Hugbúnaðarverkefni 1 / Software Project 1

13. Design Patterns

HBV501G – Fall 2018

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz Prep

- Please prepare a scrap of paper with the following information:

- ID: _____@hi.is   Date: _______
- a) ____________________________
- b) ____________________________
- c) ____________________________
- d) ____________________________
- e) ____________________________

- During class, I'll show you questions that you can answer very briefly
  - No elaboration necessary
- Hand in your scrap at the end of class
- All questions in a quiz weigh same
- All quizzes (ca. 10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

Kennslukönnun

Evaluate this course on Ugla!

(until 1 December)

Update: Assignment 5: Final Product – Schedule

- On **Thu 29 Nov**, demonstrate and explain your product in class
  - Please strive to be present for the whole time (08:30-11:30) that Thursday
    - so you can see other teams' work and learn from their experiences
    - so your classmates have an audience as well

- **Presentation rooms**
  - **Andri's teams: Árnagarður 304**
  - **Daníel's teams: Árnagarður 311**
  - **Matthias' teams: VR-II 152**

- By **Sun 2 Dec**, submit in Ugla:
  - The **slides** of the presentation you gave in class
  - The **source code** of your final product, including everything required to build and run it

Recap: Assignment 5: Final Product – Presentation

- **On Thu 29 Nov**, each team presents their product to their tutor and his other teams
  - 20 minutes per presentation (~5 minutes for each of the following parts, 5 minutes for questions)
- Your presentation must cover the following parts (in an order of your choice):
  - **A live demonstration of your final software**
    - Should include product’s key use cases
  - **An overview of your system architecture**
    - What are the key components, and how do they communicate?
    - What aspects are client and server responsible for?
    - How are you storing and accessing data?
    - Any particular aspects of your design you would like to highlight?
  - **A retrospective on your project work**
    - What went well? What difficulties did you encounter?
    - How did you plan to structure / manage your work? How did that turn out?
    - What would you do differently next time?
    - How would you avoid any difficulties you encountered?
- You can choose which teammates give which parts of the presentation.

Quiz #10 Solution: Encapsulation and Coupling

How (1-4) should shared behaviors be implemented to ensure low coupling of classes in these scenarios (a-d)?

a) Several classes share a behavior, but some perform it in different ways than others.
   - **3.** The most common form of the behavior is implemented in the superclass and inherited by all subclasses, which can override it with individual implementations if desired.

b) Several classes share a behavior, but they all perform it in different ways.
   - **4.** An abstract method in a superclass ensures that all classes will provide the behavior, but its form is implemented individually in each subclass.

c) Among several classes, some share the same behavior, but some do not (and should not look as if they did).
   - **1.** The behavior is added to those classes who should perform it by composition and delegation.

d) Several classes share a behavior, and they all perform it in the same way.
   - **2.** The behavior is implemented in a superclass, and inherited by all subclasses.

Design Patterns

(contd.)

see also:

- Larman: Applying UML and Patterns, Ch. 26
- Freeman, Robson: Head First Design Patterns

Recap: The Big Picture
Encapsulation of Change as a Driver of Good Design

One of the most fundamental modular / object-oriented design principles:

Encapsulate what varies.

- If you can foresee that something will change…
  - an algorithm, a data structure, a technical component, a business process, etc.
- …encapsulate that part of your system
  - so other parts of the system will remain unaffected by any internal changes.
  - Various encapsulation techniques exist on different levels:
    - **Programming language:** Methods, classes, visibilities
    - **Class structure:** Abstract supertypes (i.e. `abstract` classes or `interfaces`), polymorphism
    - **Class collaboration:** Object-oriented design patterns
    - **System architecture:** Component-based design, layered architectures, communication protocols

Recap: The Big Picture
Encapsulation of Change as a Driver of Good Design

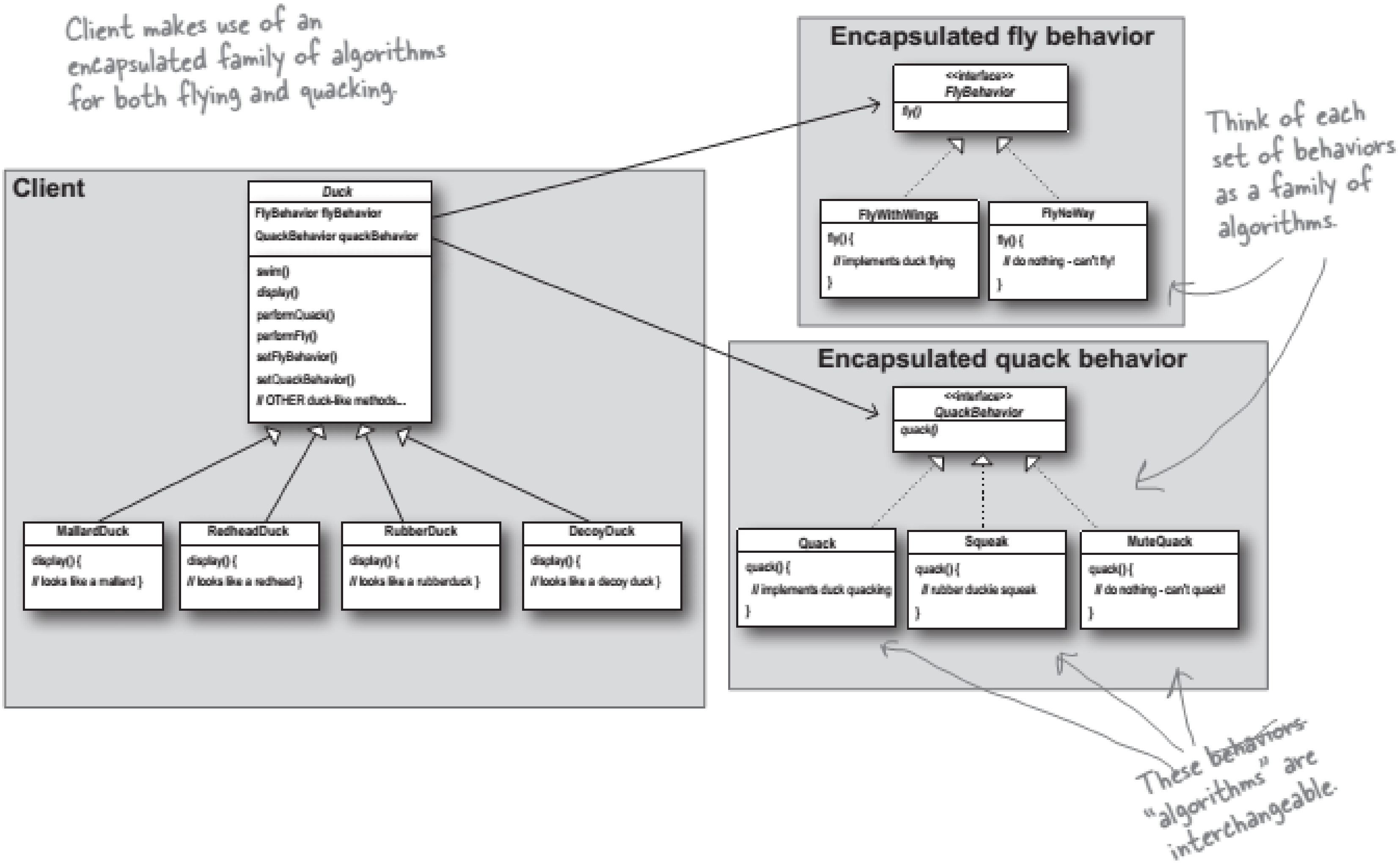
Two important corollaries to the encapsulation principle:

- **As a module user: Program to an interface, not an implementation.**
  - When you are working with something encapsulated, treat it as a black box and make no assumptions about how it works inside.
    - i.e. do not rely on knowledge/assumptions about things like data structures, sorting orders, side effects, thread-safety etc., as they may change (unless they are explicitly specified)
- **As a module provider: You can change an implementation, but never an interface.**
  - Changing an interface will break any outside code relying on it.
    - Changes include adding abstract methods to supertypes (i.e. classes or Java `interfaces`), changing method signatures, and even changing a method's documentation if it promises certain properties such as sorting order, thread-safety etc.

Recap: Strategy Pattern

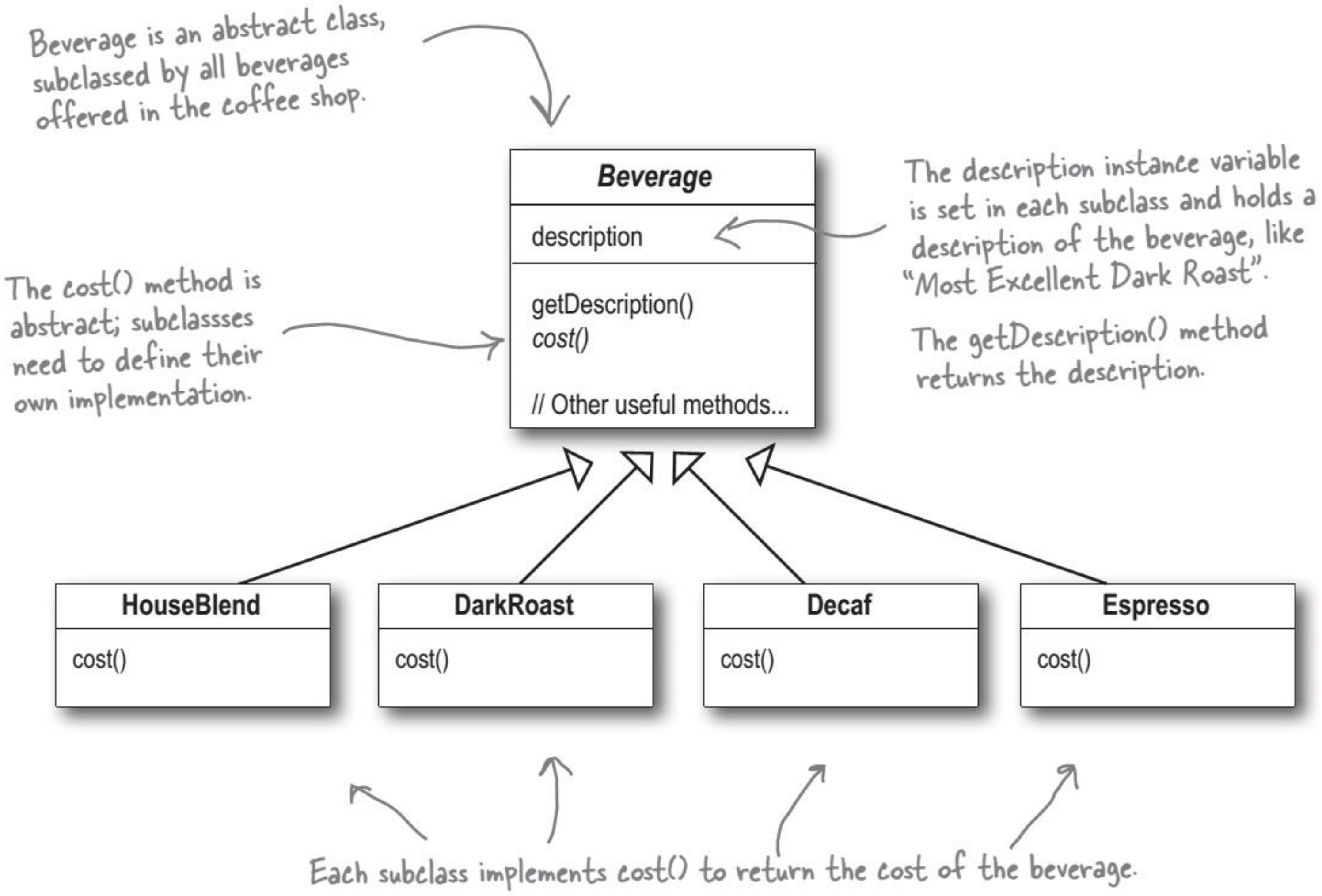
- The **Strategy pattern**
  - defines a family of algorithms,
  - encapsulates each one,
  - and makes them interchangeable.

- The Strategy pattern lets the algorithm vary independently from clients that use it.



Decorator Pattern
Motivation: Typical Inheritance Solution

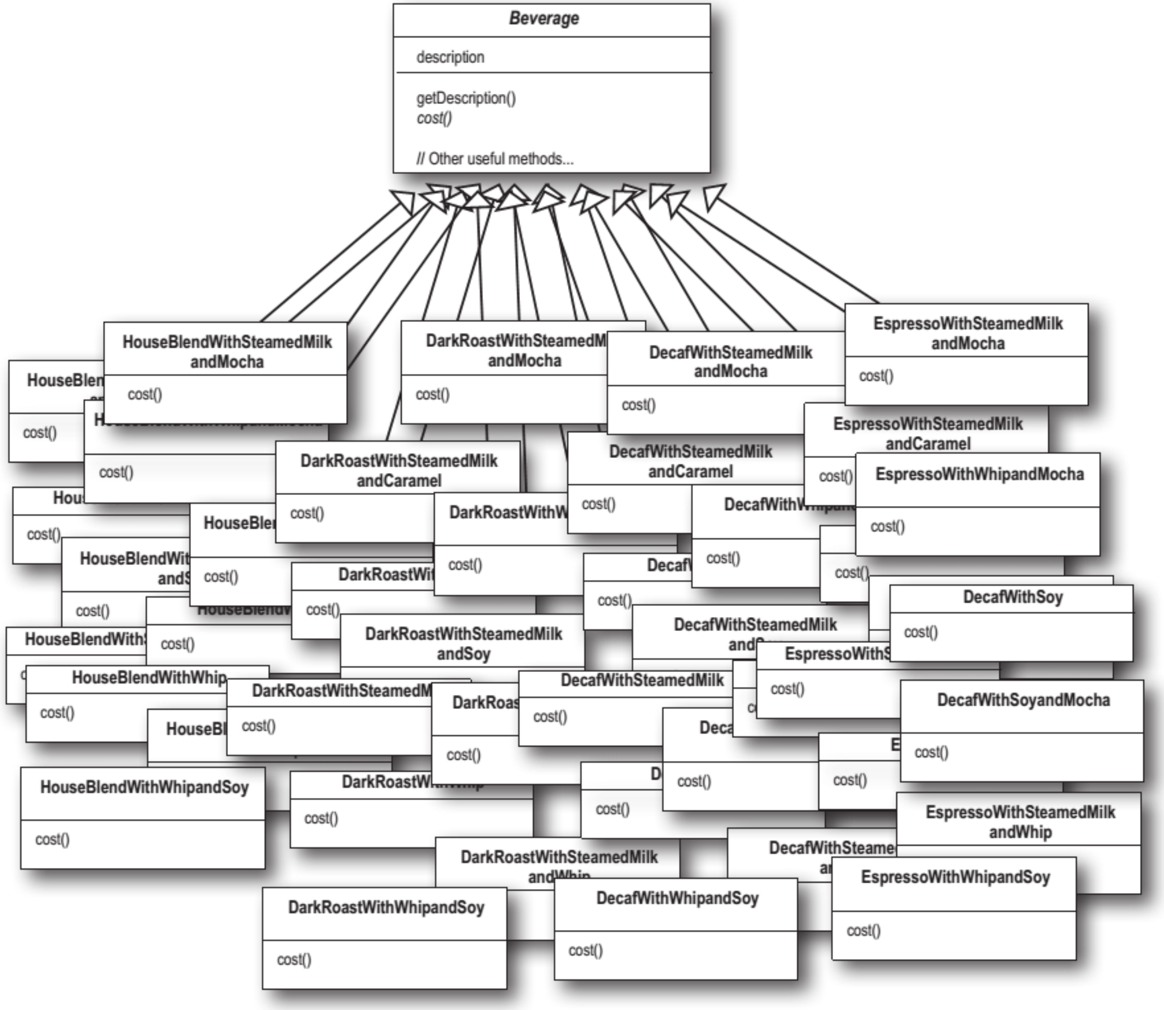
- We'd like to model all the different beverages sold in a cafeteria.
- For the four main beverages, this is easy:



Decorator Pattern
Motivation: Dealing With Variety Through Inheritance

- Adding all the different condiments (whip cream, mocha, caramel…) makes things difficult.

- Naïve solution: Just adding subclasses for each variation would lead to a maintenance nightmare:



Decorator Pattern
Motivation: Dealing With Variety Through Configuration

- A better way might be to use subclasses only for our main beverages, and make the condiments configurable in the superclass.
- However, now Beverage.cost() needs to implement quite complex price calculations, and must be adapted every time we change prices, introduce new condiments etc.

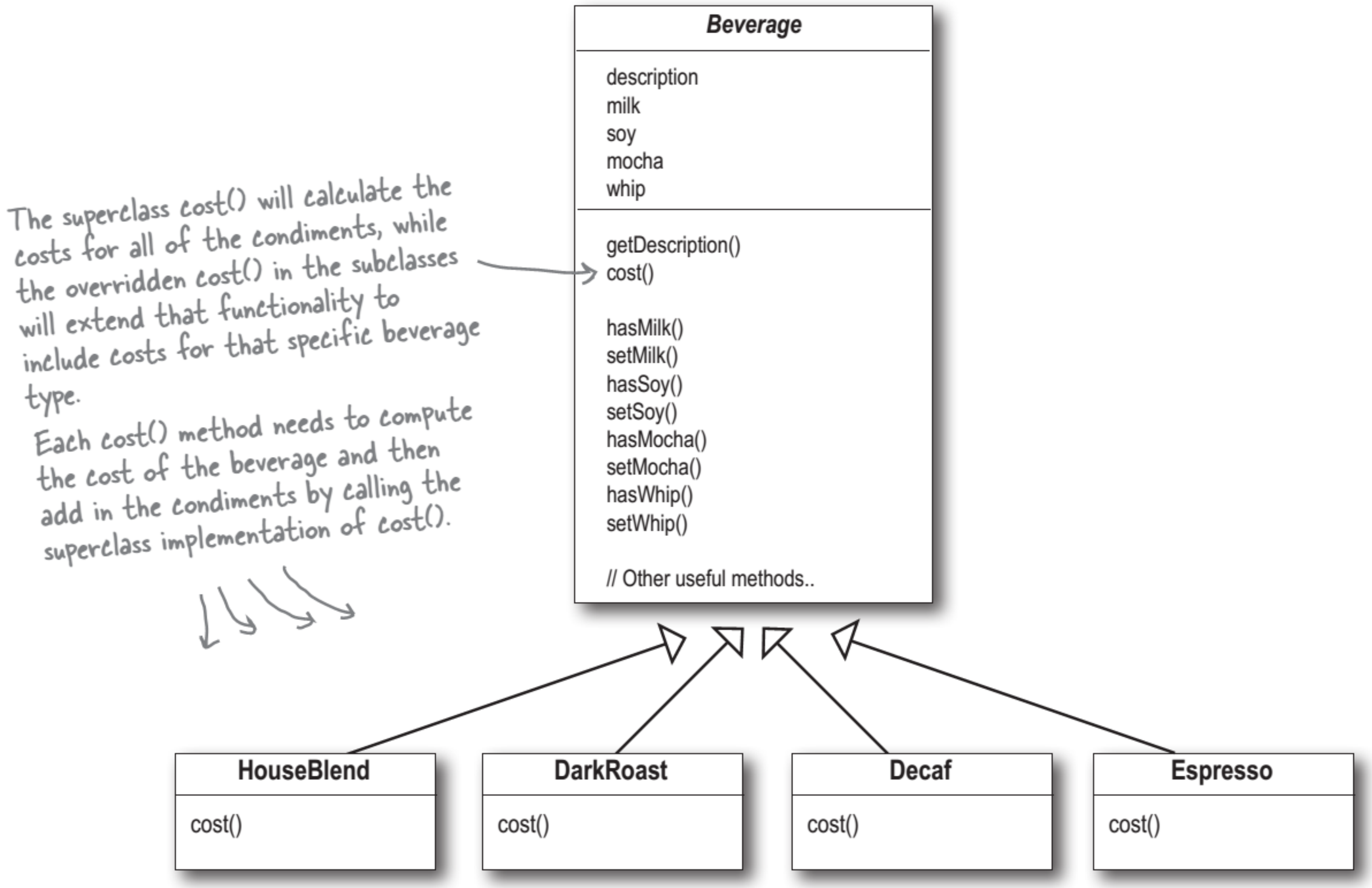
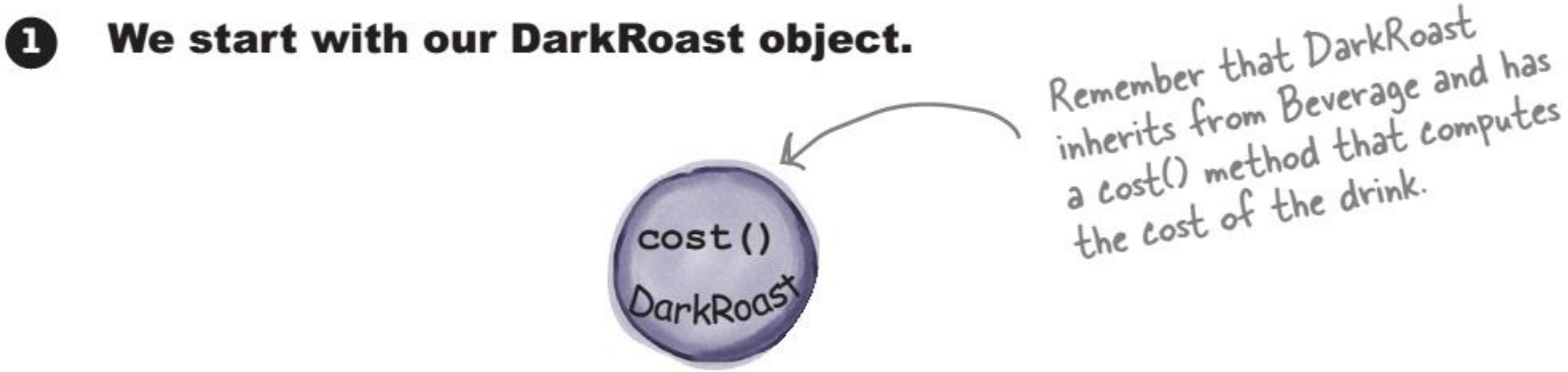


Figure: Freeman/Robson, Head First Design Patterns

Decorator Pattern
Solution: Extending an Object Without Modifying It



Solution: Extending an Object Without Modifying It

2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

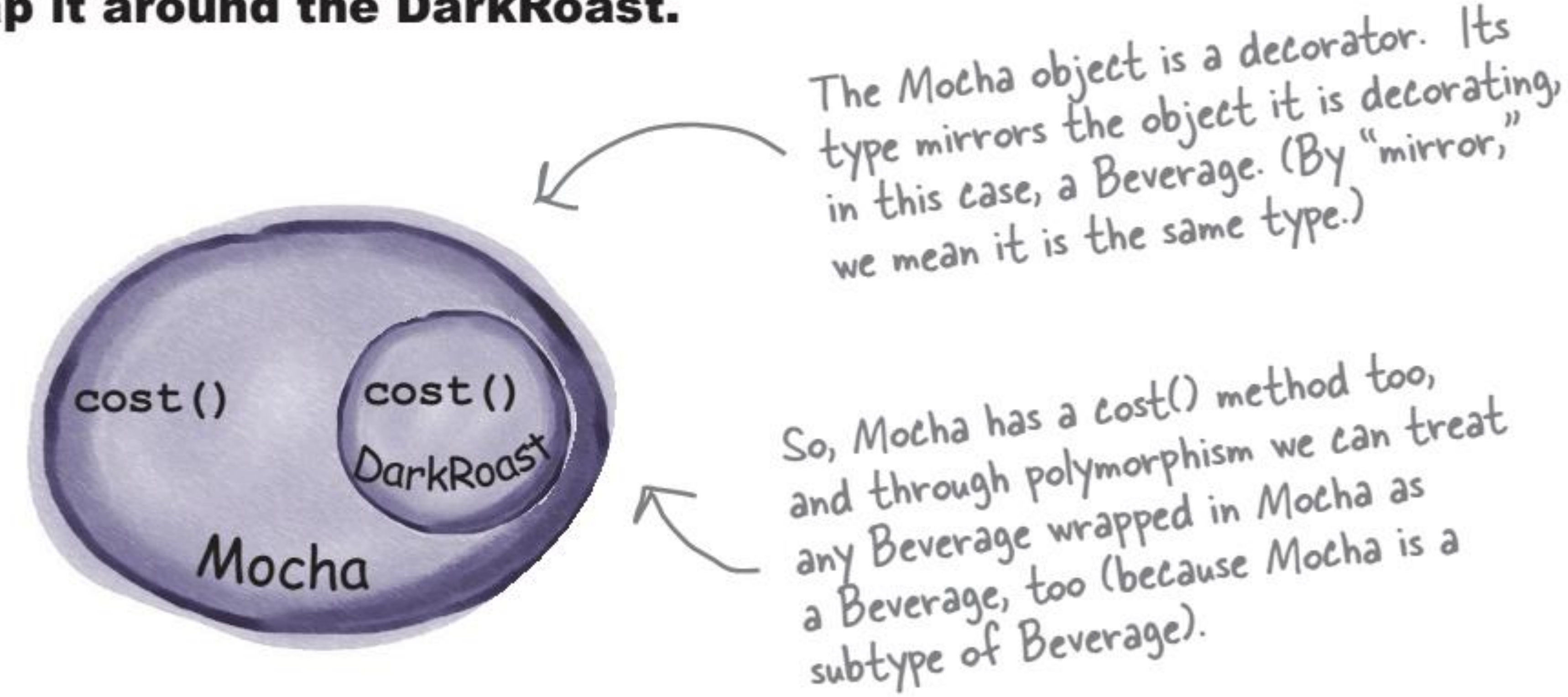


Figure: Freeman/Robson, Head First Design Patterns

Solution: Extending an Object Without Modifying It

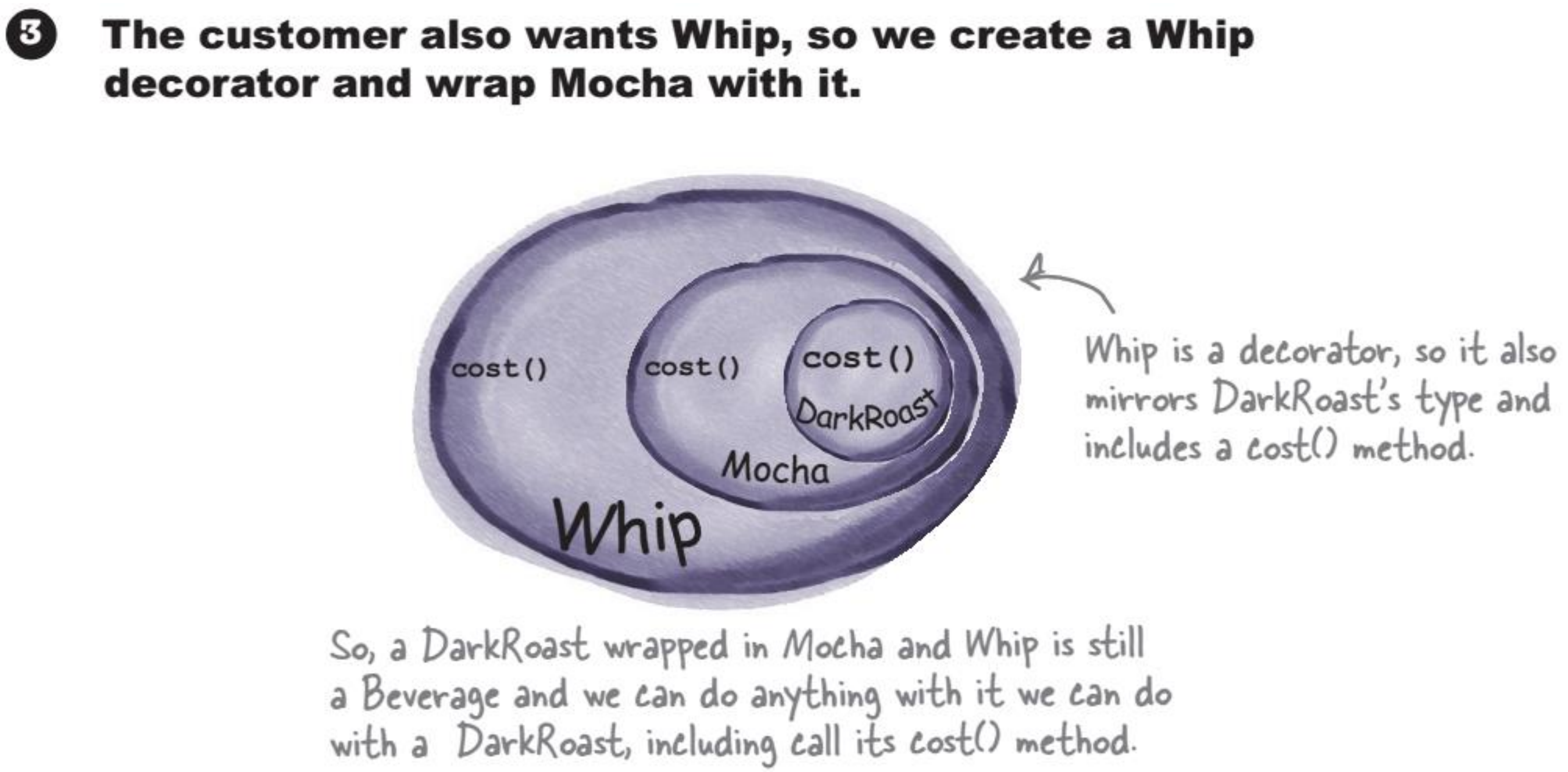
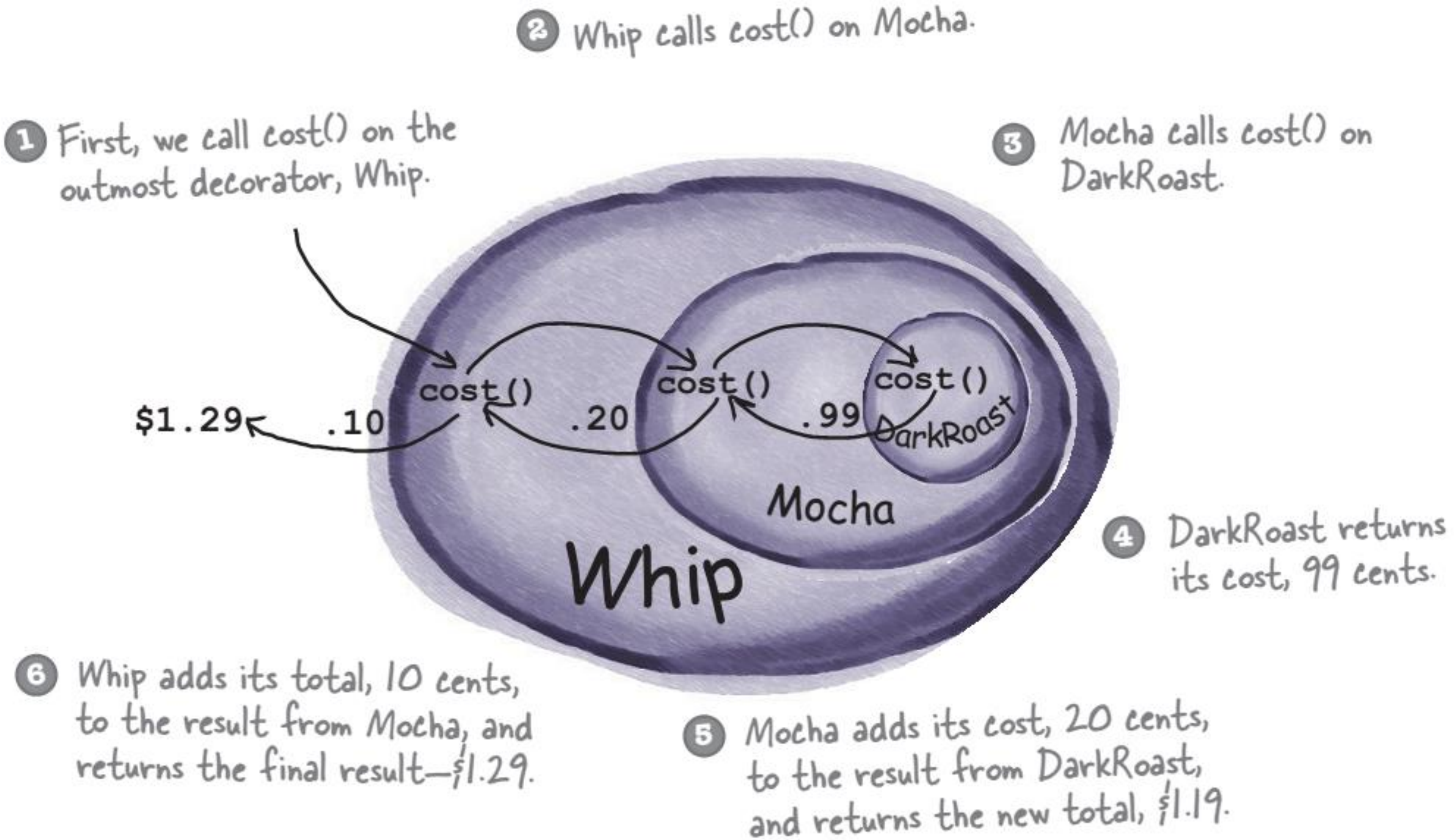


Figure: Freeman/Robson, Head First Design Patterns

Decorator Pattern
Solution: Extending an Object Without Modifying It



Decorator Pattern
General Form

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

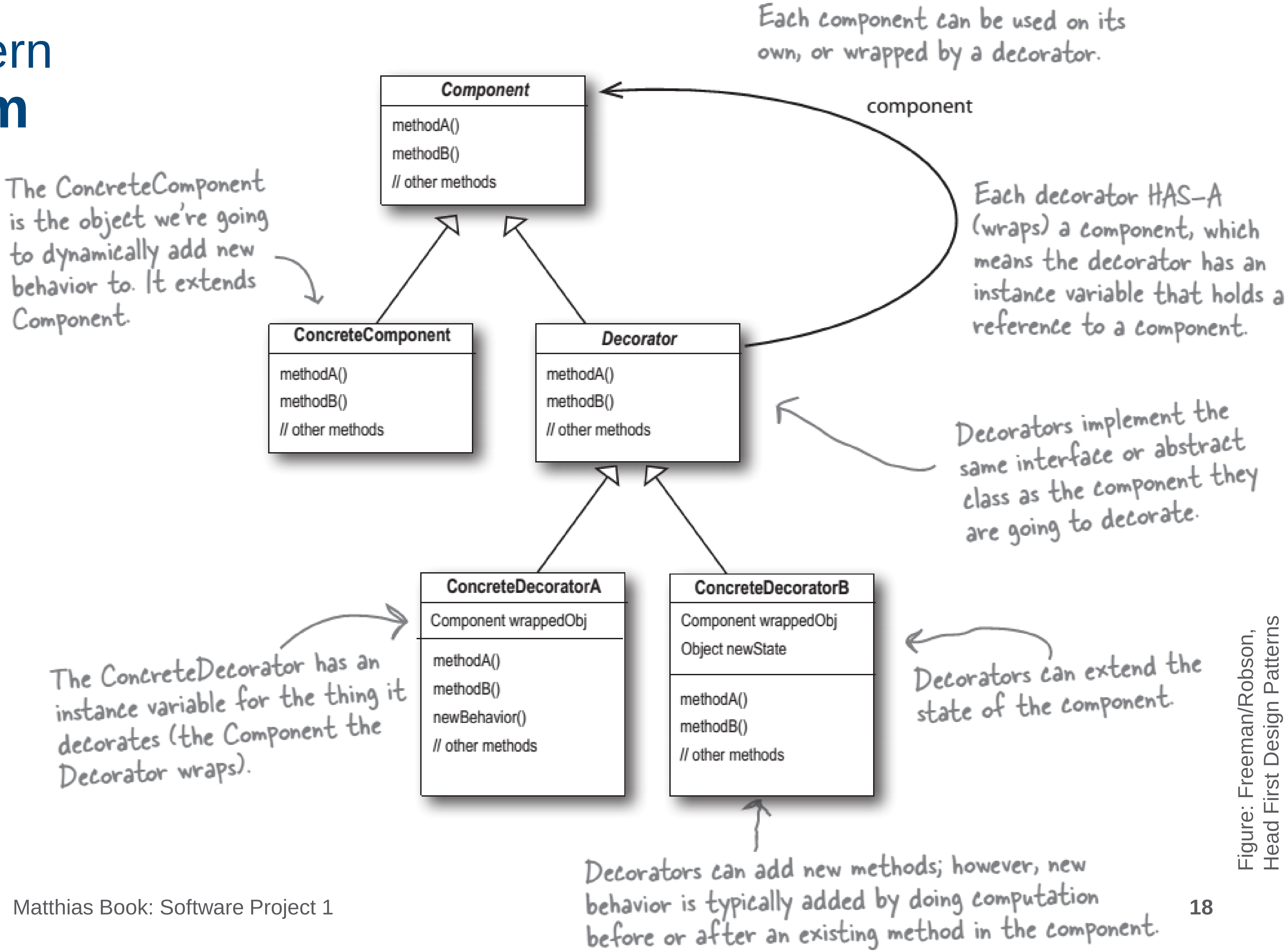


Figure: Freeman/Robson, Head First Design Patterns

Decorator **Example**

- What did we just do?

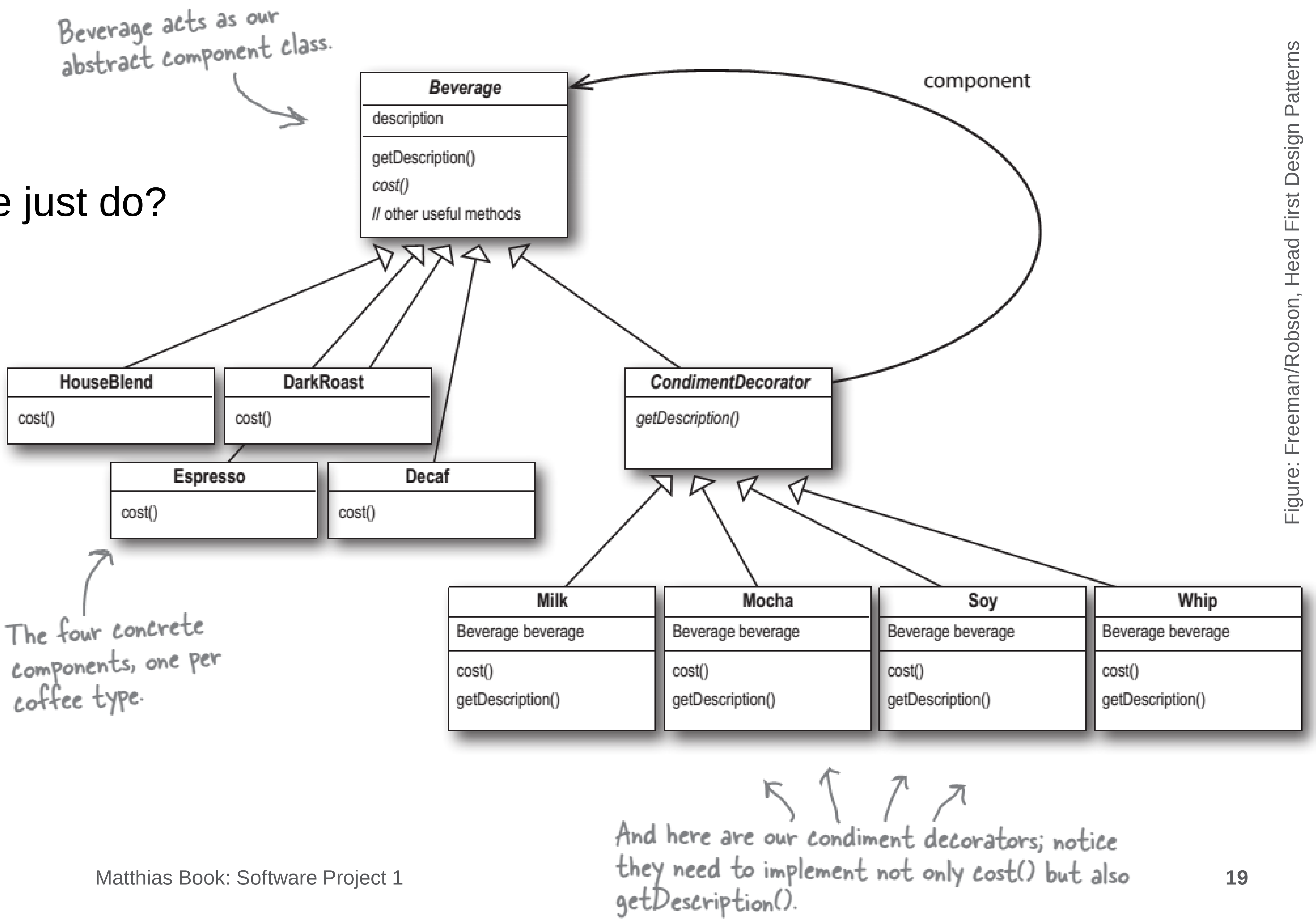


Figure: Freeman/Robson, Head First Design Patterns

Decorator Pattern
Usage in Code

```
public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription()
            + " $" + beverage.cost());

    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
}
```

Order up an espresso, no condiments, and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

File Edit Window Help CloudsInMyCoffee

```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Decorator Pattern
Another Example: Streams in Java

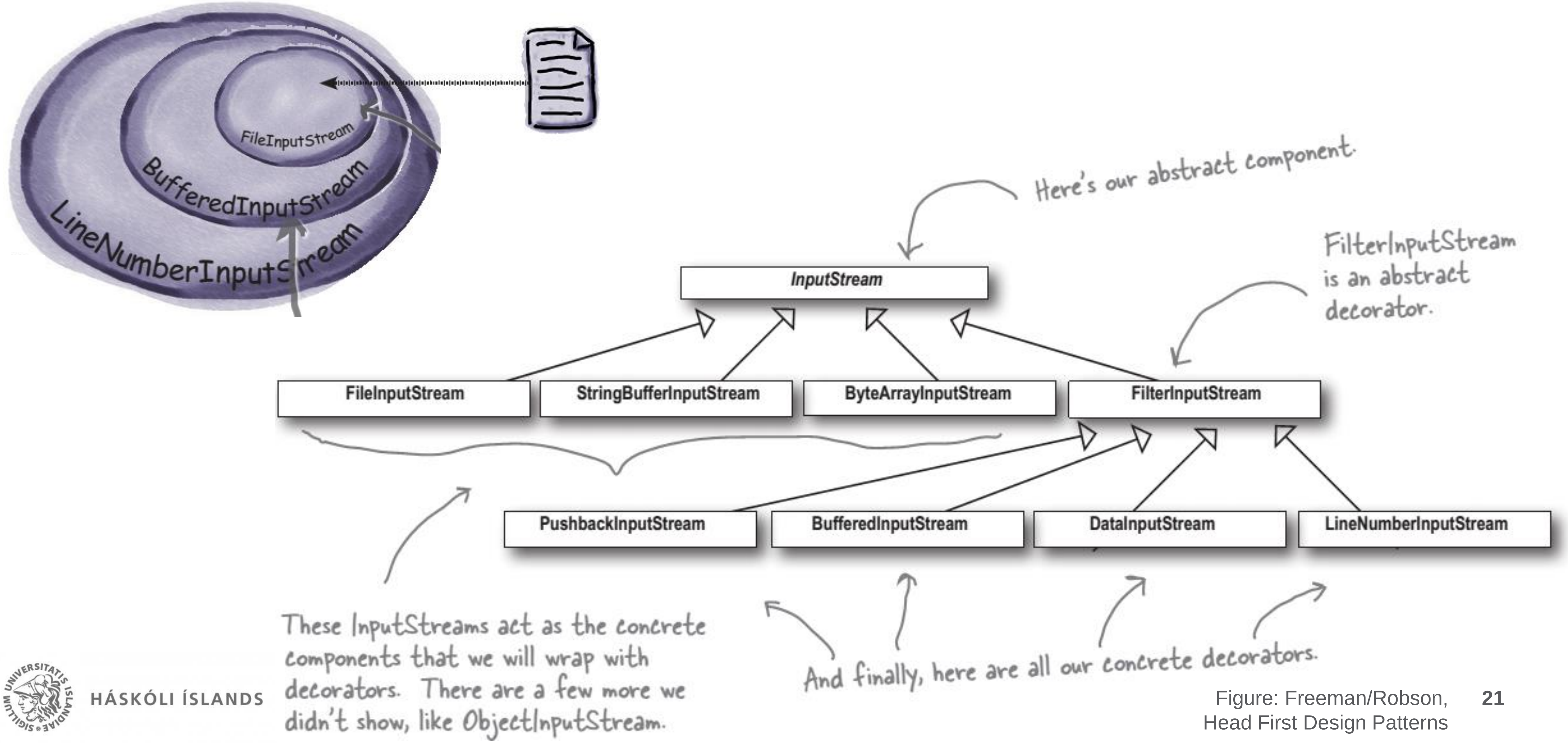


Figure: Freeman/Robson, Head First Design Patterns

Decorator Pattern
Benefits

- A decorator looks like a core object and can therefore “wrap around” it without other classes noticing that it’s there.

- This allows us to
  - **Extend** an object’s behavior
  - **Without changing** the object’s original implementation
  - And **without influencing** other classes who expected to work with the core object!

➢ A possible way to satisfy the Open-Closed Principle *(→ HBV401G, Ch. 10)*

- **Usage recommendations**
  - In many cases, simple inheritance or composition/delegation will be more straightforward
  - For more complex structures with many different variations or ongoing evolution, using the Decorator pattern can be worthwhile

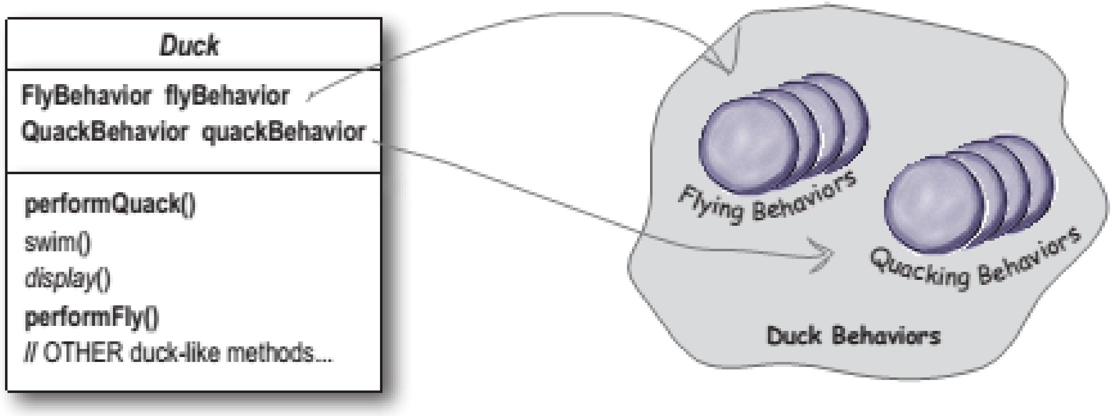
The Big Picture
Strategy or Decorator Pattern?

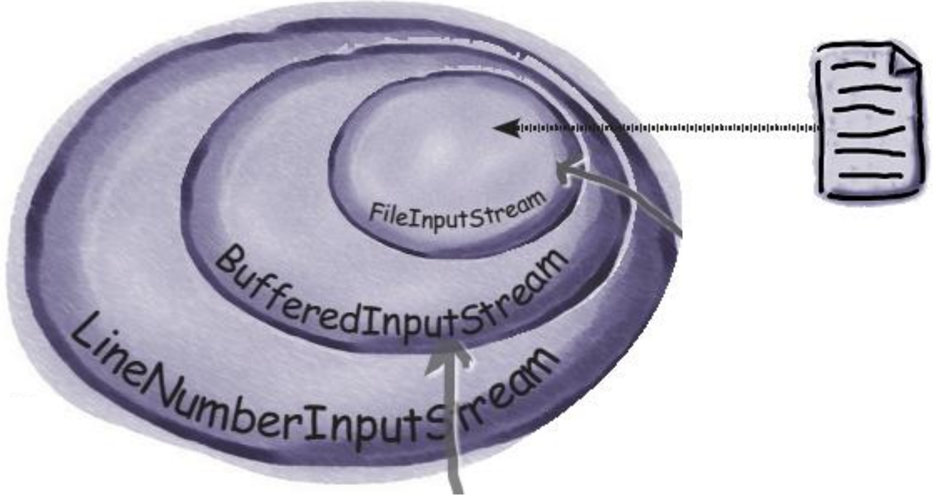
Strategy Pattern

- Useful when a class knows that it'll have to implement certain capabilities
- but wants to remain unaware of how those capabilities will be implemented

Decorator Pattern

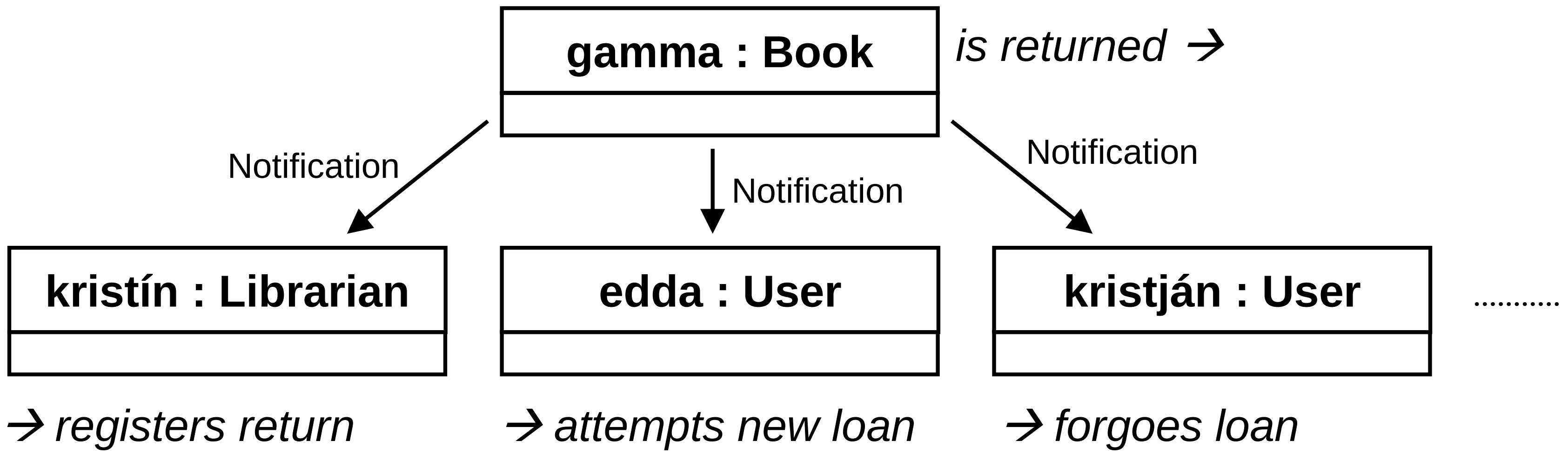
- Useful when a class should not need to be aware that its behavior will be extended in an unknown variety of ways by other classes





Observer Pattern
Problem: Distribution of Events

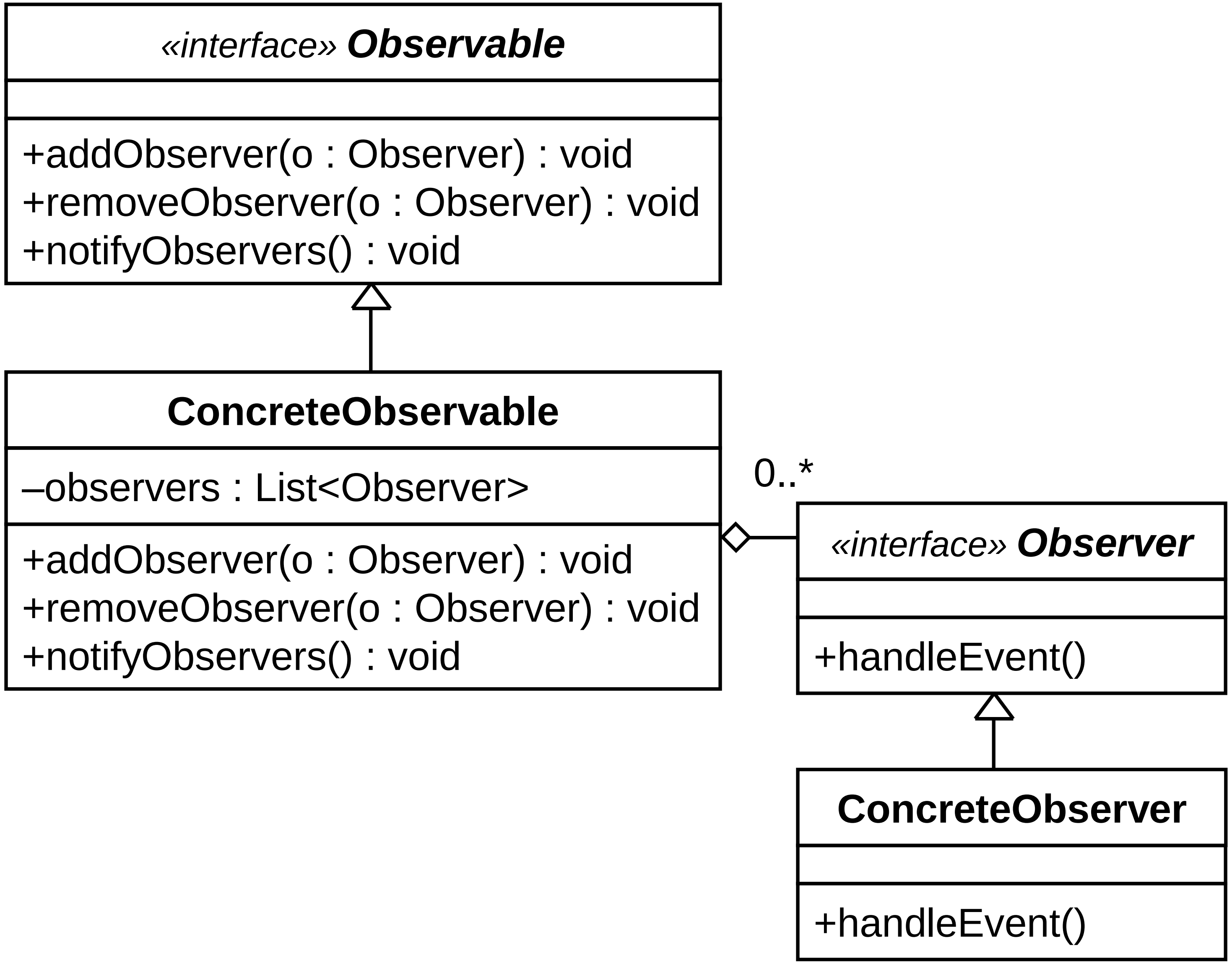
- Design requirement: A number of different objects (unknown at design time) shall react (each in their own way) to a certain event.
  - e.g. notification of librarian and all library users holding reservations when a book is returned



- Challenges:
  - Book needs to know who is interested in its return
  - Book needs to pass notification to different interested classes
  - Interested classes react differently to the notification

Observer Pattern
Solution

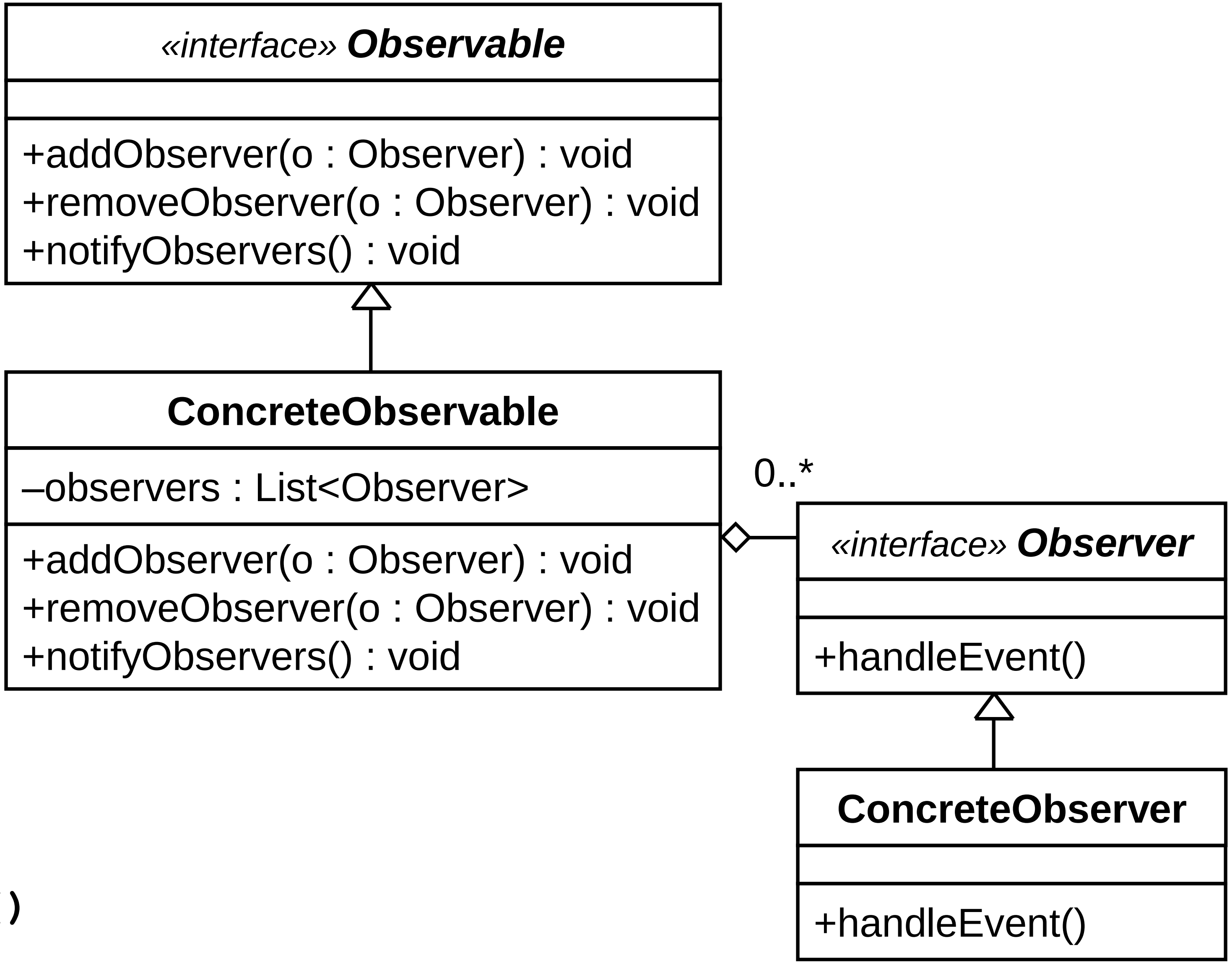
- also known as "Publisher-Subscriber"
  - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically




```
public void notifyObservers() {
  for (Observer o: observers)
    o.handleEvent();
}
```

Observer Pattern
Usage

- Register Observer with Observable:
  - Observer can do this itself…
    ```
    Observable pub = ...;
    pub.addObserver(this);
    ```
  - …or another class can:
    ```
    Observable pub = ...;
    Observer sub = ...;
    pub.addObserver(sub);
    ```
- Notify Observer upon event:
  - Observable can do this itself…
    ```
    notifyObservers();
    ```
  - …or another class can:
    ```
    Observable pub = ...;
    pub.notifyObservers();
    ```
- Let Observer react to message:
  - by implementing `handleEvent()`



Command Pattern
Motivation

- Imagine we are building a **drawing tool** that lets the user draw and manipulate shapes in many ways
  - Drawing, moving, scaling, rotating, skewing, flipping, filling, smoothing, deleting… any shape
- The number of operations our software offers is likely to evolve over time
- We want to give the user the ability to undo as many actions as desired, i.e. to revert manipulated shapes into previous states

- Thoughts on naïve solutions:
  - Hardwiring all these operations into the software would be tedious but possible
  - Undoing an arbitrary amount of shape changes would require storing a lot of information about previous states of the drawing

Command Pattern
Solution

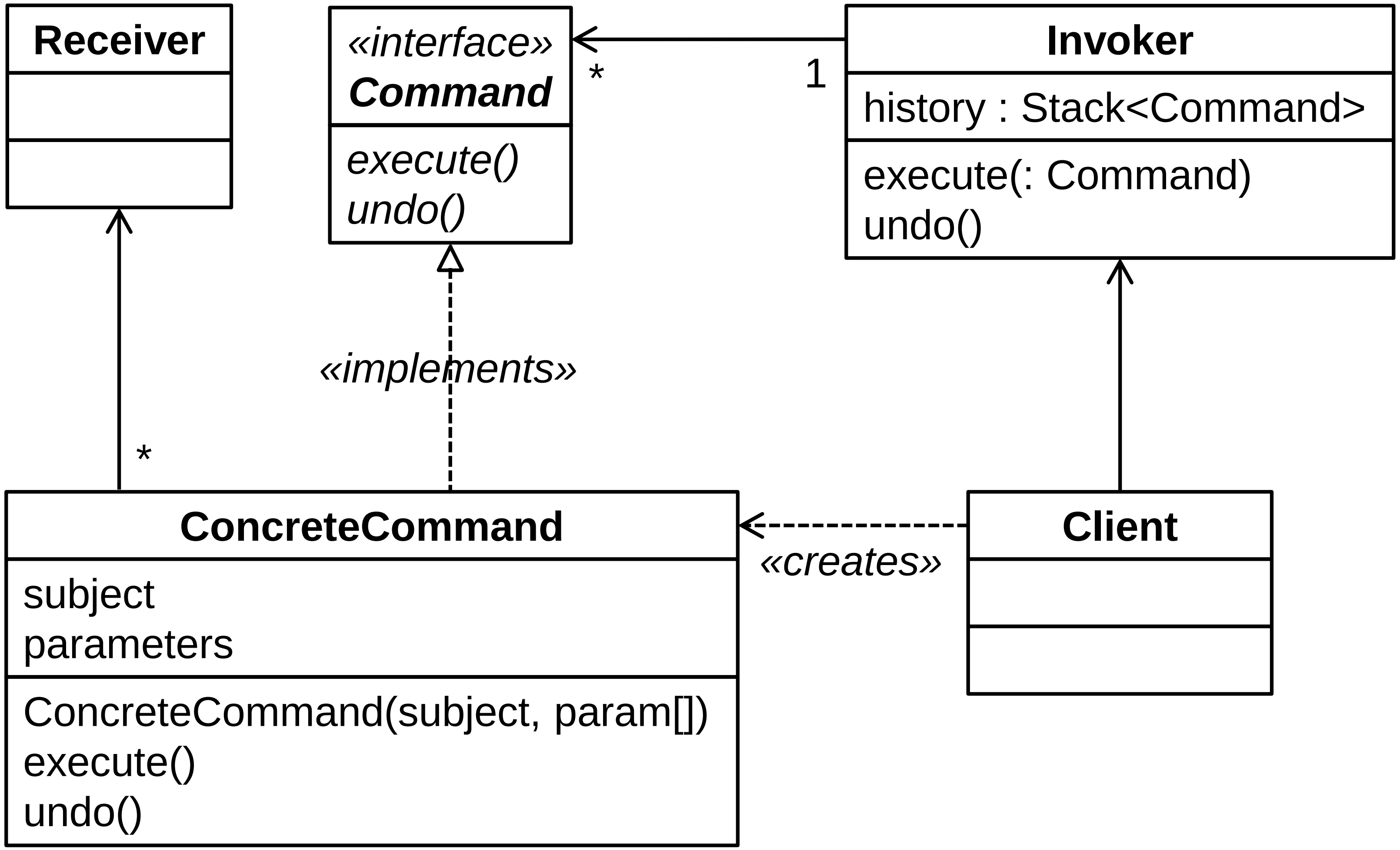
- Turn the individual drawing commands into objects!

Benefits:

- Commands become conveniently configurable
  - We can even build commands out of commands (macros)
- The set of commands can be extended without changing the command invocation logic
  - e.g. when new features are added to our software
- We can treat commands as data
  - e.g. to keep a history of executed commands on a stack
- We can not just specify how to execute, but also how to revert each command
  - Allows us to implement “undo” functionality quite elegantly

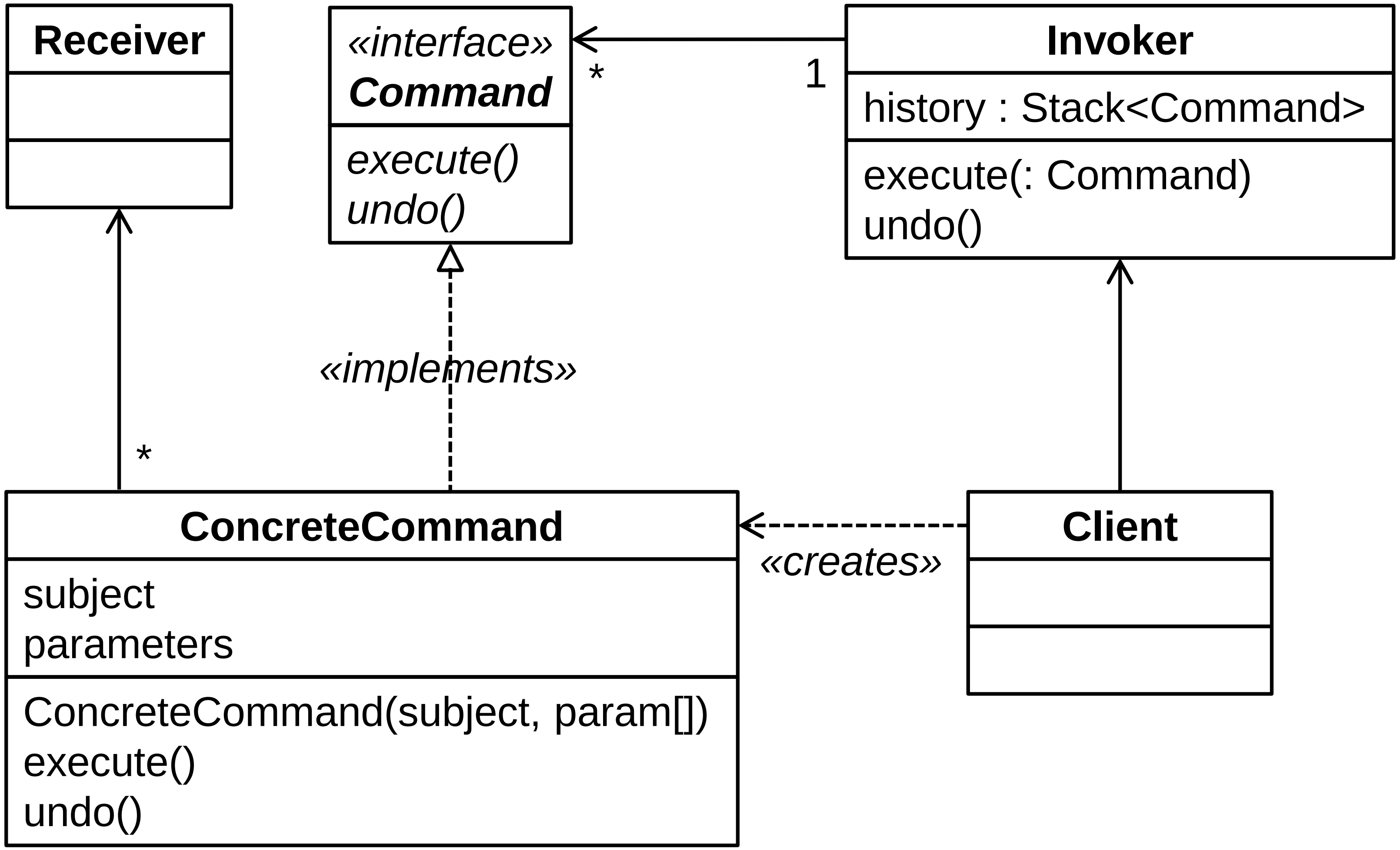
Command Pattern
Generic Example: Command Execution and Reversion

- ConcreteCommands
  - Know which Receiver(s) is/are able to perform an operation
    - e.g.: a Rotator can rotate a shape
  - Expect a *subject* to work on…
    - e.g.: the shape
  - …and *parameters* describing the operation
    - e.g.: degrees of rotation



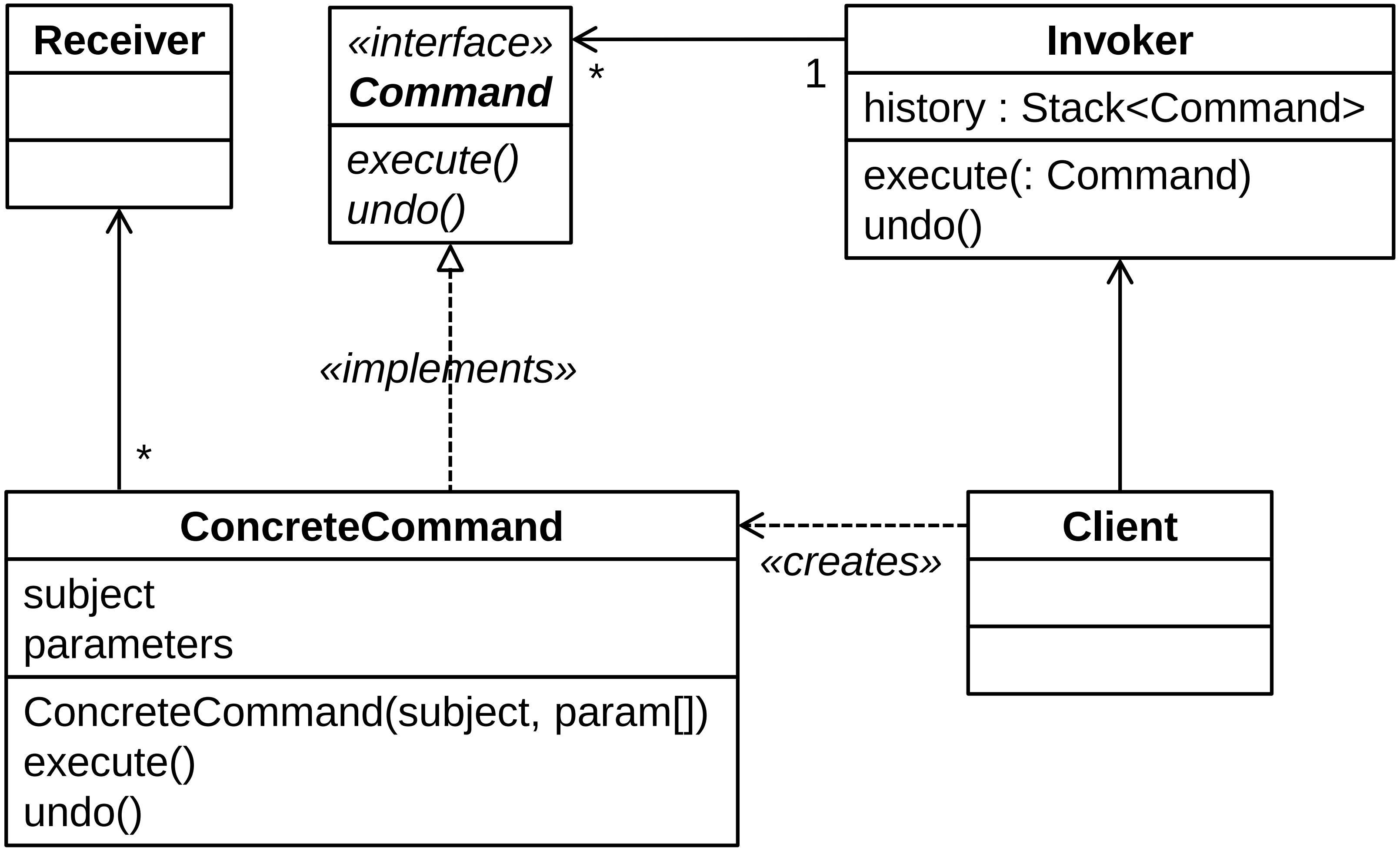
Command Pattern
Generic Example: Command Execution and Reversion

- To execute a command
  - Client instantiates and configures ConcreteCommand
  - Client asks Invoker to execute Command
  - Invoker calls Command's execute() method
  - Command calls the actual implementation of operation in Receiver
  - Invoker pushes Command on stack

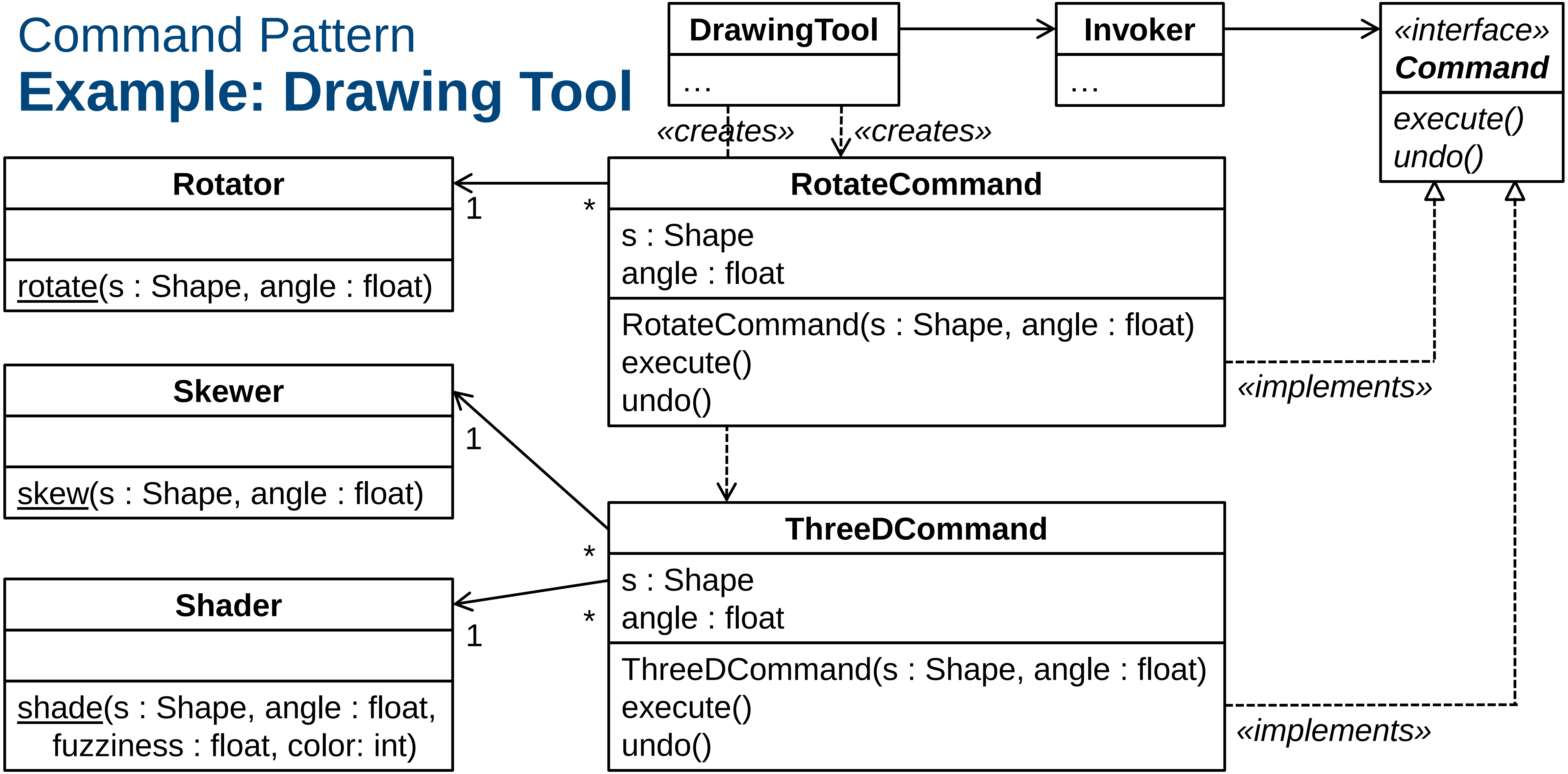


Command Pattern
Generic Example: Command Execution and Reversion

- To undo a command
  - Client invokes Invoker.undo()
  - Invoker pops top Command off stack and invokes its undo() method
  - Undo method uses stored subject and parameter information to construct a call to Receiver with instructions that will negate previous operation’s effect



Command Pattern
Example: Drawing Tool



Command Pattern
Example: Command Implementations

```
public class RotateCommand
    implements Command {
  private Shape s; private float angle;
  public RotateCommand(
      Shape s, float angle) {
    this.s = s; this.angle = angle;
  }
  public void execute() {
    Rotator.rotate(s, angle);
  }
  public void undo() {
    Rotator.rotate(s, -angle);
  }
}
```

```
public class ThreeDCommand
    implements Command {
  private Shape s; private float angle;
  public ThreeDCommand(
    Shape s, float angle) {...}
  public void execute() {
    Skewer.skew(s, angle);
    Shader.shade(s, 45, 10, 127);
  }
  public void undo() {
    Shader.clear(s);
    Skewer.skew(s, -angle);
  }
}
```

Example: DrawingTool and Invoker Implementations

```java
public class DrawingTool {
  private Invoker invoker = new Invoker();
  public testDriver() {
    Shape s = new Shape(...);
    // ...draw shape...
    invoker.execute(
      new RotateCommand(s, 90));
    invoker.execute(
      new ThreeDCommand(s, 20));
    // ...draw shape...
    invoker.undo();
    invoker.undo();
    // ...draw shape...
}
```

```java
import java.util.Stack;
public class Invoker {
  private Stack<Command> history
    = new Stack<Command>();
  public execute(Command c) {
    c.execute();
    history.push(c);
  }
  public void undo() {
    Command c = history.pop();
    c.undo();
  }
}
```

Command Pattern
Benefits

- Allows our software to not just perform operations, but be aware of what it’s doing, and e.g.
  - Undo operations
  - Facilitate end-user programming
    - Recording and bundling operations in macros
  - Choose between different appropriate operations

- **Usage recommendations**
  - In most cases, simple method calls will be more straightforward
  - Don’t be tempted to build your own command language inside the programming language unless you have good reasons to do so (e.g. need for an undo feature)

In-Class Quiz #11: Design Patterns

- **What is the purpose (1-5) of the following design patterns (a-e)?**

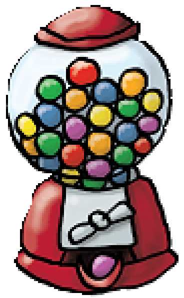
a) Command
b) Decorator
c) Observer
d) Singleton
e) Strategy

1. Ensure a class only has one instance.
2. Define a family of algorithms, encapsulate each one, and make them interchangeable in objects that use them.
3. Attach additional responsibilities to an object dynamically.
4. Notify all dependents of an object when its state changes.
5. Decouple an object making requests from objects that know how to perform those requests.

State Pattern
Background: State in Software Systems

- State can be found on all levels of software systems, e.g.
  - the state of individual objects
  - the state of the whole system
  - the state of a particular business process
  - etc.
- Some states are easily represented in simple data structures
  - e.g. the score of a player in a game
- Some states permeate the system without being explicitly “stored” anywhere
  - e.g. the view that a user has most recently navigated to
- Some states are explicitly stored and influence a system’s or object’s behavior
  - e.g. whether a user is currently logged into a system, and which rights that user has
  - e.g. how a system is reacting to outside events under different conditions
  - e.g. which capabilities/behaviors certain entities exhibit in a game

Motivation: Example



- Modeling the state transitions of a gumball machine

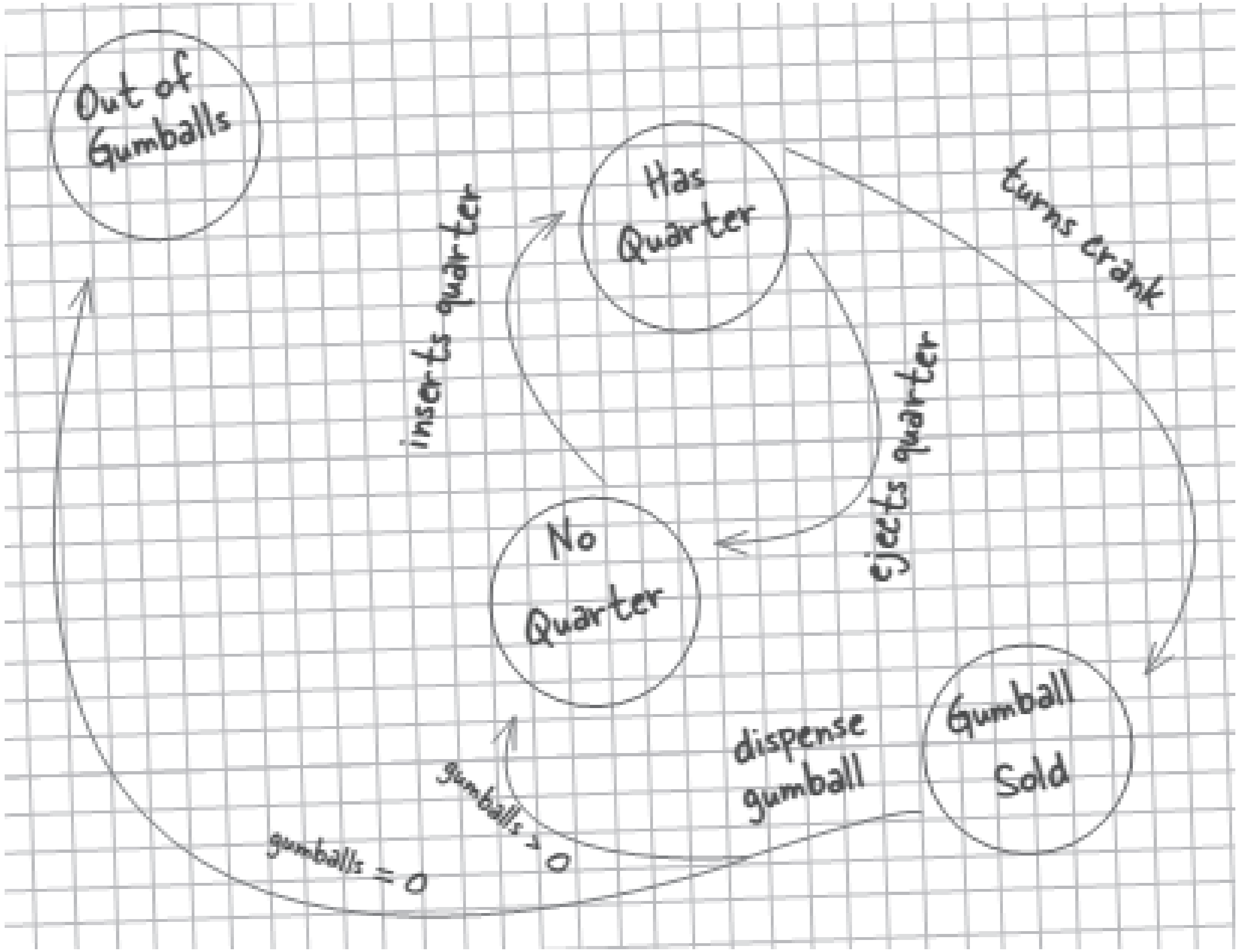


Figure: Freeman/Robson, Head First Design Patterns

State Pattern
Motivation: Impl.

Simple approach:

- Represent the states as integer values
- Represent the events as methods
- Implement actions and decide on subsequent states conditionally…

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

Now we start implementing the actions as methods....

When a quarter is inserted, if....

...a quarter is already inserted we tell the customer...

...otherwise we accept the quarter and transition to the HAS_QUARTER state.

And if the machine is sold out, we reject the quarter.

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

State Pattern
Motivation: Implementation

Simple approach:

- Represent the states as integer values
- Represent the events as methods
- Implement actions and decide on subsequent states conditionally…
- …in a complex structure for each possible combination of state and event ☹

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}

public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}

public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}

// other methods here like toString() and refill()
}
```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

You can't eject if the machine is sold out, it doesn't accept quarters!

If the customer just turned the crank, we can't give a refund; he already has the gumball!

The customer tries to turn the crank...

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

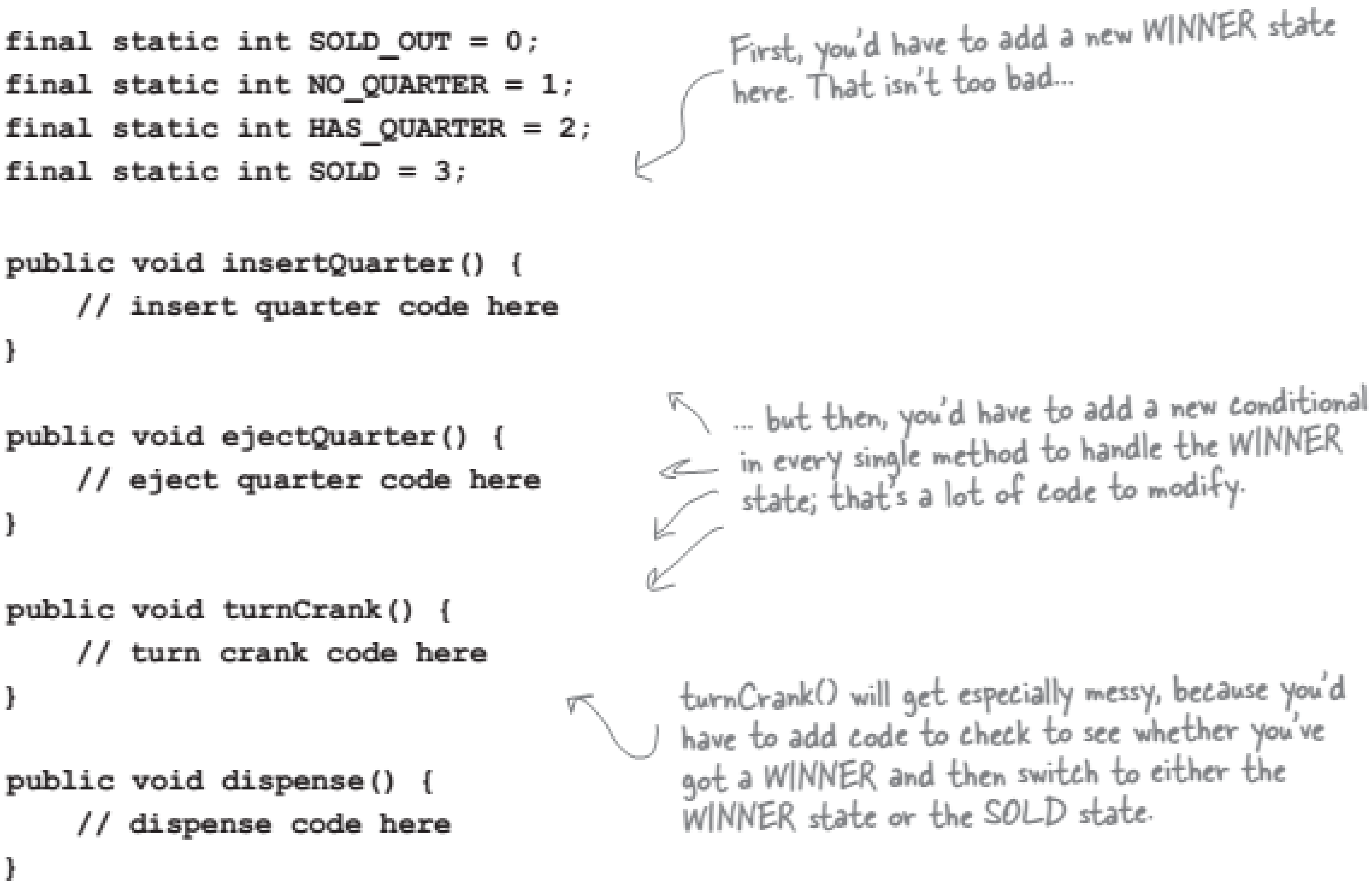
We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

State Pattern
Motivation: Dealing with Change

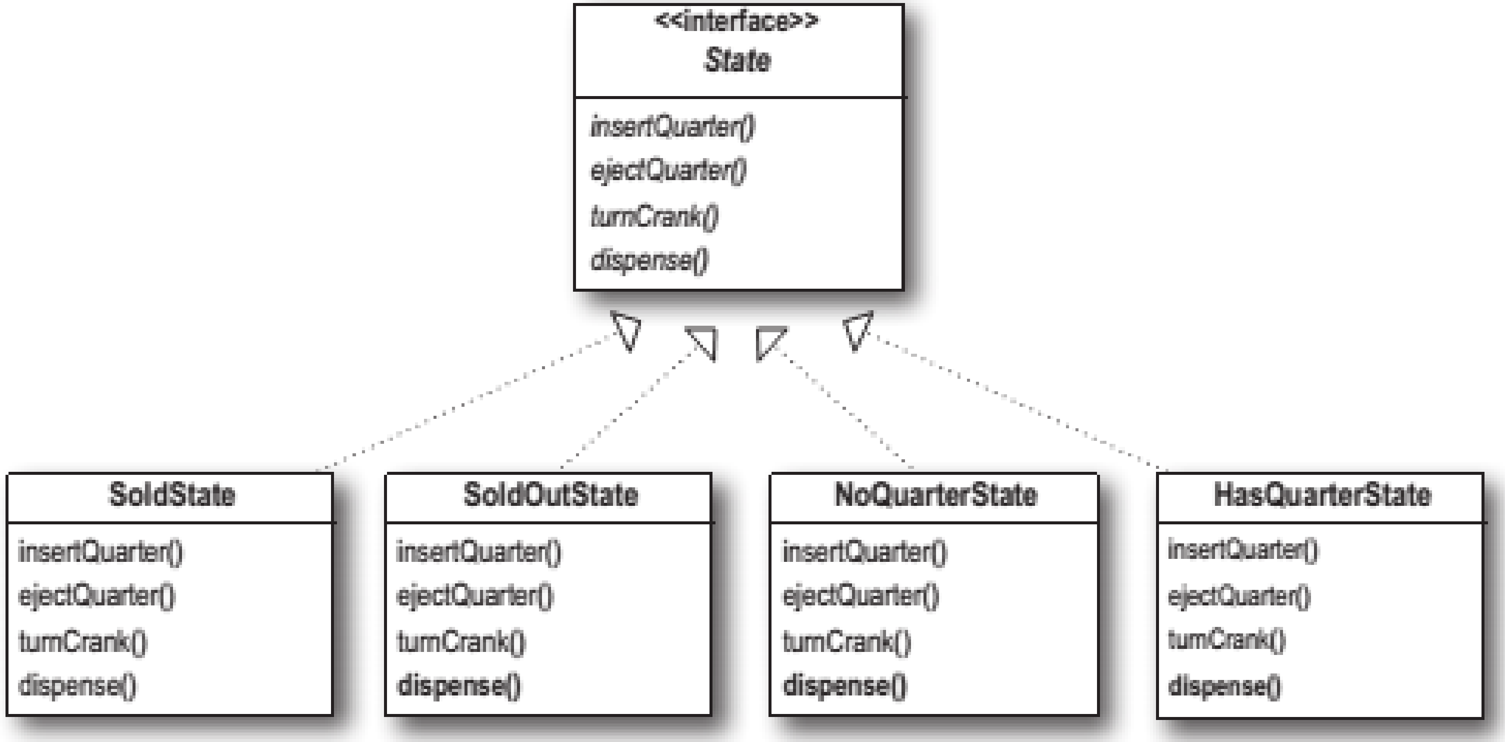
- If we want to change the structure of the state machine…
  - e.g. to add a state
    - like letting people win extra gumballs sometimes
  - e.g. to add a transition
    - e.g. to refill the machine when empty
- …we need to change the implementation of each method
  - tedious and error-prone due to all the conditionals




```java
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

State Pattern
Solution: Modeling States as Classes

- Represent each state as a class (implementing a common `State` interface)
  - with methods representing the potential events
  - and method implementations determining what should happen upon a particular event in a particular state



State Pattern
Example Implementation

- Integer state represen-tations are replaced by `State` objects.
- `State` objects are created and initial `state` is determined.

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
            state = soldOutState;
        }
    }
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState; otherwise, we start in the SoldOutState.

State Pattern
Example Implementation

- The event methods of the `Gumball-Machine` don't do anything on their own, but delegate the decision on what to do upon an event to the current `state`.

```
    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }
    // More methods here including getters for each State...
}
```

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

State Pattern
Example Impl.

- Each `State` stores a reference to the `GumballMachine` that it describes.
- Each method contains logic reacting to the respective event and possibly changing its `gumballMachine`'s state.
- (Other states are implemented analogously.)

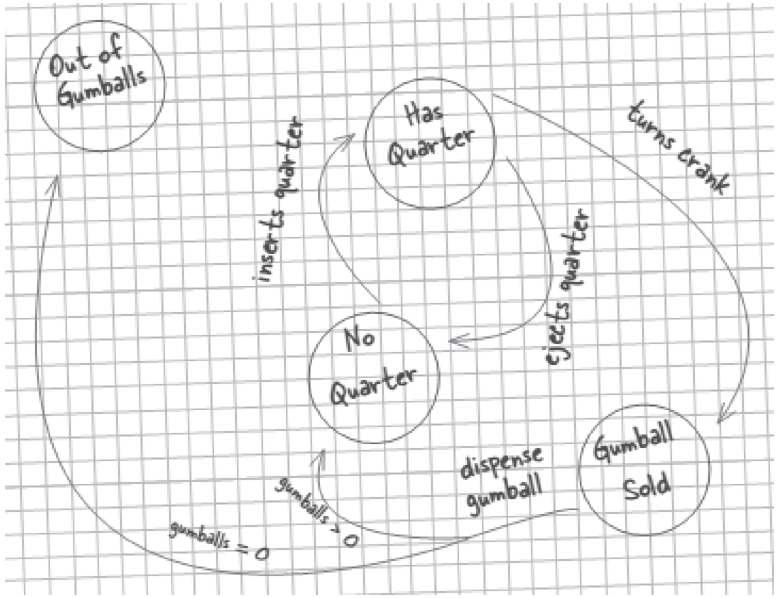
```java
public class SoldState implements State {
    GumballMachine gumballMachine;
    public SoldState(GumballMachine gm) { gumballMachine = gm; }

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

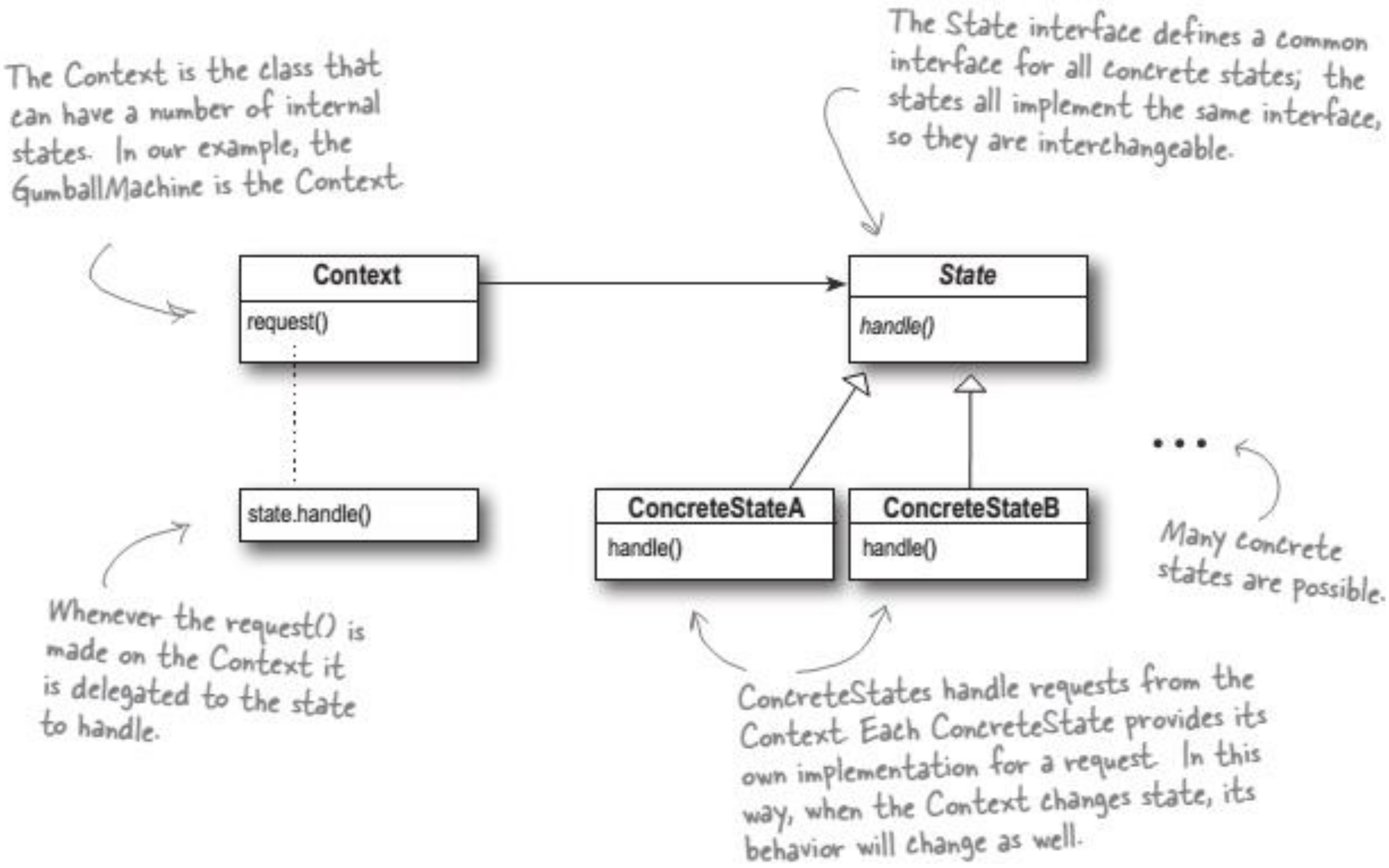
State Pattern
Protip: Dealing with Impossible Transitions

- A lot of the previous code was concerned with responding to events that
  - are not valid transitions in the business domain
  - but could be triggered by calls to the respective methods.
- In practice, you wouldn't write comments to `stdout`, but throw `UnsupportedOperationExceptions` when encountering events not valid in current state.
- An efficient way to implement this:
  - Instead of an `interface State`, use an `abstract class State` where all that each method does is `throw new UnsupportedOperationException()`.
  - In subclasses that `extend State`, you then need to override only methods for transitions that are actually allowed in the business domain, making the concrete state classes much smaller and more readable.



State Pattern
Summary

- The **State pattern** allows an object to alter its behavior when its internal state changes.
- Notice: Structurally identical to the Strategy pattern!
  - with `State` as interchangeable algorithm
  - Key difference: `State` changes itself instead of being determined from outside



State Pattern
Discussion: State vs. Usability

- In many software systems, it is discouraged to have different states (“modes”) in which the application behaves differently, especially if it is not readily distinguishable which state the application is currently in.
  - e.g. having to switch an editor from “view mode” to “edit mode” in order to change a text
- Instead, users should usually be enabled to do whatever they like at any time, without the application getting in the way
  - e.g. in a modern editor, you can view and edit a document without having to switch modes

- Sometimes, working with states/modes can be helpful if they are intuitive though
  - e.g. using different “drawing tools” in a photo editing software
  - e.g. browsing a wiki page in “viewer” or “editor” mode, depending on login state and role
  - In games, certain player or environment states can be key features of gameplay
    - e.g. temporary superpowers etc.

Summary: Selected Design Patterns

- Singleton *(→ FR5)*
- Factory *(→ FR4)*
- Adapter *(→ FR7)*
- Proxy *(→ FR11)*
- Model-View-Controller *(→ FR12)*
- Strategy *(→ FR1)*
- Decorator *(→ FR3)*
- Observer *(→ FR2)*
- Command *(→ FR6)*
- State *(→ FR10)*

- Introduced here as inspiration for how to use objects to solve problems and structure software systems cleanly
- Some rely on OO language constructs, but most can be employed in any language to facilitate better understanding and maintenance of complex software systems
- There are many more – see e.g.
  - Freeman, Robson: Head First Design Patterns, O'Reilly 2004
  - Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley 1994