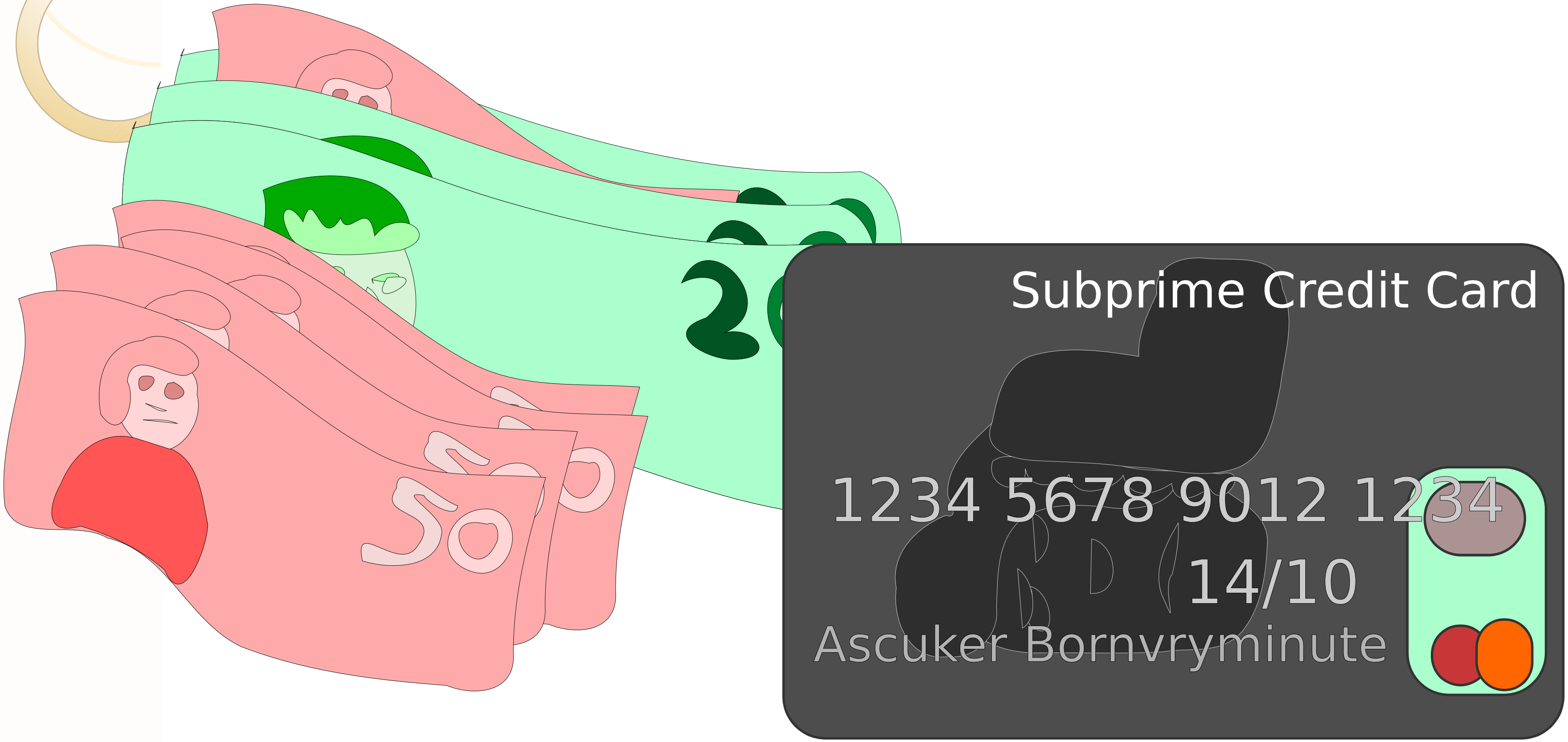
# Proxy Pattern



*the “real” thing*

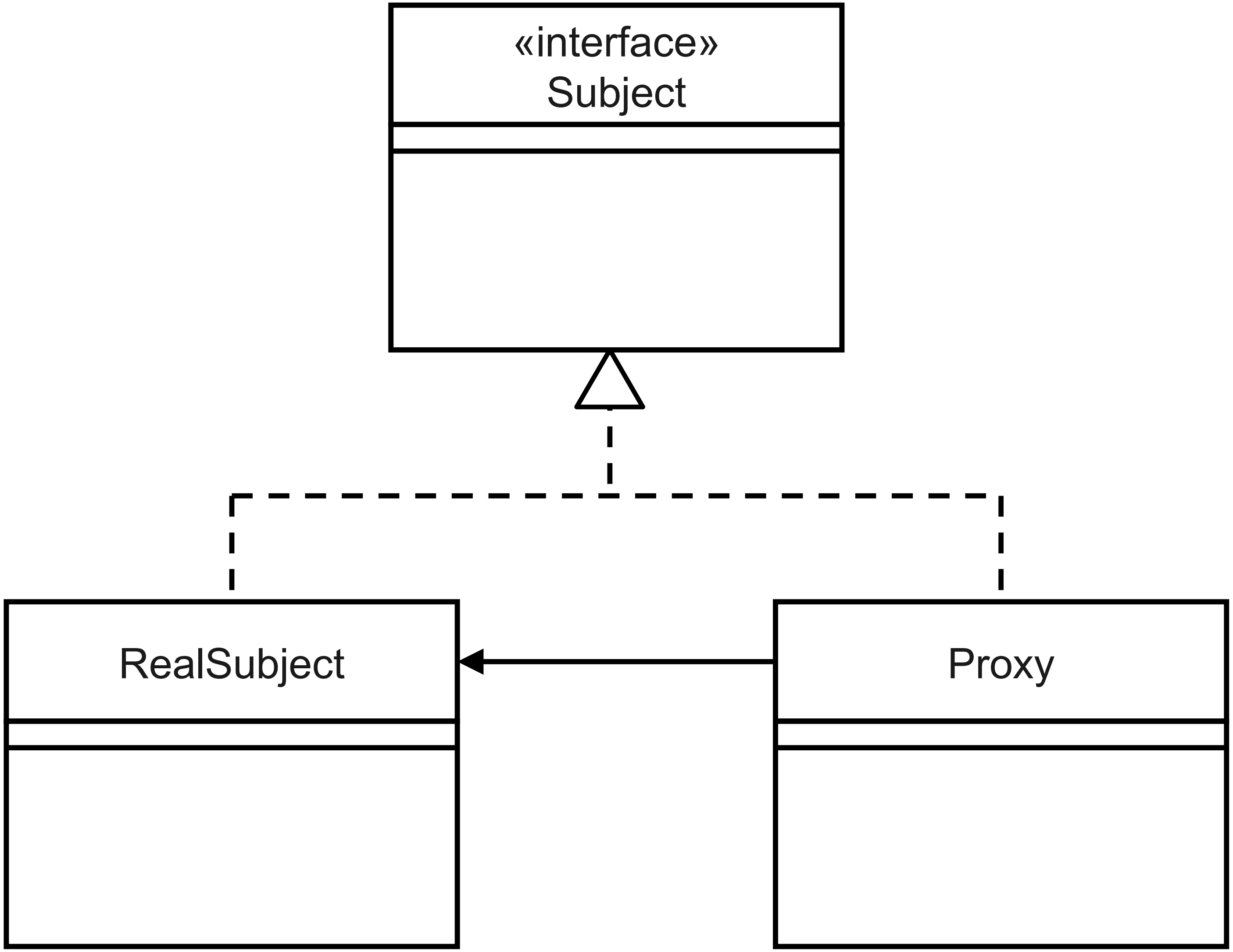
*proxy for the “real” thing*

# Proxy Pattern

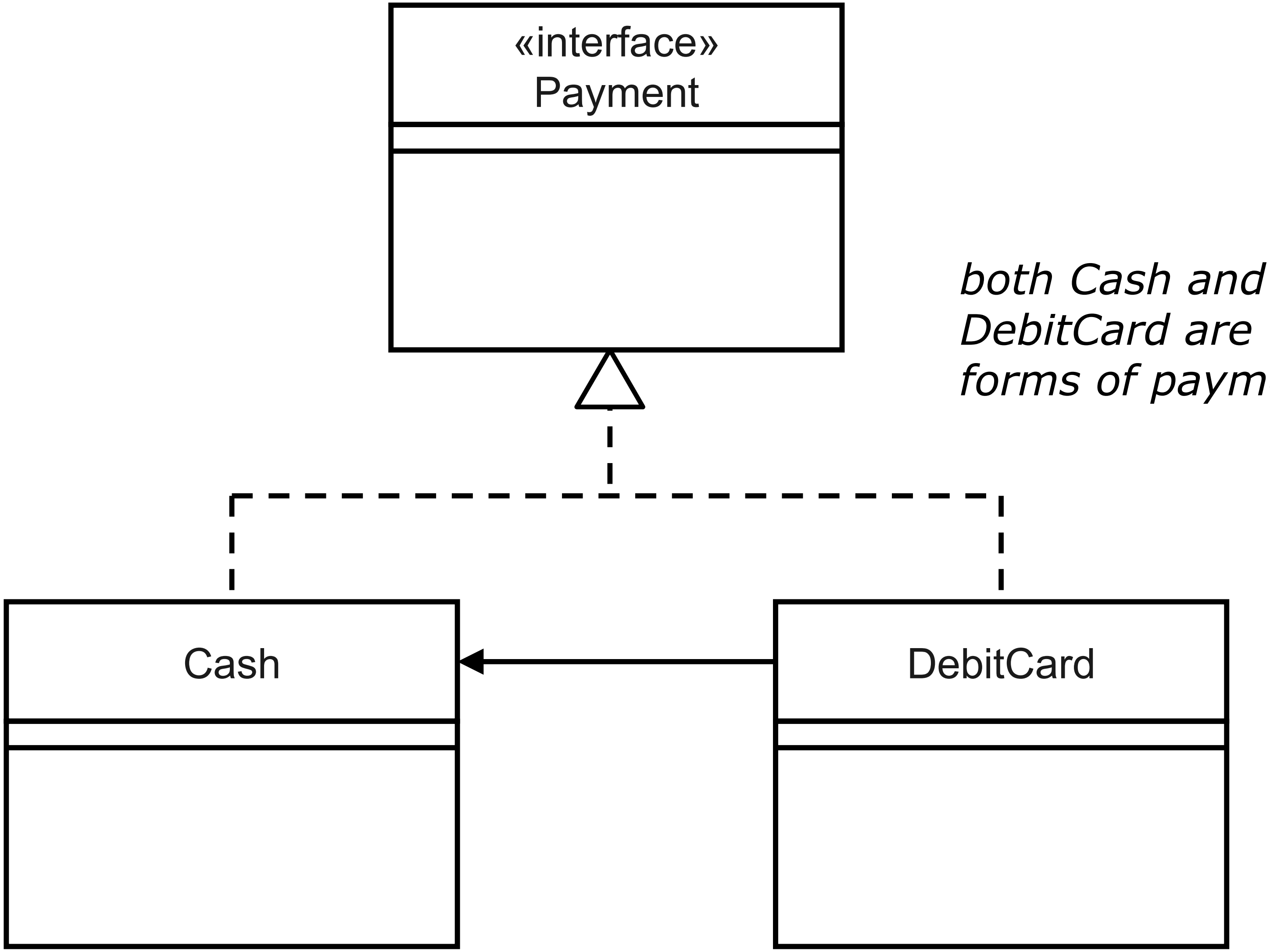
Design intent:

> "provide a surrogate or placeholder for another object to control access to it"

# Proxy Structure



# Proxy Example



*both Cash and DebitCard are forms of payment*

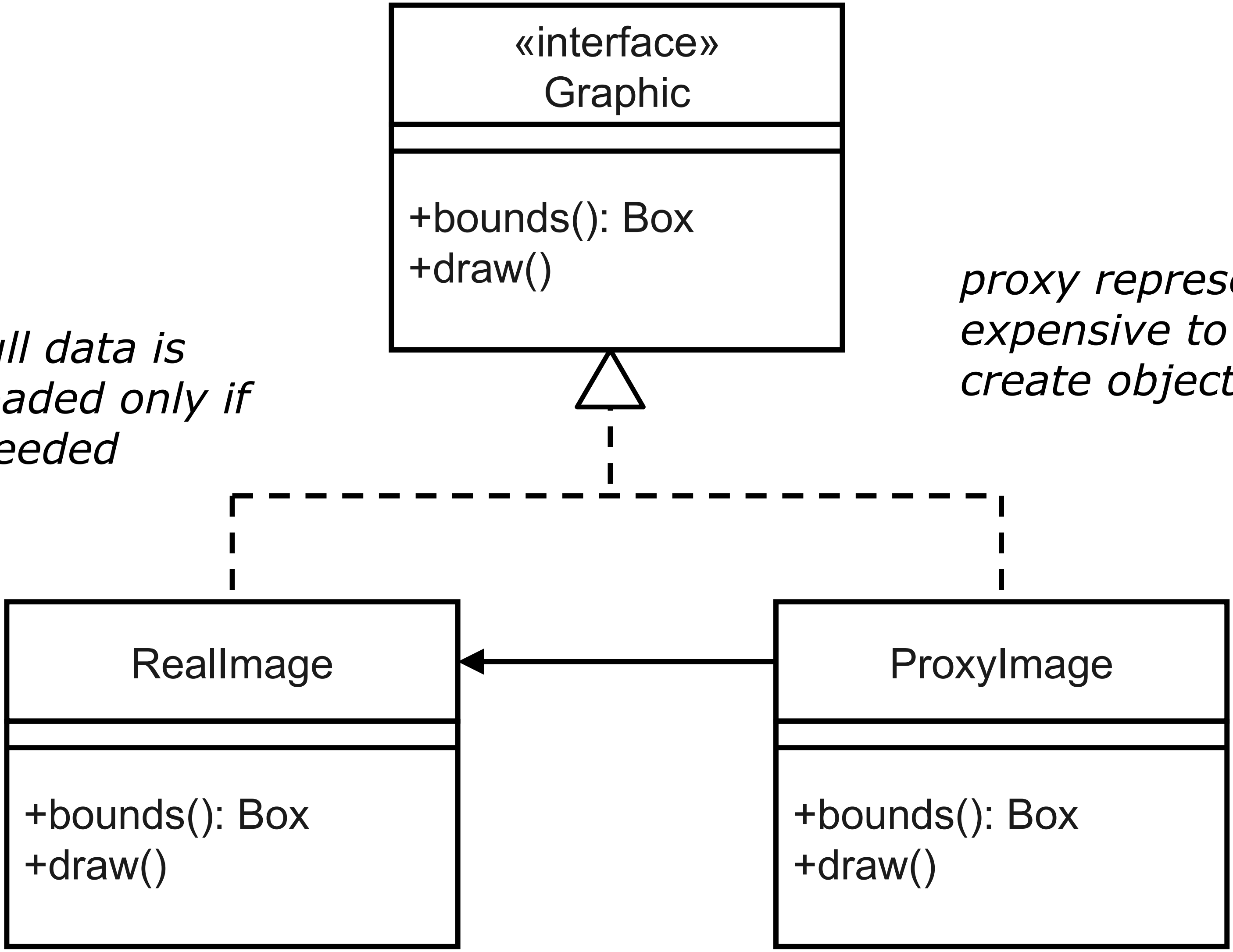
# Motivation

Use:

defer the full cost of creation and initialization of an object until we actually need to use it

- e.g., large image object and a proxy image

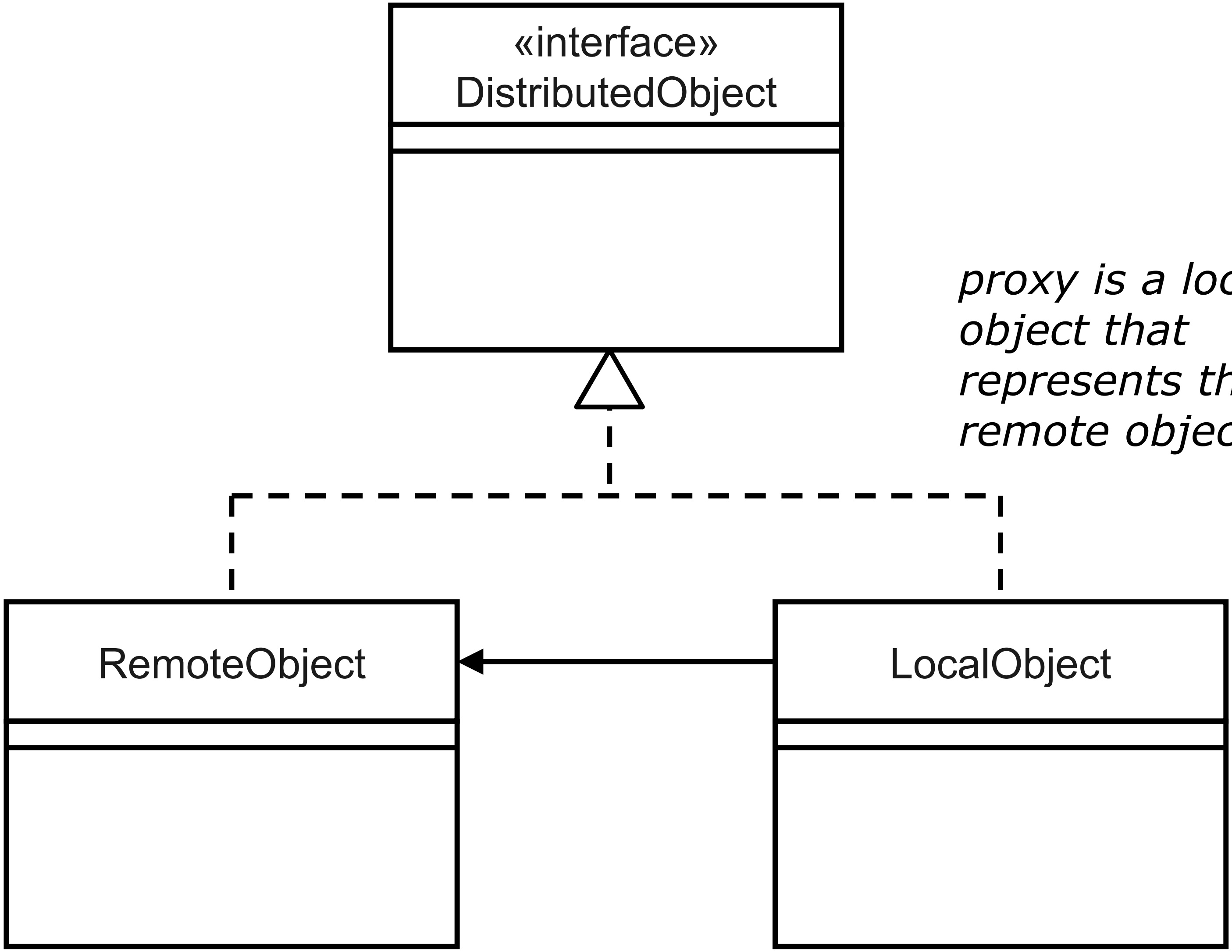
# Virtual Proxy Example



*full data is loaded only if needed*

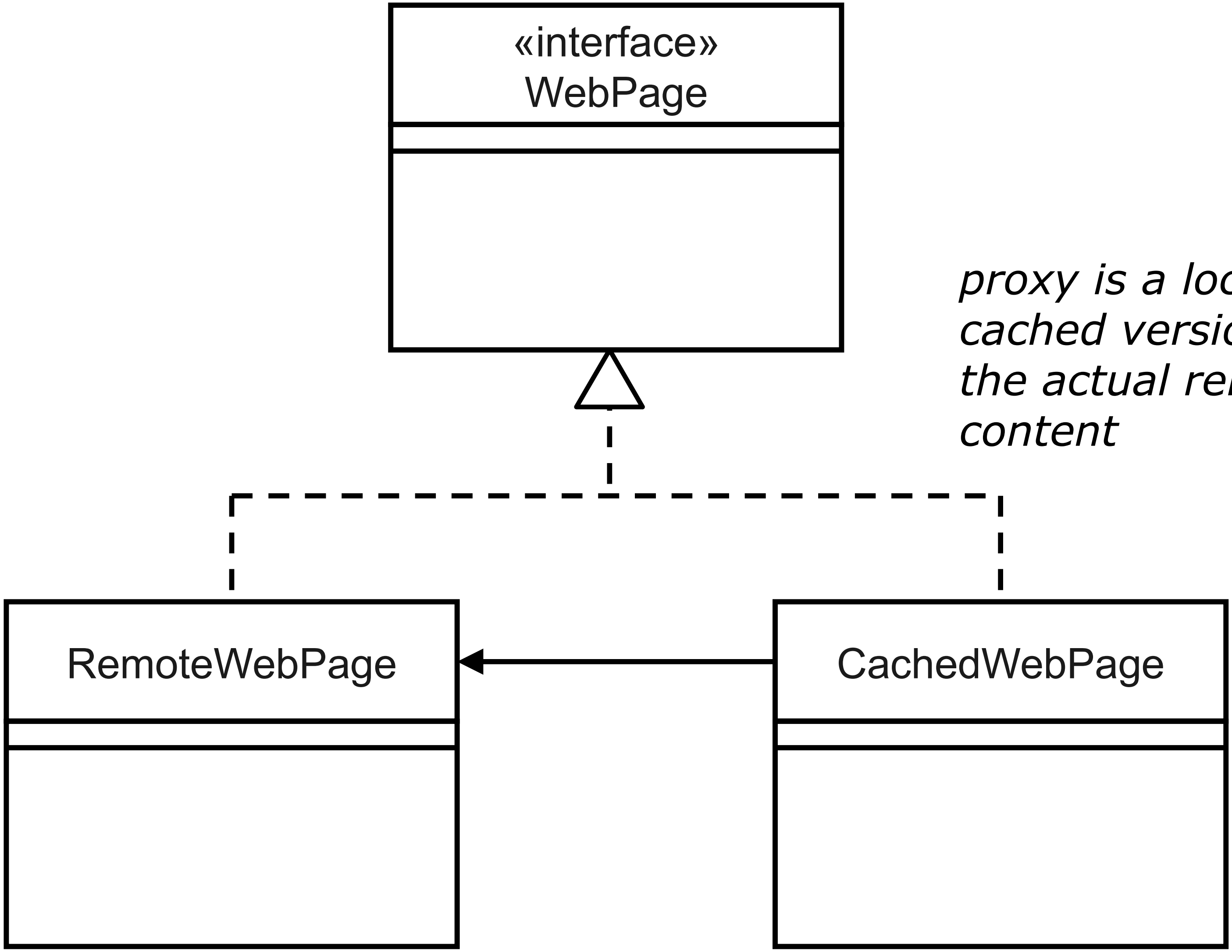
*proxy represents expensive to create object*

# Remote Proxy Example



*proxy is a local object that represents the remote object*

# Caching Proxy Example



*proxy is a locally cached version of the actual remote content*

# Facade Pattern

# State Pattern

# Problem

How to code a state model?

Example:

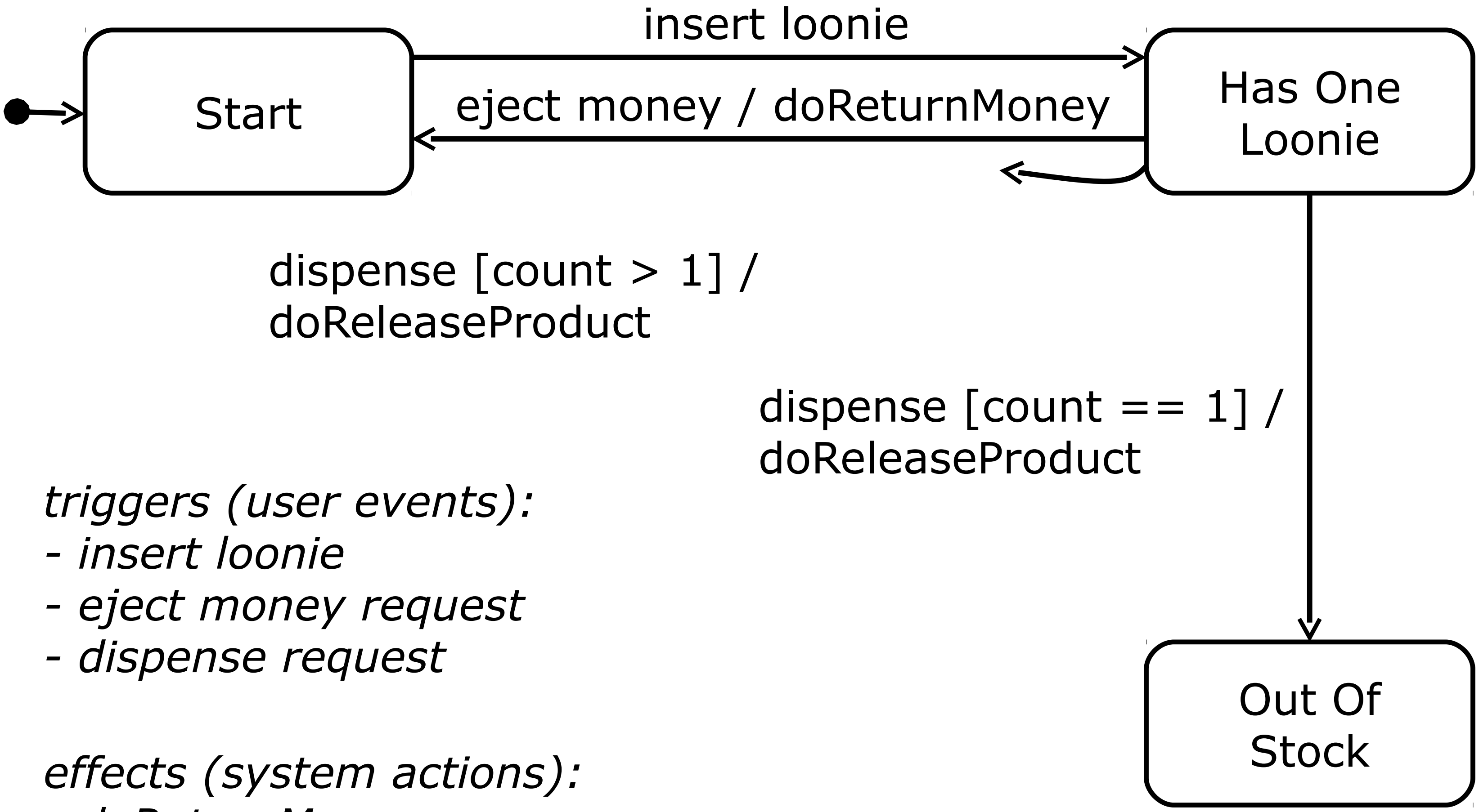
simple pop vending machine (single product)

insert loonie, press dispense button, get a pop

could eject to return money

machine has a limited supply

# Simple Pop Machine State Model


insert loonie
Start
eject money / doReturnMoney
Has One Loonie
dispense [count > 1] / doReleaseProduct
dispense [count == 1] / doReleaseProduct
Out Of Stock

*triggers (user events):*
*- insert loonie*
*- eject money request*
*- dispense request*

*effects (system actions):*
*- doReturnMoney*
*- doReleaseProduct*

# Pop Machine Class

```
public class PopMachine {
    …
    public PopMachine( int count ) {
        …
    }

    // handle user events …

    public void insertLoonie() {
        …
    }
    public void returnMoney() {
        …
    }
    public void dispense() {
        …
    }
    …
}
```

# Attempt 1

```
public class PopMachine { // constants for states

    // all potential states
    private final static int START = 0;
    private final static int HAS_ONE_LOONIE = 1;
    private final static int OUT_OF_STOCK = 2;

    private int currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = START;
            this.count = count;
        } else {
            currentState = OUT_OF_STOCK;
            this.count = 0;
        }
    }
```

# Attempt 1

```
// handle insert loonie trigger
public void insertLoonie() {
    if (currentState == START) {
        System.out.println(
            “loonie inserted”
        );
        currentState = HAS_ONE_LOONIE;
    } else if (currentState == HAS_ONE_LOONIE) {
        System.out.println(
            “already have one loonie”
        );
    } else if (currentState == OUT_OF_STOCK) {
        System.out.println(
            “machine out of stock”
        );
    }
}

…
```

# Attempt 2

```
// type-safe enumeration idiom (Joshua Bloch)

final class State { // singleton objects for states
    private State() {}

    // all potential pop machine states
    // as singletons
    public final static State START =
        new State();
    public final static State HAS_ONE_LOONIE =
        new State();
    public final static State OUT_OF_STOCK =
        new State();

}
```

# Attempt 2

```
public class PopMachine {

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    …
```

# Attempt 3

```
// using Java 5 enum

enum State {
    START,
    HAS_ONE_LOONIE,
    OUT_OF_STOCK
}
```

# Attempt 3

```
public class PopMachine { // same code as before

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    …
```

# Attempt 3

```
// handle insert loonie trigger
public void insertLoonie() {
    if (currentState == State.START) {
        System.out.println(
            “loonie inserted”
        );
        currentState = State.HAS_ONE_LOONIE;
    } else if (currentState ==
            State.HAS_ONE_LOONIE) {
        System.out.println(
            “already have one loonie”
        );
    } else if (currentState ==
            State.OUT_OF_STOCK) {
        System.out.println(
            “machine out of stock”
        );
    }
}
```

```
// handle eject money trigger
public void ejectMoney() {
    if (currentState == State.START) {
        System.out.println(
            “no money to return”
        );
    } else if (currentState ==
            State.HAS_ONE_LOONIE) {
        System.out.println(
            “returning money”
        );

        doReturnMoney();
        currentState = State.START;
    } else if (currentState ==
            State.OUT_OF_STOCK) {
        System.out.println(
            “no money to return”
        );
    }
}
```

```
// handle dispense trigger
public void dispense() {
    if (currentState == State.START) {
        System.out.println(
            “payment required”
        );
    } else if (currentState ==
            State.HAS_ONE_LOONIE) {
        System.out.println(
            “releasing product”
        );

        doReleaseProduct();
        if (count > 0) {
            currentState = State.START;
        } else {
            currentState = State.OUT_OF_STOCK;
        }
    } else if (currentState ==
            State.OUT_OF_STOCK) {
        System.out.println(
            “machine out of stock”
        );
    }
}
```

```
    // machine actions

    // return inserted money
    private void doReturnMoney() {
        …
    }

    // release one pop
    private void doReleaseProduct() {
        …
        count--;
    }


    …
} // class PopMachine
```

# Example Use and Output

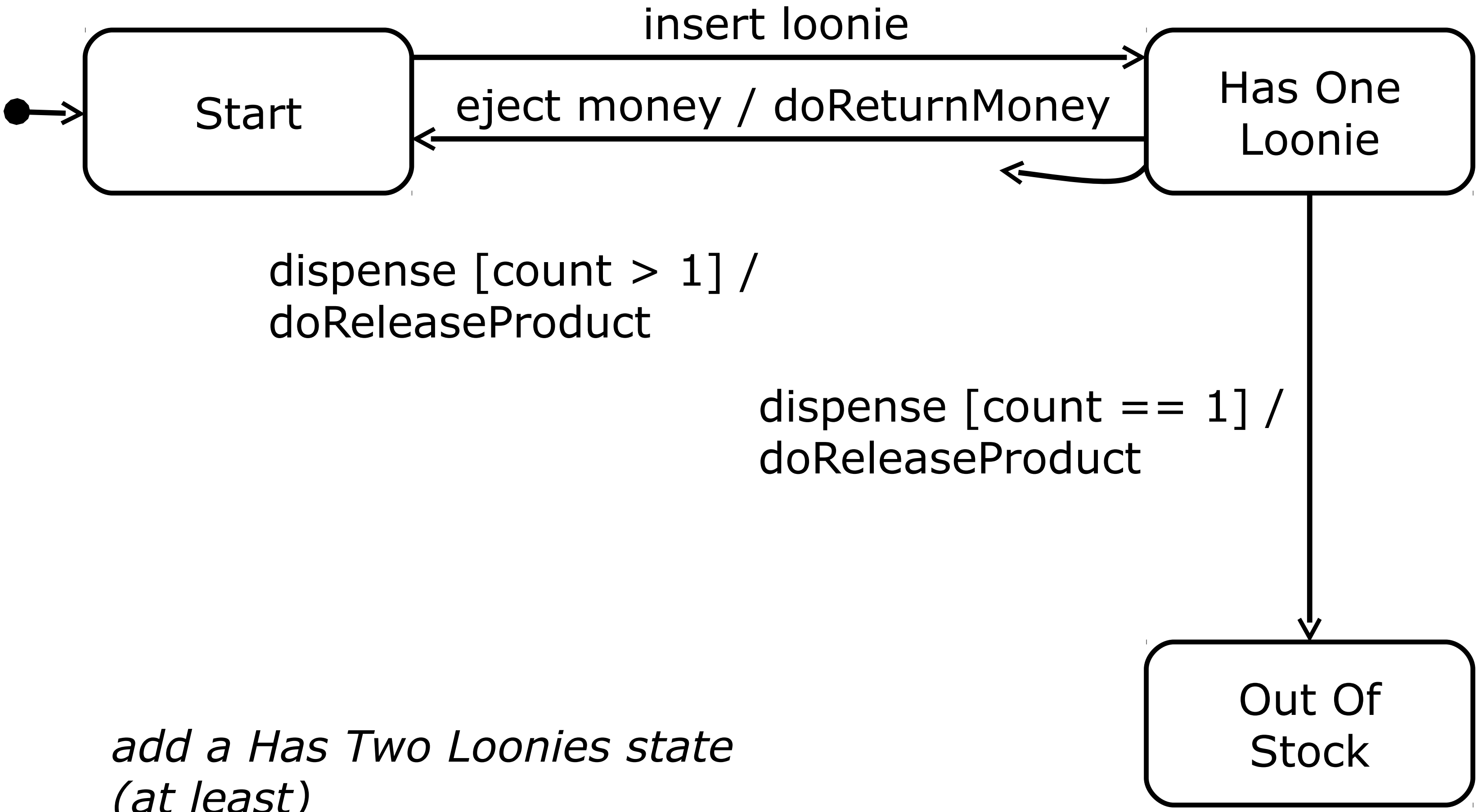
```
public static void main( String[] args ) {

    PopMachine popMachine = new PopMachine( 10 );

    // usual scenario
    popMachine.insertLoonie();                  loonie inserted
    popMachine.dispense();                      releasing product

    // no money, no sale
    popMachine.dispense();                      payment required

    // money returned, no sale
    popMachine.insertLoonie();                  loonie inserted
    popMachine.ejectMoney();                    returning money
    popMachine.dispense();                      payment required

}
```

# Change Request

Suppose:

pop machine now requires payment of two loonies

# What Needs to Change?


insert loonie
Start
eject money / doReturnMoney
Has One Loonie
dispense [count > 1] / doReleaseProduct
dispense [count == 1] / doReleaseProduct
Out Of Stock

*add a Has Two Loonies state (at least)*

# Change Request

Code changes:

need to change every trigger handling method to check for this new state

also add and adjust transitions

```
// add to insertLoonie, ejectMoney, dispense
// methods

… if (currentState == State.HAS_TWO_LOONIES) {
    …
} …
```

# Poor Design

Potential problems to address / refactor:

- blob class
  - gets increasingly larger over time
- long methods
  - forced to add cases to existing methods
  - could forget a case or introduce bugs
- conditional complexity
  - large conditional logic blocks
- passive data
  - state values not very “object-oriented”

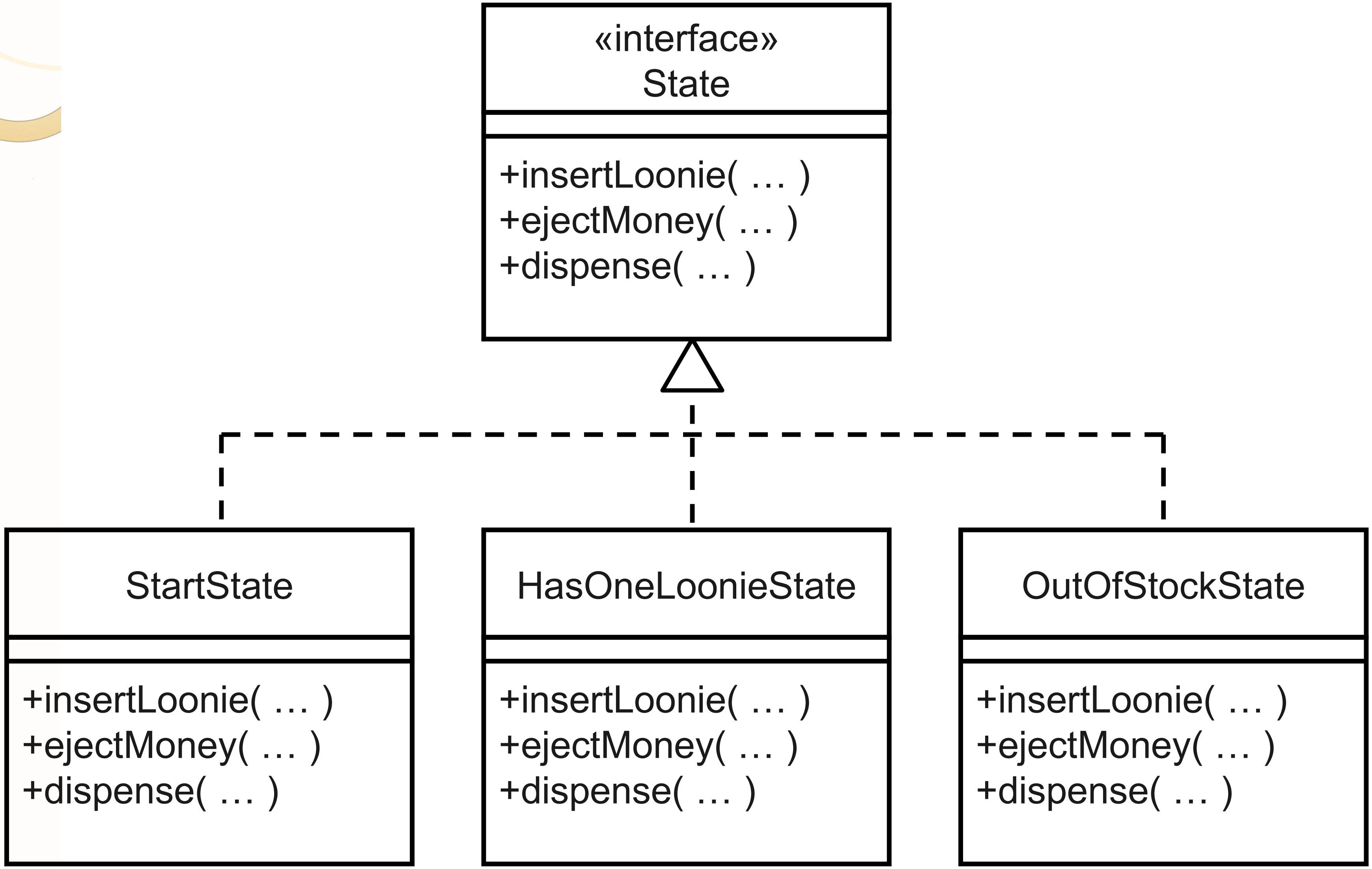
# State Pattern Approach

```
// common interface for pop machine state classes
interface State {

    // all potential triggers
    public void insertLoonie( PopMachine popMachine );
    public void ejectMoney( PopMachine popMachine );
    public void dispense( PopMachine popMachine );

}
```

*what if a new trigger is added?*

*redesign using state design pattern (state objects)*

# Pop Machine States


«interface»
State

+insertLoonie( … )
+ejectMoney( … )
+dispense( … )

StartState

+insertLoonie( … )
+ejectMoney( … )
+dispense( … )

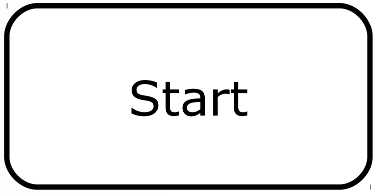
HasOneLoonieState

+insertLoonie( … )
+ejectMoney( … )
+dispense( … )

OutOfStockState

+insertLoonie( … )
+ejectMoney( … )
+dispense( … )

```
class StartState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( “loonie inserted” );

        popMachine.setState(
            popMachine.getHasOneLoonieState()
        );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( “no money to return” );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( “payment required” );
    }
}
```



Start

```
class HasOneLoonieState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( “already have one loonie” );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( “returning money” );

        popMachine.doReturnMoney();
        popMachine.setState(
            popMachine.getStartState()
        );
    }
```

Has One
Loonie

```
// class HasOneLoonieState continued

    public void dispense( PopMachine popMachine ) {
        System.out.println( “releasing product” );

        popMachine.doReleaseProduct();
        if (popMachine.getCount() > 0) {
            popMachine.setState(
                popMachine.getStartState()
            );
        } else {
            popMachine.setState(
                popMachine.getOutOfStockState()
            );
        }
    }
}
```

```
class OutOfStockState implements State {

    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( “machine out of stock” );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( “no money to return” );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( “machine out of stock” );
    }
}
```

Out Of
Stock

```
public class PopMachine {

    private State startState;
    private State hasOneLoonieState;
    private State outOfStockState;

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        // make the needed states
        startState = new StartState();
        hasOneLoonieState = new HasOneLoonieState();
        outOfStockState = new OutOfStockState();

        if (count > 0) {
            currentState = startState;
            this.count = count;
        } else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }
```

delegate behavior to current state

```
    public void insertLoonie() {
        currentState.insertLoonie( this );
    }

    public void ejectMoney() {
        currentState.ejectMoney( this );
    }

    public void dispense() {
        currentState.dispense( this );
    }

    public void setState( State state ) {
        currentState = state;
    }

    public int getCount() {
        return count;
    }

    // getters for state objects, machine actions, etc.
    …
}
```

# Example Use and Output

```
public static void main( String[] args ) {

    PopMachine popMachine = new PopMachine( 10 );

    // usual scenario
    popMachine.insertLoonie();
    popMachine.dispense();


    …
}

// popMachine.insertLoonie() delegates to
// insertLoonie() method of current state object
```

```
loonie inserted
releasing product
```

# State Pattern with Java enum

```
enum State {
  // each value is an instance of a singleton
  START { … },
  HAS_ONE_LOONIE { … },
  OUT_OF_STOCK { … };

  public abstract
  void insertLoonie( PopMachine popMachine );

  public abstract
  void ejectMoney( PopMachine popMachine );

  public abstract
  void dispense( PopMachine popMachine );
}
```

```
enum State {
  START {
    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( “loonie inserted” );

        popMachine.setState( HAS_ONE_LOONIE );
    }

    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( “no money to return” );
    }

    public void dispense( PopMachine popMachine ) {
        System.out.println( “payment required” );
    }
  },
  HAS_ONE_LOONIE {
    …
  },
  OUT_OF_STOCK {
    …
  };
  …
}
```

```
public class PopMachine {

    // no need to create state objects here

    private State currentState;
    private int count;

    public PopMachine( int count ) {
        if (count > 0) {
            currentState = State.START;
            this.count = count;
        } else {
            currentState = State.OUT_OF_STOCK;
            this.count = 0;
        }
    }

    // the rest as before
    …
}
```

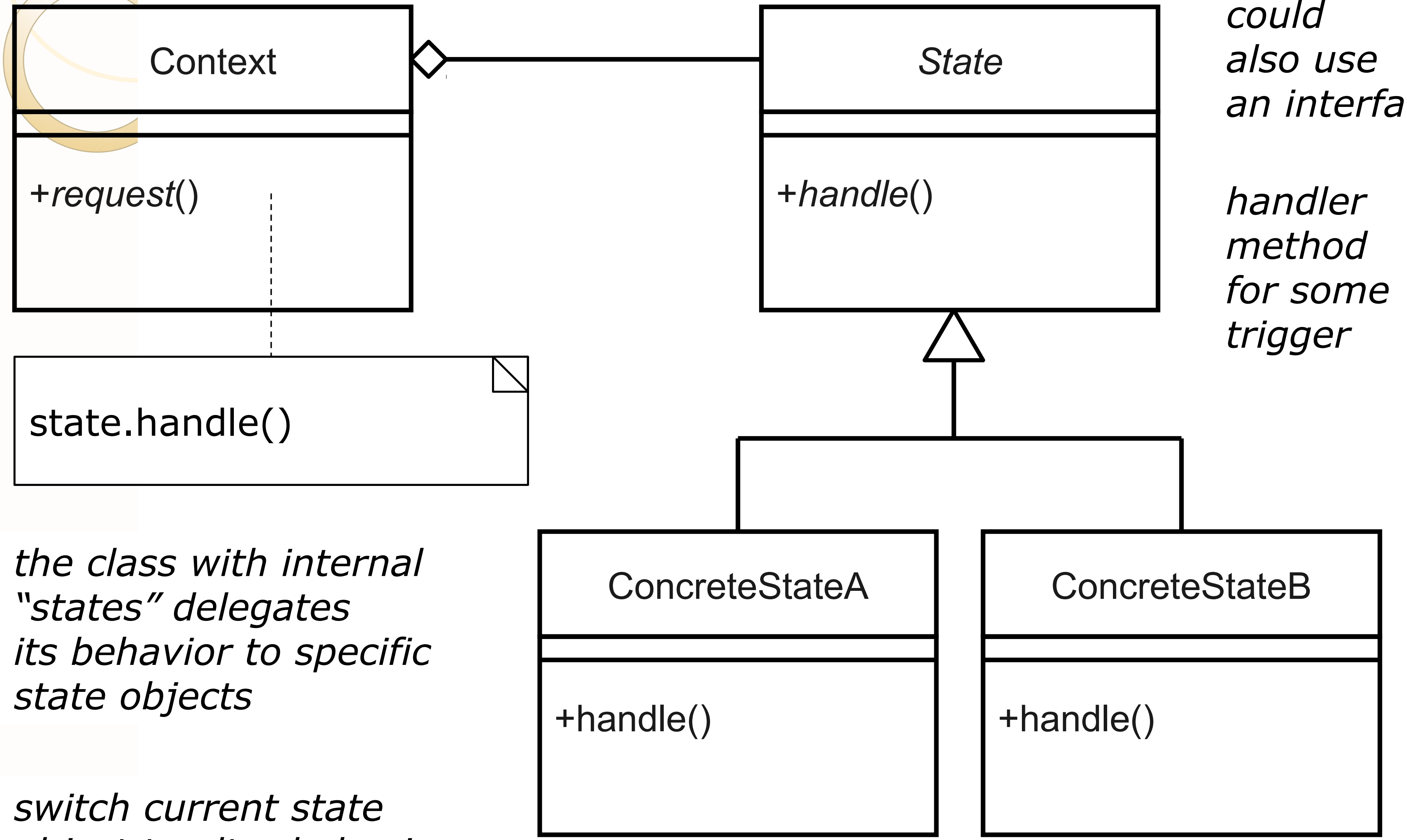
# State Pattern

Design intent:

“allow an object to alter its behavior when its internal state changes”

simplify operations with long conditionals that depend on the object’s state

# State Structure



Context

+*request*()

state.handle()

*State*

+*handle*()

ConcreteStateA

+handle()

ConcreteStateB

+handle()

*could also use an interface*

*handler method for some trigger*

*the class with internal “states” delegates its behavior to specific state objects*

*switch current state object to alter behavior*

# Decorator Pattern

# Decorator Pattern

Design intent:

“attach additional responsibilities to an object dynamically”

# Motivation

Use:

making user interface embellishments

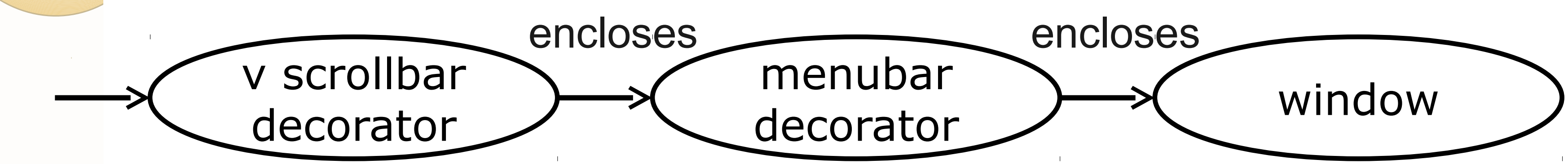
- e.g., dynamically adding “decorations” (menu bar, vertical scrollbar, horizontal scrollbar) to a basic window

don’t want too many new subclasses

use aggregation instead of inheritance

# Handling Requests

*single component "transparent" enclosures*



**draw method:**
encl.draw();
*draw itself*

**draw method:**
encl.draw();
*draw itself*

**draw method:**
*draw itself*

*this method should do everything this object "encloses" plus something extra*

*this method should do everything this object "encloses" plus something extra*

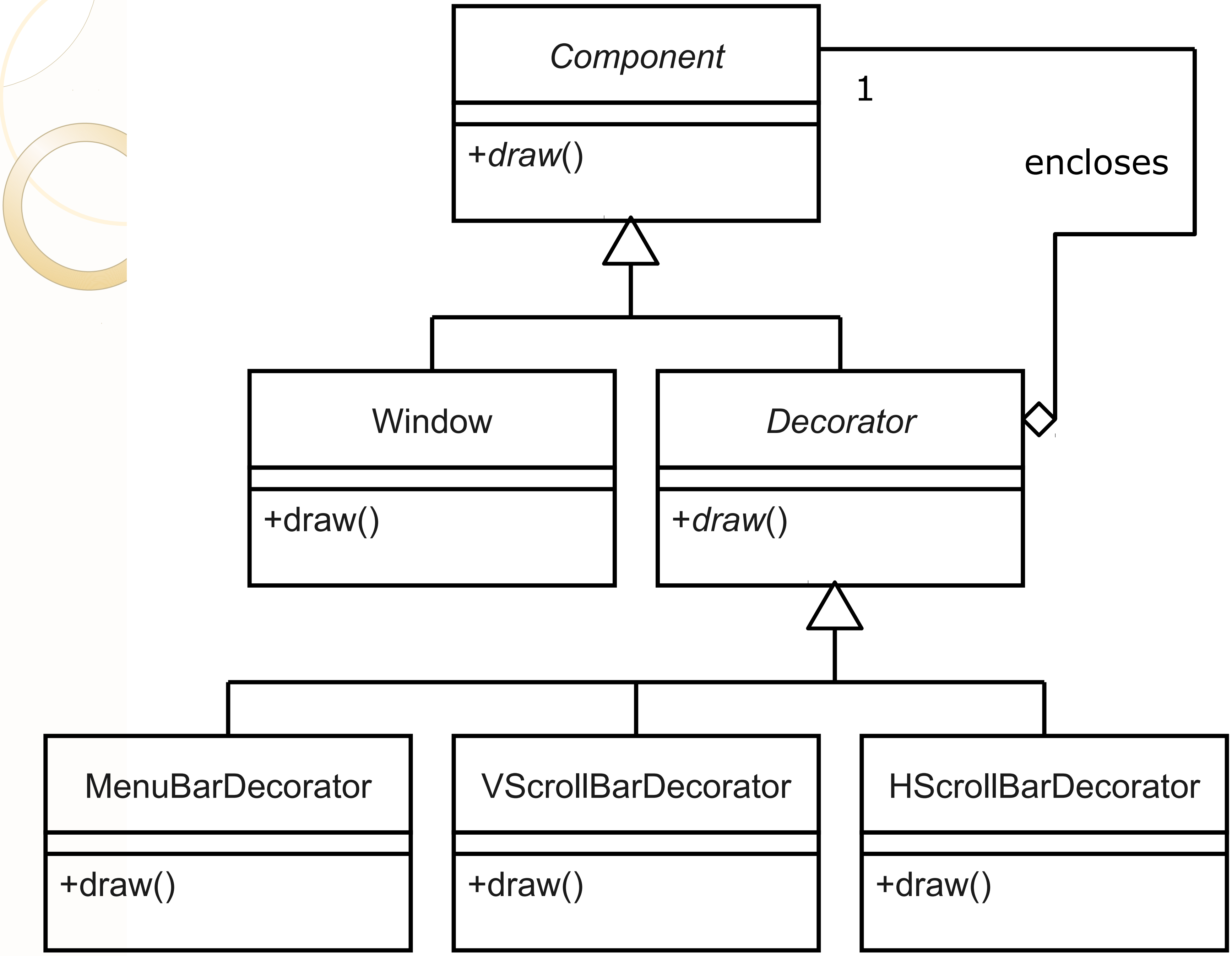
# “Transparent Enclosure”

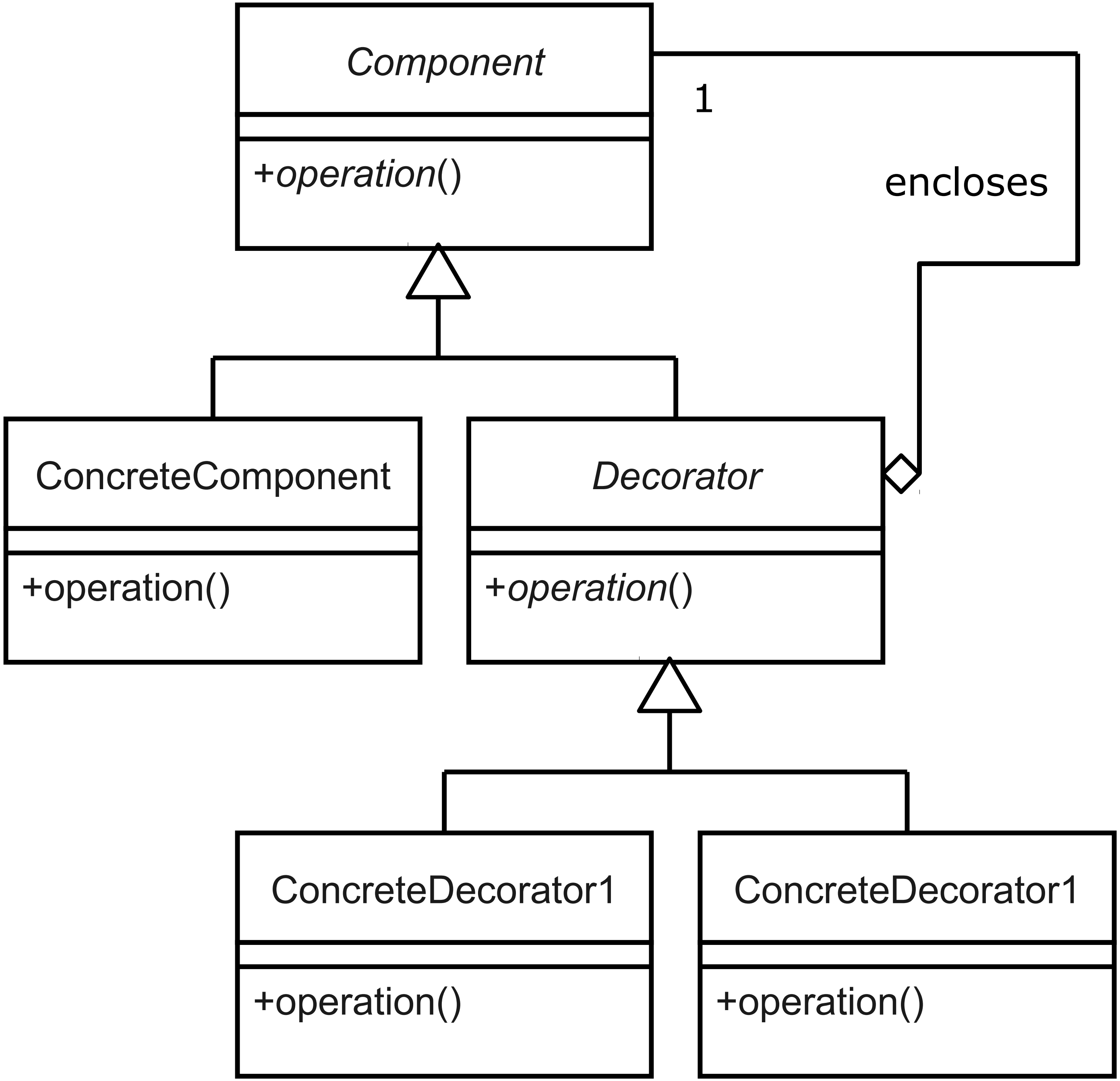
Idea:

single-component aggregation/composition

containing enclosure and contained component have compatible interfaces

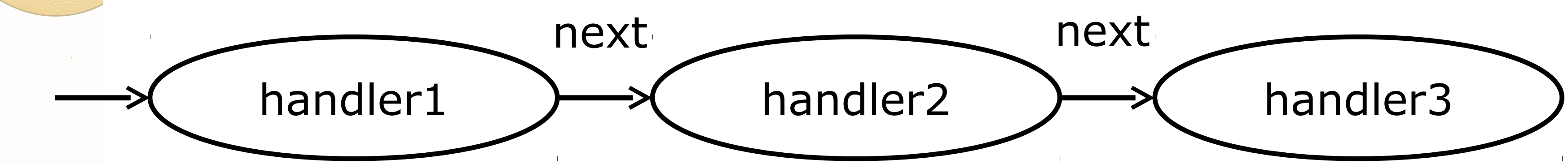
enclosure may partly delegate methods to component, and augment component behavior


Component
+draw()
1
encloses
Window
+draw()
Decorator
+draw()
MenuBarDecorator
+draw()
VScrollBarDecorator
+draw()
HScrollBarDecorator
+draw()


*Component*
+*operation*()
1
encloses
ConcreteComponent
+operation()
*Decorator*
+*operation*()
ConcreteDecorator1
+operation()
ConcreteDecorator1
+operation()

- **Chain of Responsibility Pattern**

# Handling Requests



**handle method:**

```
if (can handle) {
    handle request
} else {
    next.handle();
}
```

**handle method:**

```
if (can handle) {
    handle request
} else {
    next.handle();
}
```

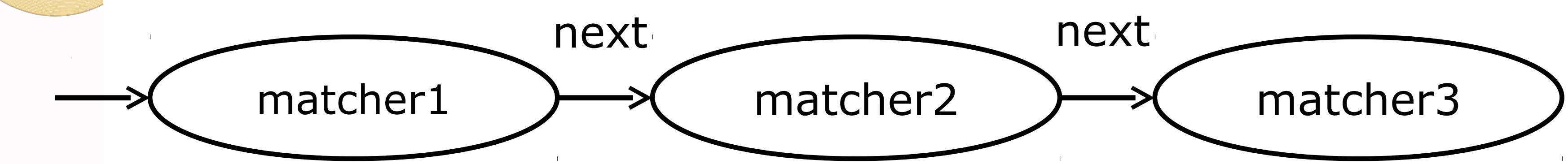
**handle method:**

```
if (can handle) {
    handle request
}
```

*request can be passed along and eventually handled by a handler (or not at all)*

*this handler not known ahead of time by the request initiator*

# Chain of Responsibility Example 1



**handle method:**

```
if (matches) {
    do match action
} else {
    next.handle();
}
```

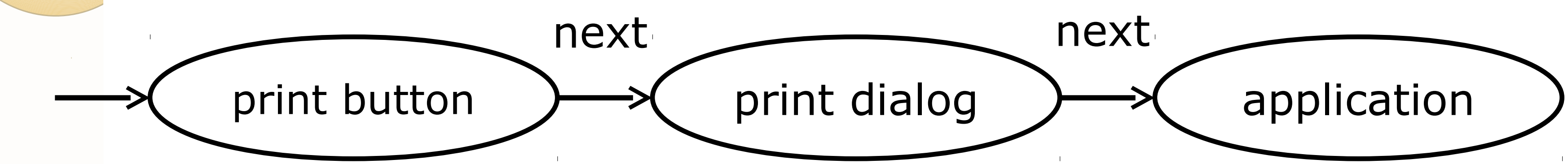
**handle method:**

```
if (matches) {
    do match action
} else {
    next.handle();
}
```

**handle method:**

```
if (matches) {
    do match action
}
```

# Chain of Responsibility Example



print button

next

print dialog

next

application

**handle method:**

```
if (has help) {
    show print
    button help
} else {
    next.handle();
}
```

**handle method:**

```
if (has help) {
    show print
    dialog help
} else {
    next.handle();
}
```

**handle method:**

```
if (has help) {
    show
    application help
}
```

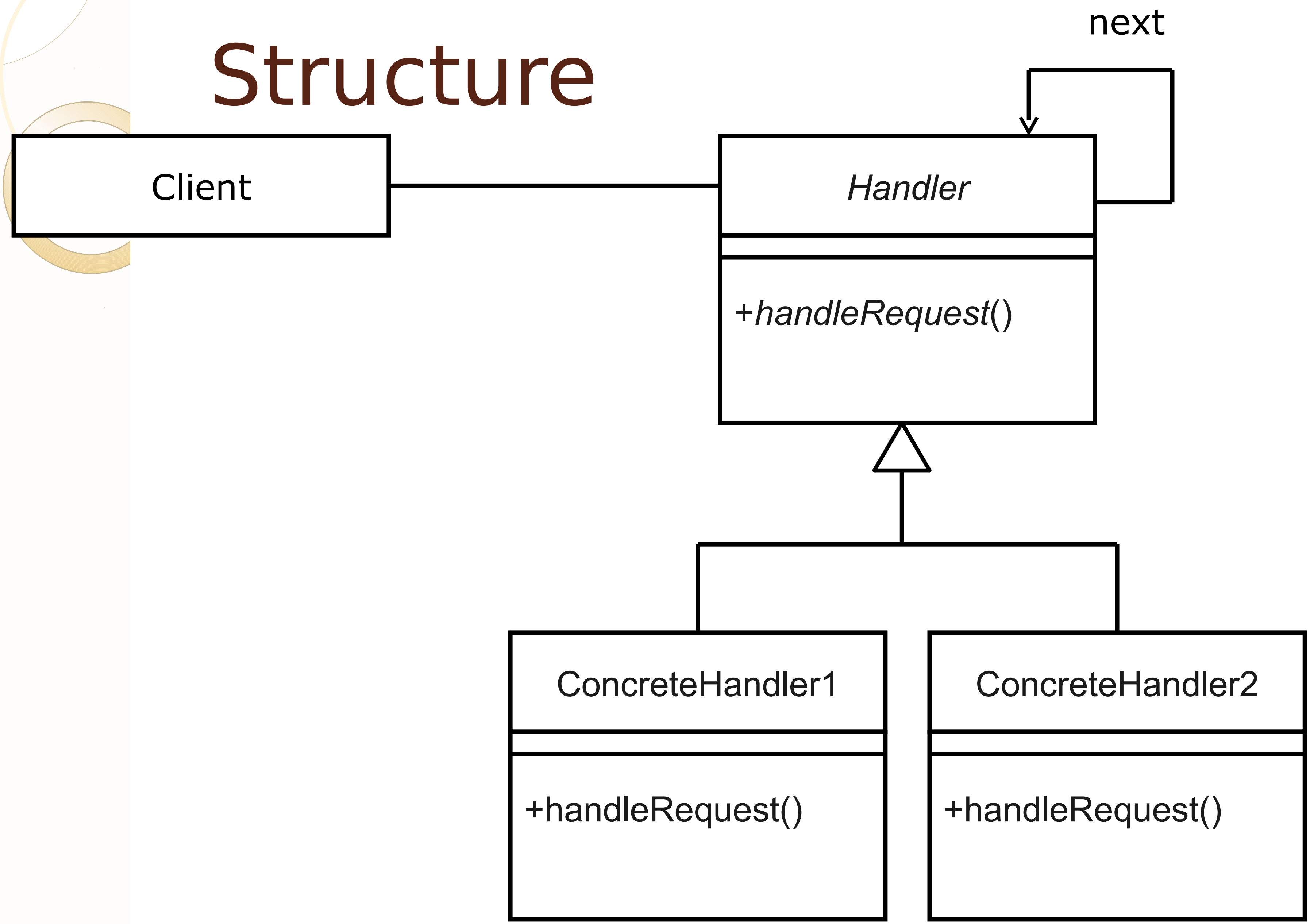
# Chain of Responsibility Pattern

Design intent:

> “avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request”

> “chain the receiving objects and pass the request along the chain until an object handles it”

# Structure


next
Client
*Handler*
+*handleRequest*()
ConcreteHandler1
+handleRequest()
ConcreteHandler2
+handleRequest()

# Consequences

Reduces coupling:

- frees an object from knowing which other object handles a request

- sender and receiver do not have direct knowledge about each other

- Design Principles

# Design Principles

Goals:

enhance flexibility under changing needs

improve reusability in different contexts

Note:

need balanced use of these guidelines

don’t overuse

# Open Closed Principle

“Classes should be open for extension, but closed for modification.”

Yes, we are …

OPEN

feel free to *extend* the classes and add new classes when needs change

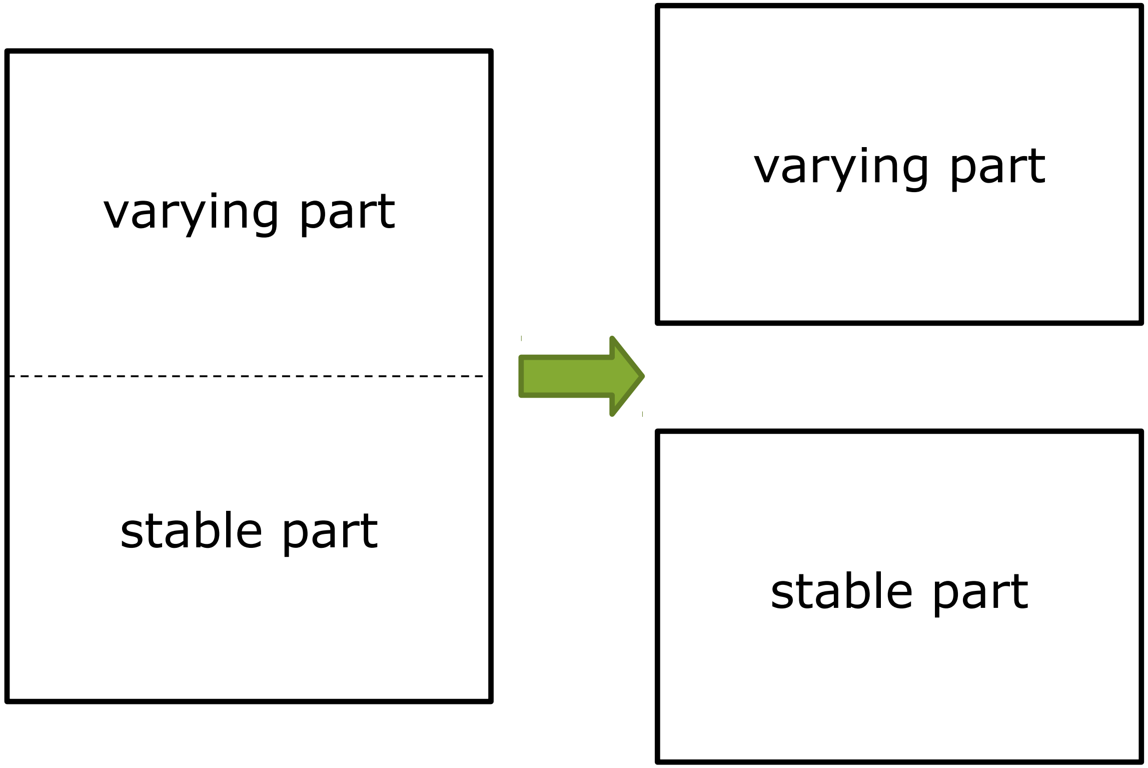
Sorry, we are …

CLOSED

existing classes are tested and work, so do not tinker with them

# Open Closed Principle

"Encapsulate what varies."
    separate and isolate into an object



*likely to change*

*use in*
*Decorator,*
*Template Method,*
*Factory,*
*State patterns*

# Open Closed Principle

What parts of a system are likely to vary?

- hardware dependencies
- business rules
- input and output formats
- user interface
- challenging design areas
- algorithms
- data structures
- …

# Dependency Inversion Principle

“Depend upon abstractions. Do not depend on concrete classes.”

Consumer

oncreteResource

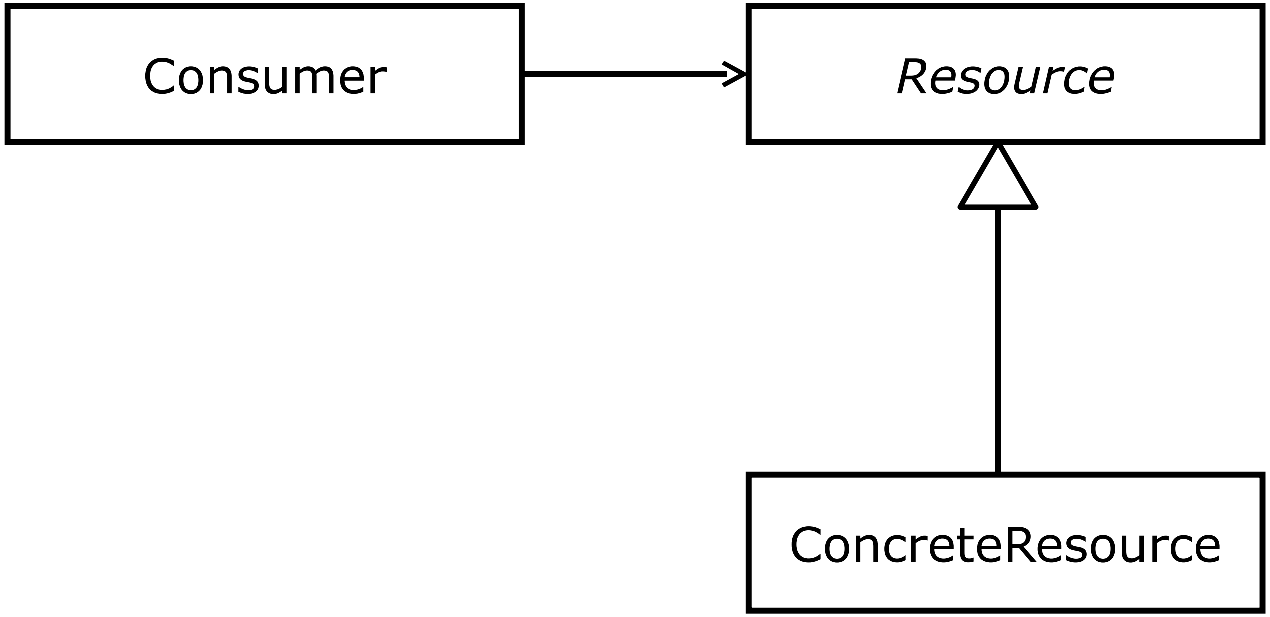
*in procedural programming, high-level modules depend on low-level modules*

*in object-oriented design, high-level classes refer to abstractions, and low-level classes depend upon these abstractions*

# Dependency Inversion Principle

“Depend upon abstractions. Do not depend on concrete classes.”

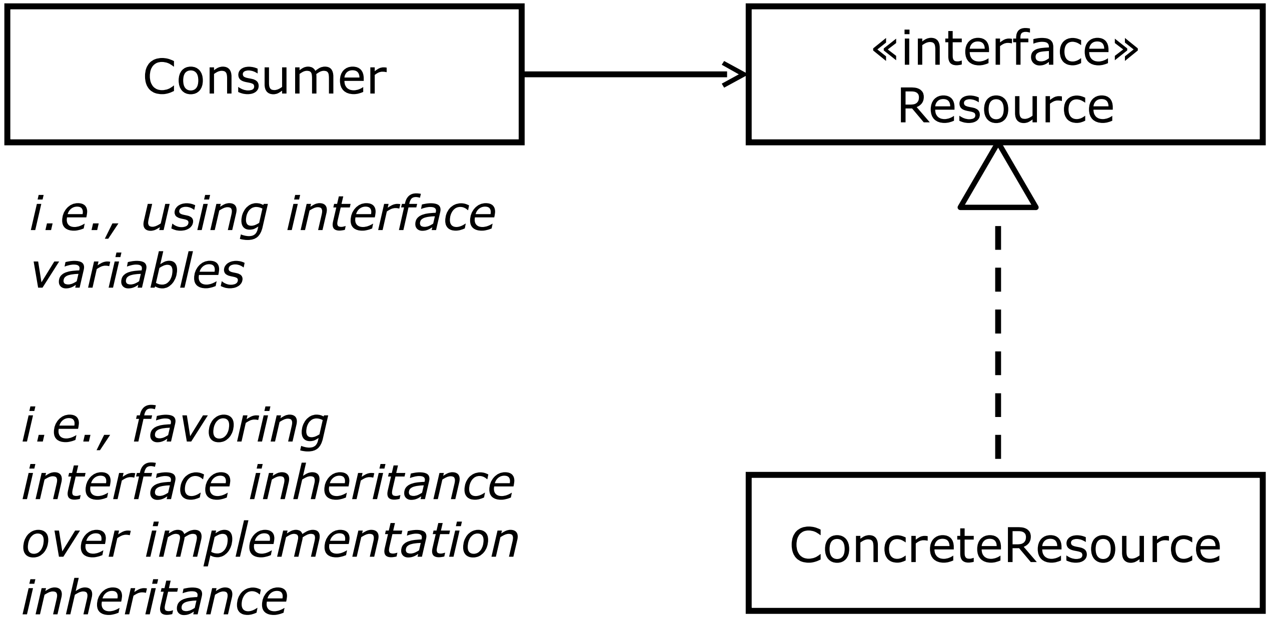
*abstract class*



*can plug in alternatives*

# Dependency Inversion Principle

“Program to interfaces, not implementations.”

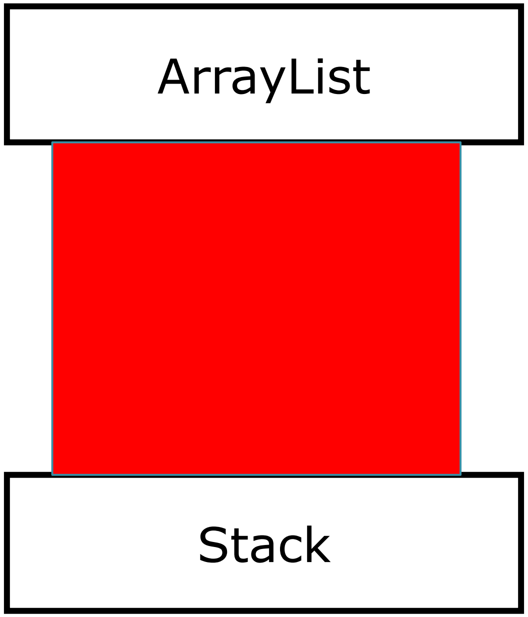


*i.e., using interface variables*

*i.e., favoring interface inheritance over implementation inheritance*

# Composing Objects

“Favor composing objects over *implementation* inheritance.”

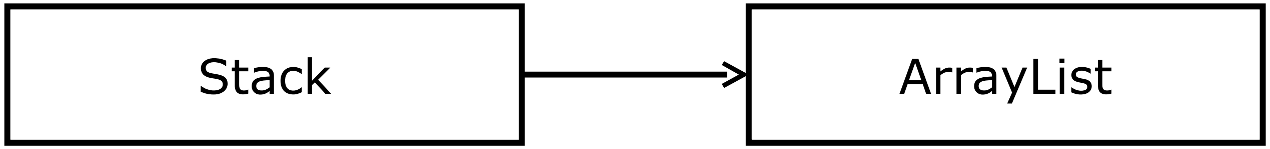


*tight coupling*

*GoF believe designers overuse implementation inheritance*

# Composing Objects

“Favor composing objects over *implementation* inheritance.”



*UML association, aggregation,*
*or composition*

*striving for loose coupling*

# Composing Objects

Implementation inheritance:

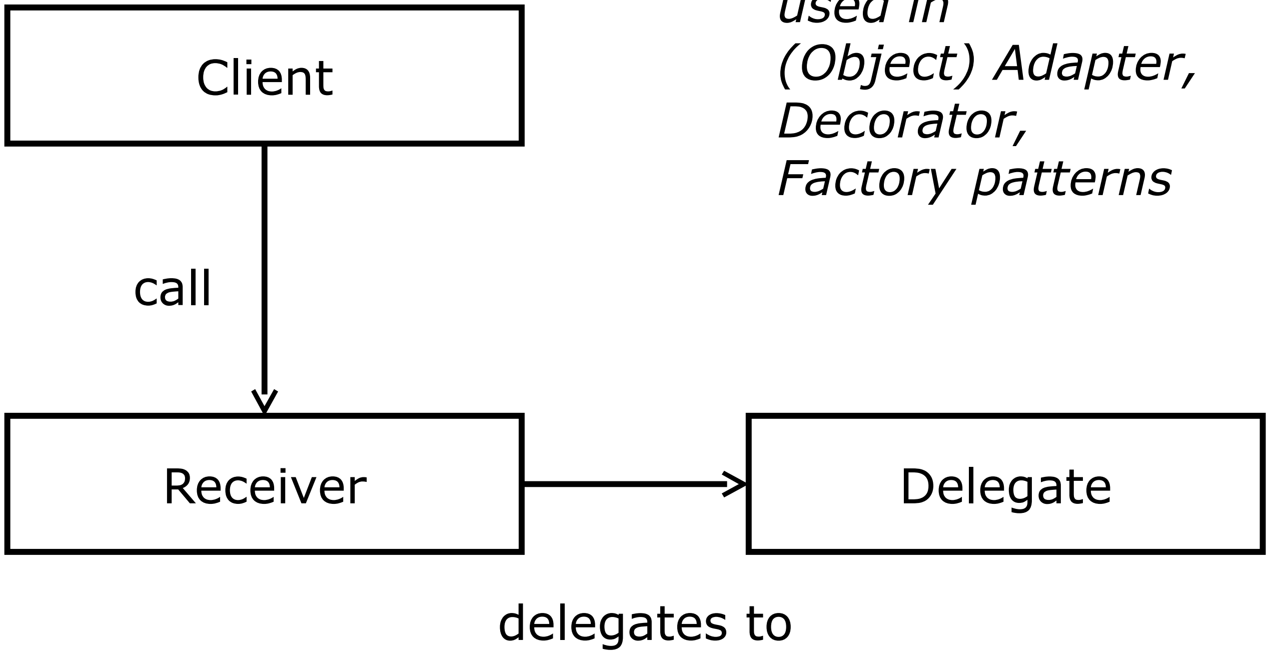
- compile-time dependency
- white-box reuse of superclass
- tight coupling, limits reuse of only subclass

Composing objects:

- run-time dependency (e.g., via injection)
- black-box “arms length” reuse via well defined interfaces
- delegation

# Composing Objects

Delegation technique:



*used in*
*(Object) Adapter,*
*Decorator,*
*Factory patterns*

*receiving object forwards to delegate object*

# Principle of Least Knowledge

“Only talk to your immediate friends.”

for an object, reduce the number of classes it knows about and interacts with

reduces coupling and changes cascading throughout the system

# Principle of Least Knowledge

“Law of Demeter”:
for method M of object O,
only call methods of the following objects

- object O itself
- parameters of method M
- any objects instantiated within method M
- direct component objects of object O

# Principle of Least Knowledge

"Law of Demeter":

avoid calling methods of objects returned by other methods (unless allowed by the law)

```
// couples this method to Preference class
Preference pref = user.getPreference();
pref.doSomething();

// equivalently
user.getPreference().doSomething();
```

i.e., "one dot only rule"

# More Information

Books:

Head First Design Patterns

- E. Freeman, E. Robson, B. Bates, and K. Sierra
- O’Reilly, 2004

# More Information

Books:

Design Patterns

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides
- Addison-Wesley, 1995

Patterns in Java

- M. Grand
- Wiley, 1998

# More Information

Links:

Source Making Design Patterns

- http://sourcemaking.com/design_patterns

Vince Huston Design Patterns

- http://www.vincehuston.org/dp/

# More Information

Links:

Speaking on the Observer Pattern

- http://www.javaworld.com/javaqa/2001-05/04-qa-0525-observer.html

Learn How to Implement the Command Pattern in Java

- http://www.javaworld.com/javatips/jw-javatip68.html

# More Information

Links:

Design Principles and Design Patterns

- http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Law of Demeter

- http://www.ccs.neu.edu/home/lieber/LoD.html

Portland Pattern Repository

- http://c2.com/ppr/