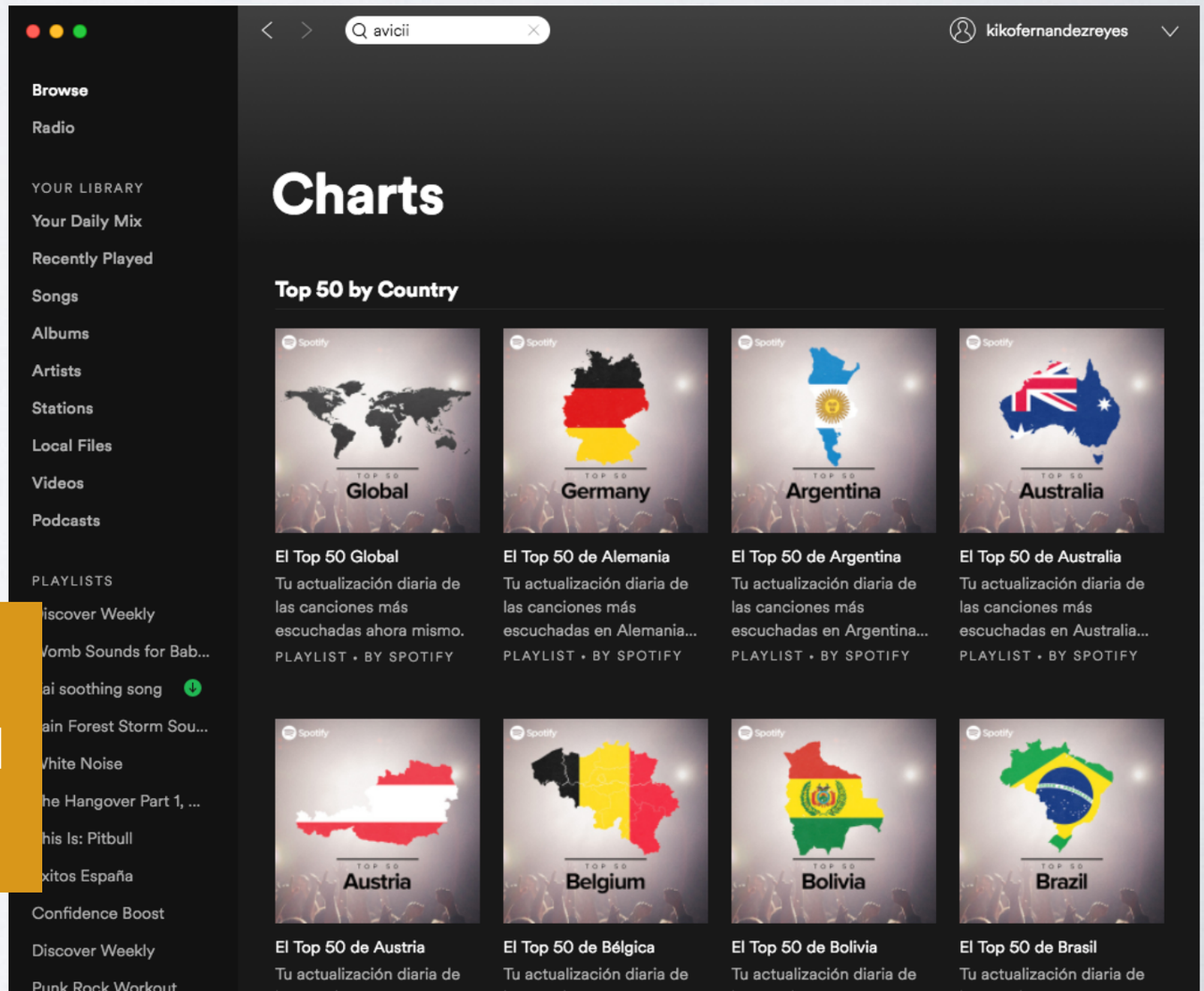
ABSTRACT FACTORY



IN-CLASS EXERCISE

Group work

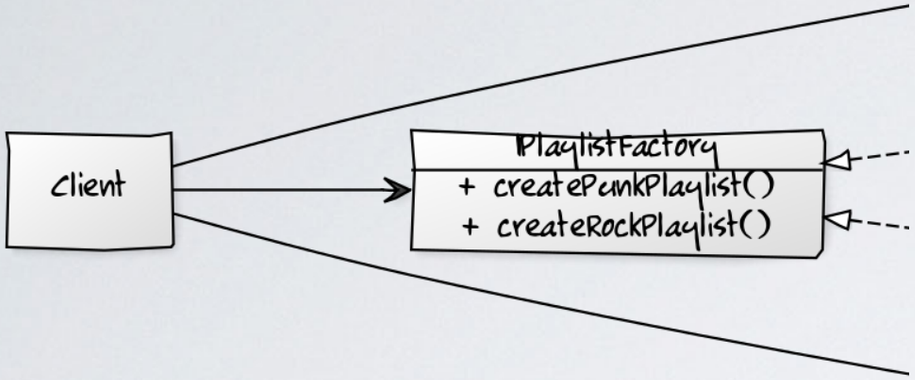
- Charts: Spotify allows you to select a country and filter songs by style (genre)



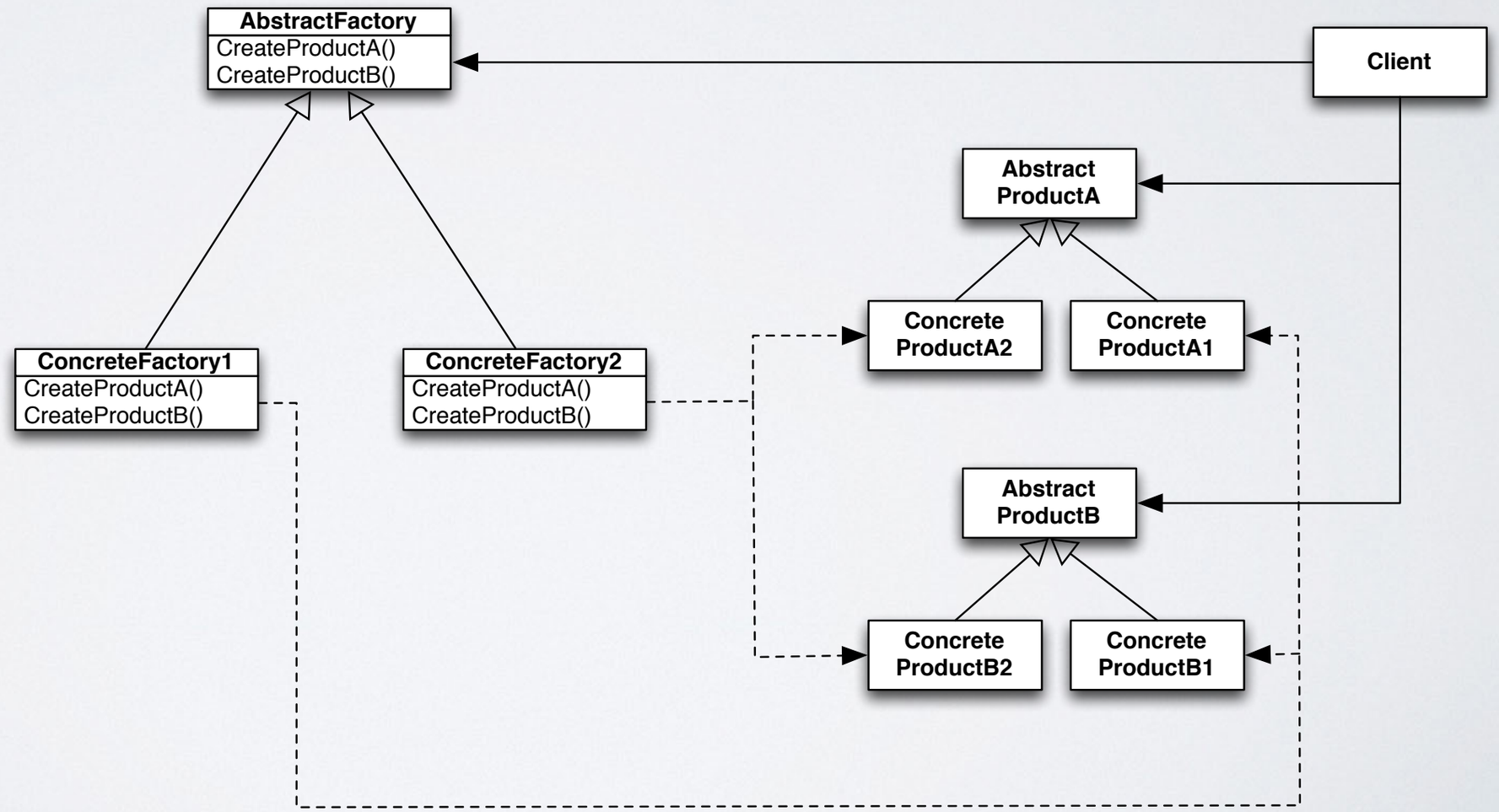
Hint: Use a PlaylistFactory to create families of Playlists based on genre

- Charts: Spotify allows you to select a country and filter songs by style (genre)

Group work



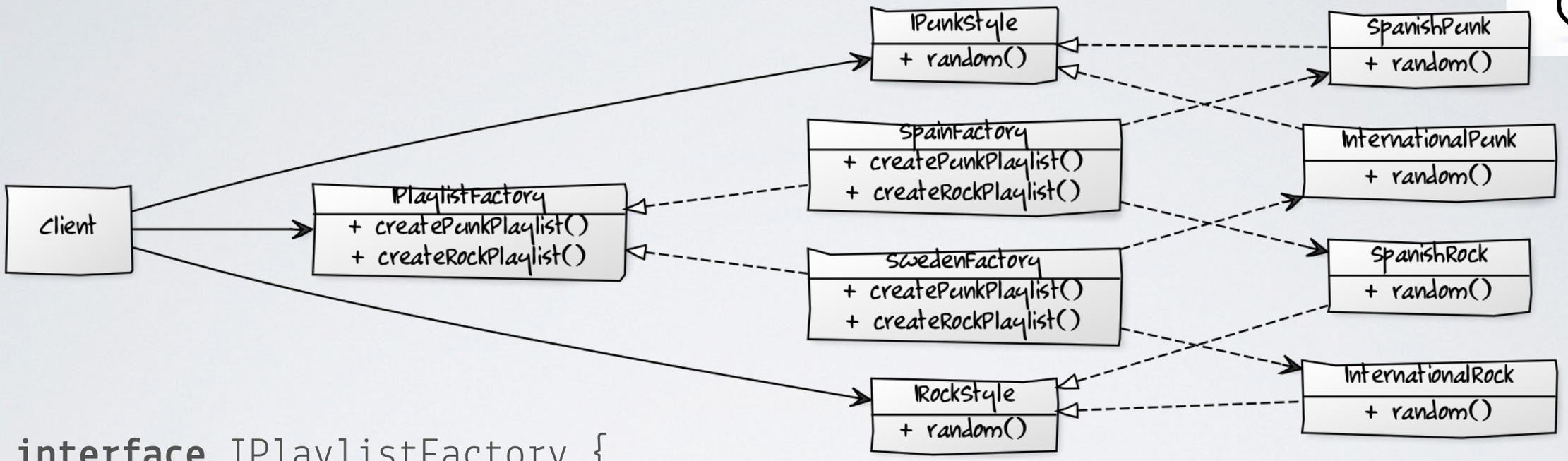
Hint



ALTERNATIVE SOLUTION

Spotify®

CODE – ABSTRACT FACTORY & CLIENT




```
interface IPlaylistFactory {
  Playlist createPunkPlaylist(uint songs);
  Playlist createRockPlaylist(uint songs);
}

public class Client {
  private IPunkStyle p;
  void myPunkPlaylist(IPlaylistFactory factory) {
    this.punk = factory.createPunkPlaylist();
  }
}
```

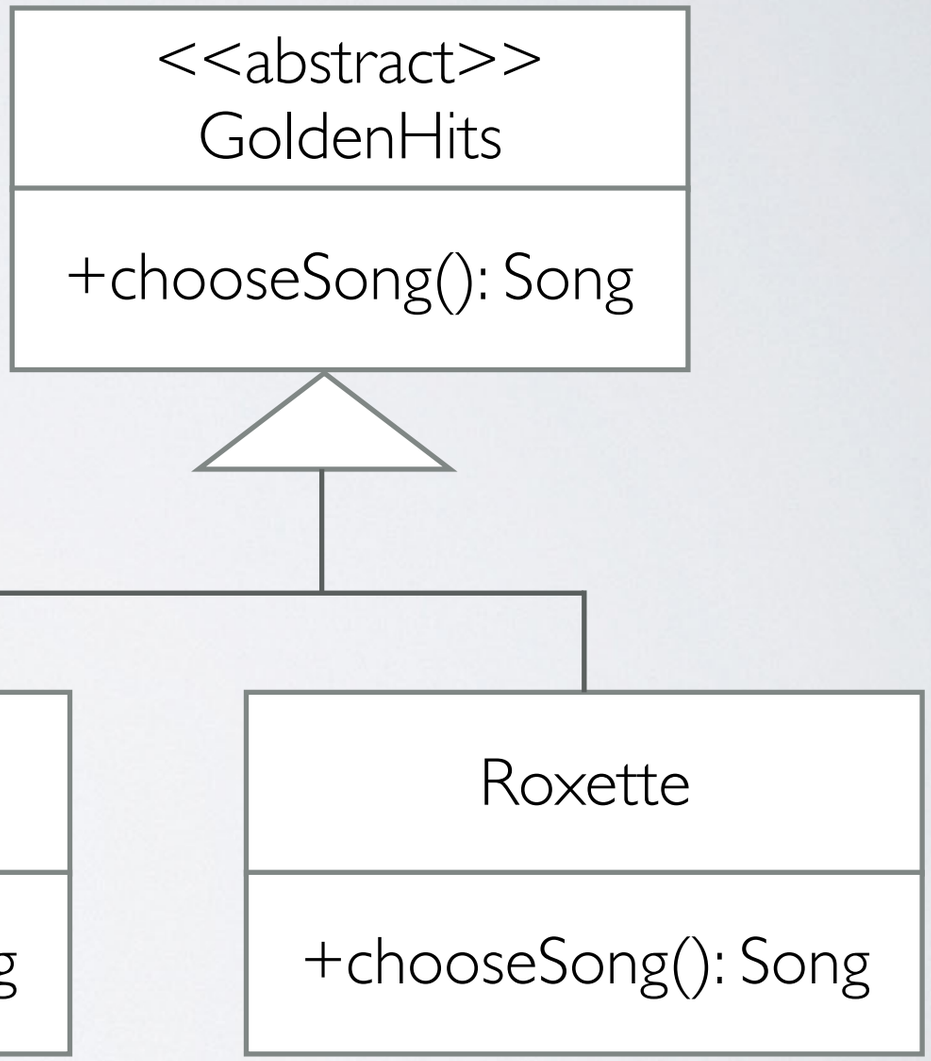
CODE – ABSTRACT FACTORY & CLIENT

```
class SwedenFactory implements IPlaylistFactory {
  Playlist createPunkPlaylist(uint nsongs) {
    Punk p = this.songs.getStyle("Punk");
     return p.random(nsongs).filter("Bad Religion");
  }

  Playlist createRockPlaylist(uint nsongs) {
    Rock r = this.songs.getStyle("Rock");
     return r.random(nsongs).filter("Metallica");
  }

}
```

Factory Method – Define an interface for creating an object, but let subclasses decide which class to instantiate.

FACTORY METHOD

```
public class GoldenHits {
    public createGoldenHits() {
        Song song1 = chooseSong();
        Song song2 = chooseSong();
        this.playlist.add(song1);
        this.playlist.add(song2);
    }
    abstract Song chooseSong();
}
```



FACTORY METHOD

```
abstract public class GoldenHits {
    public createGoldenHits() {
        Song song1 = chooseSong();
        Song song2 = chooseSong();
        this.playlist.add(song1);
        this.playlist.add(song2);
    }
    abstract Song chooseSong();
}
```

```
 public class ABA extends GoldenHits {
    Song chooseSong() {
        return this.songs.random();
    }
 }

public class Roxette extends GoldenHits {
   Song chooseSong() {
        return this.songs.random();
   }
}
```

FACTORY METHOD

```
abstract public class GoldenHits {
    public createGoldenHits() {
        Song song1 = chooseSong();
        Song song2 = chooseSong();
        this.playlist.add(song1);
        this.playlist.add(song2);
    }
    abstract Song chooseSong();
}
```

```
public class ABA extends GoldenHits {
    Song chooseSong() {
        return this.songs.random();
    }
}

public class Roxette extends GoldenHits {
    Song chooseSong() {
        return this.songs.random();
    }
}
```

FACTORY METHOD

```
abstract public class GoldenHits {
    public createGoldenHits() {
        Song song1 = chooseSong();
        Song song2 = chooseSong();
        this.playlist.add(song1);
        this.playlist.add(song2);
    }
    abstract Song chooseSong();
}
```

```
GoldenHits g = new Roxette();
g.createGoldenHits()
```

```
public class ABA extends GoldenHits {
    Song chooseSong() {
        return this.songs.random();
    }
}
```

```
public class Roxette extends GoldenHits {
    Song chooseSong() {
        return this.songs.random();
    }
}
```

BUILDER

separate the construction of a complex objects from its representation so that the same construction process can create different representations.

BUILDER

separate the construction of a complex objects from its representation so that the same construction process can create different representations.

Builder =

VAPIANO®
PASTA | PIZZA | BAR

BUILDER

```
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";

  public void setDough(String dough)     {
      this.dough = dough;
  }
  public void setSauce(String sauce)     {
      this.sauce = sauce;
  }
  public void setTopping(String topping) {
    this.topping = topping;
  }
}

/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

Recipe

BUILDER

```
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";

  public void setDough(String dough)     {
      this.dough = dough;
  }
  public void setSauce(String sauce)     {
      this.sauce = sauce;
  }
  public void setTopping(String topping) {
    this.topping = topping;
  }
}

/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

Recipe

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() {
      pizza.setTopping("ham+pineapple");
  }
}
```

```
/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  // specialised version, similar to HawaiianPizzaBuilder
}
```

LDER

```
/* "Director" */
class Waiter {
  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder(PizzaBuilder pb) {
    pizzaBuilder = pb;
  }
  public Pizza getPizza() {
    return pizzaBuilder.getPizza();
  }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}
```

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() {
      pizza.setTopping("ham+pineapple");
  }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  // specialised version, similar to HawaiianPizzaBuilder
}
```

Recipe

```
/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

LDER

```
/* "Director" */
class Waiter {
  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder(PizzaBuilder pb) {
    pizzaBuilder = pb;
  }
  public Pizza getPizza() {
    return pizzaBuilder.getPizza();
  }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}
```

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() {
      pizza.setTopping("ham+pineapple");
  }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  // specialised version, similar to HawaiianPizzaBuilder
}
```

```
/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizz

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

Recipe

```
class BuilderExample {
  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
    PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();

    waiter.setPizzaBuilder( hawaiian_pizzabuilder );
    waiter.constructPizza();

    Pizza pizza = waiter.getPizza();
  }
}
```

ABSTRACT FACTORY, FACTORY METHOD, BUILDER

Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method – Define an interface for creating an object, but let subclasses decide which class to instantiate.

Builder – separate the construction of a complex objects from its representation so that the same construction process can create different representations.

MATCH SYSTEM

- When would you use…
  - Abstract Factory
  - Factory Method
  - Builder

MATCH SYSTEM

- When would you use…
  - Abstract Factory Generic Match System → Families of requests
  - Factory Method
  - Builder

MATCH SYSTEM

- When would you use…
  - Abstract Factory — Generic Match System → Families of requests
  - Factory Method — Log as different Objects
  - Builder

PROBLEM

Separate creation of objects
from the objects they depend on

```
class Song {
  Artist artist;
  String title;

  public Song() { // constructor
    artist = new Artist(“Avicii”); // artist fixed
    title = new String(“Wake me up”);
  }
}

song = new Song();
```

DEPENDENCY INJECTION

Dependency Injection – Move dependency of clients on services out of client.

Pass in service as parameter to constructor or via setter.

BEFORE DEPENDENCY INJECTION

```
class Song {
   Artist artist;
   String title;

   public Song() { // constructor
      artist = new Artist(“Avicii”); // artist fixed
      title = new String(“Wake me up”);
   }
}

song = new Song();
```

AFTER DEPENDENCY INJECTION

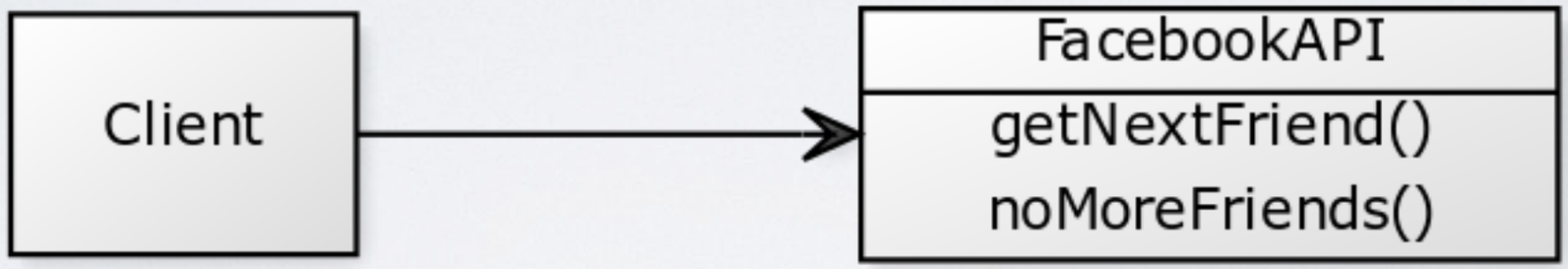
```
class Song {

   Artist artist;
   String title;

   public Song(Artist a, String t) {
      artist = a;
      title = t;
   }
}


song = new Song(new Artist(“Avicii”), “Wake Me up”);
```

PROBLEM

Accessing, for example, a third party API, but protecting your application from changes to the API.

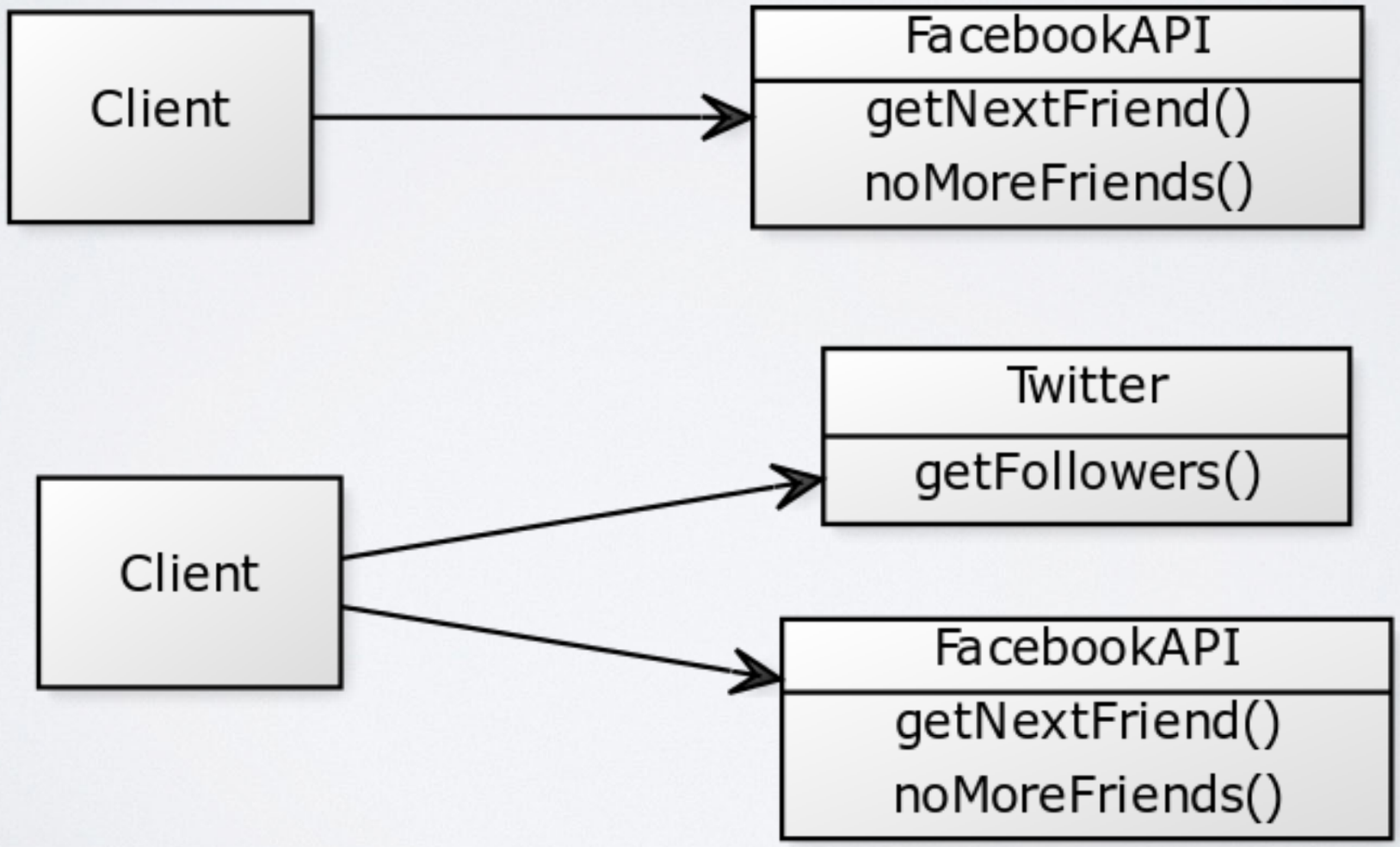
ADAPTOR

Adaptor – convert the interface of a class into another interface classes expect.

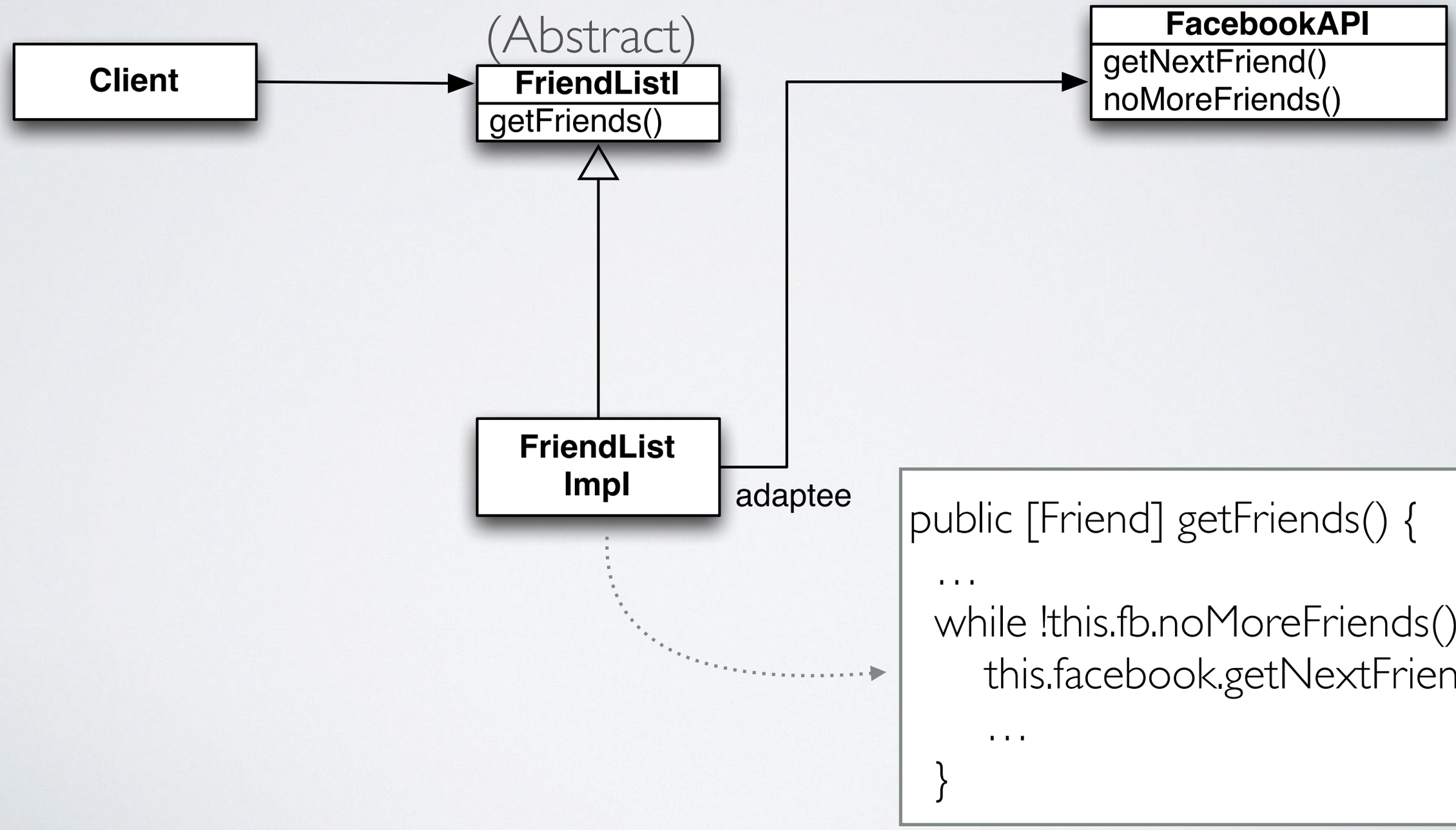


ADAPTOR

Adaptor – convert the interface of a class into another interface classes expect.



ADAPTOR


(Abstract)
Client
FriendListI
getFriends()
FacebookAPI
getNextFriend()
noMoreFriends()
FriendList
Impl
adaptee
public [Friend] getFriends() {
…
while !this.fb.noMoreFriends() {
this.facebook.getNextFriend();
…
}

IN-CLASS EXERCISE

Group work

- Discuss where the **Adapter** can be used in the **MatchSystem**

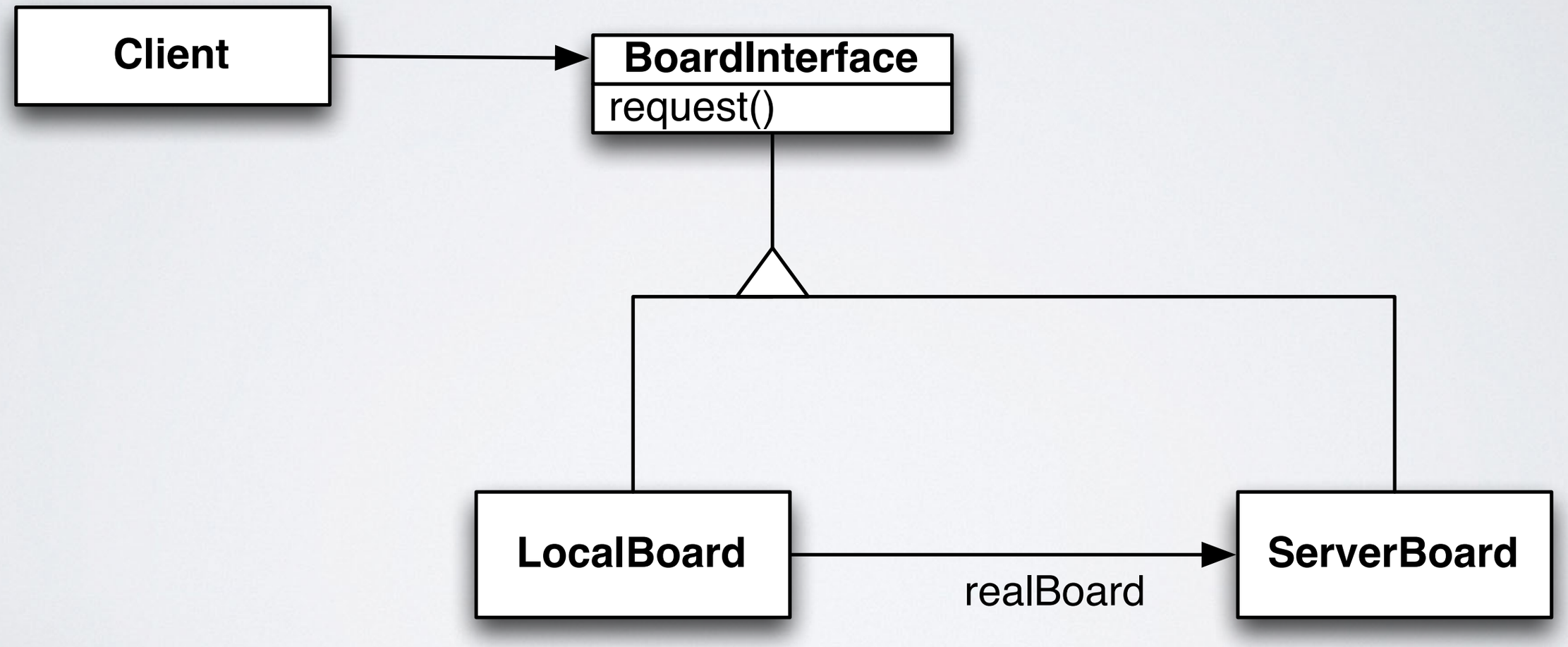
PROBLEM

Client accesses a copy of an object that resides on a server.

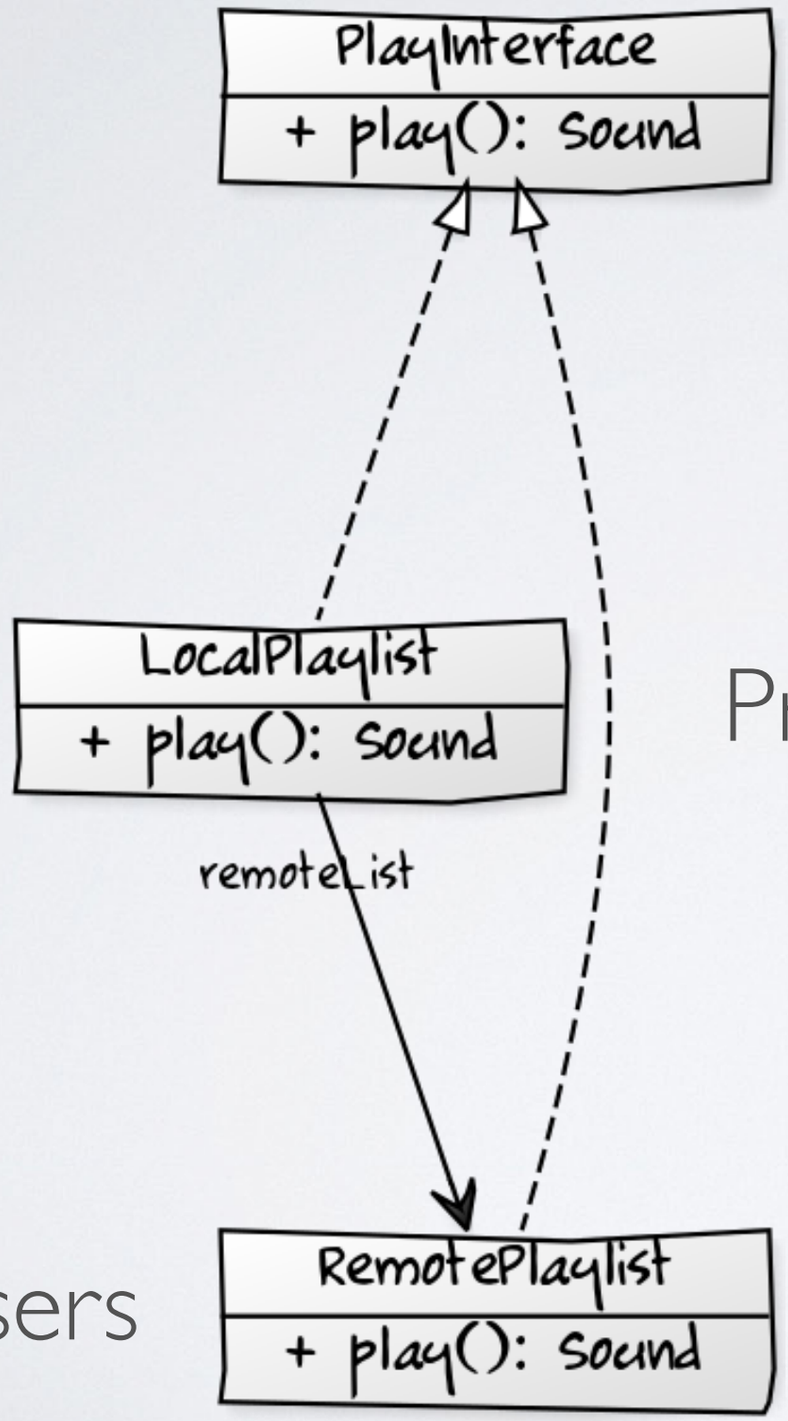
Getting attributes and values from the server takes quite some time, can we solve this?

PROXY

Proxy – provide a surrogate or placeholder for another object to control access to it.



PROXY



Premium users

Free users

Spotify®

PROBLEM

Some objects are expensive to create and need to be carefully managed.

OBJECT POOL

Object pool uses a set of initialised objects kept ready for use in a pool rather than allocating and destroying them on demand

OBJECT POOL

Object pool uses a set of initialised objects kept ready for use in a pool rather than allocating and destroying them on demand

```
public class Board implements PooledObject
{
  Board() {
    // do expensive operations to set up board
  }

  void cleanUp() {
    // clear tiles from board
  }
}
```

OBJECT POOL

```
public static class BoardPool {
    private static List<Board> _available = new List<Board>();
    private static List<Board> _inUse = new List<Board>();

    public static Board getBoard() {
       if (_available.count > 0) {
            Board po = _available[0];
            _inUse.Add(po);
            _available.removeAt(0);
            return po;
        } else {
            Board po = new Board();
            _inUse.Add(po);
            return po;
        }
     }

     public static void releaseObject(Board po)
     {
         po.cleanUp(); _available.Add(po); _inUse.Remove(po);
    }
}
```

OBJECT POOL

```
public static class BoardPool {
    private static List<Board> _available = new List<Board>();
    private static List<Board> _inUse = new List<Board>();

    public static Board getBoard() {
       if (_available.count > 0) {
            Board po = _available[0];
            _inUse.Add(po);
            _available.removeAt(0);
            return po;
        } else {
            Board po = new Board();
            _inUse.Add(po);
            return po;
        }
    }

    public static void releaseObject(Board po)
    {
        po.cleanUp(); _available.Add(po); _inUse.Remove(po);
   }
}
```

PROBLEM

How to deal with semantics of "null" implementation without having an if statement.

NULL OBJECT

Recursive implementation to get tree size

```
public class Tree implements ITree
   left: ITree
   right: ITree
   def size(): uint
      sum = 1
      if this.left != null then
         sum += this.left.size()
      end
      if this.right != null then
         sum += this.right.size()
      end
      return sum
   end
end
```

NULL OBJECT

```
var leftBranch = new Tree(null, null)
var rightBranch = null
var root = new Tree(leftBranch, rightBranch)
```

```
public class Tree implements ITree
  left: ITree
  right: ITree
  def size(): uint
    sum = 1
    if this.left != null then
      sum += this.left.size()
    end
    if this.right != null then
      sum += this.right.size()
    end
    return sum
  end
end
```

NULL OBJECT

```
var leftBranch = new Tree(null, null)
var rightBranch = null
var root = new Tree(leftBranch, rightBranch)
```

```
public class Tree implements ITree
    left: ITree
    right: ITree
    def size(): uint
        sum = 1
        if this.left != null then
            sum += this.left.size()
        end
        if this.right != null then
            sum += this.right.size()
        end
        return sum
    end
end
```

Checking for null → antipattern

NULL OBJECT

```
var leftBranch = new Tree(null, null)
var rightBranch = null
var root = new Tree(leftBranch, rightBranch)
```

```
public class Tree implements ITree
  left: ITree
  right: ITree
  def size(): uint
    sum = 1
    if this.left != null then
      sum += this.left.size()
    end
    if this.right != null then
      sum += this.right.size()
    end
    return sum
  end
end
```

Checking for null → antipattern

Conditional logic → adds complexity

NULL OBJECT

Recursive implementation to get tree size

```
public class Tree implements ITree
   left: ITree
   right: ITree
   def size(): uint
     return 1 + this.left.size() + this.right.size()
   end
end
```

NULL OBJECT

Recursive implementation to get tree size

```
public class Tree implements ITree
   left: ITree
   right: ITree
   def size(): uint
     return 1 + this.left.size() + this.right.size()
   end
end

public class EmptyTree implements ITree
 def size(): uint
     return 0
   end
end
```

NULL OBJECT

```
var leftBranch = new Tree(new EmptyTree(), new EmptyTree())
var rightBranch = new EmptyTree()
var root = new Tree(leftBranch, rightBranch)
```

```
public class Tree implements ITree
   left: ITree
   right: ITree
   def size(): uint
     return 1 + this.left.size() + this.right.size()
   end
end

public class EmptyTree implements ITree
 def size(): uint
     return 0
   end
end
```

PROBLEM

Ensure that there is just one object of a certain kind, potentially accessible from everywhere

SINGLETON

Singleton – Ensure a class has only one instance, and provide a global point of access to it.

```
class Statistics {
  static private Statistics theStatistics =
    new Statistics();

  static public getInstance() {
    return theStatistics;
  }

  // private constructor
  private Statistics() { … }
}
```

IN-CLASS EXERCISE

MatchSystem:

- Name potential Singleton classes
- Drawbacks?

IN-CLASS EXERCISE

MatchSystem:

- Name potential Singleton classes
- Drawbacks?

  - Be careful with threads

IN-CLASS EXERCISE

MatchSystem:

- Name potential Singleton classes
- Drawbacks?

  - Be careful with threads

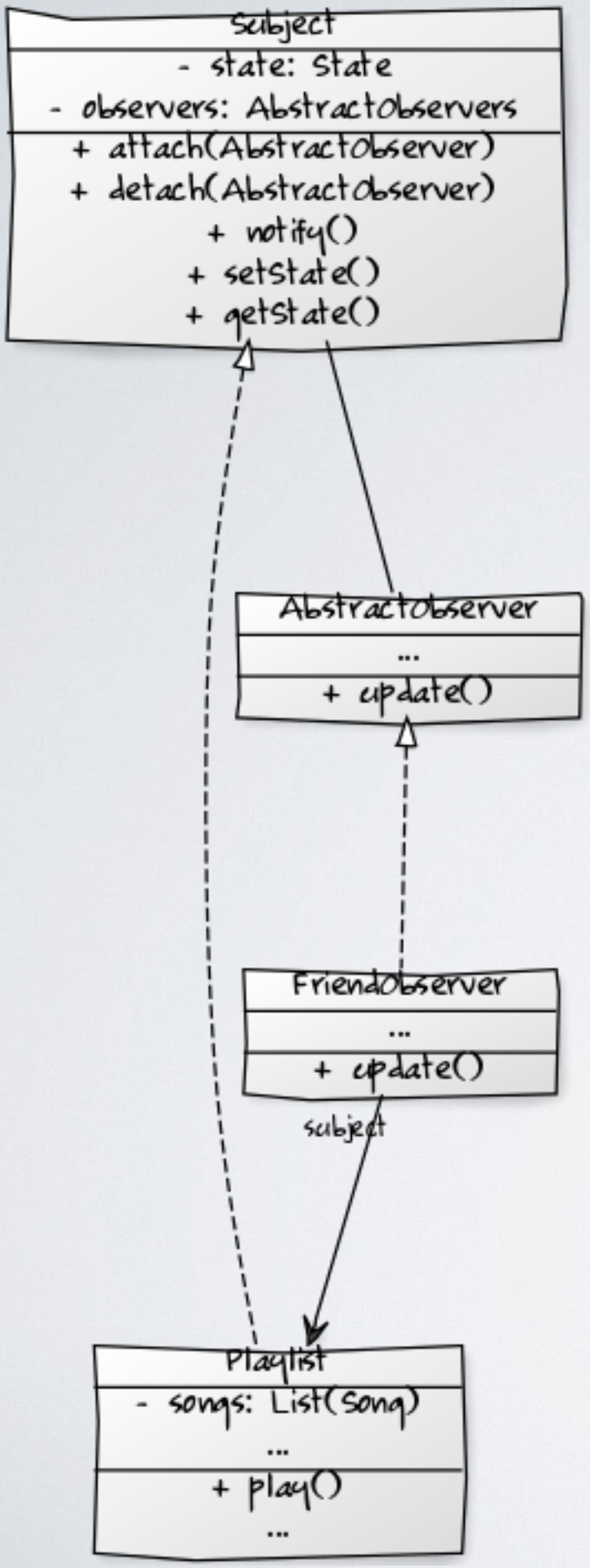
  - Singletons live until the program finishes. Too many of them not good

PROBLEM

Propagate updates from one object to all interested parties, without coupling all objects together.

OBSERVER

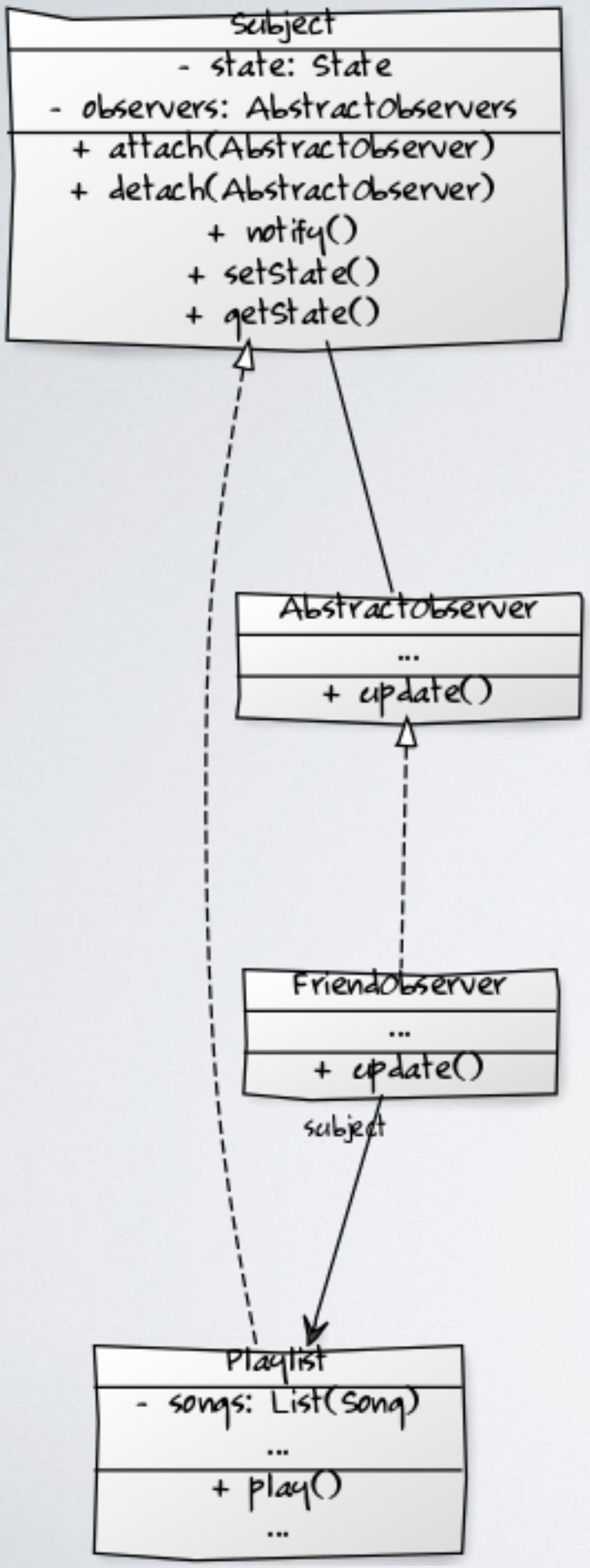
Observer – define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.



OBSERVER

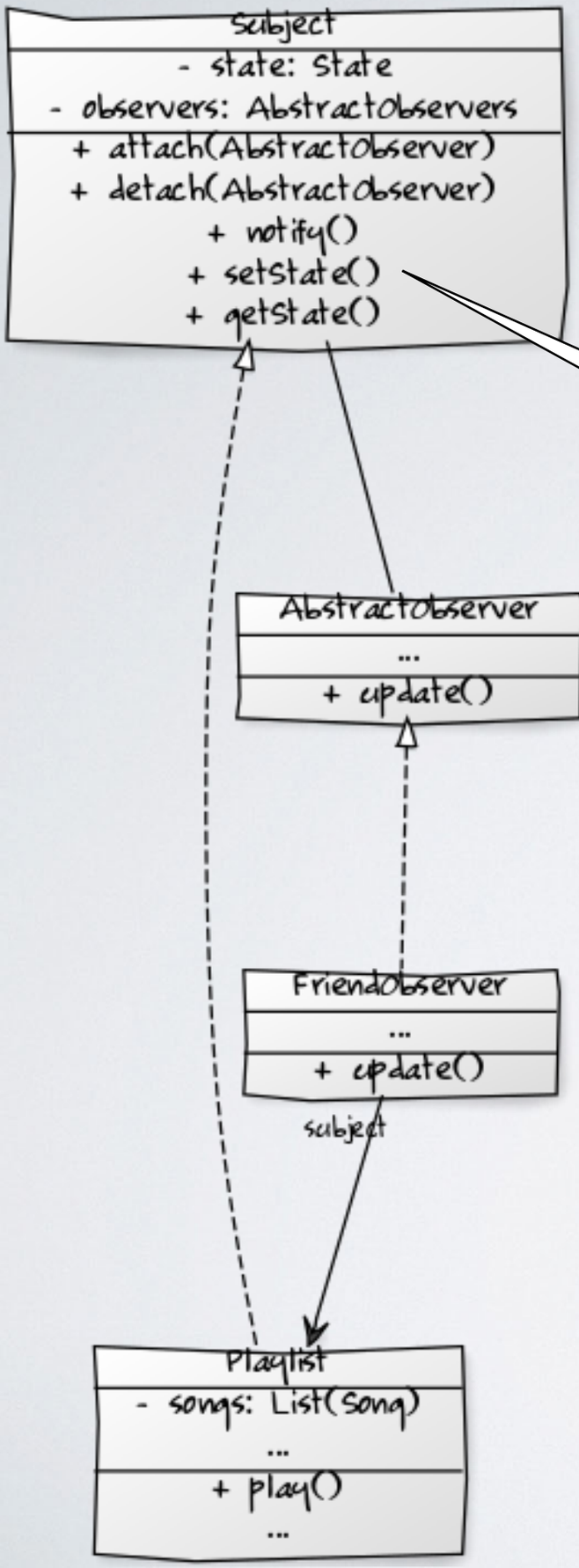
Observers get attached to *Subject*

```
subject.attach(observer1)

subject.attach(observer2)
```



OBSERVER

If *State* of *Subject* changes, notify *Observers*

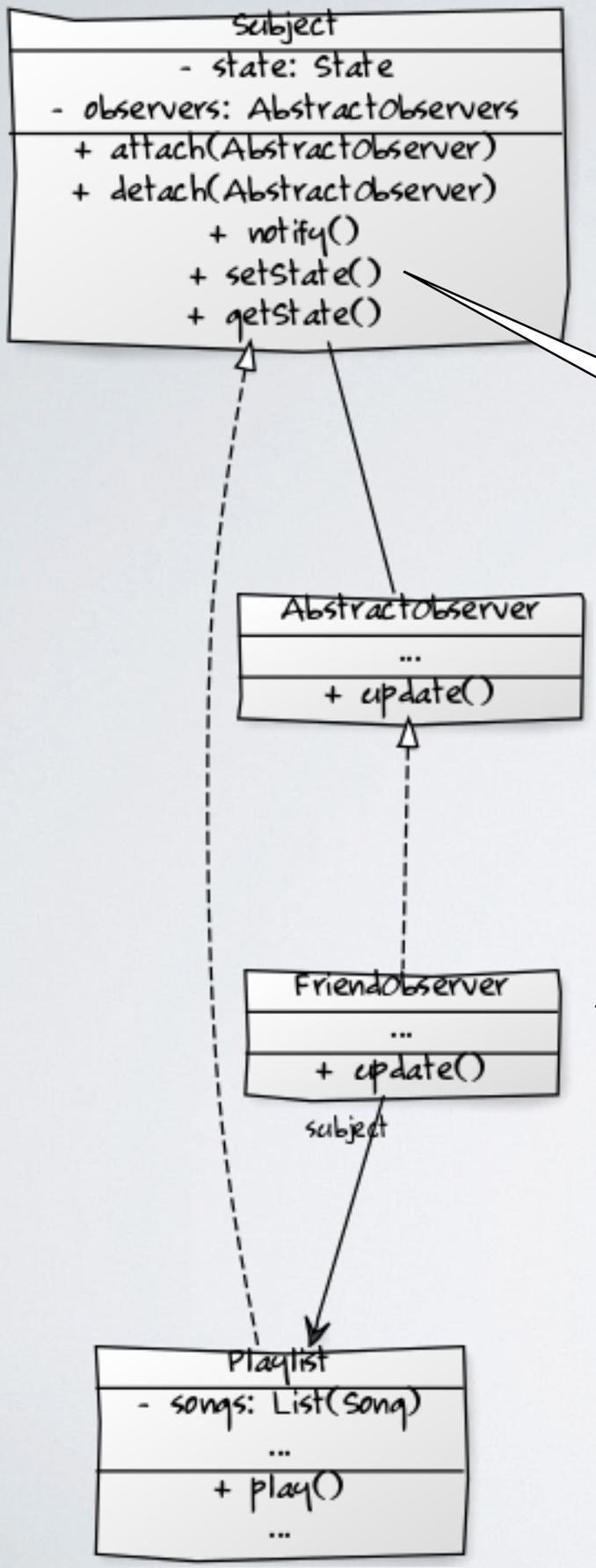


OBSERVER

If *State* of *Subject* changes, notify *Observers*

```
foreach o in this.observers:
    o.update()
```

OBSERVER



If *State* of *Subject* changes, notify *Observers*

```
foreach o in this.observers:
    o.update()
```

```
public class FriendObserver
  subject: Playlist
  def update(): void
    var s = this.subject.getState()
    -- do something with s
  end
end
```

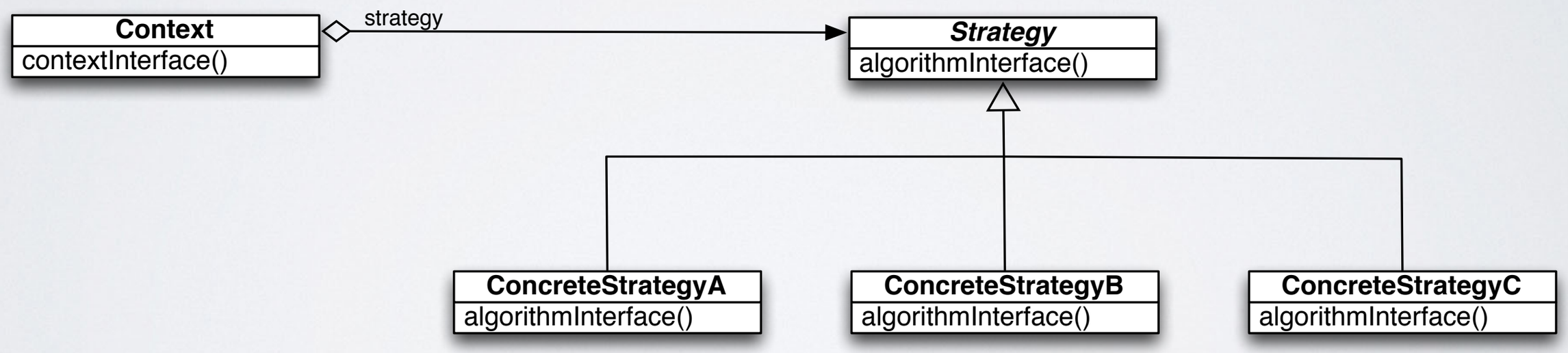
STRATEGY & TEMPLATE METHOD

Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable.

Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

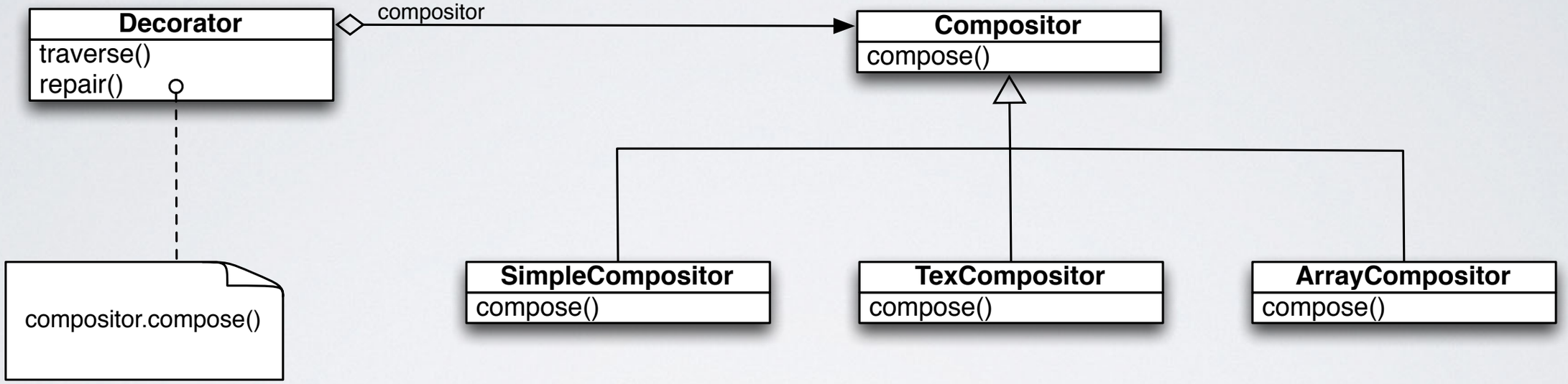
STRATEGY PATTERN

Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable.



Context may have an interface to allow Strategy to access its data

STRATEGY EXAMPLE



Algorithms for breaking a stream of text into lines.

Different algorithms appropriate at different times.

Avoid mingling all such algorithms with main code.

Allow new algorithms to be added.

BEFORE STRATEGY

```
class Composition {
  void repair() {
    switch (_breakingStrategy) {
      case SimpleStrategy:
        composeWithSimpleCompositor();
        break;
      case TexStrategy:
        composeWithTexCompositor();
        break;
      // …
    }
  }
}
```

Strategies handled within case statement.
Restricted to hardcoded strategies

AFTER STRATEGY

```
class Composition {
  CompositionStrategy _compositor; // set in constructor
  void repair() {
   _compositor.compose(this);
  }
}

class SimpleStrategy extends CompositionStrategy {
  void compose(Composition ctx) {
    // …
  }
}
class TexStrategy extends CompositionStrategy {
  void compose(Composition ctx) {
    // …
  }
}
```

Strategy set in constructor
– determined outside Composition class.

New strategies are possible.
ctx object provides information
for strategy to use.

Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

TEMPLATE METHOD

```
class View {
  abstract void doDisplay(Canvas cs);

  // generic algorithm
  void display(Canvas cs) {
    setFocus();
    doDisplay(cs);
    resetFocus();
}

class MyView extends View {
  void doDisplay(Canvas cs) {
    // render view's contents on cs
  }
}
```

algorithm parameter

parameter instantiated

TEMPLATE METHOD

```
class View {
  abstract void doDisplay(Canvas cs);

  // generic algorithm
  void display(Canvas cs) {
    setFocus();
    doDisplay(cs);
    resetFocus();
}

class MyView extends View {
  void doDisplay(Canvas cs) {
    // render view's contents on cs
  }
}
```

algorithm parameter

parameter instantiated

deferring some steps to subclasses

TEMPLATE METHOD

```
class View {
  abstract void doDisplay(Canvas cs);

  // generic algorithm
  void display(Canvas cs) {
    setFocus();
    doDisplay(cs);
    resetFocus();
}

class MyView extends View {
  void doDisplay(Canvas cs) {
    // render view's contents on cs
  }
}
```

deferring some steps to subclasses

```
View view = new MyView();
controller.update(view)
```

```
public class Controller
{
  private Canvas canvas;

  public void update(View v){
    v.display(this.canvas)
  }
}
```

COMPARISON

Key differences are:

Strategy is defined outside the object and is preserved throughout the object lifetime.

Template Method uses inheritance to vary part of the algorithm, whereas **Strategy** uses delegation.

TOO MANY DESIGN PATTERNS FOR TODAY…