Template Method Pattern

Acknowledgement: Some material in these slides is adopted from Head First Design patterns

Starbuzz Barista Training Manual

Starbuzz Coffee recipe

- Boil some water
- Brew coffee in boiling water
- Pour coffee in cup
- Add sugar and milk

Starbuzz Tea Recipe

- Boil some water
- Steep Tea in boiling water
- Pour tea in cup
- Add lemon

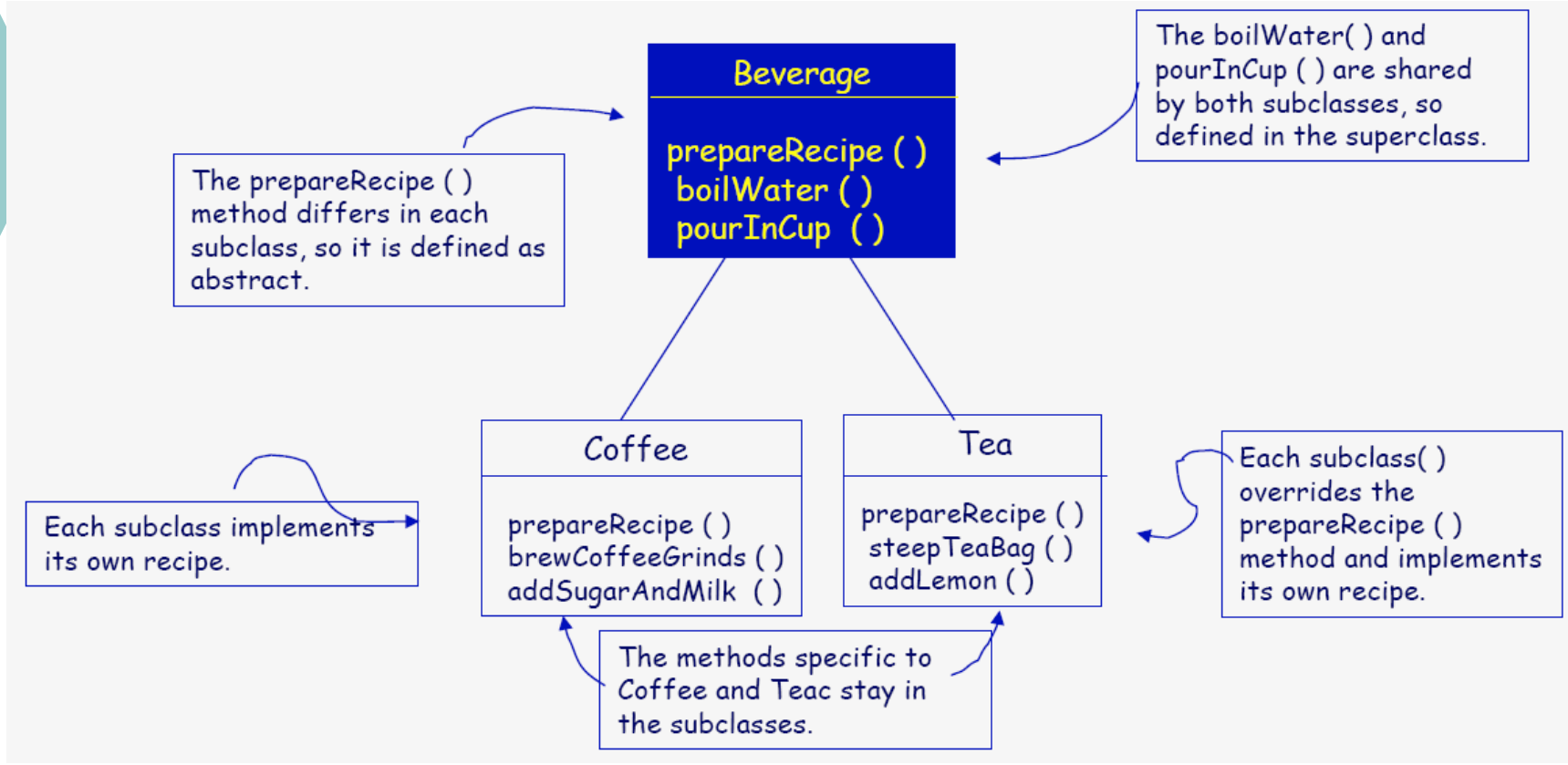
Coffee Code

```
public class Coffee {
    void prepareRecipe () {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println (“Boiling Water”)
    }
    public void brewCoffeeGrinds() {
        System.out.println (“Dripping coffee though filter”)
    }
    public void pourInCup() {
        System.out.println (“Pouring into Cup”)
    }
    public void addSugarAndMilk() {
        System.out.println (“Adding Sugar and Milk”)
    }
}
```

Tea Code

```
public class Tea {
    void prepareRecipe () {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println (“Boiling Water”)
    }
    public void steepTeaBag() {
        System.out.println (“Steeping the Tea”)
    }
    public void pourInCup() {
        System.out.println (“Pouring into Cup”)
    }
    public void addLemon() {
        System.out.println (“Adding Lemon”)
    }
}
```

Exercise
Redesign class diagram to remove redundancy



Did we do a good job on the redesign? Are we overlooking some other commonality? What are the other ways that Coffee and Tea are similar

Taking the design further

Coffee recipe

- Boil some water
- Brew coffee in boiling water
- Pour coffee in cup
- Add sugar and milk

Tea Recipe

- Boil some water
- Steep Tea in boiling water
- Pour tea in cup
- Add lemon

Both recipes follow same algorithm

- Boil some water
- Use hot water to extract Coffee or Tea
- Pour coffee in cup
- Add the appropriate condiments to the beverage

Taking the design further

```
void prepareRecipe () {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
```

```
void prepareRecipe () {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }
```

```
void prepareRecipe () {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
```

Beverage Code

```
public abstract class CaffeineBeverage {

    void prepareRecipe () {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    public void boilWater() {
        System.out.println (“Boiling Water”)
    }

    abstract void brew();

    public void pourInCup() {
        System.out.println (“Pouring into Cup”)
    }

    abstract void addCondiments();
}
```

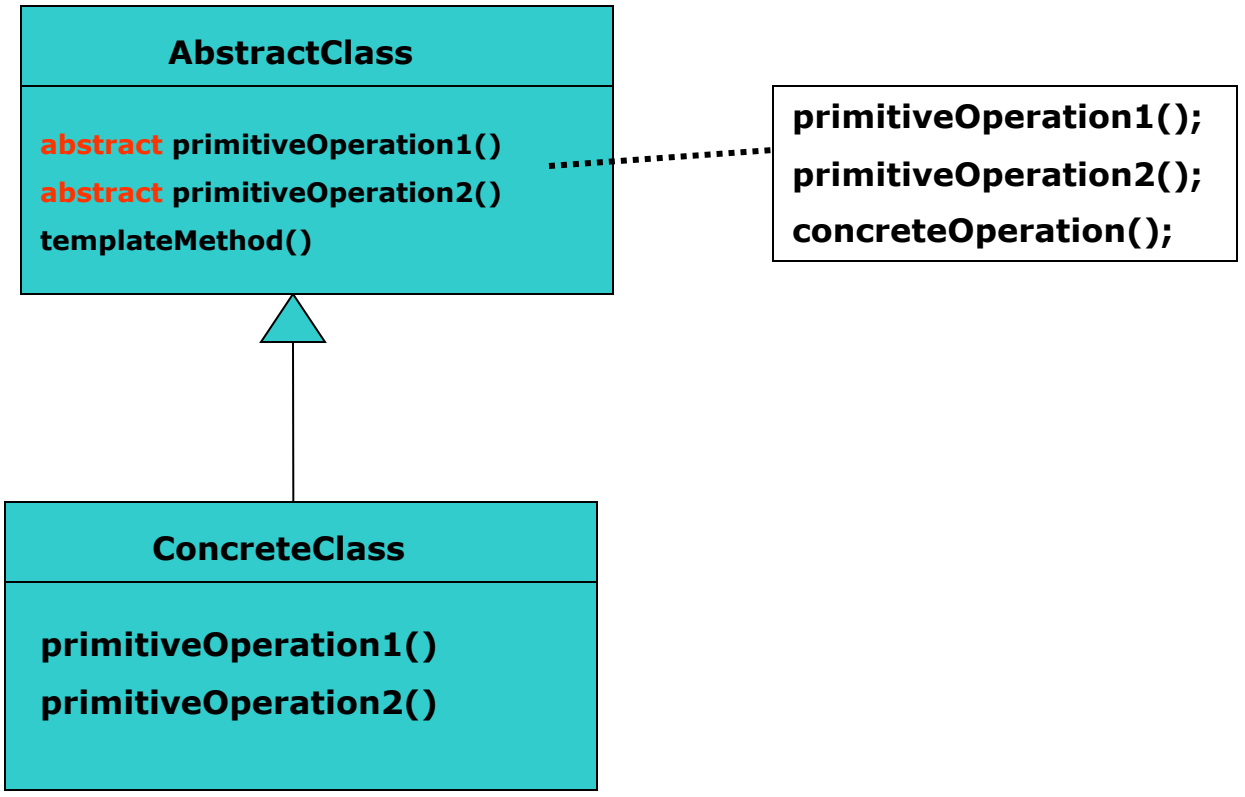
Tea and Coffee Code

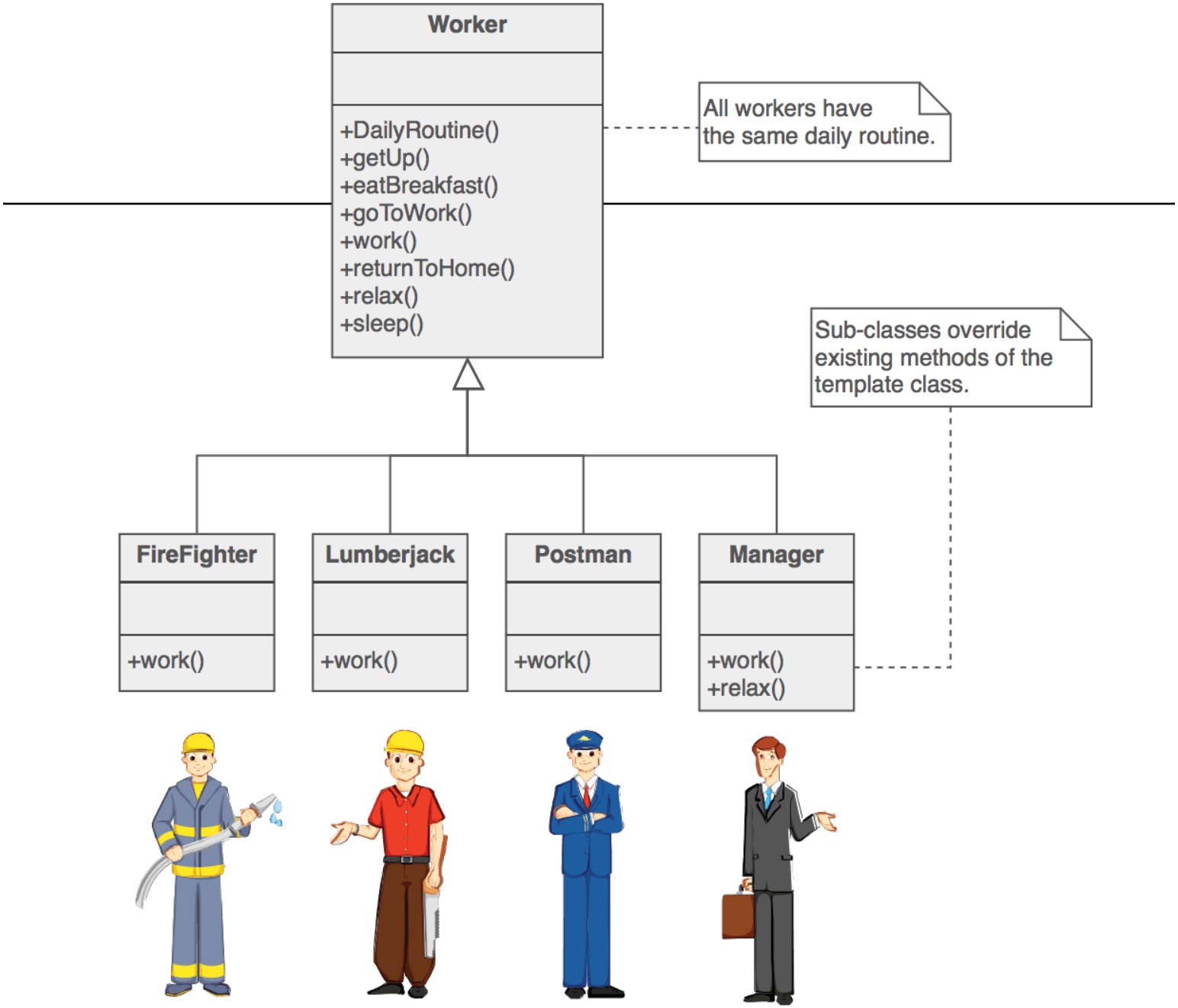
```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println (“Dripping coffee though filter”)
    }
    public void addCondiments() {
        System.out.println (“Adding Sugar and Milk”)
    }
}

public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println (“Steeping the tea”)
    }
    public void addCondiments() {
        System.out.println (“Adding Lemon”)
    }
}
```

Template Method Pattern Defined

- The Template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template method – class diagram




Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template class.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()

When to use...

- Let subclasses implement alternate behavior by method overriding
- Avoid code duplication
- Super class method still maintains control compared to a simple polymorphic override, where the base method is entirely rewritten

Hollywood Principle

Don’t Call us, We’ll Call you!

Important OO Principle - Composition over inheritance !

- Classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance
  - Better Testability
  - Inheritance may break encapsulation
  - More flexibility (with replacement of composed class implementation) – for example, if you are using a Comparator class, changing to a different type is easier at run-time

The Strategy Pattern

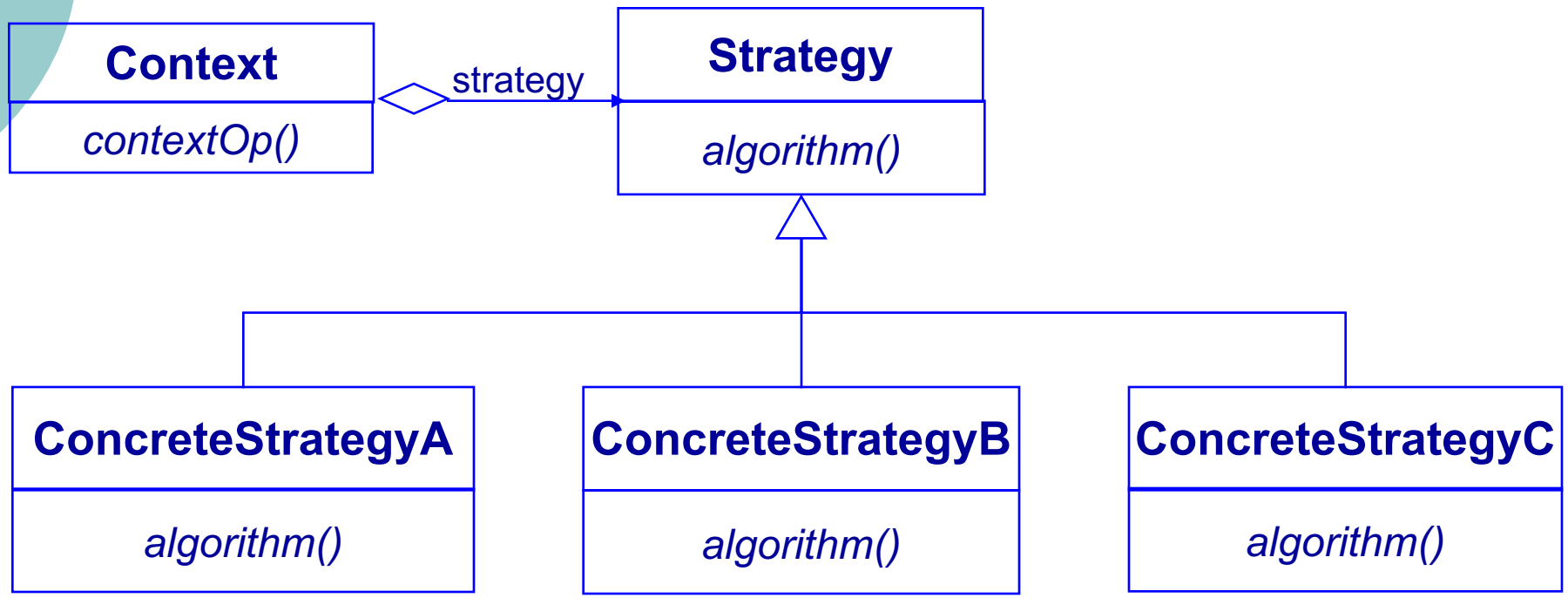
Motivation

- Problem
  - You have a family of related algorithms
    - Sorting and Searching
    - Line Breaking and Page Layout
    - AWT Layout Managers
- Desire
  - Encapsulate the algorithms so they can be used interchangeably
  - Isolate clients from the specific algorithm employed
- Solution
  - Define a common interface for each group of algorithms
  - Encapsulate algorithms in a *Strategy* (aka Objects as Algorithms)

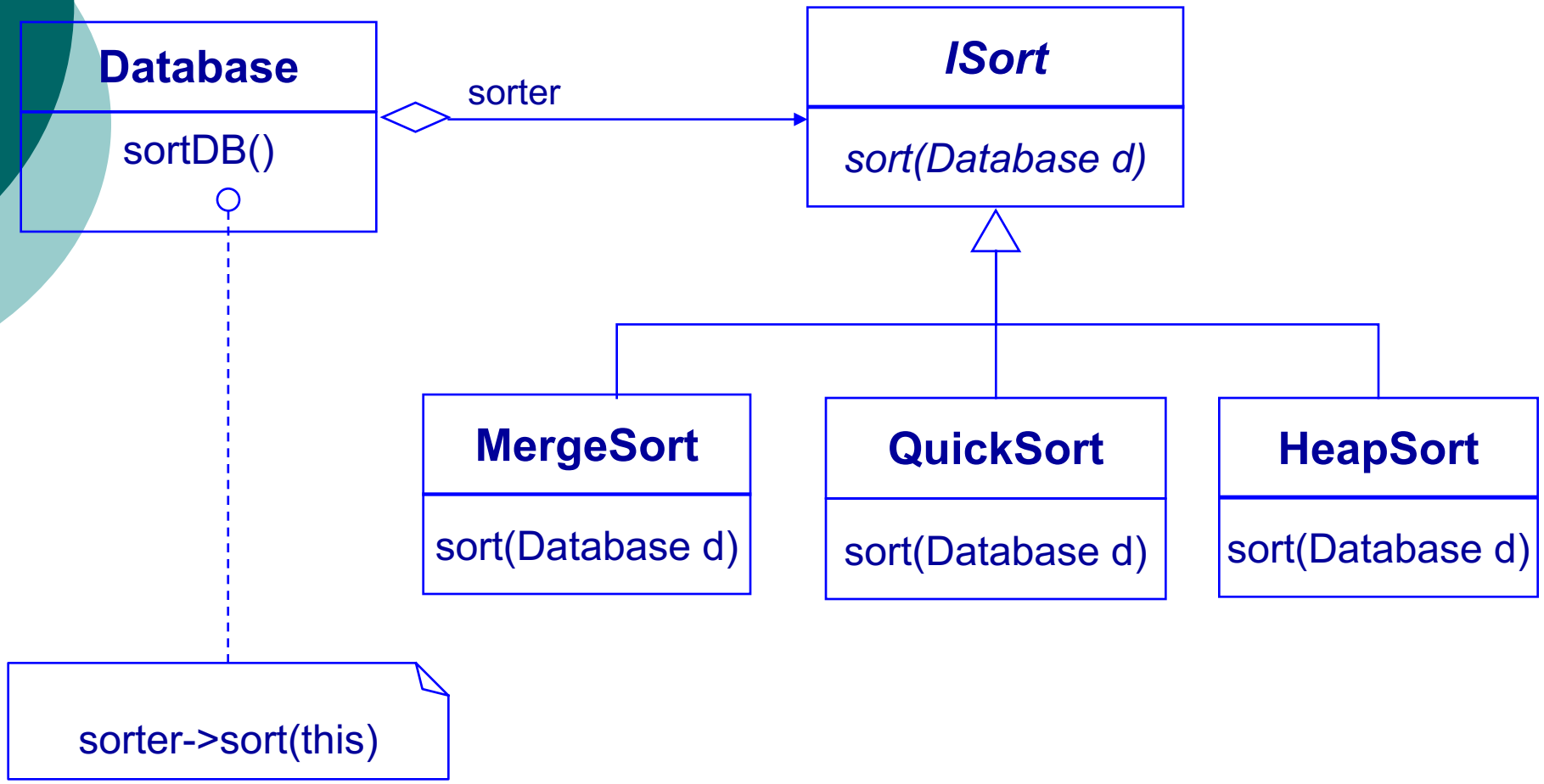
Strategy Pattern

- Provide invariant interface to varying algorithms:
  - Java interface
  - Java abstract class / C++ pure virtual functions
- Subclass for each algorithm variant
- Parameterize clients by interface

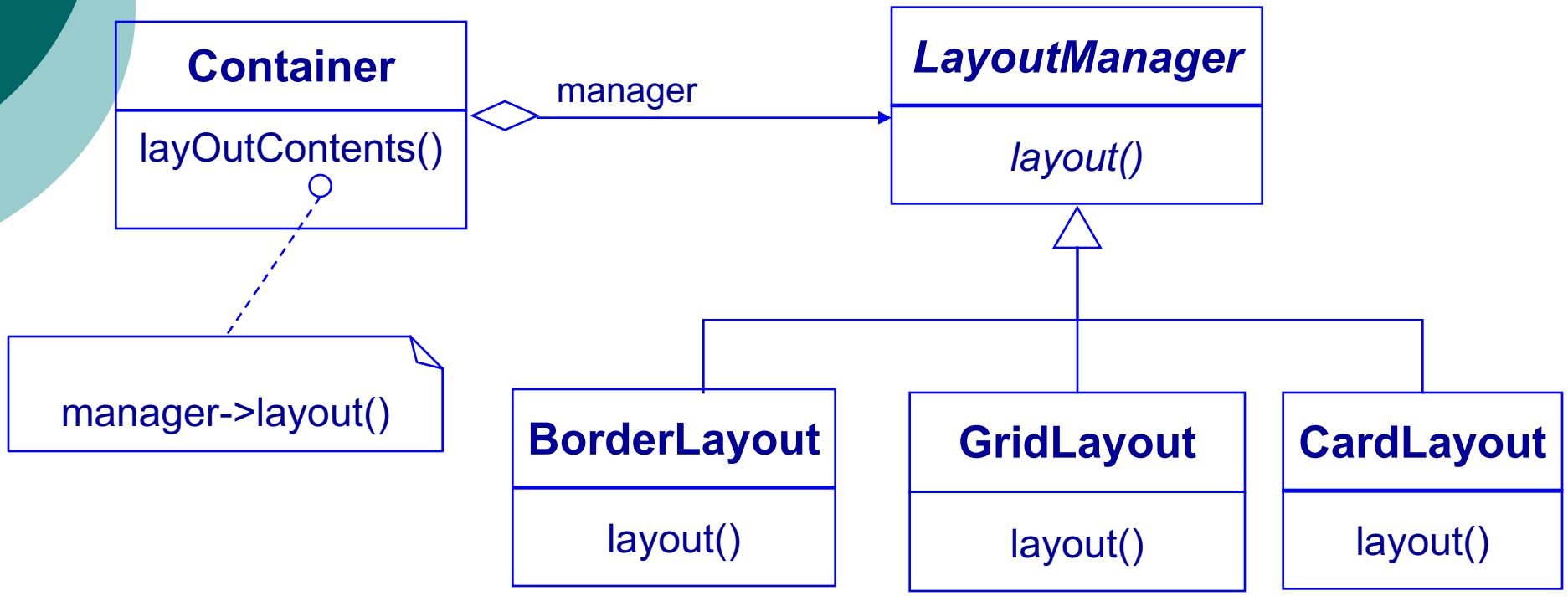
Strategy Pattern Structure



Example: Sorting a Database



Example: AWT Container Layout



Participants

- Context
  - Configured with a particular ConcreteStrategy to use
  - Keeps reference to (current) Strategy
  - May define interface for strategy to call back.
- Strategy
  - Common interface to supported algorithm(s)
  - Used by context to access the actual algorithm(s)
  - Abstract / virtual
- Concrete Strategy
  - Implements specific algorithm(s)

Applicability

- Many related classes differ only in specific reaction to requests.
- Need different variants of an algorithm.
  - Sorting?
  - Retrieval?
- Hide algorithm specific data structures
- To eliminate multiple conditional tests to select behavior in a client / context

Potential Benefits

- Encapsulates families of algorithms
- Alternative to subclassing *Context*
  - *Context* could be the top of a hierarchy.
  - Subclasses only change algorithm
  - Algorithm's implementation mingled with its use
  - Separate what varies from what is constant
- Eliminates often used conditionals and switches
- Context can choose alternatives
  - Choice is dynamic
  - Could choose different variants of the same algorithm

Potential Drawbacks

- Client must be aware of different strategies
  - At least some implementation issues exposed
  - Otherwise, how can client choose strategies?
- Communications overhead
  - Same interface for all Concrete Strategies whether simple or complex
  - Simple strategies may not use all provided information

The State Pattern

Motivation

- Problem:
  - An object responds differently based on its state
- Example: Fax transmitter
  - Basic operations:
    - boolean dial()
    - boolean getSpeed()
    - void send(Doc d)
    - Status hangUp()
  - Problems:
    - Not all operations legitimate at all times
    - Meaning of operation depends on current state

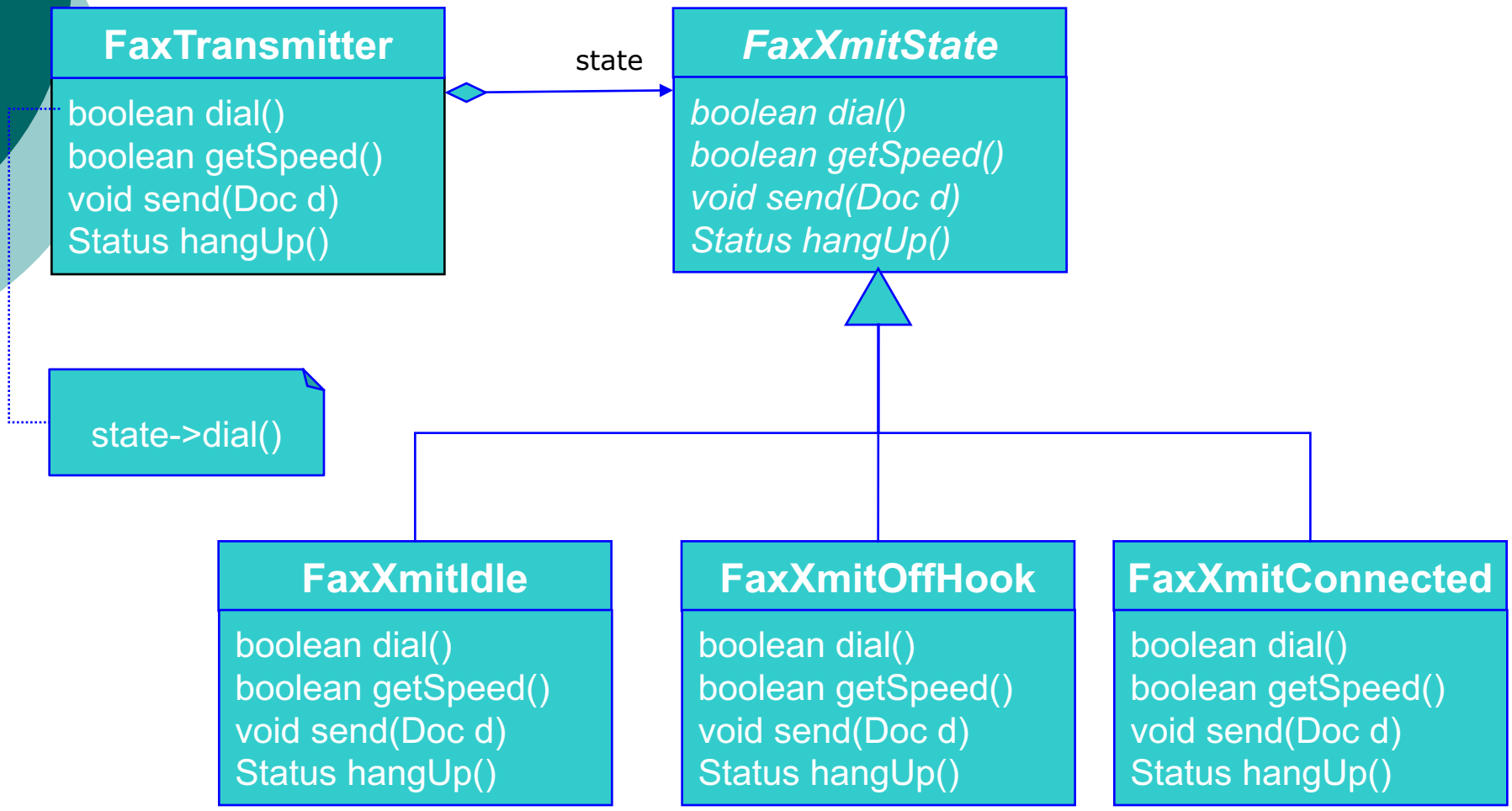
Naive Solution

- Create class FaxTransmitter
- Implement each operation directly
- Use variables to record state
- Use conditionals w/ state variables:
  - To decide whether request is legal
  - To decide on method of handling request
  - To decide whether to ignore request
- Issues
  - Complex, logic in methods
  - Hard to add new states (retry?)
  - Inflexible, hard-to-extend design

Better Solution

- Define basic interface abstractly
- Subclass for distinct response sets (state)
- Client's communicate with a *Context* object
- *Context* switches state object under the hood
- To clients, *Context* seems to change its class

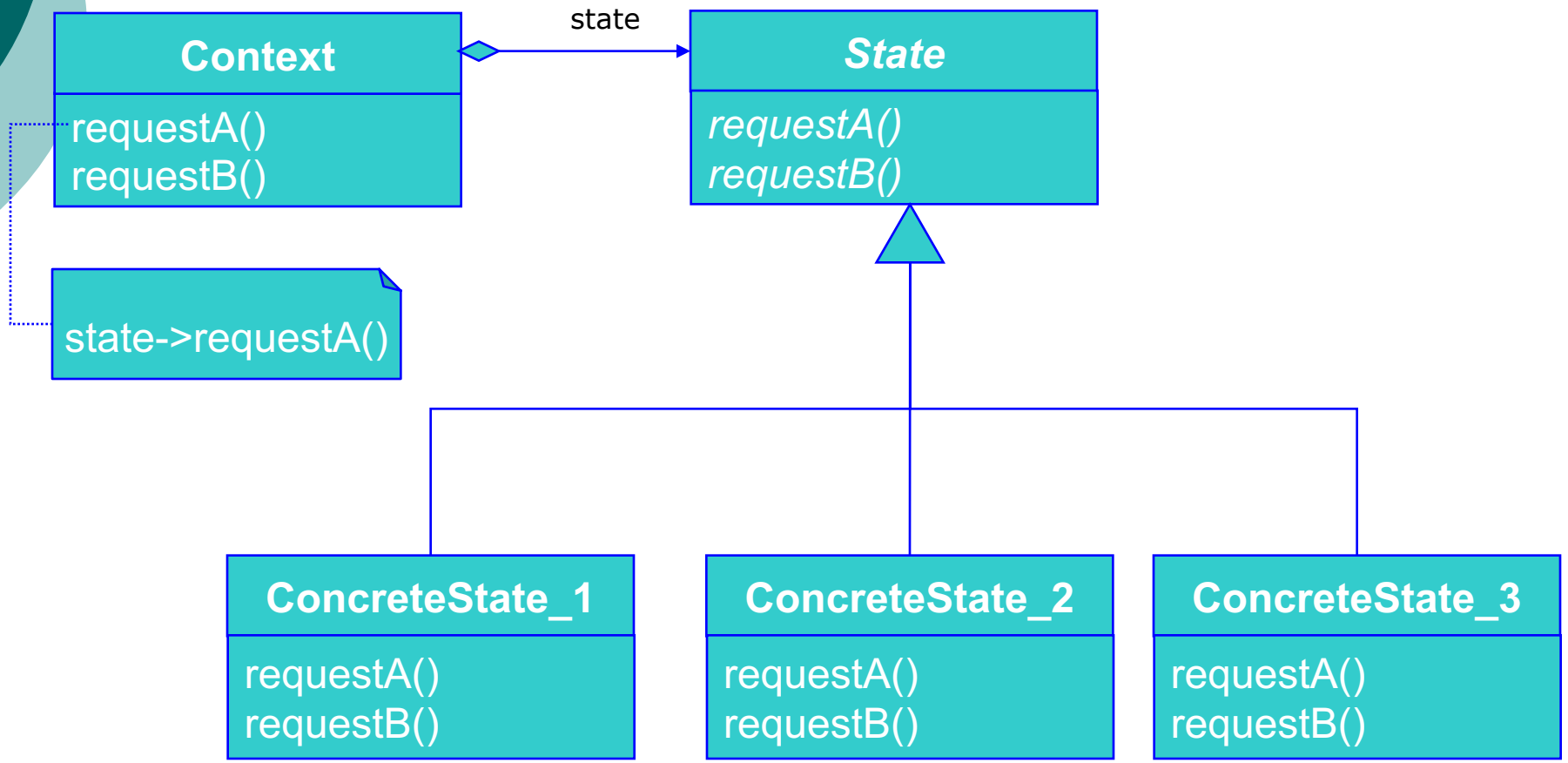
Example: FaxTransmitter



The State Pattern

- Context (FaxTransmitter) object defines interface
- Forwards request to a delegate (FaxXmitState) object representing current state
- Delegates have same / similar interface as Context
- Subclass Delegate for each concrete state
- State switch => delegate switch
- Clients see Context as having changed state

Pattern Structure



Participants

- Context (FaxTransmitter)
  - Defines interface of interest to clients
  - Keeps reference to current state
  - Forwards state-specific requests to current state object
- State (FaxXmitState)
  - Abstracts state specific behavior (methods)
- Concrete State (FaxXmitIdle)
  - Implements behavior for a specific state

Applicability

- Use when object's behavior depends on state that changes at run-time
- To factor out operations with multi-part conditionals based on object's state
  - State pattern puts each conditional in different concrete state class
  - Treats states as entities that can vary independent of context

Consequences

Localizes & partitions state-specific behavior

- Replaces large, complex conditionals with many state objects
- Eases addition of new states and transitions