Course Application Design

Design Patterns

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Design patterns

- Design patterns are proven solutions to a wide variety of design challenges
- Here we'll deal with a selection of often-used ones

The Patterns dealt with here

Behavioral

- **Strategy**
- **Template method**
- **Observer**
- **Command**
- State
- **Filter**
- **Null/Special Case Object**

Creational

- **Singleton**
- **Factory method**
- Abstract class factory
- **Builder**

Structural

- Facade
- **Decorator**

The patterns in **bold** may be tested on the exam

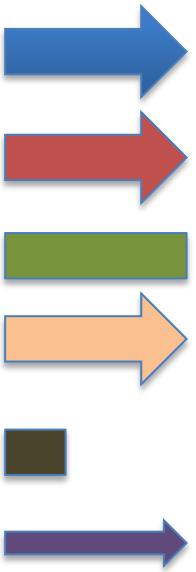
Behavioral

Strategy pattern

A genome browser

- Suppose you are creating a genome browser where many different features can be visualized on different tracks: genes, transcripts, CDSs, repeats etc:

Feature	Visualization
Gene	Arrow
Transcript (mRNA)	Arrow
CDS	Rectangle
Repeat	Arrow
SNP	Rectangle
Probe location	Arrow

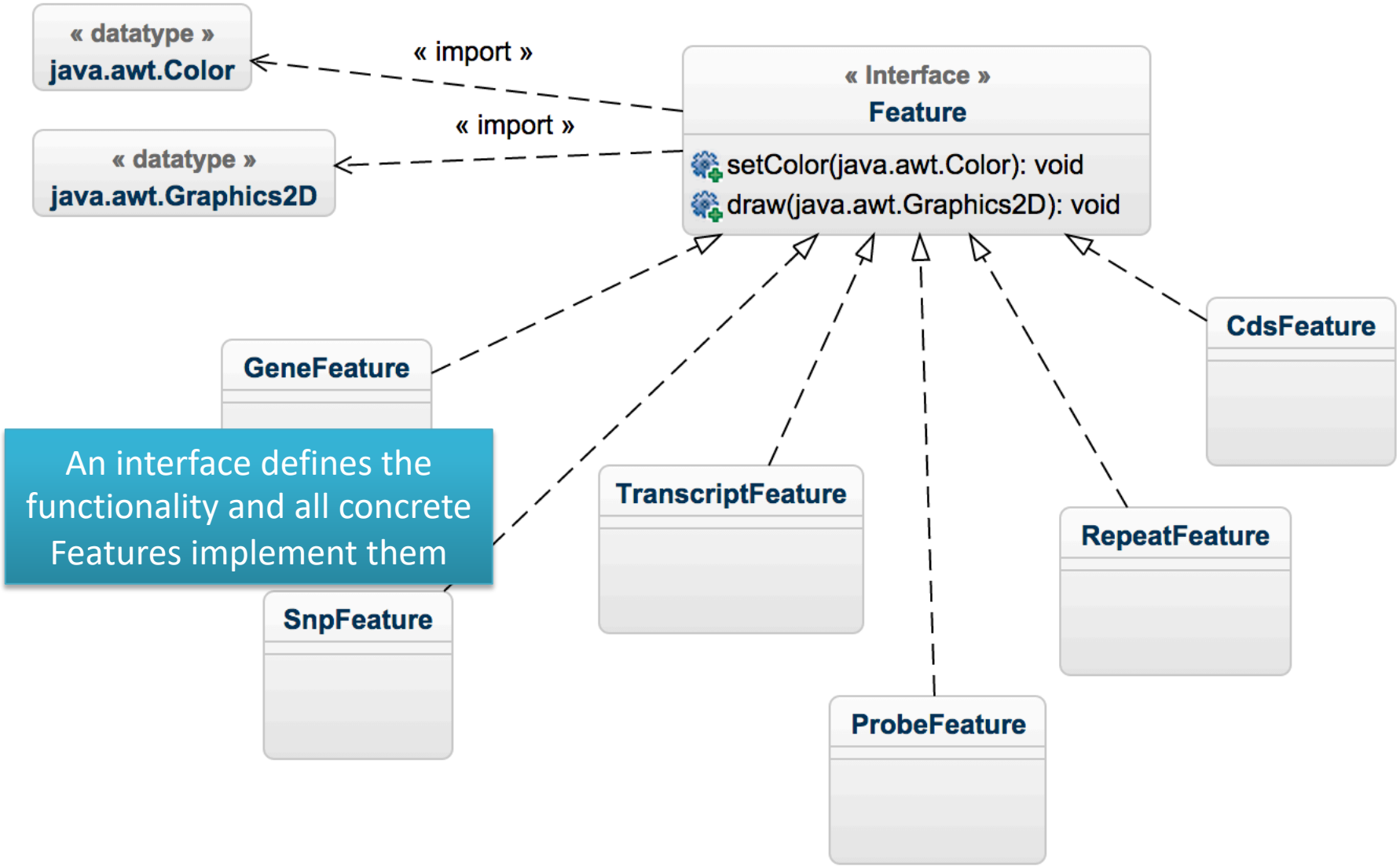


A genome browser

- Given the specs from the previous slide, take a few minutes to create a design for this

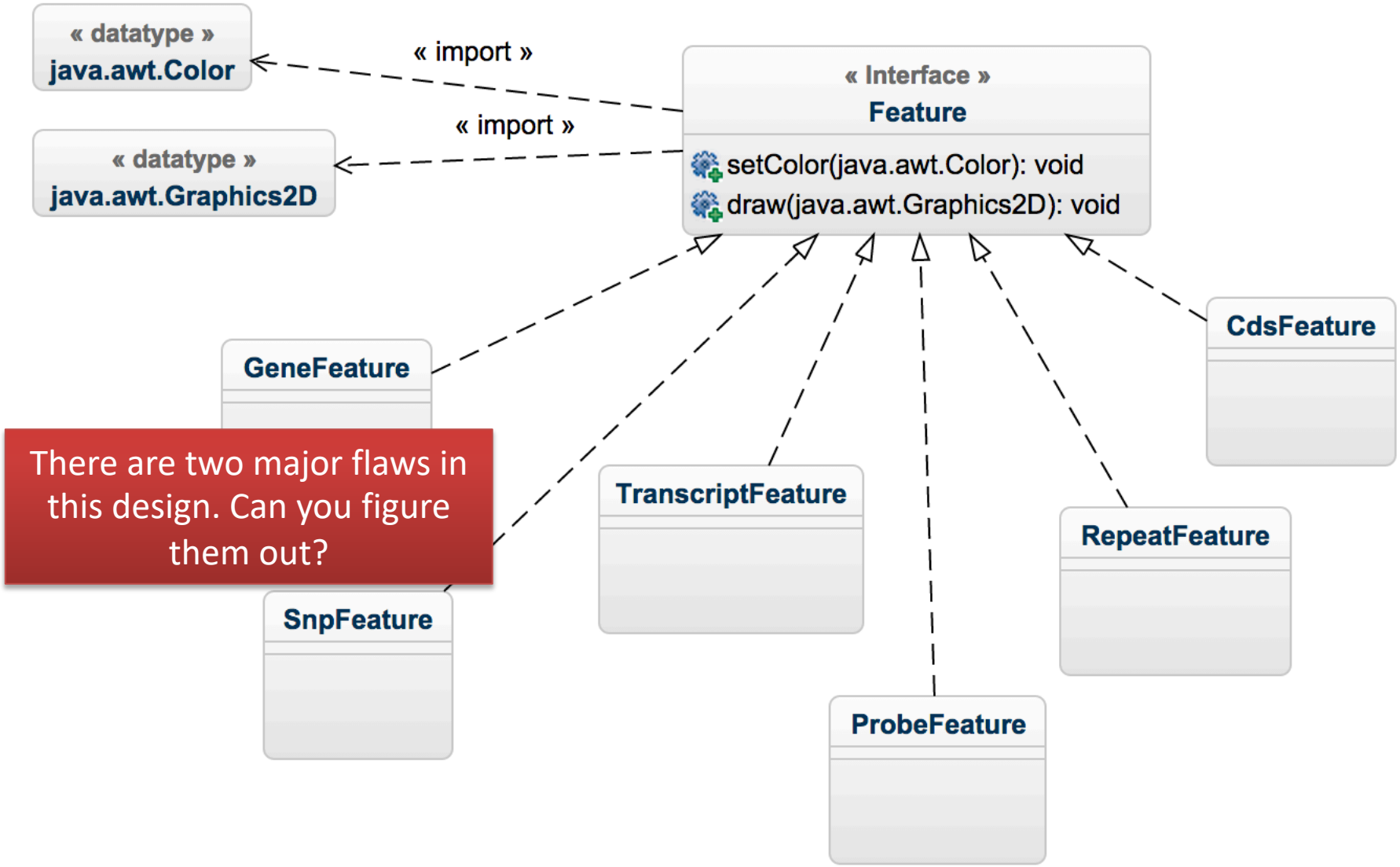
A genome browser

- Do you have something like this?



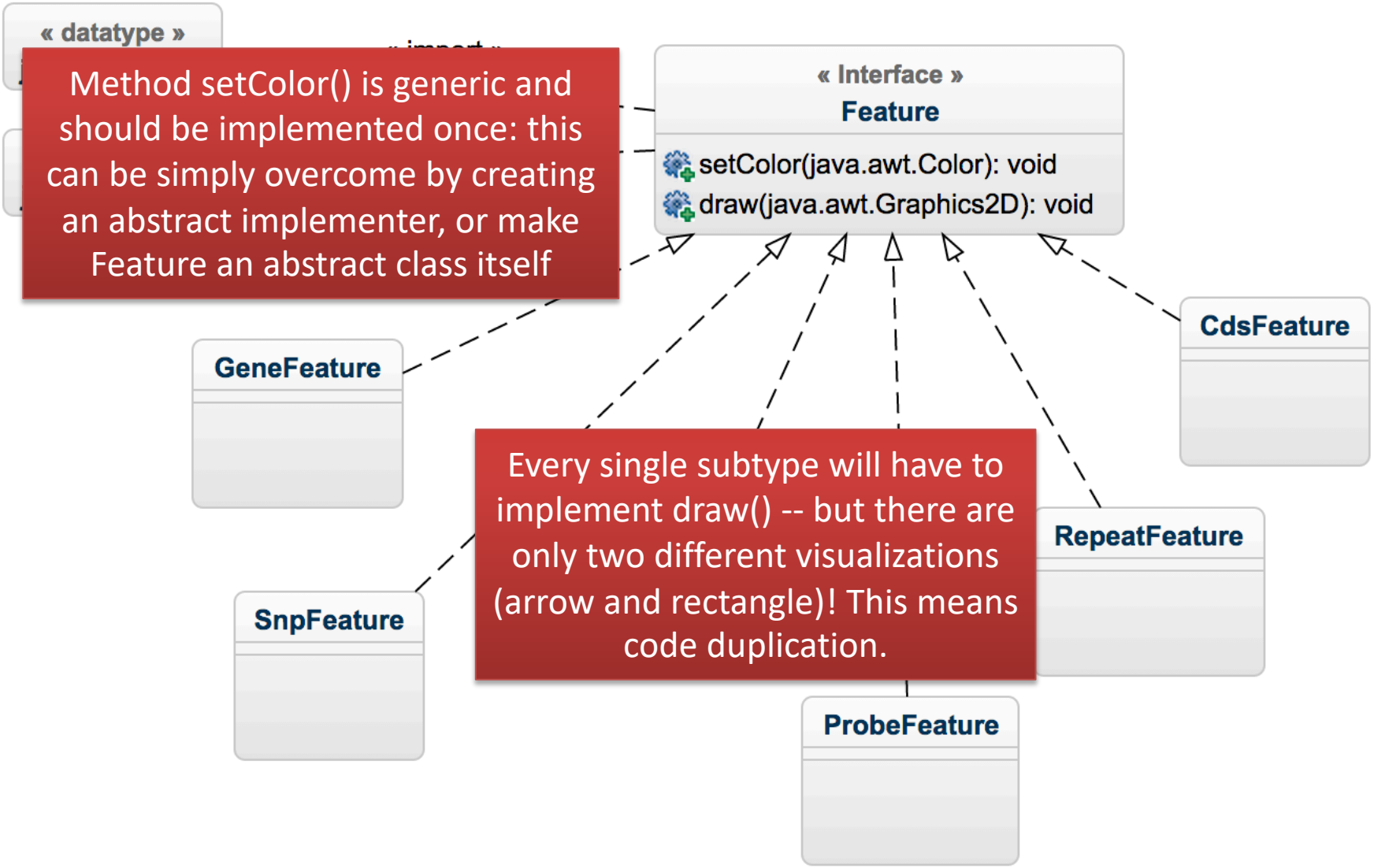
A genome browser

- Do you have something like this?



A genome browser

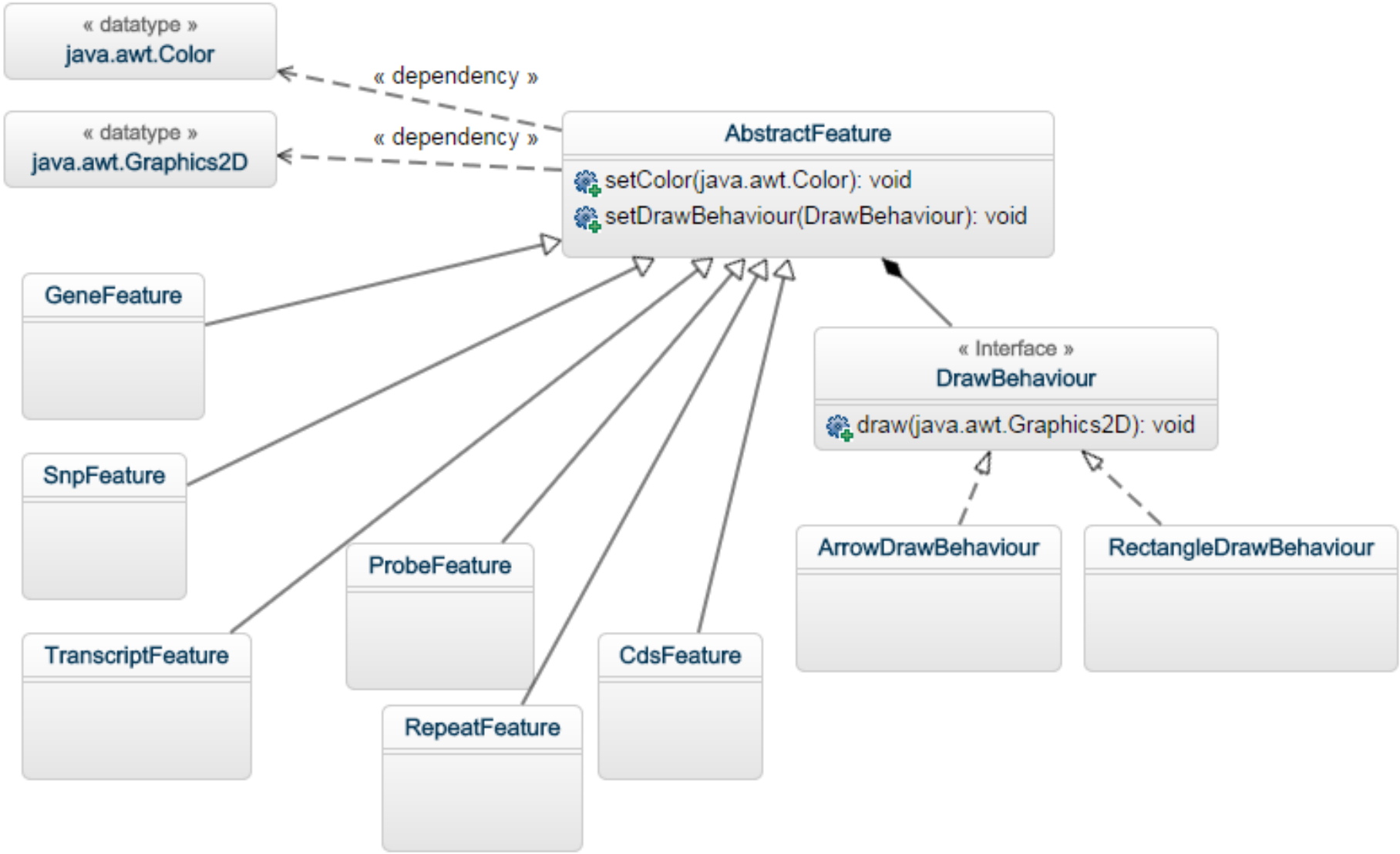
- Do you have something like this?



Strategy Pattern to the rescue

- Given the specs from the previous slide, take a few minutes to create a design for this

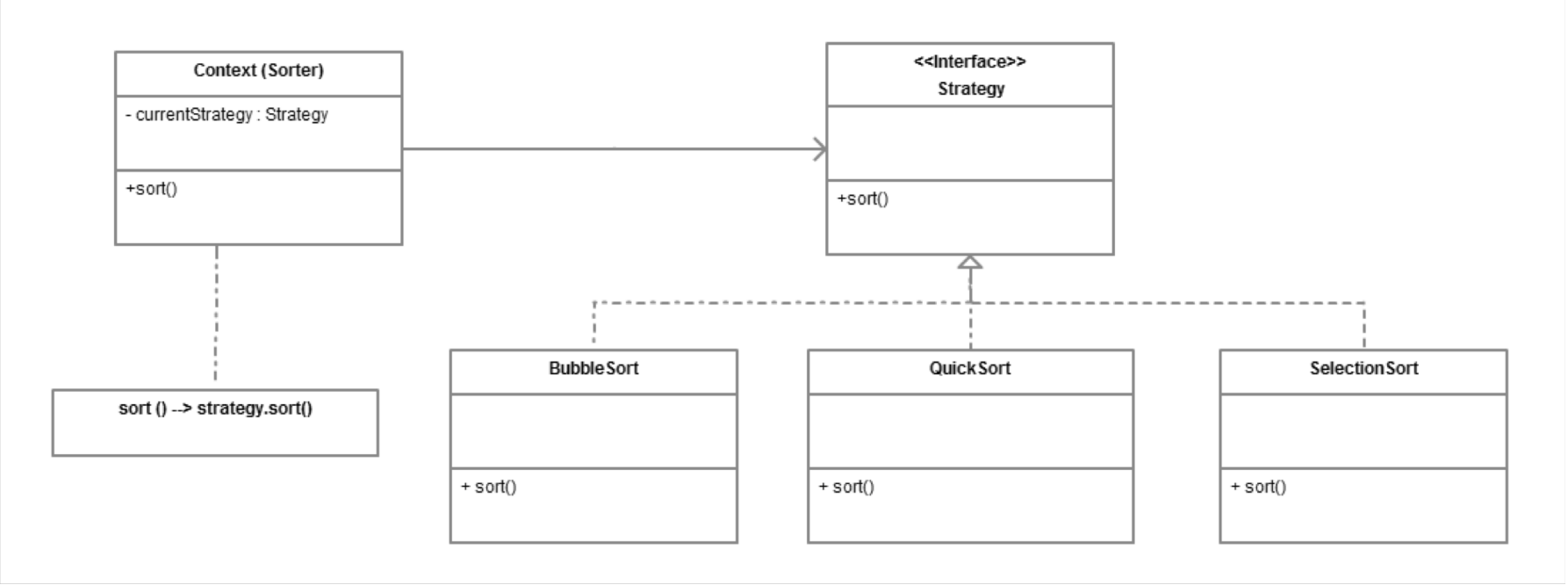
Drawing with the Strategy pattern



Strategy pattern

- The **Strategy** pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Strategy lets the algorithm vary independently from the clients that use it

Strategy pattern UML



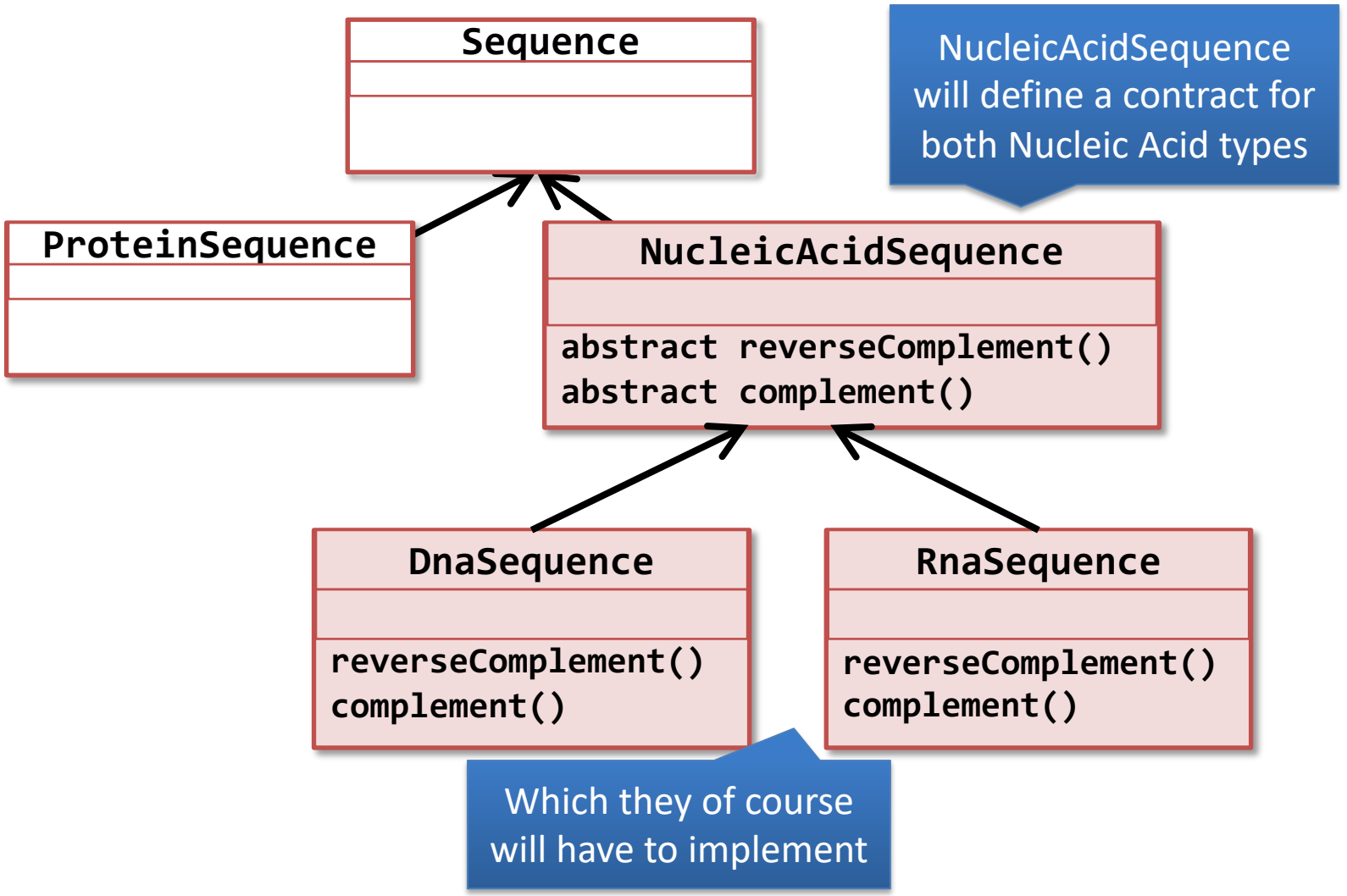
Behavioral

Template method

Template method

- Take five minutes to think about implementing a model for nucleic acid sequences, in particular the complement() and reverseComplement() functionality
- Does it look something like this?

Template method



Template method

- So here's the code for the abstract superclass

```
public abstract class NucleicAcidSequence {
    protected String sequence;

    public void reverse() {
        StringBuilder sb = new StringBuilder(sequence);
        sb.reverse();
        this.sequence = sb.toString();
    }

    public abstract void complement();

    public abstract void reverseComplement();
}
```

Template method

- and here are our friends DNA and RNA

RNA

```
public class RnaSequence extends NucleicAcidSequence{
    public static final HashMap<Character, Character> complements = new HashMap<>();
    static{
        complements.put('A', 'U');          complements.put('U', 'A');
        complements.put('G', 'C');          complements.put('C', 'G');
    }
    @Override
    public void complement() {
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(complements.get(nuc));
        }
        super.sequence = newSequence.toString();
    }

    @Override
    public void reverseComplement() {
        reverse();
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(complements.get(nuc));
        }
        super.sequence = newSequence.toString();
    }
```

DNA

```
public class DnaSequence extends NucleicAcidSequence{
    public static final HashMap<Character, Character> complements = new HashMap<>();
    static{
        complements.put('A', 'T');          complements.put('T', 'A');
        complements.put('G', 'C');          complements.put('C', 'G');
    }
    @Override
    public void complement() {
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(complements.get(nuc));
        }
        super.sequence = newSequence.toString();
    }

    @Override
    public void reverseComplement() {
        reverse();
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(complements.get(nuc));
        }
        super.sequence = newSequence.toString();
    }
```

Template method

- Or were you awake enough to do this?

Template method

```
public class RnaSequence extends NucleicAcidSequence{
    public static final HashMap<Character, Character> complements =
         new HashMap<>();
    static{
        complements.put('A', 'U');          complements.put('U', 'A');
        complements.put('G', 'C');          complements.put('C', 'G');
    }
    @Override
    public void complement() {
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(complements.get(nuc));
        }
        super.sequence = newSequence.toString();
    }
    @Override
    public void reverseComplement() {
        reverse();
        complement();
}
```

Template method

- Now take it a bit further and apply the template method
- What is the **only** code that varies between the two classes?

Template method

```
public abstract class NucleicAcidSequence {
    protected String sequence;
    public void reverse() {
        //same
    }
    public void complement() {
        StringBuilder newSequence = new StringBuilder();
        for (Character nuc : sequence.toCharArray()) {
            newSequence.append(getComplementChar(nuc));
        }
        sequence = newSequence.toString();
    }

    public void reverseComplement() {
        reverse();
        complement();
    }

    public abstract Character getComplementChar(Character nucleotide);
}
```

Yes, you can call an unimplemented abstract method from within an abstract class!

The only thing that differs between DNA and RNA is the complementing nucleotide so lets make that the only varying thing in this design

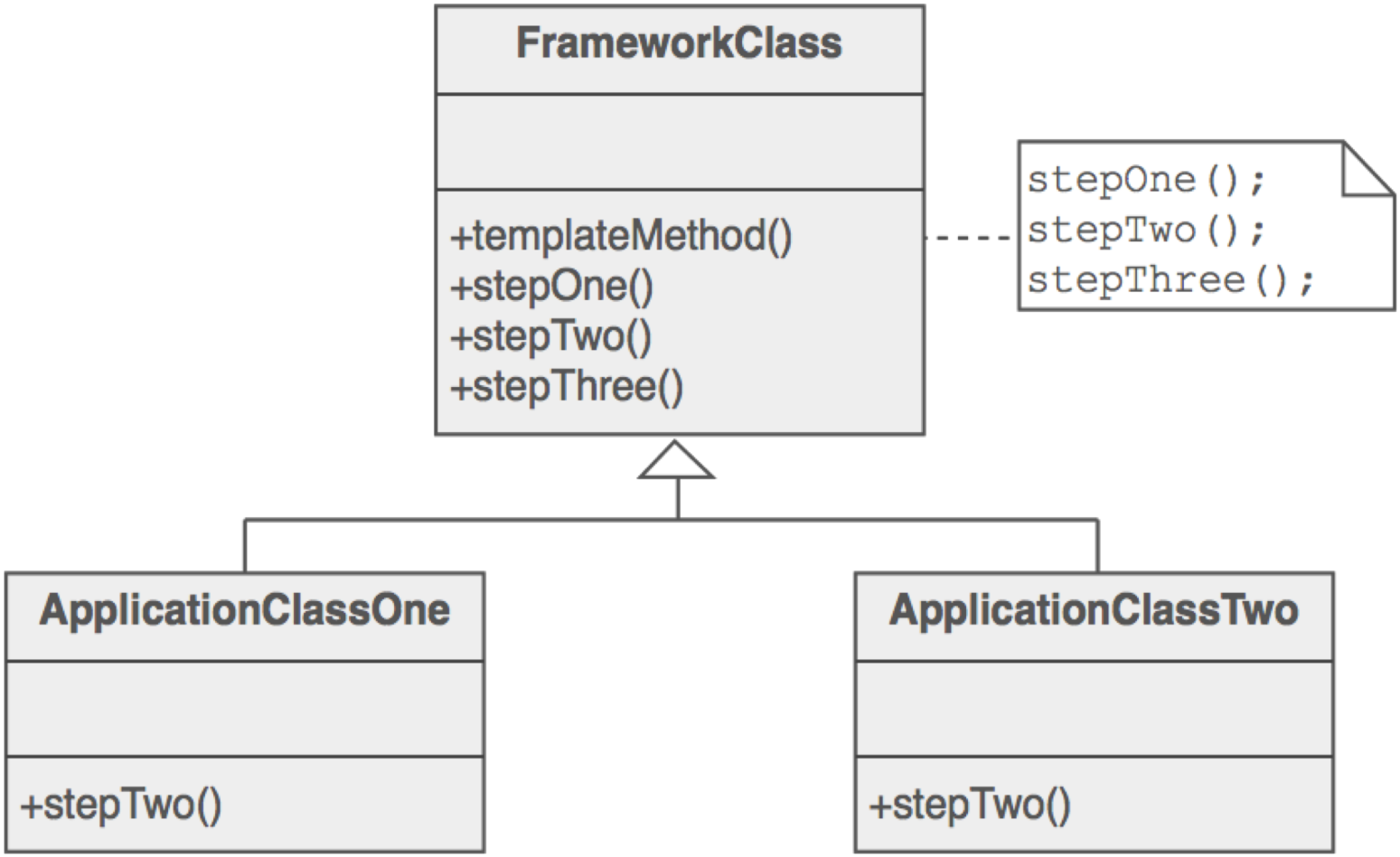
Template method

```
public class RnaSequence extends NucleicAcidSequence{
    public static final HashMap<Character, Character>
                                        complements = new HashMap<>();

    static{
        complements.put('A', 'U');
        complements.put('U', 'A');
        complements.put('G', 'C');
        complements.put('C', 'G');
    }

    @Override
    public Character getComplementChar(Character nucleotide) {
        return complements.get(nucleotide);
    }
```

Template method pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

Template method



Template Method vs Strategy

- Actually, Template Method and Strategy achieve the same through different means
- Strategy through composition and template method through inheritance
- Can you implement the previous Nucleic acids complementing solution using Strategy instead of Template?

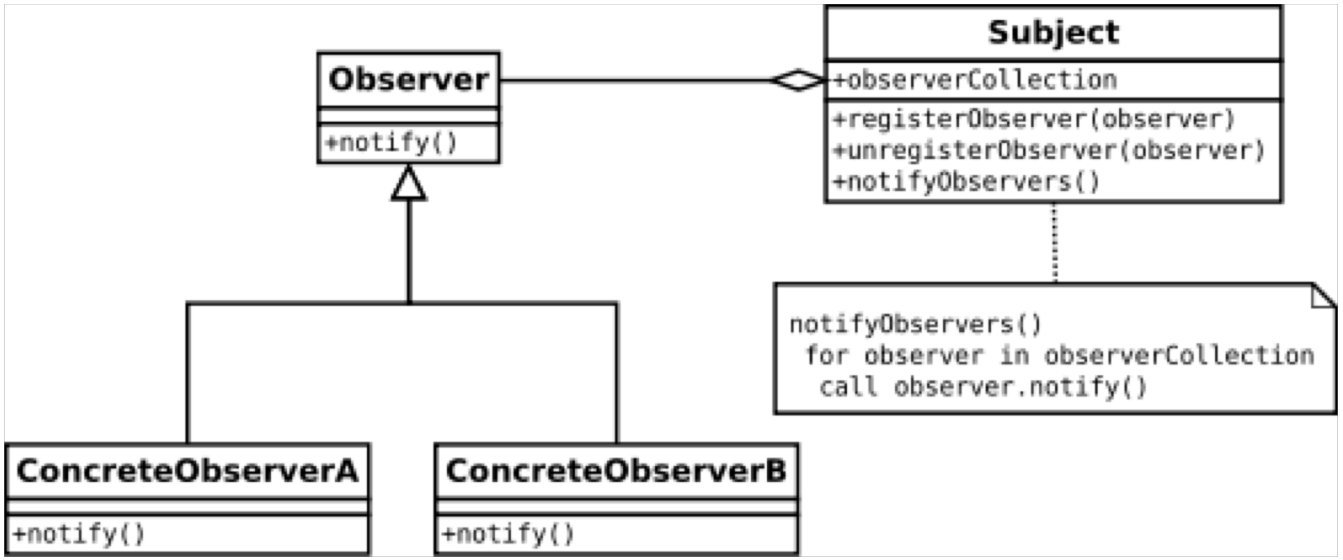
Behavioral

Observer Pattern

Observer

- An object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes
- It is mainly used to implement distributed event handling systems.
- Although the Observer pattern is used primarily in GUI applications, there are other uses for it, e.g. in (parallel) streaming processing settings

Observer UML



Behavioral

Command Pattern

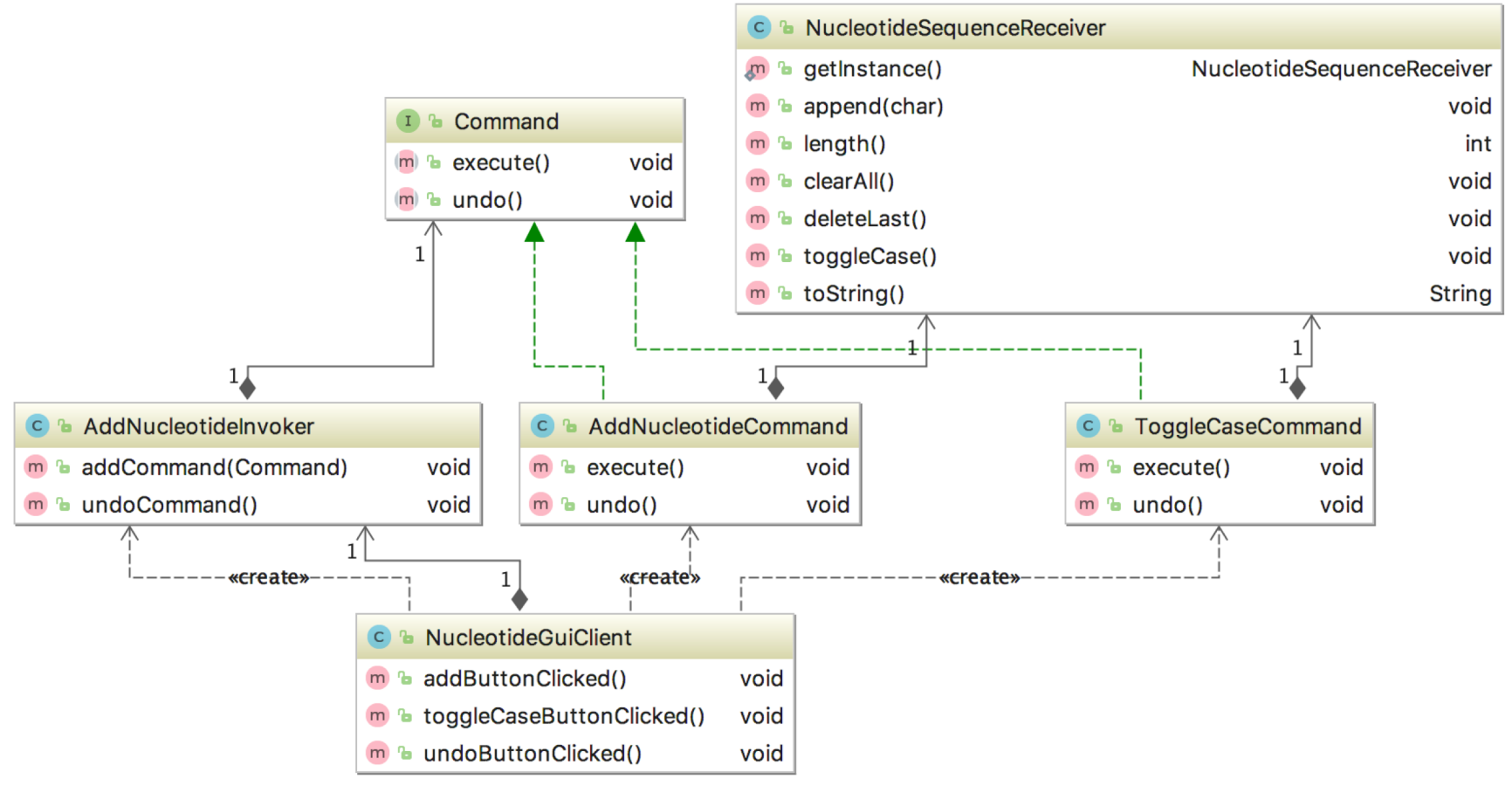
Purpose

- Encapsulates a request allowing it to be treated as an object
- This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

Purpose

- Encapsulates a request (transaction), allowing it to be treated as an object
- Request can be handled in object based relationships such as queuing and callbacks.
- Use When
  - You need callback functionality
  - Requests need to be handled at variant times or in variant orders
  - A history of requests is needed (e.g. for ***undo*** functionality)
- The invoker should be decoupled from the object handling the invocation

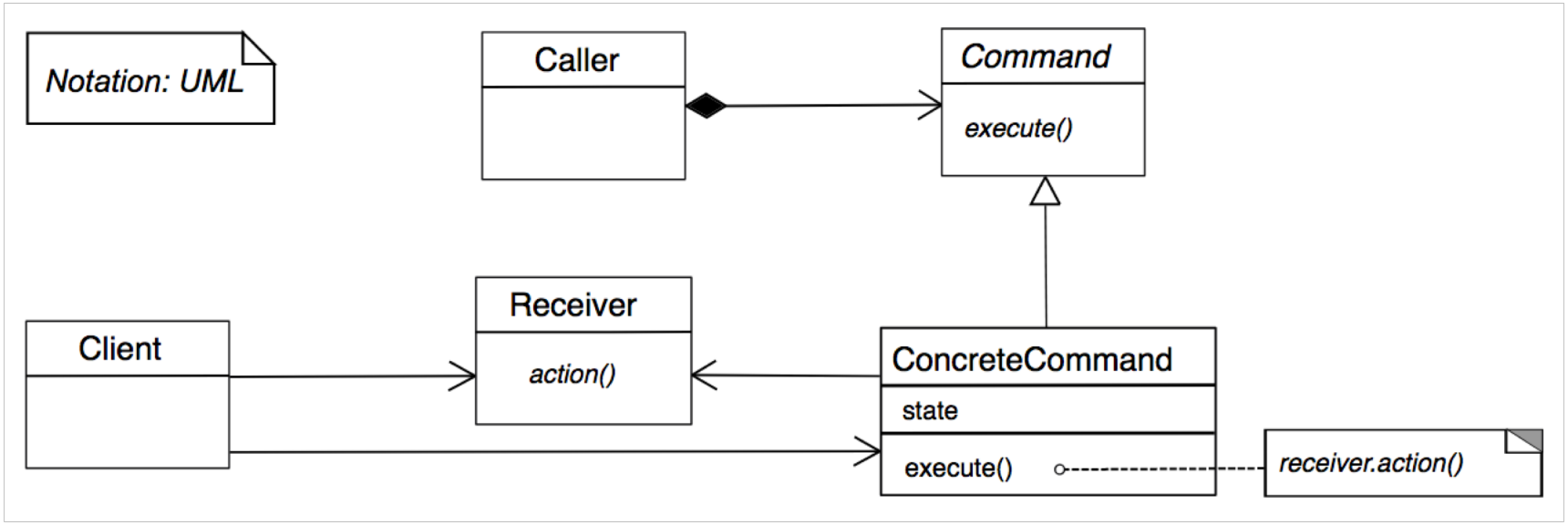
Command Pattern



Command pattern demo

```
after add button clicked five times : sequence=GAGTG
toggled case: sequence=gagtg
added again: sequence=gagtgA
after undo button clicked : sequence=gagtg
after undo button clicked : sequence=GAGTG
toggled case: sequence=gagtg
after undo button clicked : sequence=GAGTG
after undo button clicked : sequence=GAGT
after undo button clicked : sequence=GAG
```

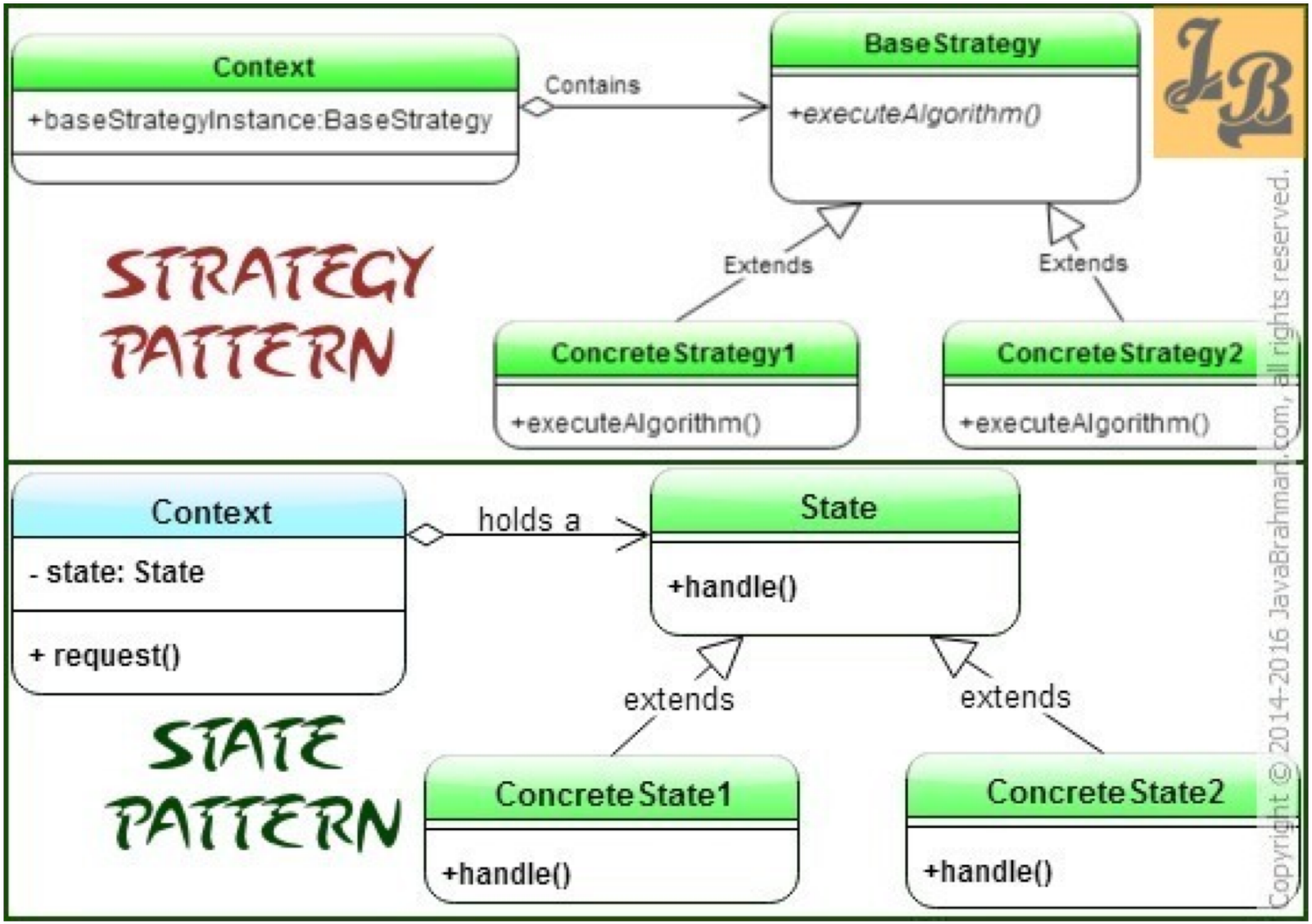
Command Pattern classic



Behavioral

State Pattern

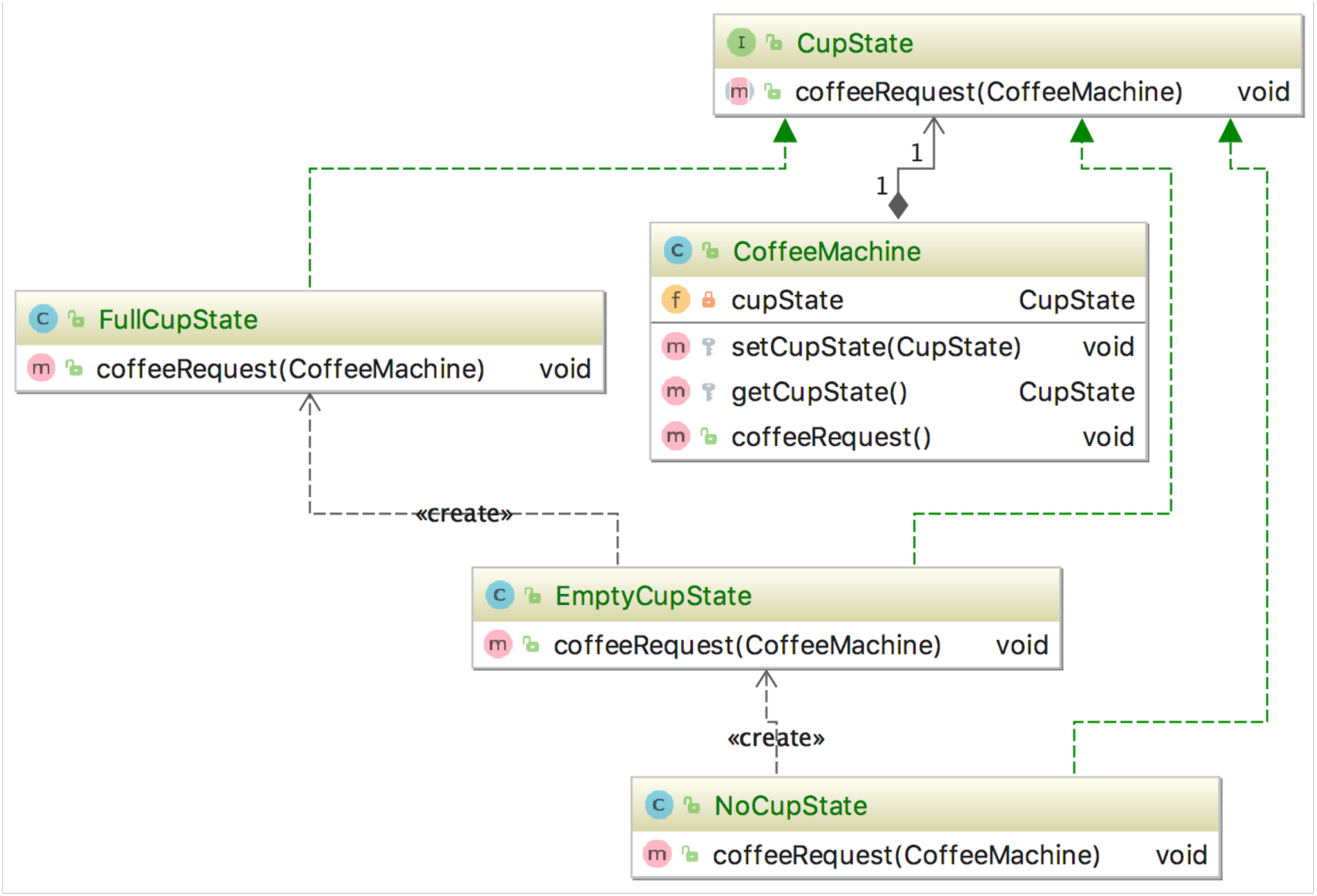
State pattern looks a lot like…



…but the intent is different

- Intent of Strategy Pattern is to have a family of interchangeable algorithms which can be chosen based on the context and/or client needs
- On the other hand, State Pattern's intent is to manage states of the object along with object's behavior which changes with its state.

A coffee machine



Coffee Machine Code (1)

```
public interface CupState {
    void coffeeRequest(CoffeeMachine context);
}


public class NoCupState implements CupState {
    @Override
    public void coffeeRequest(CoffeeMachine context) {
        System.out.println("Placing a cup first..");
        context.setCupState(new EmptyCupState());
        context.coffeeRequest();
    }
}

public class EmptyCupState implements CupState {
    @Override
    public void coffeeRequest(CoffeeMachine context) {
        System.out.println("Pouring a nice Java!");
        context.setCupState(new FullCupState());
    }
}
```

Coffee Machine Code (2)

```
public class FullCupState implements CupState {
    @Override
    public void coffeeRequest(CoffeeMachine context) {
        ...println("Remove your cup before getting a new one!");
    }
}


public class CoffeeMachine {
    private CupState cupState;

    protected void setCupState(CupState cupState) {
        this.cupState = cupState;
    }

    public void coffeeRequest() {
        this.cupState.coffeeRequest(this);
    }
}
```

Coffee machine test

```
CoffeeMachine coffeeMachine = new CoffeeMachine();
//machine is empty
coffeeMachine.setCupState(new NoCupState());
coffeeMachine.coffeeRequest(); //request 1
coffeeMachine.coffeeRequest(); //request 2
//somebody uses her own mug
coffeeMachine.setCupState(new EmptyCupState());
coffeeMachine.coffeeRequest(); //request 3
```

Placing a cup first.. `//request 1`
Pouring a nice Java! `//request 1 with new state`
Remove your cup before getting a new one please! `//request 2`
Pouring a nice Java! `//request 3`

Behavioral

Filter Pattern

Filtering SNPs

- Suppose you are working on an application for primer analysis
- This includes several optional and configurable filter steps:
  - GC percentage filter
  - Length filter
  - Homopolymer filter
  - …
- Again, take a minute to think about how you would implement this

Primer.java

```
public class Primer {
    private String sequence;

    public double getGcPercentage() {
        //solve this the Java 8 way
        final int[] gcCount = new int[]{0};
        this.sequence.chars().forEach(
            (n) -> {if(n == 67 || n == 71){gcCount[0]++;}}
        );
        return (double) gcCount[0] / this.sequence.length();
    }

    public double getMeltingTemperature() {
        //Tm logic
    }
//more code
```

PrimerFilter.java

```
public interface PrimerFilter {
    /**
     * checks the given primer.
     * @param primer the primer
     * @return primerOK
     */
    boolean isOK(Primer primer);
    /**
     * returns this filter name.
     * @return name
     */
    String getName();
}
```

Creating a filter (anonymous local inner class)

```
List<PrimerFilter> filters = new ArrayList<>();
//adds length filter
filters.add(new PrimerFilter(){
    @Override
    public boolean isOK(Primer primer) {
        return (primer.getLength() >= 18
                && primer.getLength() < 25);
    }
    @Override
    public String getName() {
        return "Length filter [18..24]";
    }
});
```

Creating a filter (static inner class)

```
private static class HomopolymerFilter implements PrimerFilter {
    private final int maxPolymer;
    private List<String> polymers = new ArrayList<>();

    public HomopolymerFilter(int maxHomopolymer) {
        this.maxPolymer = maxHomopolymer;
        createPolymers(); //method not shown
    }

    @Override
    public boolean isOK(Primer primer) {
        for (String hp : this.polymers) {
            if (primer.getSequence().contains(hp)) return false;
        }
        return true;
    }

    @Override
    public String getName() {
        return "Homoplolymer filter (<";
    }
}
```

Using the filters

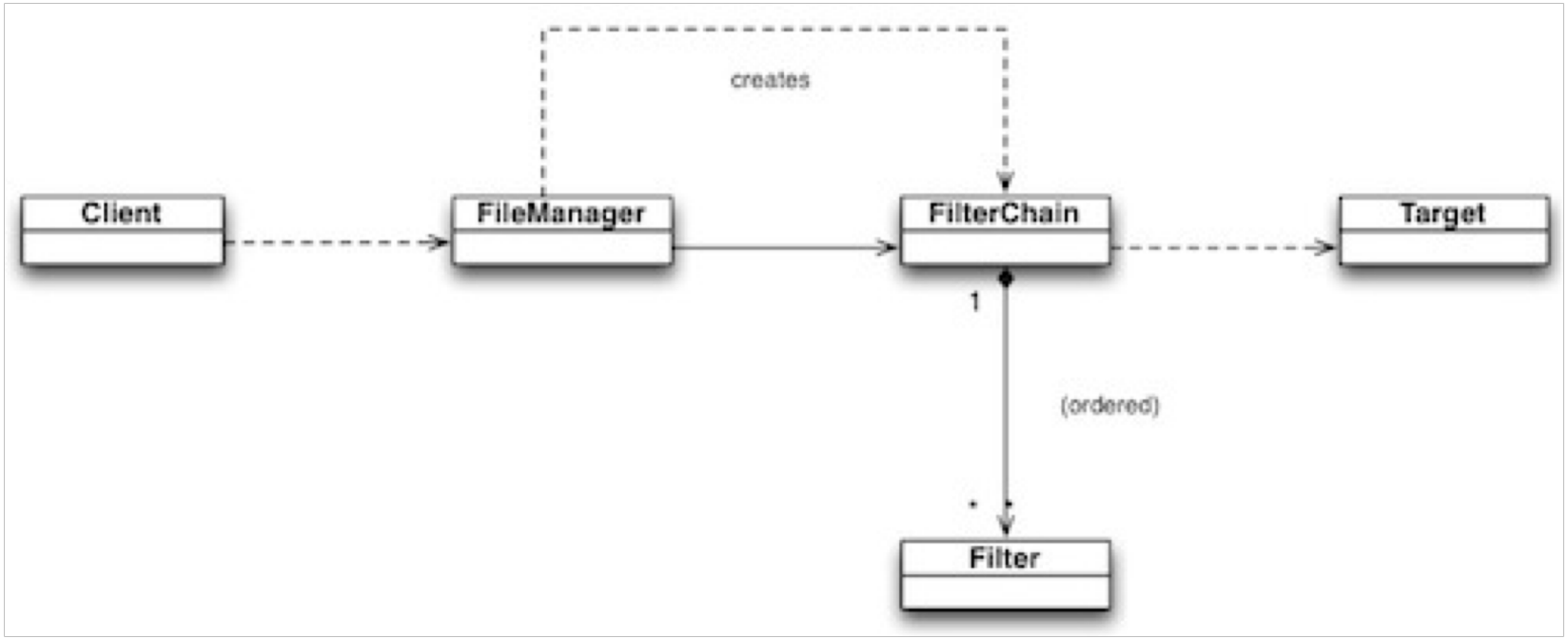
```
for (Primer p : primers) {
    for (PrimerFilter pf : filters) {
        System.out.println("primer " + p + ": pf = "
                + pf.getName() + " says: " + pf.isOK(p));
    }
}
```

```
Primer{GC%=0.45, Tm=58.0, l=20} Length filter [18, 24] OK: true
Primer{GC%=0.45, Tm=58.0, l=20} GC% filter [35, 60) OK: true
Primer{GC%=0.45, Tm=58.0, l=20} Tm filter [50, 65) OK: true
Primer{GC%=0.45, Tm=58.0, l=20} Homoplolymer filter OK: true

Primer{GC%=0.6363636363636364, Tm=72.0, l=22} Length filter [18, 24] OK: true
Primer{GC%=0.6363636363636364, Tm=72.0, l=22} GC% filter [35, 60) OK: false
Primer{GC%=0.6363636363636364, Tm=72.0, l=22} Tm filter [50, 65) OK: false
Primer{GC%=0.6363636363636364, Tm=72.0, l=22} Homoplolymer filter OK: true

Primer{GC%=0.4642857142857143, Tm=82.0, l=28} Length filter [18, 24] OK: false
Primer{GC%=0.4642857142857143, Tm=82.0, l=28} GC% filter [35, 60) OK: true
Primer{GC%=0.4642857142857143, Tm=82.0, l=28} Tm filter [50, 65) OK: false
Primer{GC%=0.4642857142857143, Tm=82.0, l=28} Homoplolymer filter OK: true
```

Filter (aka Criteria) pattern

- The previous example represented the simplest implementation of the pattern
- It enables you to filter a set of objects, using different criteria, chaining them in a decoupled way through logical operations

Filter pattern UML



Behavioral

Null Object or Special Case Pattern

Null is coming!

- When working with streaming processing (see next presentation), it may happen there is an occasional null object.
  - You do NOT want your app to crash
  - you do NOT want to put null checks or try/catch all over the place
- This is how to solve this issue

Support our homeless!

- Suppose you have this simple class:

```
public class User {
    private long id;
    private String name;
    private int numberOfLogins;
    private Address address;
    //code omitted
}
```

- What happens if you are going to process millions of Users, like this, and some won't have an address?

```
users
    .stream()
    .forEach(x -> printUser(x.getName() + ":" + x.getAddress()));
```

Support our homeless!

- So are you going to do this?

```
users
    .stream()
    .forEach(x -> printUser(x.getName() + ":"
         + x.getAddress() == null ? "HOMELESS" : x.getAddress()));
```

- Or this?

```
users.stream()
  .forEach(x -> {
    try{
      System.out.println(x.getName() + ":" + x.getAddress());
    } catch (NullPointerException ex) {
      System.out.println(x.getName() + ": HOMELESS");
    }
  });
```

Support our homeless!

- Or do you use the Null Object?

```
public class Address {
    public static final Address;

    //code omitted
    static {
        DEFAULT_NO_ADDRESS = new Address();
        DEFAULT_NO_ADDRESS.street = "HOMELESS";
    }
    //code omitted
```

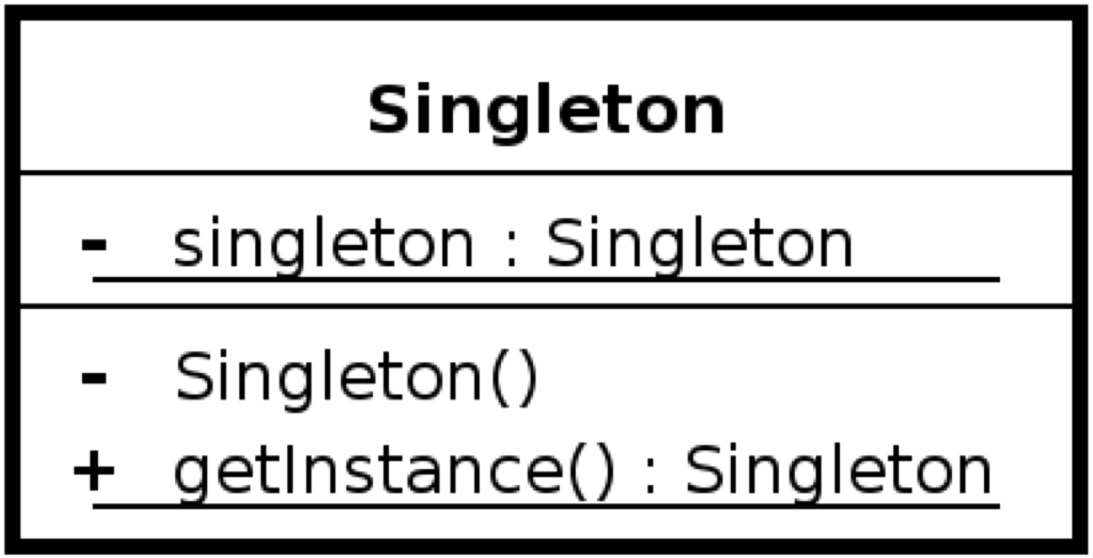
- User objects are by default (or explicitly) instantiated with this default Address.
- No other check required!

Creational

Singleton Pattern

Singleton Pattern

- Restricts the instantiation of a class to one object only
- Used often in Controllers, DAO objects etc
- It's UML is really simple



Note the private constructor!

Classic Singleton in code

Not thread safe!

```
public class ClassicSingleton {
    private static ClassicSingleton instance;
    /**
     * private constructor ensures no one can
     * instantiate it beside its own class!
     */
    private ClassicSingleton() { }
    /**
     * The only means to get hold of the instance.
     * Uses lazy instantiation.
     * @return
     */
    public static ClassicSingleton getInstance() {
        if (instance == null)
            instance = new ClassicSingleton();
        return instance;
    }
}
```

Thread-safe Singleton

```
public class ThreadSafeSingleton {
  private static volatile ThreadSafeSingleton instance = null;
  private ThreadSafeSingleton() {}

  public static ThreadSafeSingleton getInstance() {
    if (instance == null) {
      synchronized(ThreadSafeSingleton.class) {
        if (instance == null) {
          instance = new ThreadSafeSingleton();
        }
      }
    }
    return instance;
  }
}
```

Synchronized?

- Java synchronized blocks can be used to avoid race conditions
- A synchronized block in Java is synchronized on some object
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time

Volatile?

- Volatile is used to indicate that a variable's value will be modified by different threads
- Declaring a volatile Java variable means:
  - The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory”
  - Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

Creational

Factory

Factory

- It comes in several flavors, depending on the complexity of your model:
  - Factory Method
  - Factory (class)
  - Abstract Factory

Creational

Factory Method Pattern

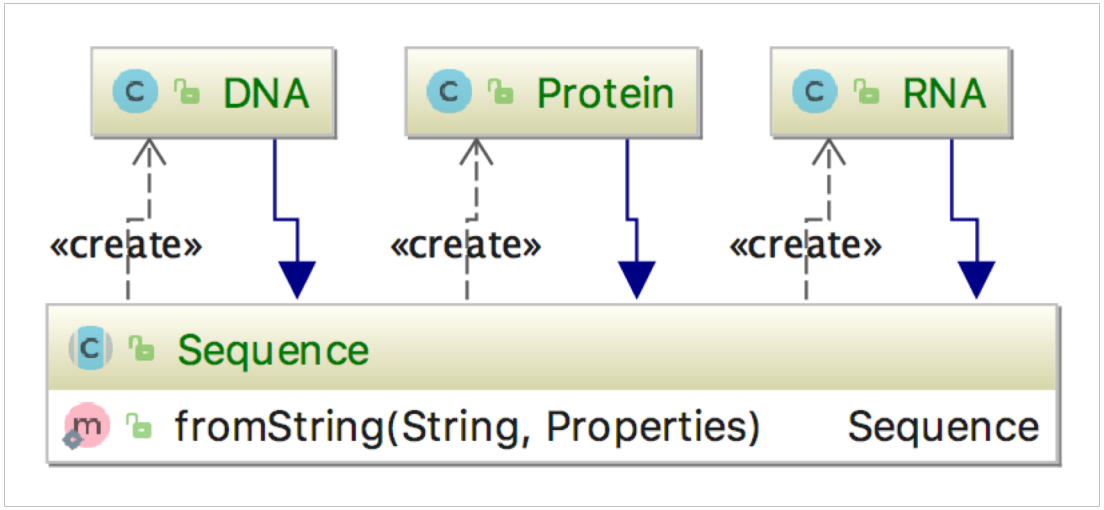
Factory

- In Factory pattern, we create an object without exposing the creation logic to the client and refer to newly created object using a common interface

More advanced:

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- The Factory method lets a class defer instantiation it uses to subclasses.

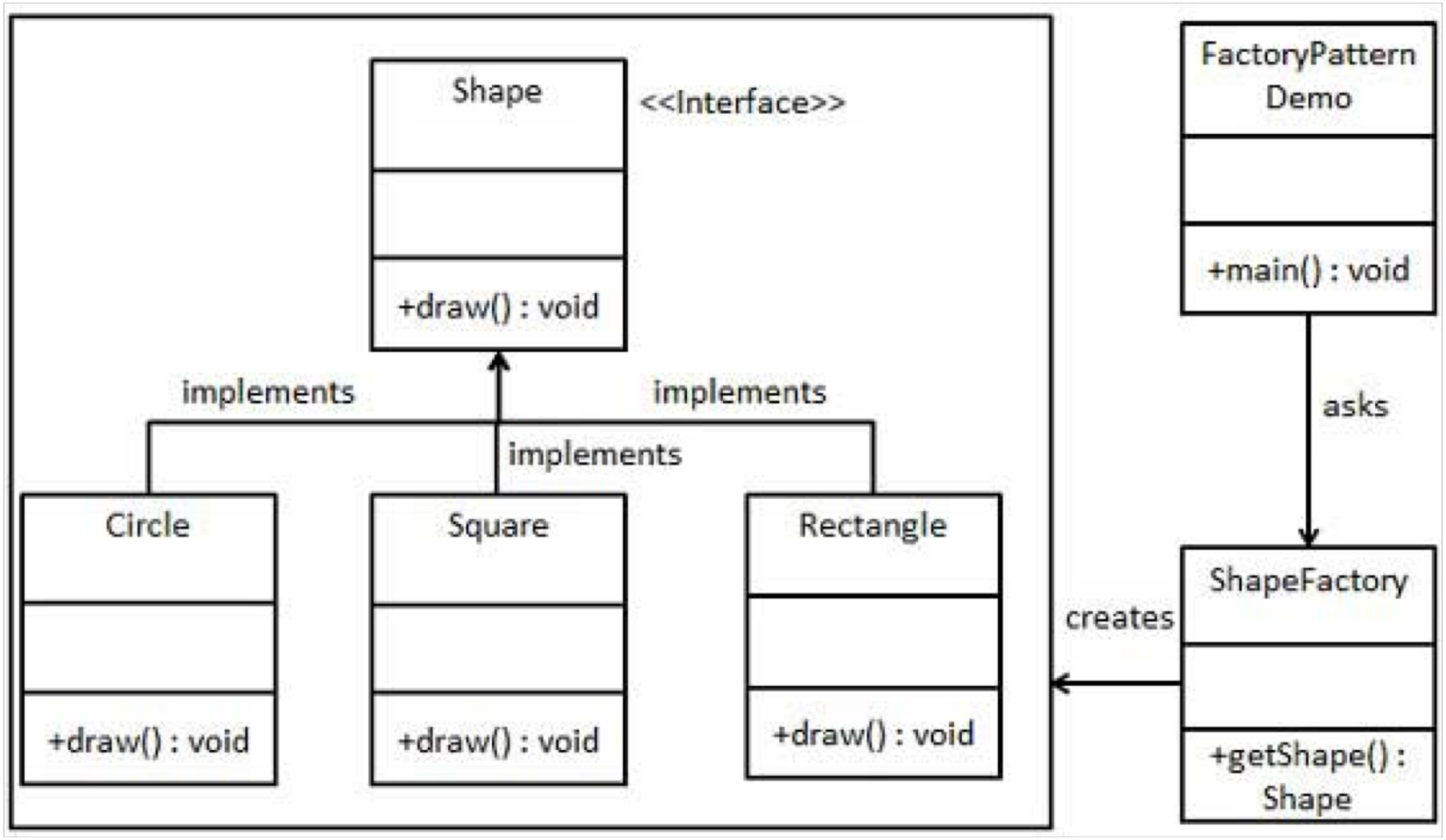
A Sequence Factory method



A Sequence Factory method

```
public static Sequence fromString(
                    String sequence, Properties properties){
    sequence.toUpperCase();
    Sequence newSeq;
    //determine what type the string represents
    //create the correct subtype (DNA, RNA, ...)
    if (sequence.contains("T"))
        newSeq = new DNA();
    else if (sequence.contains("U"))
        newSeq = new RNA();
    else
        newSeq = new Protein();
    //process the properties and return the created object
    //nobody needs to know what subtype is being dealt with
    return newSeq;
}
```

Factory pattern classic UML



Creational

Factory Class Pattern

Best explained through a use case

Building composite filter objects for

- Probe filtering for microarray
- Primer filtering for qPCR

Will involve complex construction that can be abstracted away in an Abstract Factory Class, subtype of FilterFactory:

- MicroarrayProbeFilterFactory
- PcrPrimerFilterFactory

Can you implement this model?

Creational

Builder Pattern

A class with many properties

- Suppose you have a class with many (>3) properties that can be set, and you don't want null values.
- Here is a valid way of implementing this scenario, using the ***telecoping constructor pattern***.

```
public class Sequence {
    private String sequence;
    private String accession;
    private String name;
    private SequenceType type;

    public Sequence(String sequence) {
        this(sequence, "_UNKNOWN_ACCNO_");
    }

    public Sequence(String sequence, String accession) {
        this(sequence, accession, ”_ANONYMOUS_");
    }

    public Sequence(String sequence, String accession, String name) {
        this(sequence, accession, name, SequenceType.UNKNOWN);
    }

    public Sequence(String sequence,
                    String accession,
                    String name,
                    SequenceType type) {
        this.sequence = sequence;
        this.accession = accession;
        this.name = name;
        this.type = type;
    }
```

A class with many properties

- The problem here is that this leaves you with
  - hard-to-read client code
  - high risk of mistaken argument position
  - only certain combinations have defaults
  - objects at risk of corrupted data (no atomic construction)

Enter the builder pattern!

- Implementing such a class with a Builder gives you
  - atomic construction
  - easy combination of parameters
  - harder-to-misplace arguments
  - very readable client code
- Here is the implementation, in three steps

A static inner Builder class

```
public static class Builder {
    //required parameter
    private final String sequence;
    //optional parameters
    private String name = "_ANONYMOUS_";
    private String accession = "_UNKNOWN_ACCNO_";
    private SequenceType type = SequenceType.UNKNOWN;

    private Builder(String sequence) {
        this.sequence = sequence;
    }

    public Builder name(String name) {
        this.name = name; return this;
    }

    public Builder accession(String accession) {
        this.accession = accession; return this;
    }

    public Builder type(SequenceType type) {
        this.type = type; return this;
    }

    public Sequence build() {
        return new Sequence(this);
    }
}
```

A private Sequence constructor

- A single private one
- Taking a Builder object as argument

```
private Sequence(Builder builder) {
    this.sequence = builder.sequence;
    this.accession = builder.accession;
    this.name = builder.name;
    this.type = builder.type;
}
```

A Builder provider

- A static method serving an instance of the inner Builder class
- Taking the required sequence string as argument

```
//in class Sequence
public static Builder builder(String sequence) {
    return new Builder(sequence);
}
```

Test drive

- Chained construction of the Sequence instance makes for very readable code

```
Sequence sequence = Sequence.builder("GAATTC")
        .accession("GB|123456")
        .name("RNA Polymerase III")
        .type(SequenceType.DNA)
        .build();
```

- Chained construction with same-type arguments is even simpler to implement:

```
private static class Builder {
    private final PizzaBase base;
    private List<Ingredient> ingredients = new ArrayList<>();

    private Builder(PizzaBase base) {
        this.base = base;
    }

    public Builder ingedient(Ingredient ingredient) {
        this.ingredients.add(ingredient); return this;
    }

    public Pizza build() {
        return new Pizza(this);
    }
}
```

```
Pizza pizza = Pizza
        .builder(PizzaBase.EXTRA_THICK)
        .ingedient(new Ingredient("cheese"))
        .ingedient(new Ingredient("onions"))
        .ingedient(new Ingredient("peppers"))
        .ingedient(new Ingredient("gorgonzola"))
        .build();
```

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Builder pattern builds a complex object using simple objects using a step by step approach.

Structural

Facade

Facade pattern

- Facade pattern hides the complexities of the system and provides a simple interface to the client
- The client uses only this interface to access the system
- A well-known example is the use of DAO classes to abstract away the complexities of database interaction

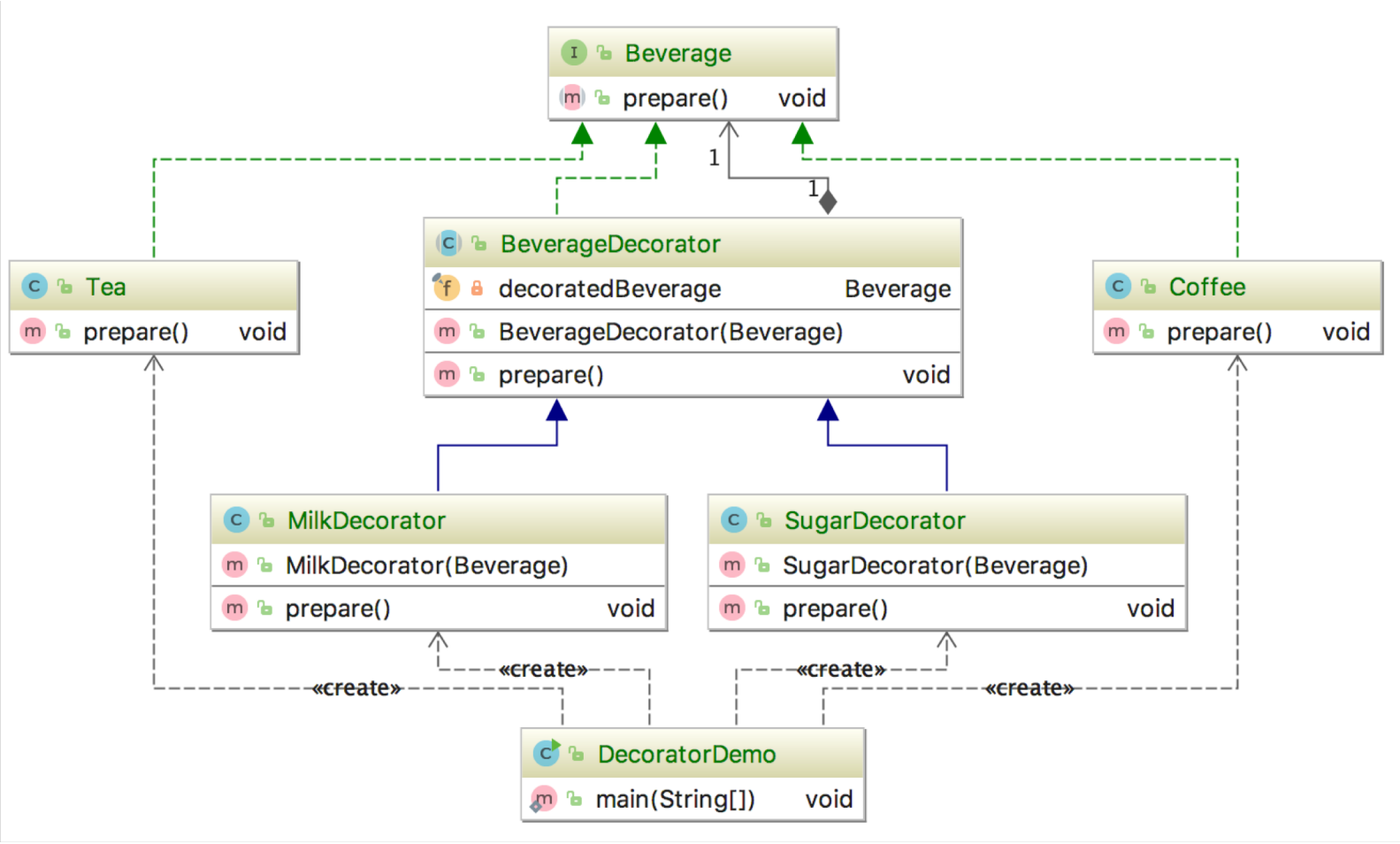
Structural

Decorator

Decorator intent

- Attach additional responsibilities to an object dynamically
- Decorators provide a flexible alternative to subclassing for extending functionality
- An example best explains

Making a brew



Some code

```java
public interface Beverage {
    void prepare();
}
```

```java
public class Coffee implements Beverage {
    @Override
    public void prepare() {
        System.out.println("Preparing a nice hot Java!");
    }
}
```

```java
public abstract class BeverageDecorator implements Beverage {
    private final Beverage decoratedBeverage;

    public BeverageDecorator(Beverage beverage) {
        this.decoratedBeverage = beverage;
    }

    @Override
    public void prepare() {
        decoratedBeverage.prepare();
    }
}
```

Testing

```
//create coffee with milk and sugar
Beverage coffee = new MilkDecorator(
                        new SugarDecorator(
                            new Coffee()));
coffee.prepare();

//create tea with sugar only
Beverage tea = new SugarDecorator(new Tea());
tea.prepare();
```

```
preparing a nice hot Java!
..adding some sugar
..adding milk
Making plain tea
..adding some sugar
```

- Of course, there are (many) more patterns out there
- Whenever you have smelly code, go on an internet hike to find out if there is an elegant solution

The end