SENG 350
- Software Architecture & Design

Shuja Mughal

Design Patterns

Fall 2024

University of Victoria

The Observer Pattern

University of Victoria

Observer Pattern

A behavioral (object) pattern:

- Concerns about objects and their behavior.

Applicability

- Vary and reuse 2 different abstractions independently.
- Change to one object requires change in (one or more) other objects – whose identity is not necessarily known

Observer Pattern - Participants

Subject

Has a list of observers;
interfaces for attaching/detaching an observer

Observer

An updating interface for objects that gets notified of changes in a subject.

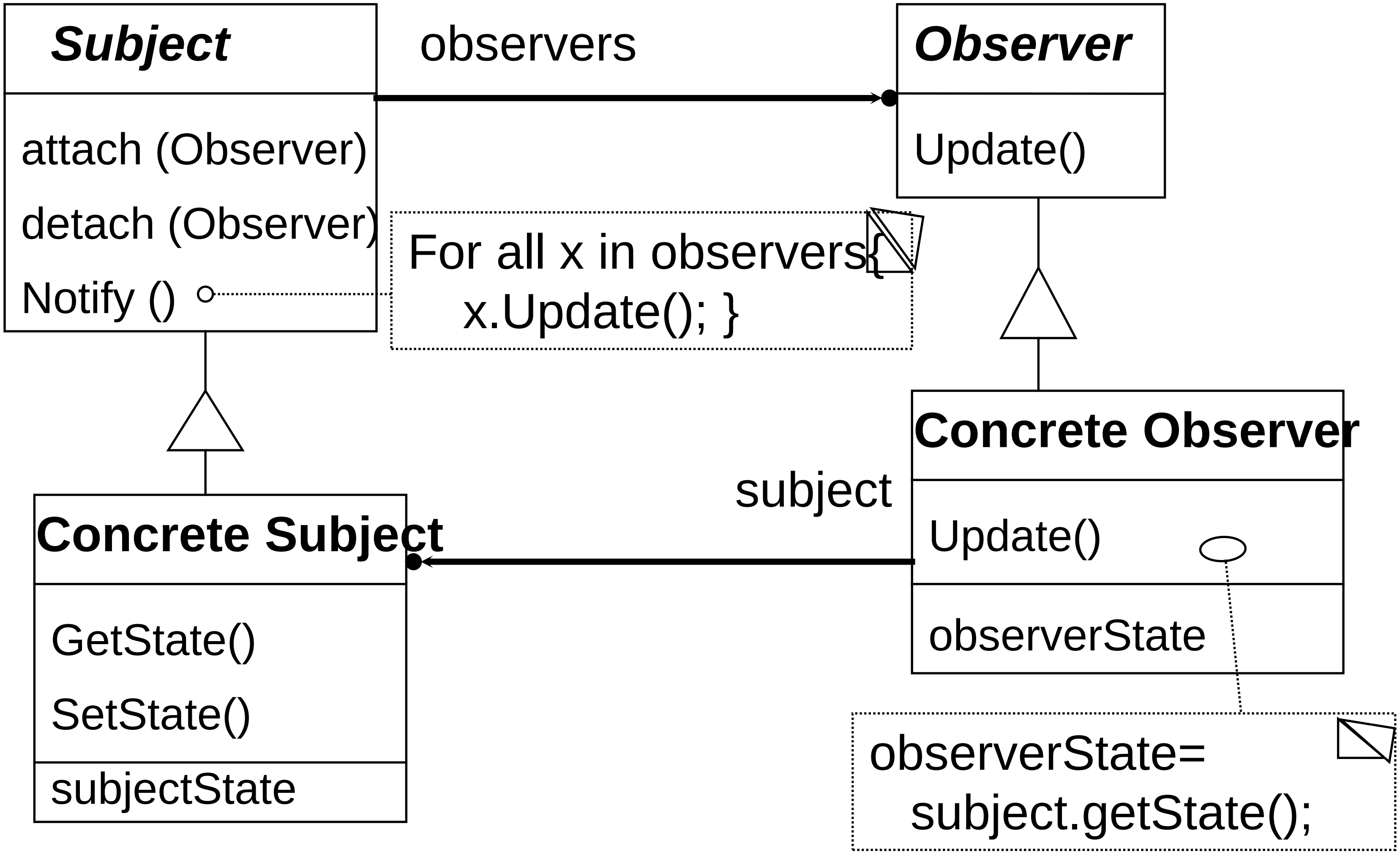
ConcreteSubject

Stores state of interest to observers
Sends notification when state changes.

ConcreteObserver

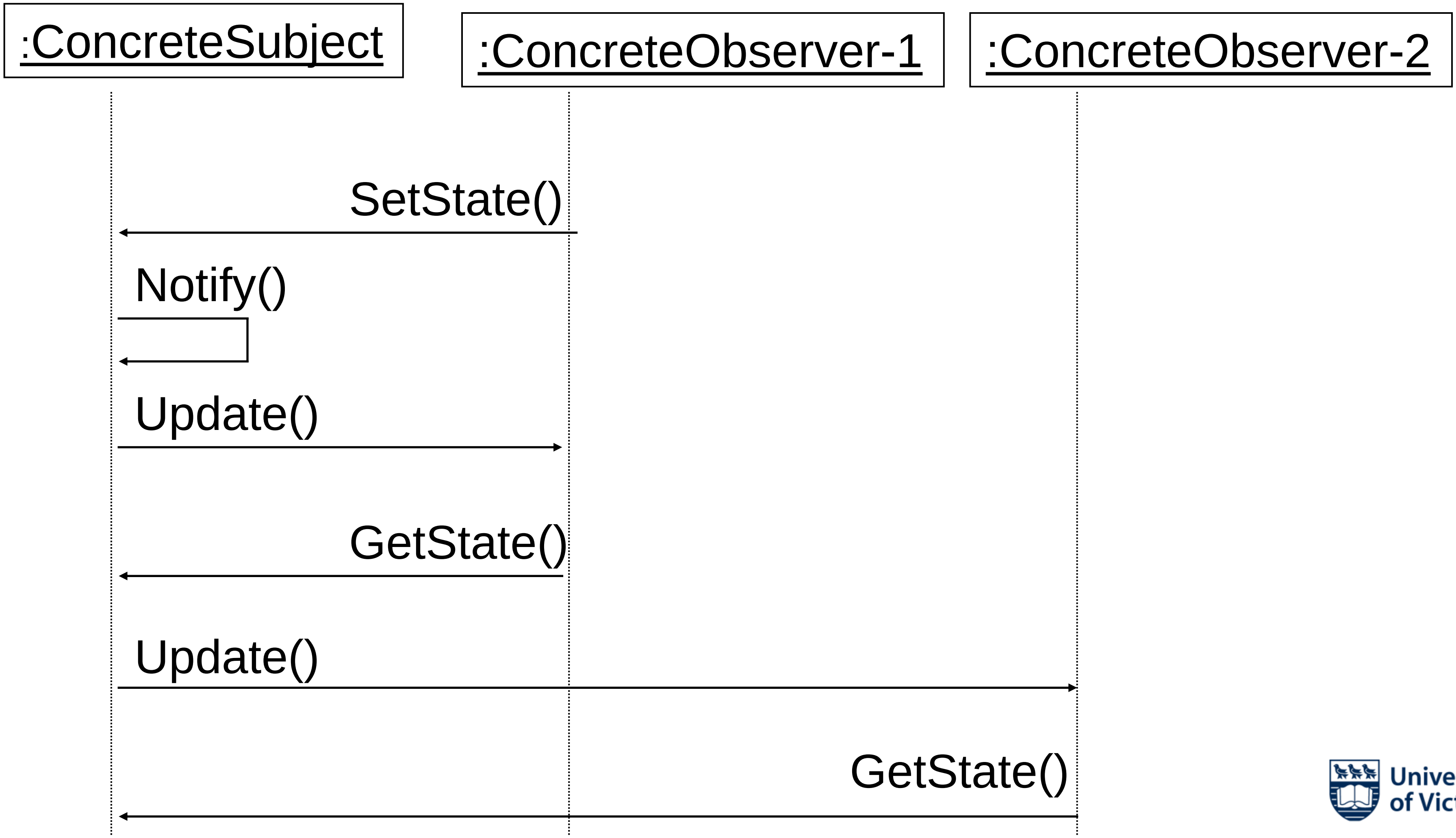
Implements updating interface.

Observer Pattern – Structure



University of Victoria

Observer Pattern - Collaborations



Observer Pattern - Implementation

```
interface Observer {
        void update (Observable sub, Object arg)
}
```

Java terminology for Subject.

```
class Observable {

           public void addObserver(Observer o) {}

           public void deleteObserver (Observer o)  {}

           public void notifyObservers(Object arg) {}

           public boolean hasChanged() {}
 ...

}
```

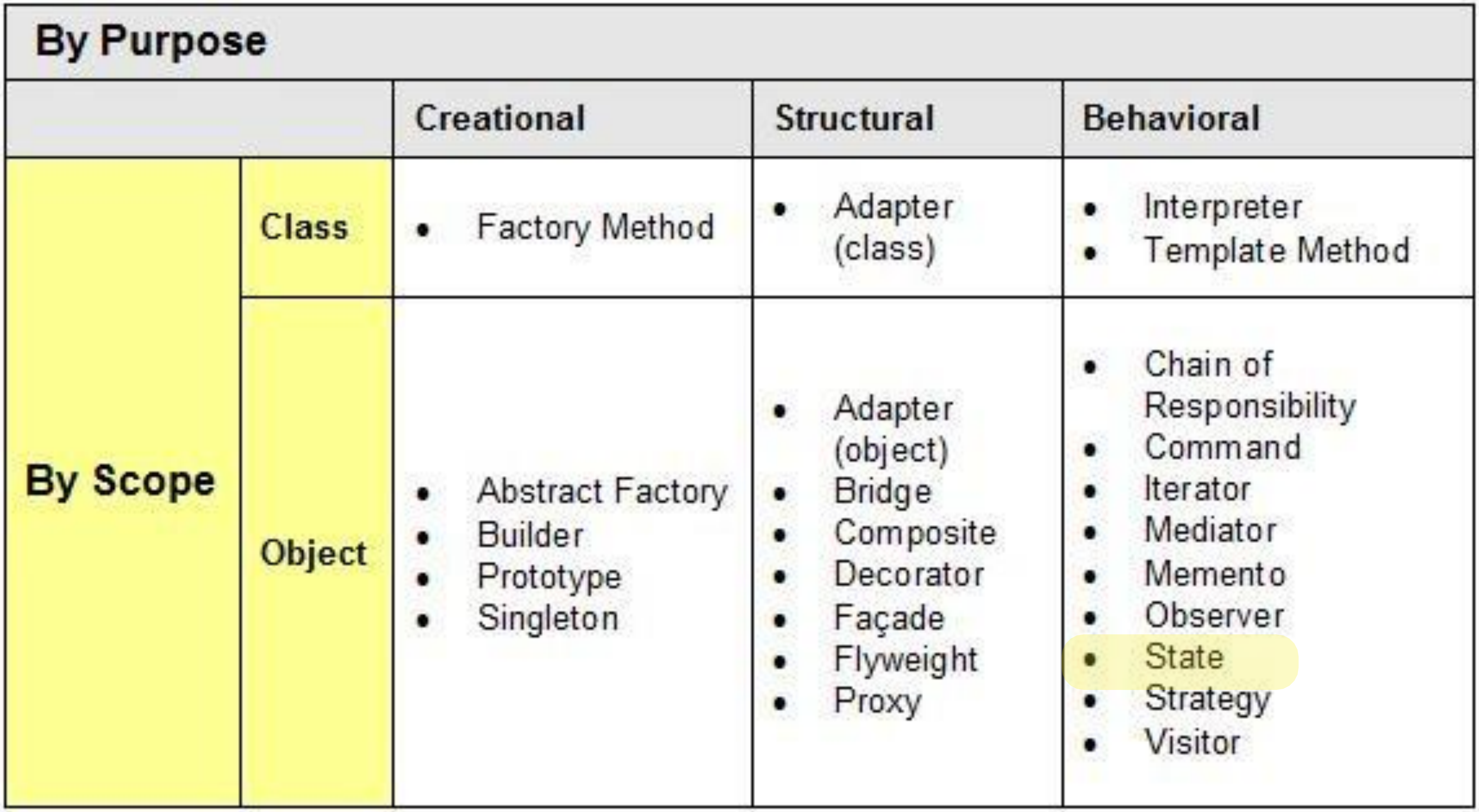
Observer Pattern - Implementation

```
   public PiChartView implements Observer {

            void update(Observable sub, Object arg) {
               // repaint the pi-chart
            }
    }

 class StatsTable extends Observable{

          public boolean hasChanged() {

                   // override to decide when it is considered changed

          }

 }
```

A Concrete Observer.

Observer Pattern - Consequences

- Abstract coupling between subject and observer. (subject need not know concrete observers)
- Support for broadcast communication (all observers are notified)
- Unexpected updates (observers need not know when updates occur)



By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

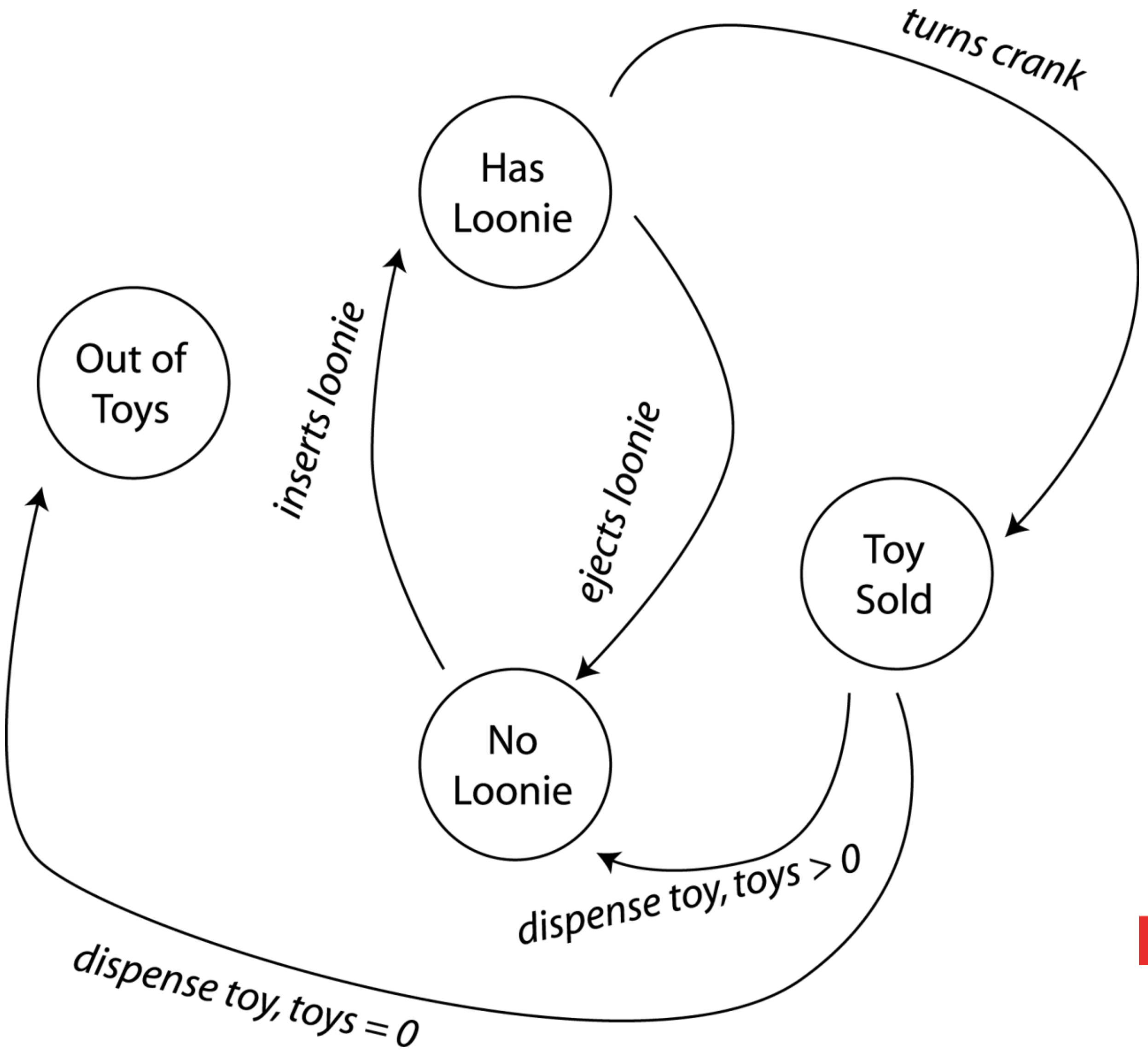
The State Pattern

University of Victoria

The State Pattern

Example: Toy vending machine

A State Machine Model



How to implement this?

Straight-forward Implementation

State as variable

```
final static int SOLD_OUT = 0;
final static int NO_LOONIE = 1;
final static int HAS_LOONIE = 2;
final static int SOLD = 3;

int state = SOLD_OUT;

public void insertLoonie() {
    if (state == HAS_LOONIE) {
        System.out.println (“You cannot insert another loonie, eh?”);
    } else if (state == SOLD_OUT) {
        System.out.println (“You cannot insert a loonie, the machine is sold out.”);
    } else if (state == SOLD) {
        System.out.println (“Please wait, we’re already giving you a toy!”);
    } else if (state == NO_LOONIE) {
        state = HAS_LOONIE;
        System.out.println (“You inserted a loonie.”);
    }
}
```

Transitions as methods

University of Victoria

```
public void ejectLoonie() {
    if (state == HAS_LOONIE) {
       System.out.println (“Loonie returned”);
       state = NO_LOONIE;
   } else if (state == NO_LOONIE) {
       System.out.println (“You have not inserted a loonie yet, kid!”);
   } else if (state == SOLD) {
       System.out.println (“Sorry, you already turned the crank.”);
   } else if (state == SOLD_OUT) {
       System.out.println (“You cannot eject, you haven’t inserted a loonie yet!”);
   }
}


public void turnCrank() {
    if (state == SOLD) {
      System.out.println (“Turning twice doesn’t get you another toy!”);
    } else if (state == NO_LOONIE) {
       System.out.println (“You turned, but there is no loonie”);
    } else if (state == SOLD_OUT) {
       System.out.println (“You turned, but there are no toys left.”);
    } else if (state == HAS_LOONIE) {
       System.out.println (“You turned…”);
       state = SOLD;
       dispense();
    }
}
```

```
public void dispense() {
    if (state == SOLD) {
      System.out.println (“A toy comes rolling out the slot”);
      count = count - 1;
      if (count == 0) {
         System.out.println (“Oops, out of toys!”);
         state = SOLD_OUT;
      } else {
         state = NO_LOONIE;
      }
   } else if (state == NO_LOONIE) {
      System.out.println (“You need to pay first, eh?”);
   } else if (state == SOLD_OUT) {
      System.out.println (“No toy dispensed”);
   } else if (state == HAS_LOONIE) {
      System.out.println (“No toy dispensed”);
   }
}
```

The “out of toys” situation handled for us within this code.

This should never happen! But if it does, we print an error message rather than give a toy.

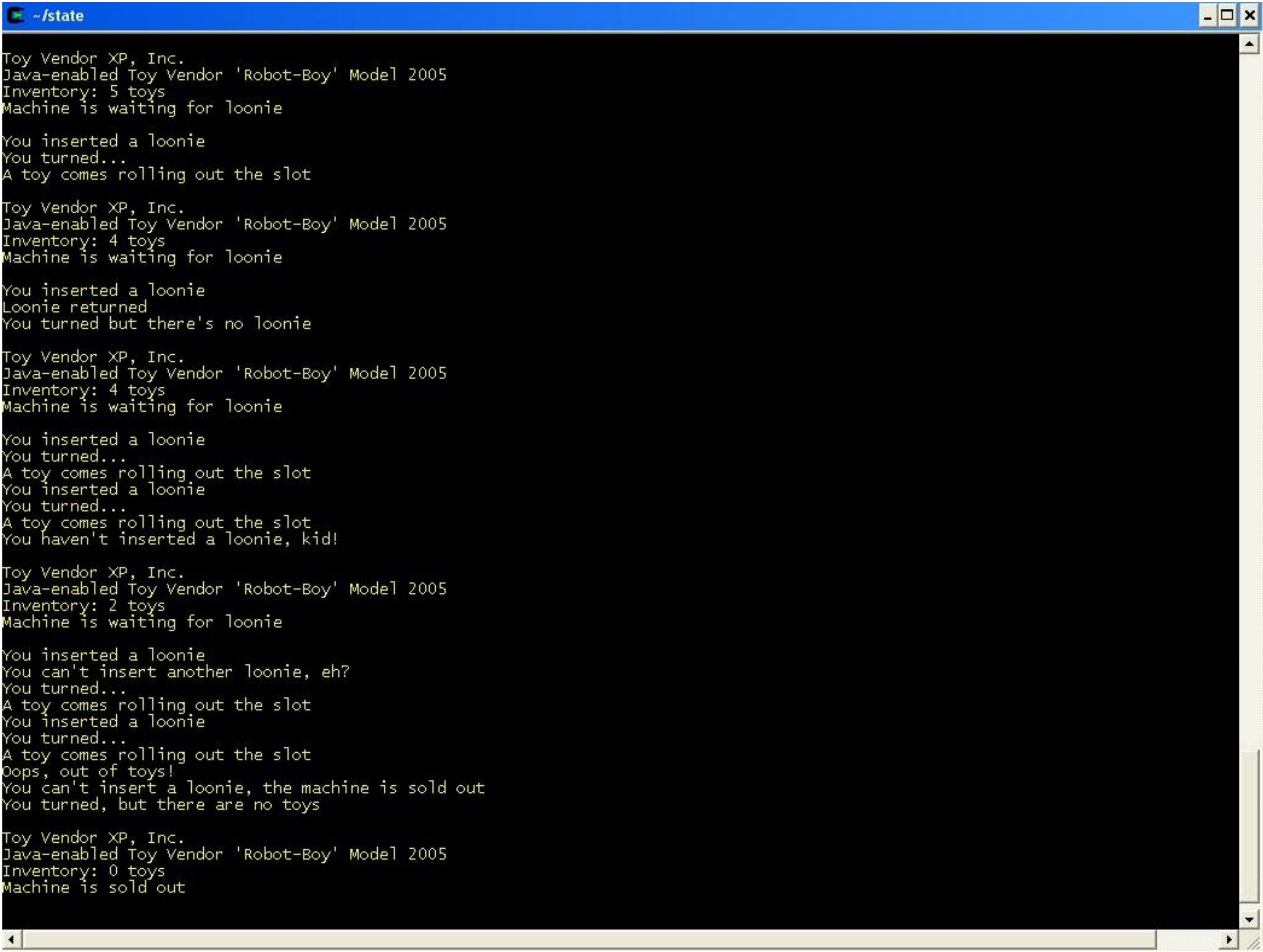
```
public ToyVendor (int count) {
   this.count = count;
   if (count > 0) {
      state = NO_LOONIE;
   }
}
```

Constructor takes an initial inventory of toys. If the inventory is not zero, the machine enters state NO_LOONIE, meaning it is waiting for some kid to insert a loonie.

Test Drive the Machine

```
public class ToyVendorTestDrive { public static void
    main(String[] args) {
        ToyVendor vendMachine = new
ToyVendor(5);
        System.out.println (vendMachine);
        vendMachine.insertLoonie ();
        vendMachine.turnCrank();
        System.out.println (vendMachine);
        vendMachine.insertLoonie();
        vendMachine.ejectLoonie();
        vendMachine.turnCrank();
        System.out.println (vendMachine);
        vendMachine.insertLoonie ();
        vendMachine.turnCrank();
        vendMachine.insertLoonie();
        vendMachine.turnCrank();
        vendMachine.ejectLoonie();
        System.out.println (vendMachine);
```

```
        vendMachine.insertLoonie();
        vendMachine.insertLoonie();
        vendMachine.turnCrank();
        vendMachine.insertLoonie();
        vendMachine.turnCrank();
        vendMachine.insertLoonie();
        vendMachine.turnCrank();
        System.out.println (vendMachine);
}
```



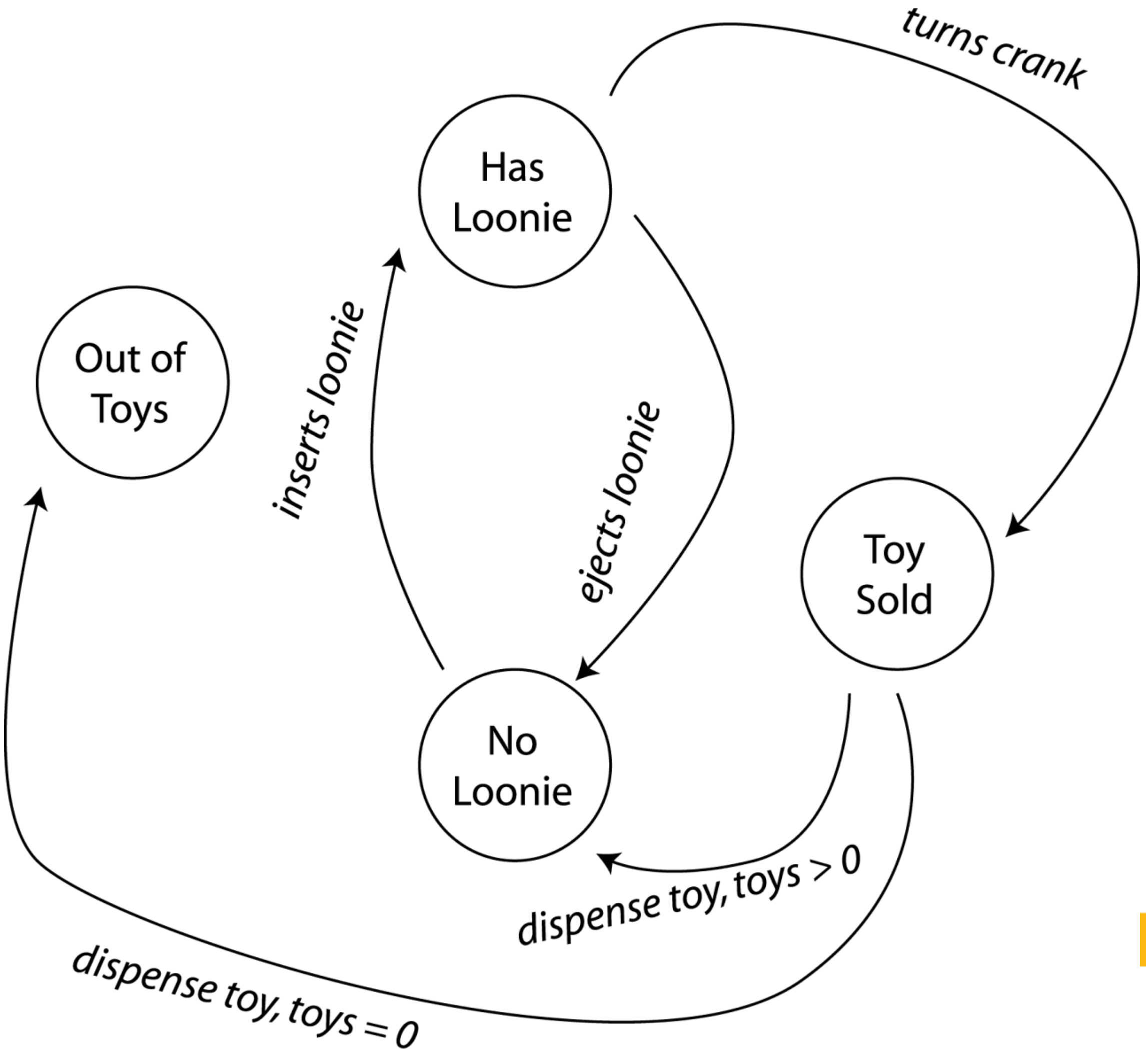
```
Toy Vendor XP, Inc.
Java-enabled Toy Vendor 'Robot-Boy' Model 2005
Inventory: 5 toys
Machine is waiting for loonie

You inserted a loonie
You turned...
A toy comes rolling out the slot

Toy Vendor XP, Inc.
Java-enabled Toy Vendor 'Robot-Boy' Model 2005
Inventory: 4 toys
Machine is waiting for loonie

You inserted a loonie
Loonie returned
You turned but there's no loonie

Toy Vendor XP, Inc.
Java-enabled Toy Vendor 'Robot-Boy' Model 2005
Inventory: 4 toys
Machine is waiting for loonie

You inserted a loonie
You turned...
A toy comes rolling out the slot
You inserted a loonie
You turned...
A toy comes rolling out the slot
You haven't inserted a loonie, kid!

Toy Vendor XP, Inc.
Java-enabled Toy Vendor 'Robot-Boy' Model 2005
Inventory: 2 toys
Machine is waiting for loonie

You inserted a loonie
You can't insert another loonie, eh?
You turned...
A toy comes rolling out the slot
You inserted a loonie
You turned...
A toy comes rolling out the slot
Oops, out of toys!
You can't insert a loonie, the machine is sold out
You turned, but there are no toys

Toy Vendor XP, Inc.
Java-enabled Toy Vendor 'Robot-Boy' Model 2005
Inventory: 0 toys
Machine is sold out
```

The Requirements Change

Be a winner!
One in TEN
kids get a
FREE TOY!

Version 2: Win a toy in a game of luck

Need a new State: “Winner”



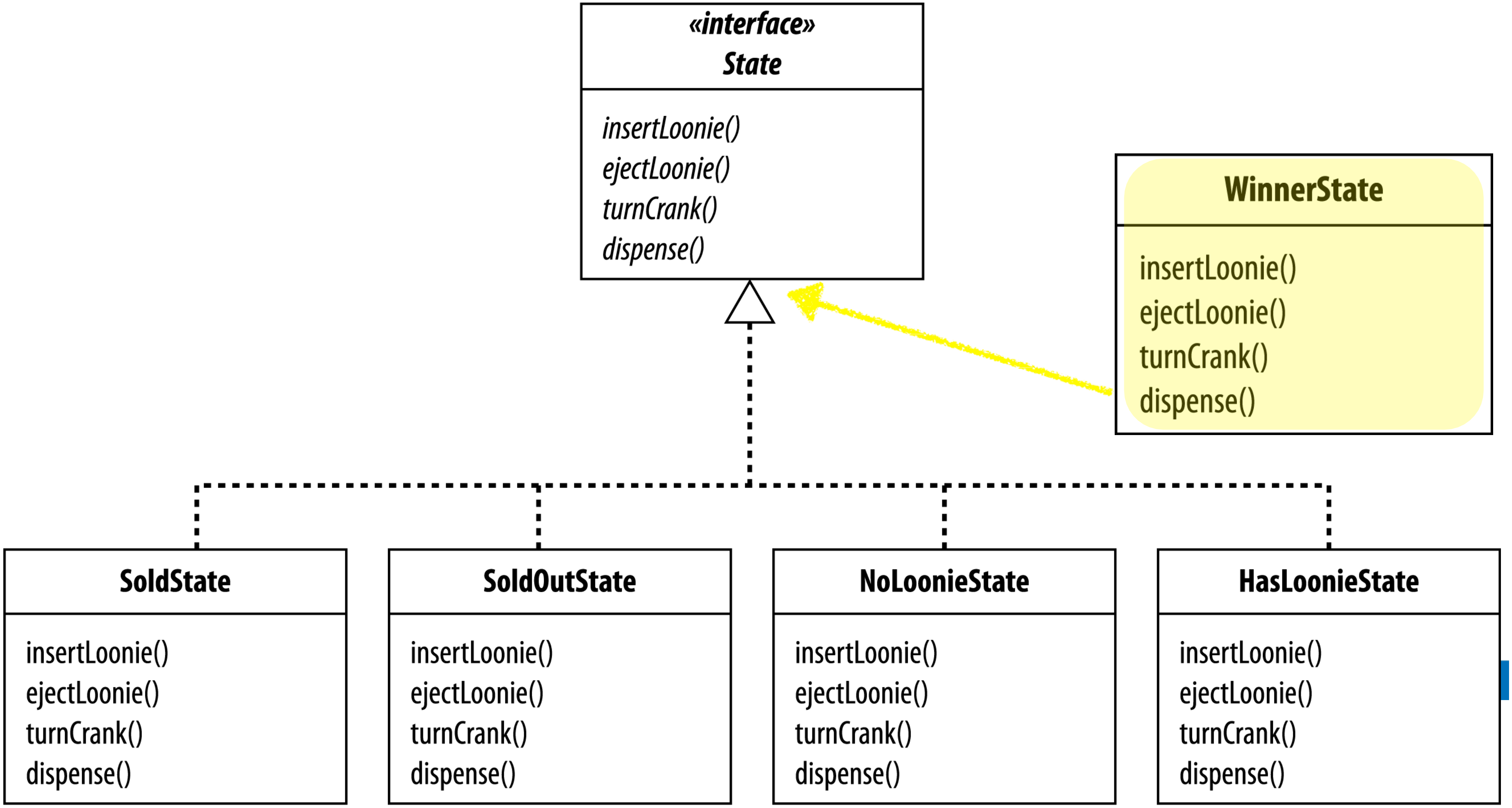
What does that mean for our implementation?

Every Transition Method has to change (has to handle the new State)

```
final static int SOLD_OUT = 0;
final static int NO_LOONIE = 1;
final static int HAS_LOONIE = 2;
final static int SOLD = 3;
int state = SOLD_OUT;

public void insertLoonie() {
   if (state == HAS_LOONIE) {
      System.out.println (“You cannot insert another loonie, eh?”);
   } else if (state == SOLD_OUT) {
      System.out.println (“You cannot insert a loonie, the machine is sold out.”);
   } else if (state == SOLD) {
      System.out.println (“Please wait, we’re already giving you a toy!”);
   } else if (state == NO_LOONIE) {
      state = HAS_LOONIE;
      System.out.println (“You inserted a loonie.”);
   }
}
```

violates open-closed principle

A Better Design:
Map States to Classes / Objects



Refactor the code to the new design

```
public class NoLoonieState implements State {
    ToyVendor toyVendor;
    public NoLoonieState (ToyVendor toyVendor) {
        this.toyVendor = toyVendor;
    }
    public void insertLoonie () {
        System.out.println (“You inserted a loonie.”);
        this.toyVendor.setState (toyVendor.getHasLoonieState());
    }
    public void ejectLoonie() {
        System.out.println (“You have not inserted a loonie, eh?”);
    }
    public void turnCrank() {
        System.out.println (“You turned, but there’s no loonie!”);
    }
    public dispense() {
        System.out.println (“You need to pay first”);
    }
}
```

Each state class implements the behaviours appropriate for its state

this may involve a state transition

State objects are now members of the class.

The special member “state” refers to the *current* state object.

```
public class ToyVendor {

    final static int SOLD_OUT = 0;
    final static int NO_LOONIE = 1;
    final static int HAS_LOONIE = 2;
    final static int SOLD = 3;
    int state = SOLD_OUT;
    int count = 0;
    // …
```

```
public class ToyVendor {

    State soldOutState;
    State noLoonieState;
    State hasLoonieState;
    State soldState;
    State  state = soldOutState;
    int count = 0;
    // …
```

```
public class ToyVendor {
    State soldOutState;
    State noLoonieState;
    State hasLoonieState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public ToyVendor (int numberToys) {
        soldOutState = new SoldOutState (this);
        noLoonieState = new NoLoonieState (this);
        hasLoonieState = new HasLoonieState (this);
        soldState = new SoldState (this);
        this.count = numberToys;
        if (numberToys > 0) {
            state = noLoonieState;
        }
    }
        //….
```

The new constructor

ToyVendor implementation (cont'd)

```
public void insertLoonie() {
    state.insertLoonie();
}
public void ejectLoonie() {
    state.ejectLoonie();
}
public void turnCrank() {
    state.turnCrank();
    state.dispense();
}
void setState (State state) {
    this.state = state;
}
void releaseToy() {
    System.out.println (“A toy rolls out the slot…”);
    if (count != 0) {
        count = count – 1;
    }
}
// …
```

The ToyVendor methods are now easy to implement

Implementing more states

```
public class HasLoonieState implements State {
    ToyVendor toyVendor;
    public HasLoonieState (ToyVendor toyVendor) {
        this.toyVendor = toyVendor;
    }
    public void insertLoonie () {
        System.out.println (“You cannot insert another loonie.”);
    }
    public void ejectLoonie() {
        System.out.println (“Loonie returned.”);
        toyVendor.setState(toyVendor.getNoLoonieState());
    }
    public void turnCrank() {
        System.out.println (“You turned…”);
        toyVendor.setState(toyVendor.getSoldState());
    }
    public dispense() {
        System.out.println (“No gumball dispensed”);
    }
}
```

We need the reference to ToyVendor in order to transition it (when needed) to a different state.

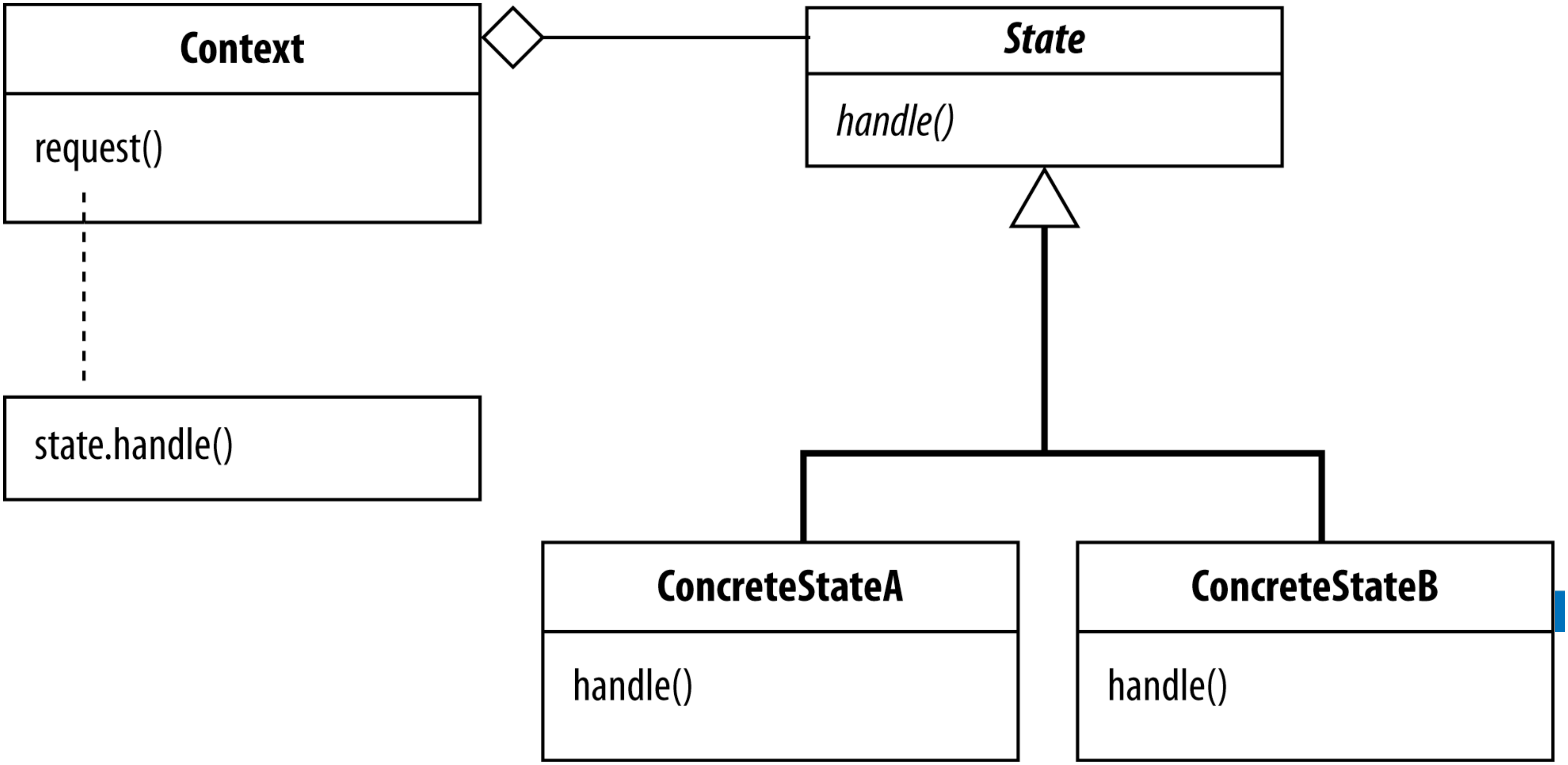
When crank is turned, machine is transitioned to SoldState by call the machine’s setState method and passing it the machine’s SoldState object.

Implementing more states

```
public class SoldState implements State {
   // constructor, instance variables here…
   public void insertLoonie () {
      System.out.println (“Please wait! We’re already giving “
         + “a toy!”);
   }
   public void ejectLoonie() {
      System.out.println (“Sorry, you already turned the crank.”);
  }
   public void turnCrank() {
      System.out.println (“Turning twice does not get you “
         + “another toy!”);
   }
   public dispense() {
      toyVendor.releaseToy();
      if (toyMachine.getCount() > 0) {
         toyMachine.setState  (toyMachine.getNoLoonieState());
      } else {
         System.out.println (“Oops, out of toys!”);
         toys.setState  (toyMachine.getSoldOutState());
      }
}
```

conditional transition

The State Pattern allows an object to alter its behavior when its internal state changes.



So, what extensions do we need to do for introducing the “Winner” functionality?

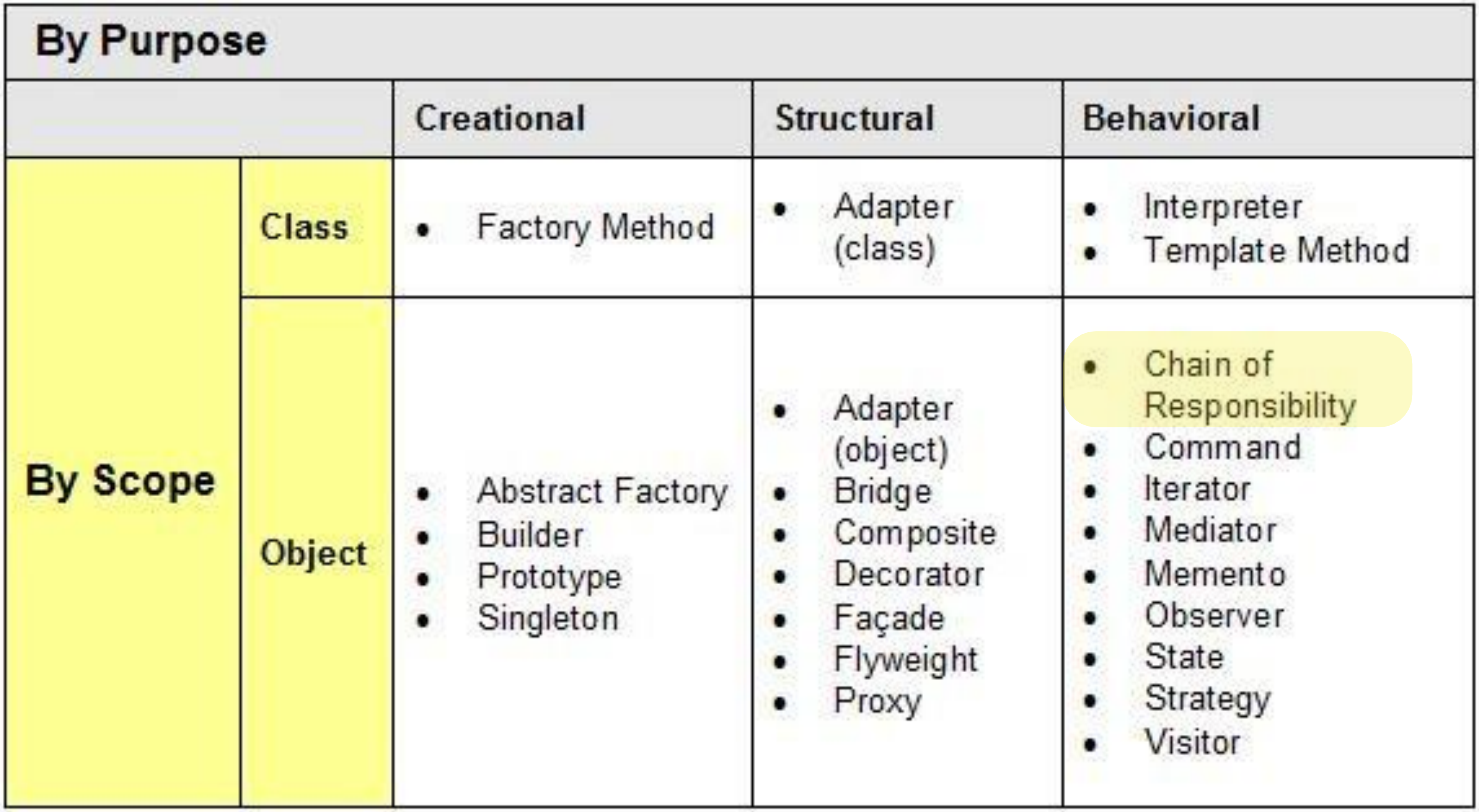
```
public class ToyVendor {
    State soldOutState;
    State noLoonieState;
    State hasLoonieState;
    State soldState;
    State winnerState;

    State = soldOutState;
    int count = 0;
//….
```

```
public class WinnerState implements State {
    // constructor, instance variables here…
    // insertLoonie error message
    // ejectLoonie error message
    // turnCrank error messages
    public dispense() {
        System.out.println (“YOU’RE A WINNER! You get two toys “
        + “for your loonie”);
        toyVendor.releaseToy();
        if (toyVendor.getCount() == 0) {
            toyVendor.setState(toyVendor.getSoldOutState());
        } else {
            toyVendor.releaseToy();
            if (toyVendor.getCount() > 0) {
                toyVendor.setState(toyVendor.getNoLoonieState());
            } else {
                System.out.println (“Oops, out of toys!”);
                toyVendor.setState(toyVendor.getSoldOutState());
            }
        }
    }
}
```

One new state - one new class / object - that’s how it should be

```
public class HasLoonieState implements State {
    Random randomWinner = new Random (System.currentTimeMillis());
    ToyVendor toyVendor;

    public HasLoonieState (ToyVendor toyVendor) {
        this.toyVendor = toyVendor;
    }

    public void insertLoonie () {
        System.out.println (“You cannot insert another loonie.”);
    }

    public void ejectLoonie() {
        System.out.println (“Loonie returned.”);
        toyVendor.setState(toyVendor.getNoLoonieState());
    }

    public void turnCrank() {
        System.out.println (“You turned…”);
        int winner = randomWinner.nextInt(10);
        if ((winner == 10) && (toyVendor.getCount() > 1)) {
            toyVendor.setState(toyVendor.getWinnerState());
        } else {
            toyVendor.setState(toyVendor.getSoldState());
        }
    }

    // code for dispense()

}
```

And - of course - we need a transition into it.



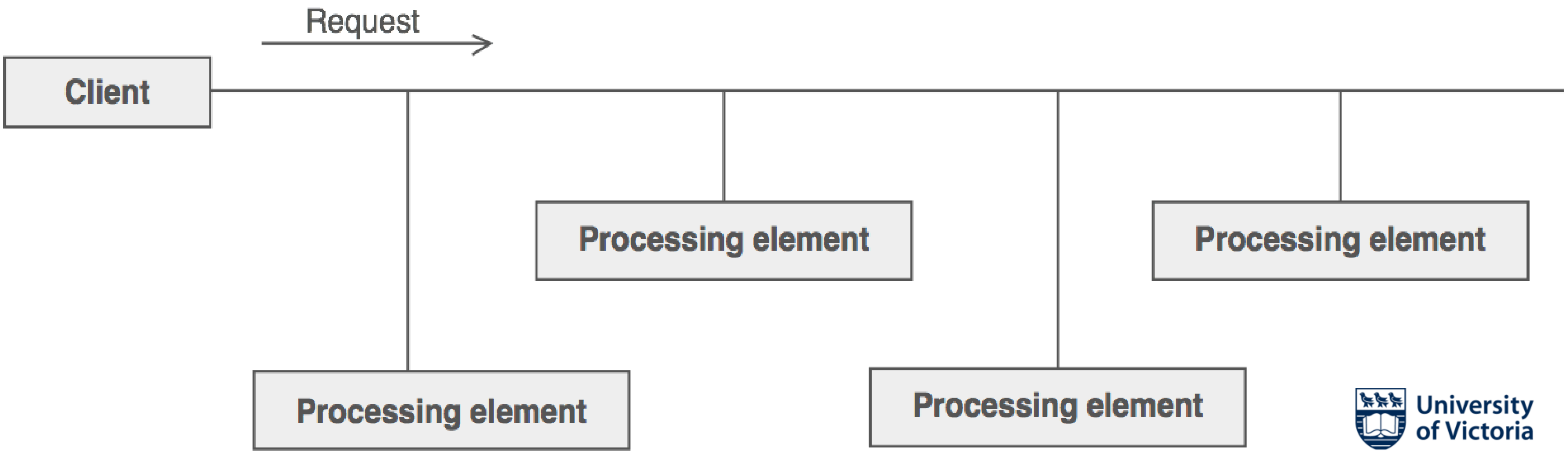
By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

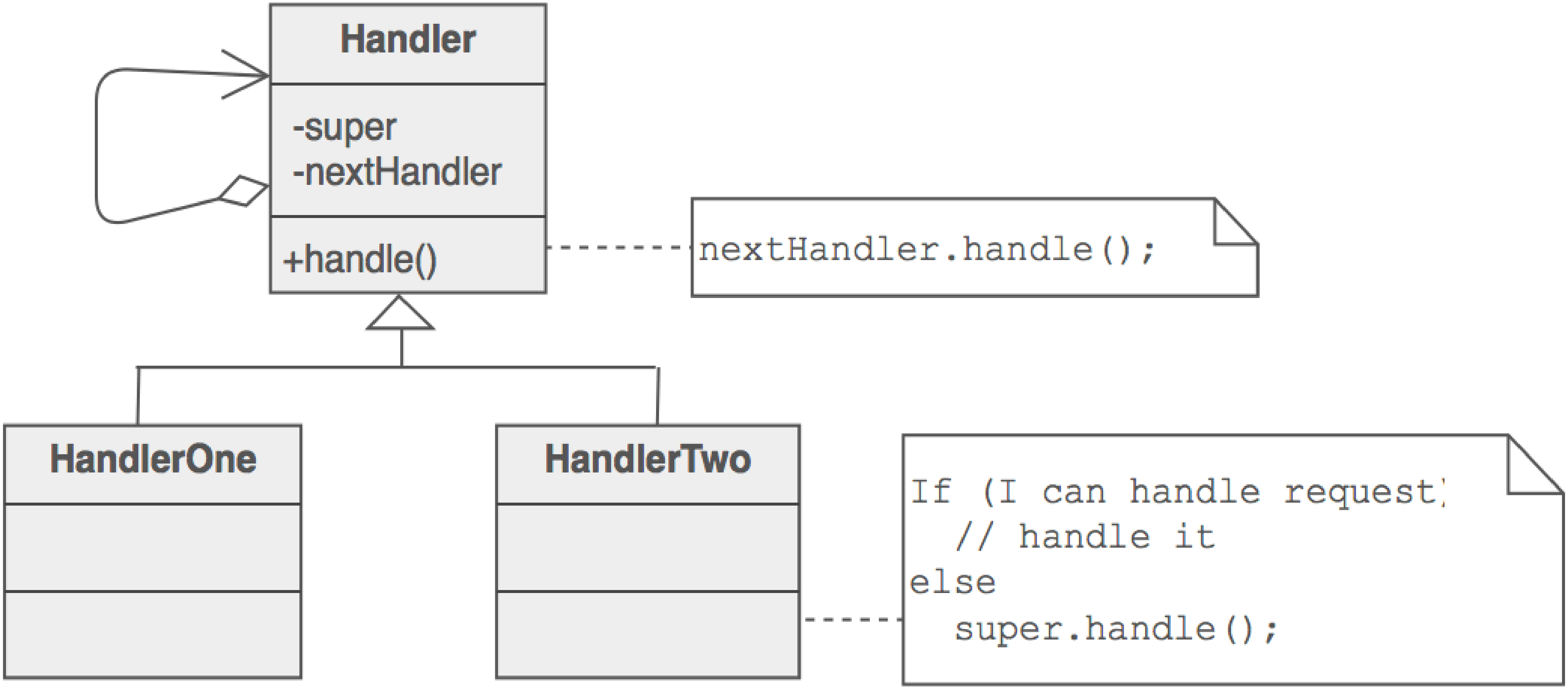
Chain of Responsibility Pattern

University of Victoria

Chain of Responsibility Pattern

Intent: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request





Handler

-super
-nextHandler

+handle()

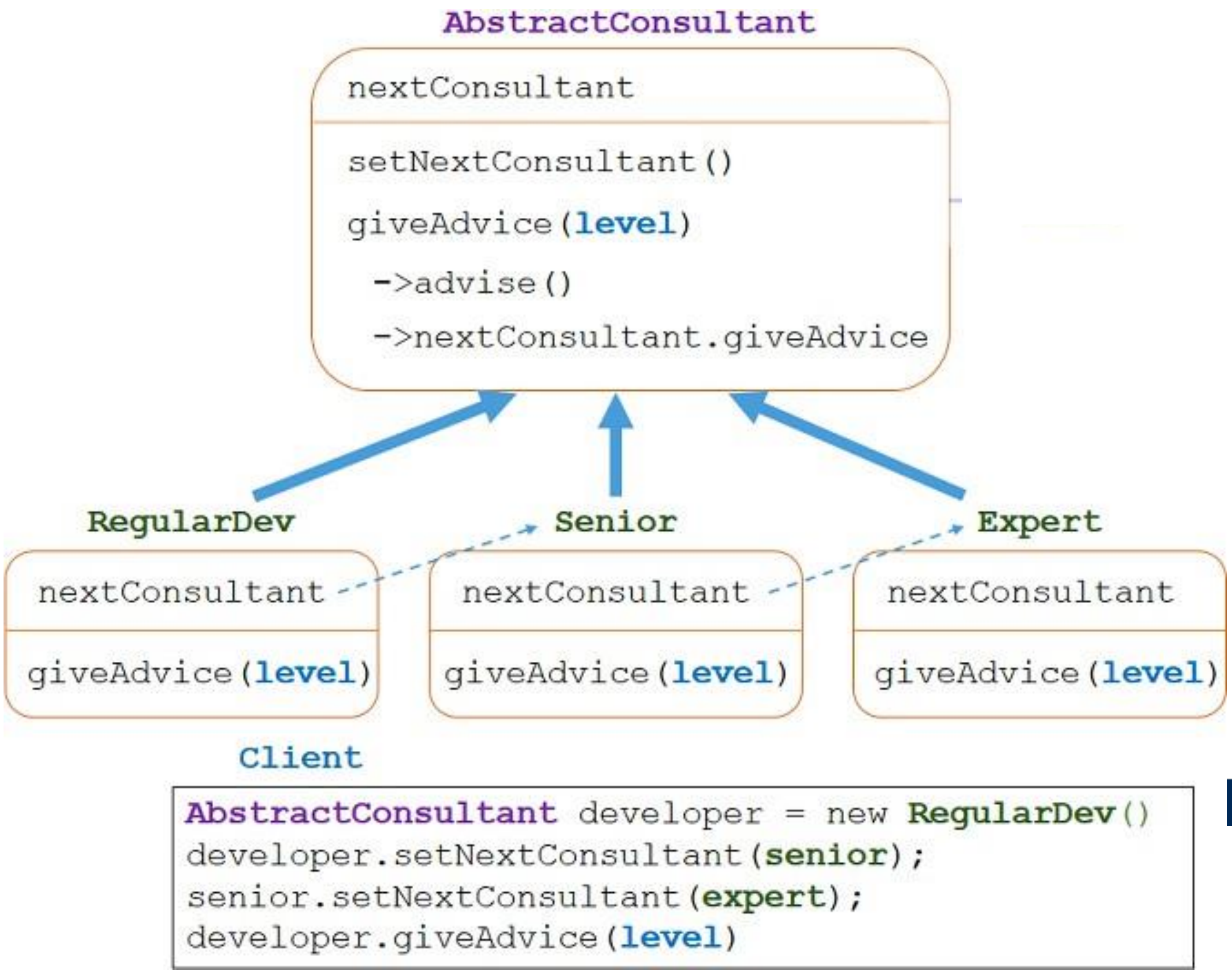
```
nextHandler.handle();
```

HandlerOne

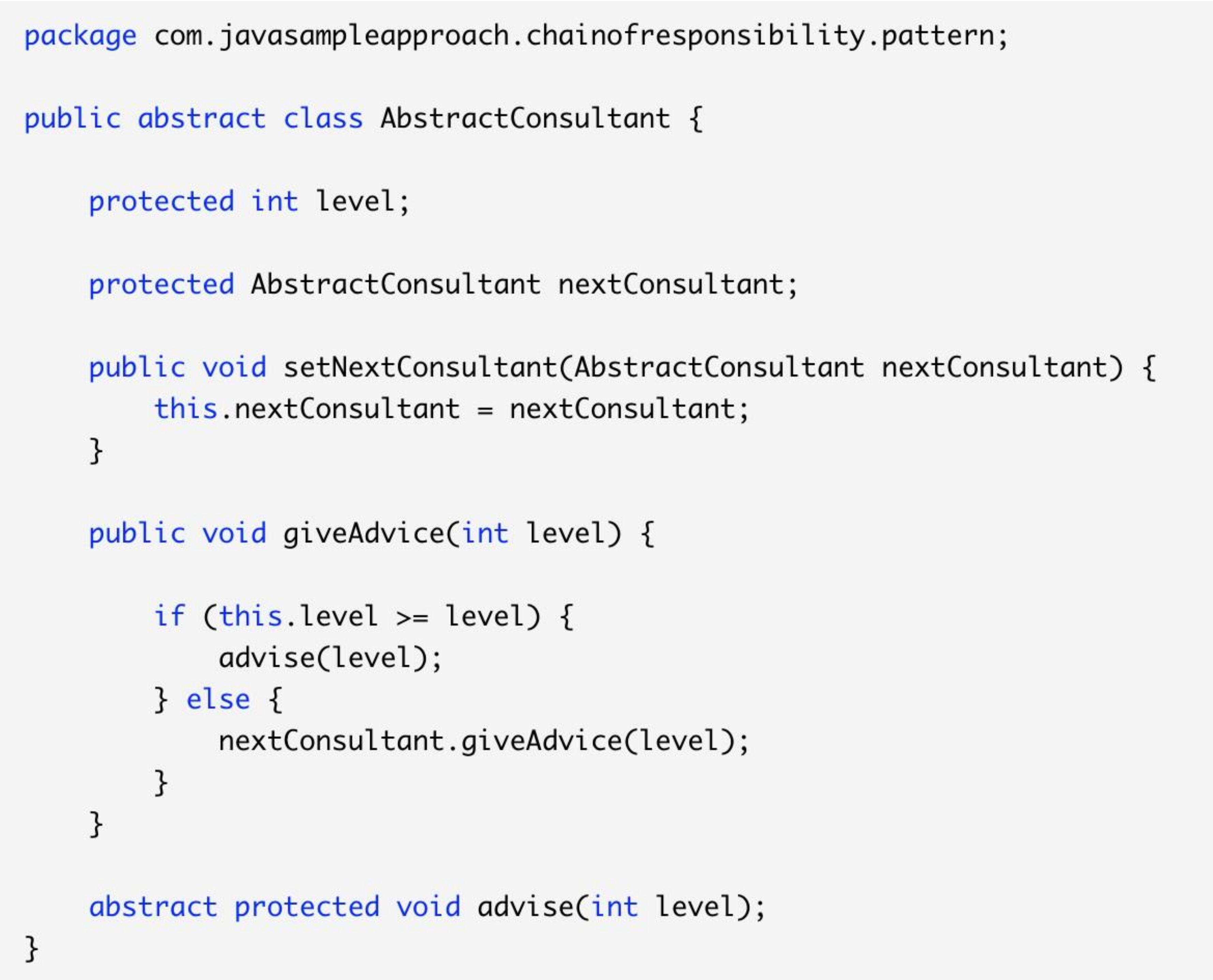
HandlerTwo

```
If (I can handle request)
  // handle it
else
  super.handle();
```

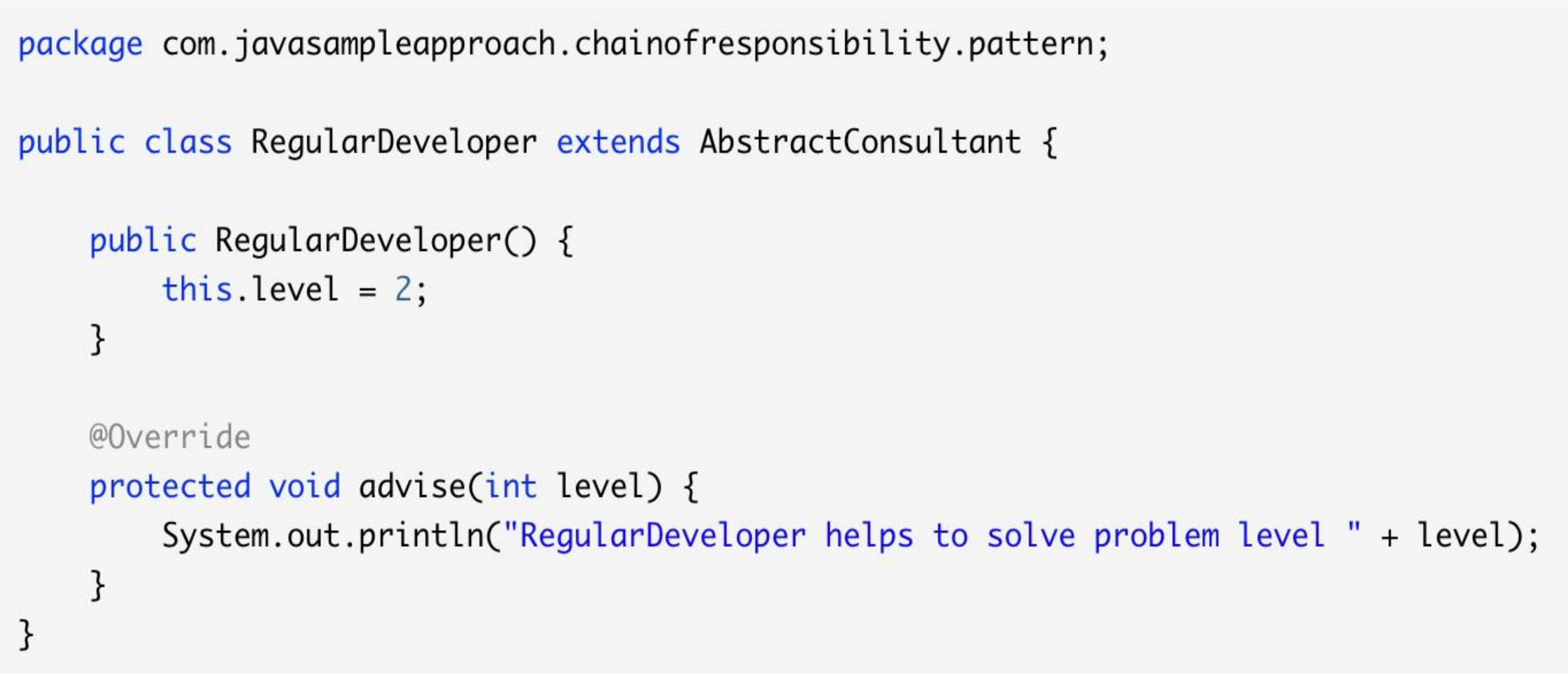
Several Variations of the COR pattern exist



```
AbstractConsultant developer = new RegularDev()
developer.setNextConsultant(senior);
senior.setNextConsultant(expert);
developer.giveAdvice(level)
```



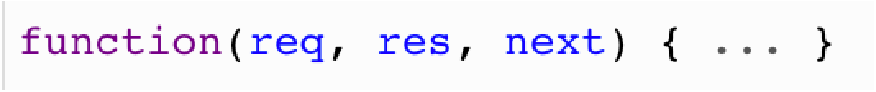
```java
package com.javasampleapproach.chainofresponsibility.pattern;

public abstract class AbstractConsultant {

    protected int level;

    protected AbstractConsultant nextConsultant;

    public void setNextConsultant(AbstractConsultant nextConsultant) {
        this.nextConsultant = nextConsultant;
    }

    public void giveAdvice(int level) {

        if (this.level >= level) {
            advise(level);
        } else {
            nextConsultant.giveAdvice(level);
        }
    }

    abstract protected void advise(int level);
}
```



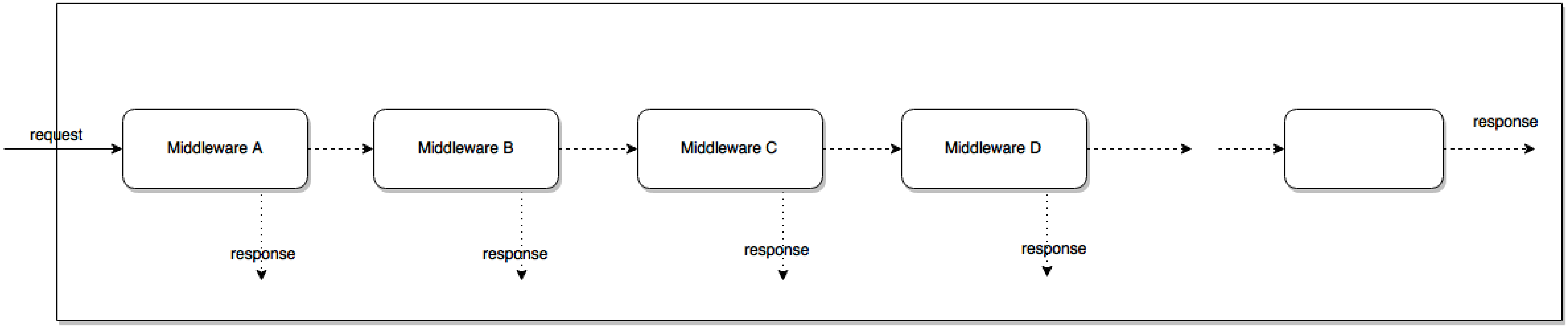
```java
package com.javasampleapproach.chainofresponsibility.pattern;

public class RegularDeveloper extends AbstractConsultant {

    public RegularDeveloper() {
        this.level = 2;
    }

    @Override
    protected void advise(int level) {
        System.out.println("RegularDeveloper helps to solve problem level " + level);
    }
}
```

Example: ExpressJS Middleware

Middleware function signature



```
function(req, res, next) { ... }
```



In-class activity

- Find another “example” of the State pattern
- Make a State pattern UML of it.
- Submit to Week 6
- 10 minutes