# Design Principles Behind Design Patterns

Chuck Allison

# Objective

- Investigate some *timeless design principles*
- Observe how they are manifest in selected design patterns
  - as well as in other ways
- Look briefly at some software patterns that aren't specifically related to design
  - Architectural
  - Organizational

# Agenda

- The History and Impact of Design Patterns in Software Development
- The Anatomy of a Design Pattern
- Design Principles
  - and how they drive pattern creation
- Other Types of Patterns

# Context

- The Design Patterns "movement" has revolutionized software development
- Most everyone is familiar with the "Gang of Four" book
  - *Design Patterns: Elements of Reusable Object-oriented Software*, Gamma et al
- Terms like *Strategy* and *Adapter* have crept into our technical vocabularies

# The Impact of Design Patterns

- They constitute a catalog of reusable software artifacts
  - They apply in many situations
  - They share design expertise
- They give us a useful, shared vocabulary
  - "It looks like we need a Composite here"
- Most importantly, they improve our thinking about design
  - Because they adhere to sound principles

# 23 is **not** a Magic Number

- The 23 GoF design patterns are *protypical*, but **not** *sacrosanct*
- There are many more design patterns
  - New ones still emerge
- There are other, non-design software-related patterns
  - Architectural, organizational, testing, refactoring, process

# Strange But True…

The enormous success of design patterns is a testimonial to the commonality seen by object programmers. The success of the book *Design Patterns*, however, has stifled any diversity in expressing these patterns.

-- Kent Beck

# A Quick Pattern History

- Developers have long sought a way of preserving and communicating *design decisions*
- The Hillside Group
  - Smart People found inspiration in Christopher Alexander's patterns of building architecture
  - *The Timeless Way of Building*, 1979
- Early publications
  - Coplien's *Advanced C++*, GoF, Coplien's *Software Patterns*

# Anatomy of a Design Pattern

- Summary
  - State what is trying to be accomplished in succinct, high-level terms
- Problem
  - Describes the context for the problem, the forces that cause the "dilemma", and the negative consequences of not resolving those forces
- Solution
  - Describes the structure and behavior of the solution. Shows how forces are resolved. Include sketches as needed.

# What a Design Pattern *Is*

- A solution to a design problem in a given context
- It *balances the forces* in a given context to achieve a design goal
- Design patterns are independent of programming language and platform
  - They can be manifest in many ways

# What a Design Pattern is *Not*

- Just a diagram
  - sketches help, but different patterns have identical sketches
  - sketches illustrate forces and their resolution in a general manner
- Code
- A step-by-step recipe
  - they're more of a heuristic
- A Panacea

# Patterns and Principles

- Patterns emerge from *principles*
  - "Program to an Interface, not an Implementation"
  - "Minimize coupling; maximize cohesion"
  - "Don't Repeat Yourself"
- The principles have long been with us
  - Long before design patterns were around
  - It takes effort to master them
  - Studying and using patterns helps

# DESIGN PRINCIPLES

# A Fundamental Principle

- *Separate things that vary from things that stay the same*
- The benefit is obvious:
  - The static part is not affected by changes in other related components
- Not always adhered to by developers!
- Manifests itself in different ways…

# Commonality vs. Variability
## *Take 1 – Designing a Function*

**The Abstraction**: A *function* encapsulates a group of related operations at the *statement level*.

| What Stays the Same | Coupling Mechanism | What Changes |
|---|---|---|
| Procedure Logic | Function Parameters | Input data |

```
int f(int n, string data) {…}
```

# Commonality vs. Variability
## *Take 2 – Designing a Class Hierarchy*

**The Abstraction**: A *top-level class* defines an interface. Subclasses *implement* the interface.

| **What Stays the Same** | **Coupling Mechanism** | **What Changes** |
| --- | --- | --- |
| The Interface | Inheritance, subtype polymorphism | Implementations of individual methods |

A related design pattern: **Template Method**

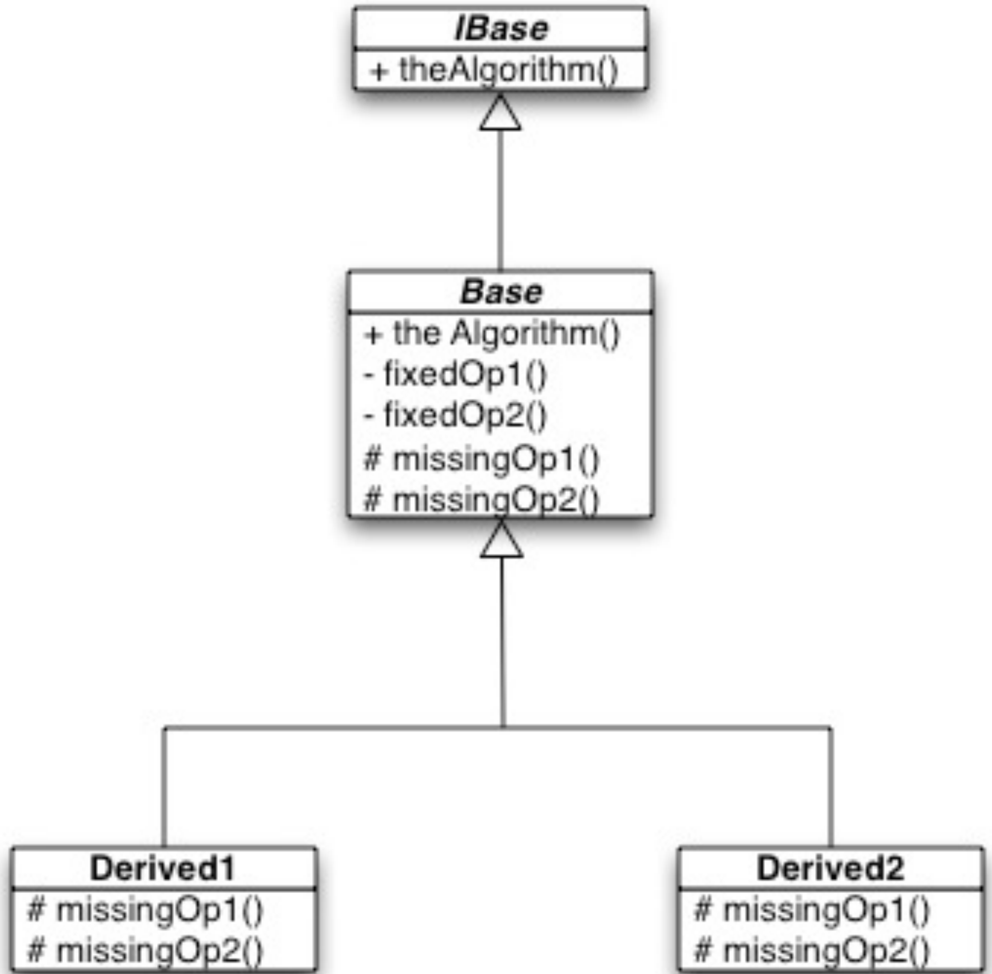
# Template Method

- Used in some *multi-step* algorithms
- The top-level, public method calls upon other methods for each step
- Some steps don't vary, some do
- The parts that vary are *separated out* into *polymorphic* methods
  - overridden by subclasses
- The top-level method is *non-polymorphic*
  - it *controls* the entire process

# Template Method Description

- Summary
  - Define the *skeleton* of an algorithm, deferring some steps to subclasses. Subclasses can customize an algorithm without changing the overall algorithm structure.
- Problem
  - You want to control the steps of the algorithm, but some of the steps vary. You want to factor common behavior among subclasses into the base class to avoid duplication. You want to allow subclasses to customize behavior in a controlled way.
- Solution
  - Provide a fixed interface for clients, but have the implementation call upon *hidden*, polymorphic methods as needed.

# Template Method Class Sketch



# Java Code

```
abstract class Base implements IBase {
    public final void theAlgorithm() {
        fixedop1();
        missingop1();
        fixedop2();
        missingop2();
    }
    final void fixedop1() {
        System.out.println("fixedop1");
    }
    final void fixedop2() {
        System.out.println("fixedop2");
    }
    protected abstract void missingop1();
    protected abstract void missingop2();
};
```

# Java Code (continued)

```
class Derived extends Base {
    protected void missingop1() {
        System.out.println("missingop1");
    }
    protected void missingop2() {
        System.out.println("missingop2");
    }
};

class Skeleton {
    public static void main(String[] args) {
        Derived d = new Derived();
        d.theAlgorithm();
    }
}
```

# Commonality vs. Variability
## *Take 3 – Designing Families of Implementations*

**The Abstraction**: A *client class* depends on other classes for part of its *behavior*. A specific implementation can be selected on demand.

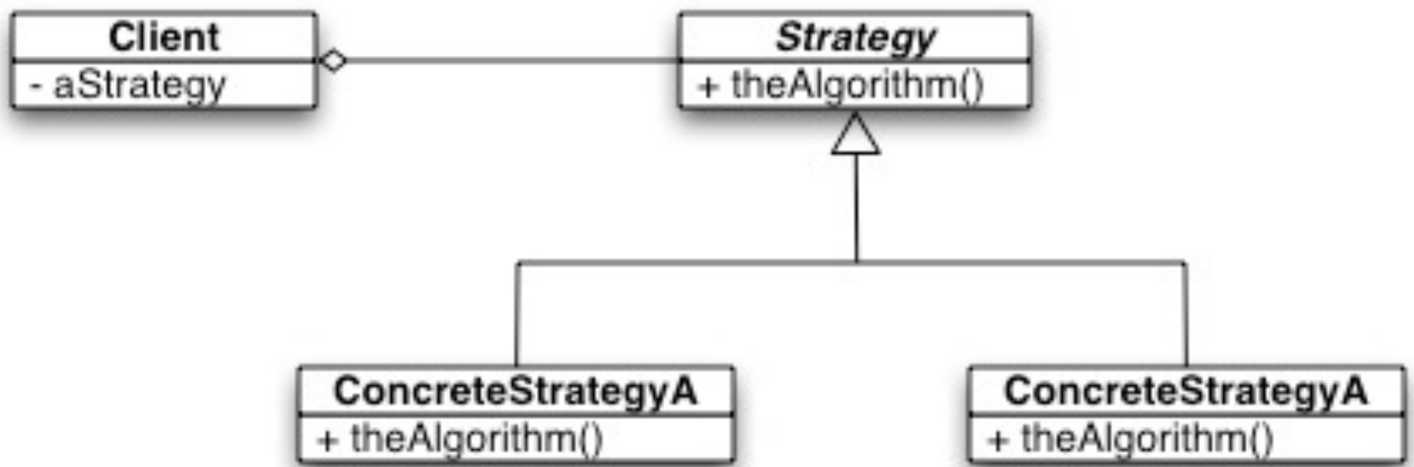
| What Stays the Same | Coupling Mechanism | What Changes |
|---|---|---|
| High-level Solution Structure | Separate Class Hierarchies | Implementation of solution facets |

Related design patterns: **Strategy**, **Bridge**, (most…)

# Strategy Description

- Summary
  - Define an interchangeable family of algorithms. Let implementations vary independently from clients.
- Problem
  - A client may need variants of an algorithm, configurable at runtime. Without encapsulating the related variants, significant amounts of code must change when an selected implementation changes.
- Solution
  - Define an interface for the family of algorithms. Encapsulate each variant in a subclass. Clients keep polymorphic references to implementations.

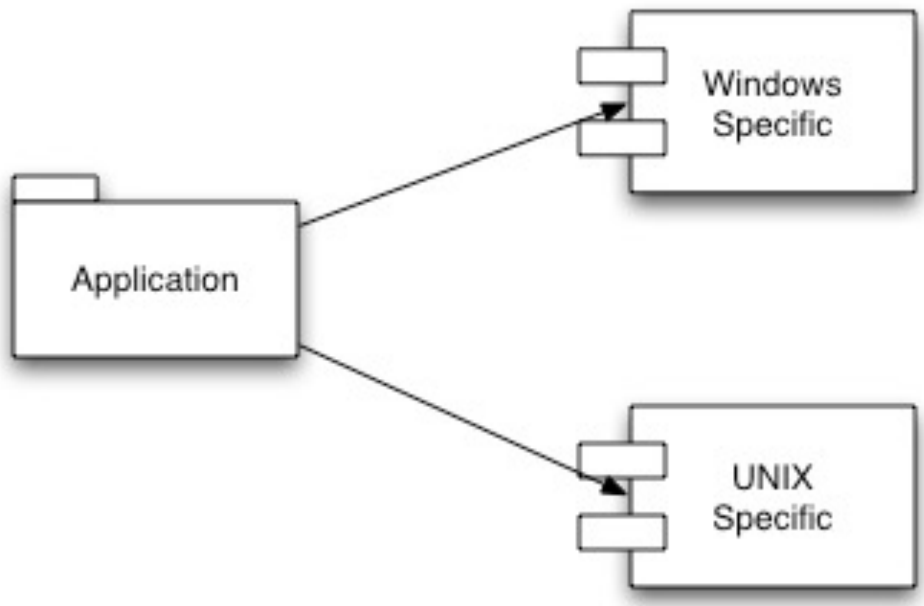
# Strategy Class Sketch



# Compile-time Applications of **Strategy**

- Isolating platform-specific code
- C++ Template Idioms
  - Traits
  - Policies
- C++ Container Adaptors
- All use *implicit* interfaces

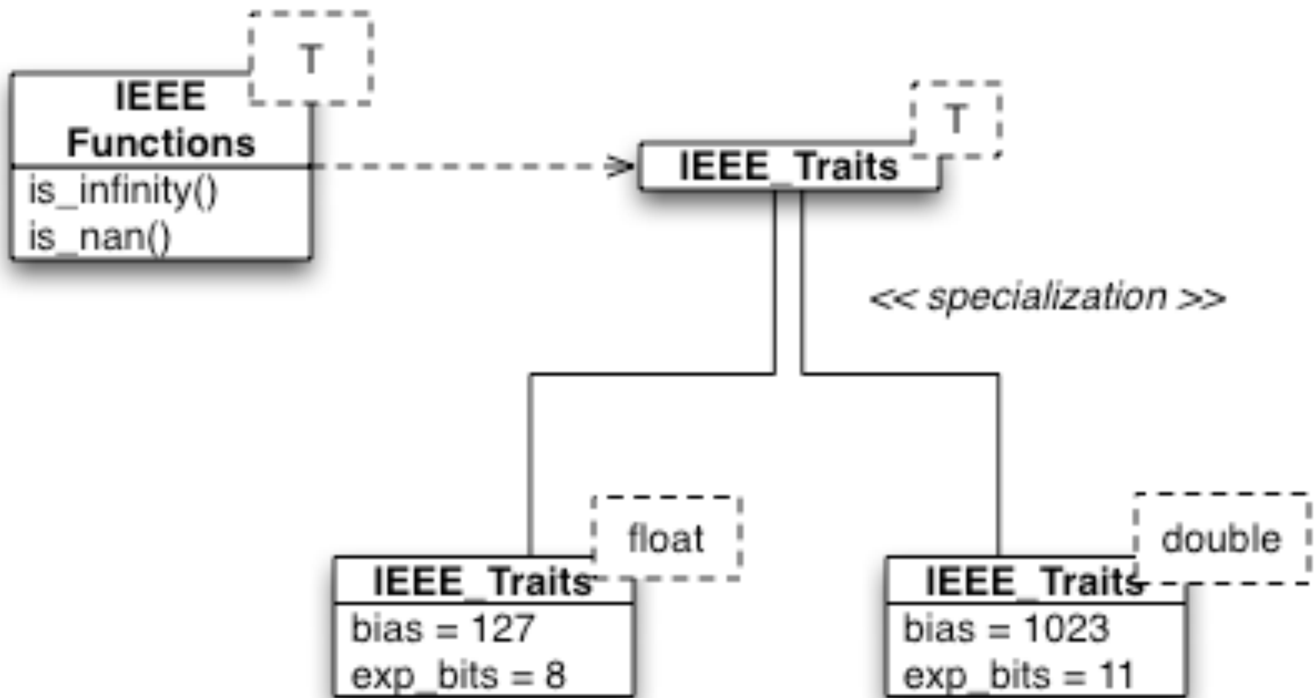
# Isolating Platform-Specific Code




Accomplished with conditional compilation, etc.

# C++ Template Traits

- A way of factoring variable data from a template



# IEEE Traits

```
template<typename T>
struct IEEE_traits {};

template<>
struct IEEE_traits<float>
{
  typedef float FType;
  enum {
     nbytes = sizeof(float),
     nbits = nbytes*8,
     exp_bits = 8,
     bias = 127
   };
};
```

```
template<>
struct IEEE_traits<double>
{
  typedef double FType;
  enum {
     nbytes = sizeof(double),
     nbits = nbytes*8,
     exp_bits = 11,
     bias = 1023
  };
};
```

# Using IEEE_Traits

```
template<typename FType>
bool is_infinity(FType x) {
   return exponent(x) == IEEE_traits<FType>::bias+1 &&
          fraction(x) == FType(0);
}

template<typename FType>
bool is_nan(FType x) {
   return exponent(x) == IEEE_traits<FType>::bias+1 &&
          fraction(x) != 0;
}
```

# Policies

- Classes with implementation strategies are *template arguments*
- Example – C++ Container Adaptors:

```
queue<int> q1;                     // Default policy
queue<int, list<int> > q2; // Explicit policy
```

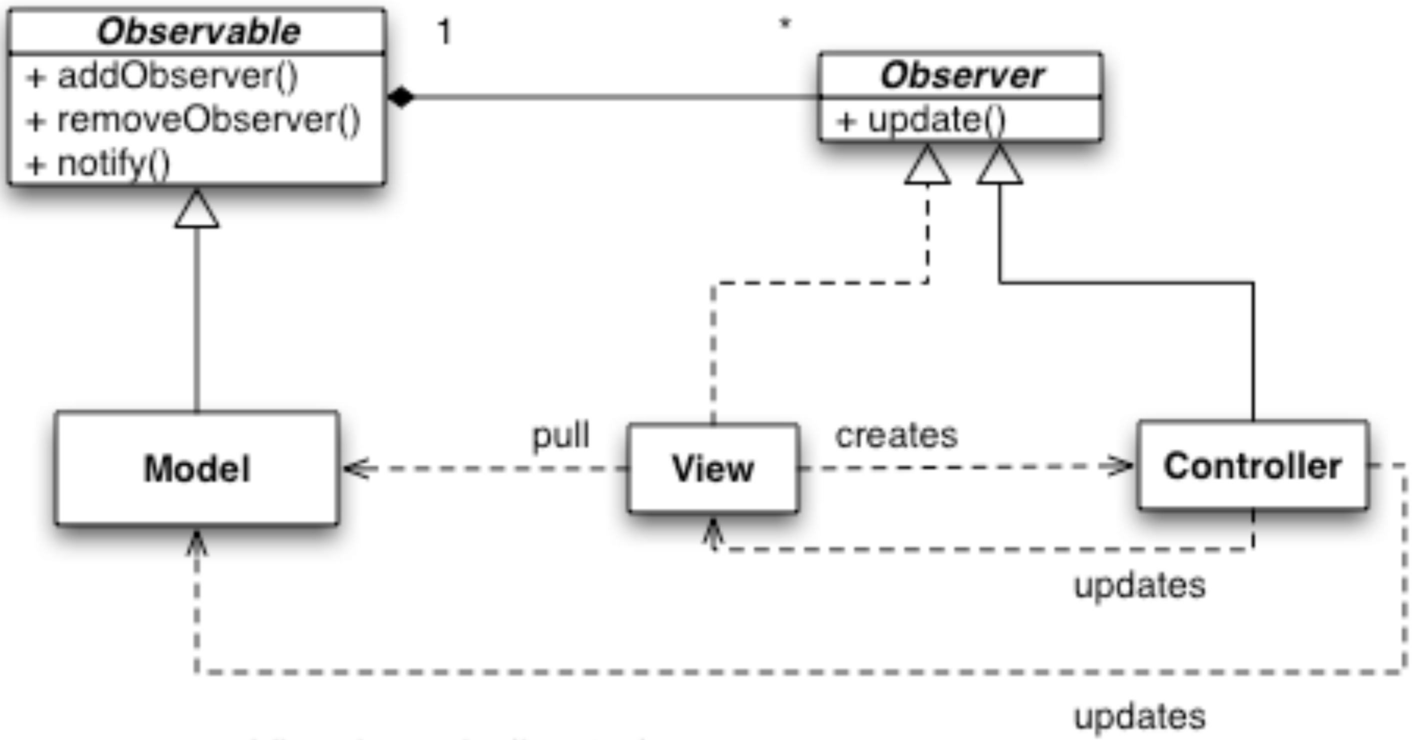
# Commonality vs. Variability
## *Take 4 – Designing User Interfaces*

**The Abstraction**: Data can be presented to users in different ways. Views vary independently of data.

| What Stays the Same | Coupling Mechanism | What Changes |
|---|---|---|
| The data (structure of model) | Complex! (MVC) | The current user view |

Related design patterns: **Model-View-Controller** (**Observer** + **Composite** + **Strategy**)

# Model-View-Controller



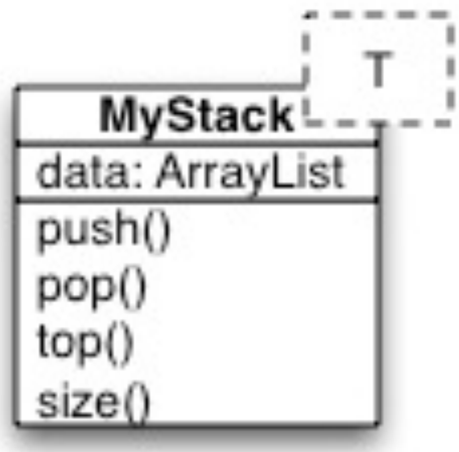

View is an indirect observer.
Each view has an associated controller
(The controller is the view's *Strategy*.)
Views are Composites.

# Another Fundamental Principle

- *Program to an Interface, not an Implementation*
  - Same benefit as before (shield clients from changes)
- Actually, just a *special case* of the previous principle
  - *interfaces* stay the same, *implementations* vary
  - You can't program exclusively to an interface unless it *exists separately* from the implementation
- Moral: Many design principles "overlap"

# OOP 101

- Design a Stack Class



# MyStack in Java

```
class MyStack<T> {
    private ArrayList<T> data = new ArrayList<T>();
    public void push(T t) {
        data.add(t);
    }
    public T pop() {
        return data.remove(data.size()-1);
    }
    public T top() {
        return data.get(data.size()-1);
    }
    public int size() {
        return data.size();
    }
}
```

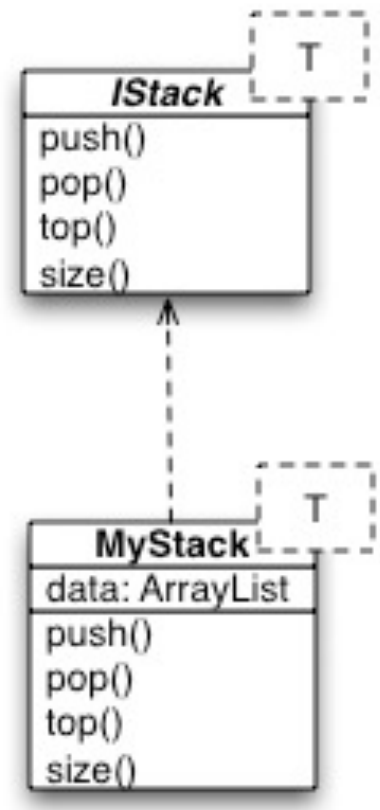
# Using **MyStack**

```
 public static void main(String[] args {
   MyStack<Integer> stk = new MyStack<Integer>();
   stk.push(1);
   stk.push(2);
   System.out.println(stk.size());  // 2
   System.out.println(stk.pop());   // 2
   System.out.println(stk.pop());   // 1
   System.out.println(stk.size());  // 0
}
```

# How is Our Design?

- Is the user really shielded from changes in implementation?
- No…
  - The fact that we use an **ArrayList** introduces a dependency for the user
  - If we *change it later*, the user is affected
  - Or a better class with a *different name* may come along
  - Users should *program to an interface*

# Separate The Implementation

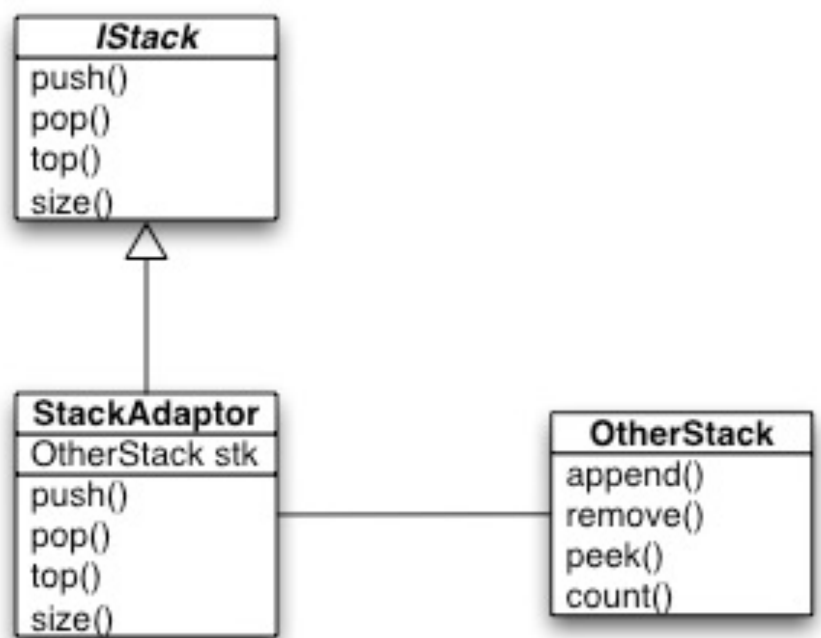


# Using IStack

```
static void test(IStack stk) { // Transparency
    stk.push(1);
    stk.push(2);
    System.out.println(stk.size());
    System.out.println(stk.pop());
    System.out.println(stk.pop());
    System.out.println(stk.size());
}
public static void main(String[] args) {
    IStack<Integer> stk = new MyStack<Integer>();
    test(stk);
}
```

# Using a Different Implementation

- Programming to an interface facilitates *adapting* to a different implementation
- The **Adapter Pattern**:



# A Variation on Adaptor

- The essence of Adapter allows clients to use a familiar interface with an implementation with a different interface
- The interfaces can be *implicit*
- Example: C++ *function-object adapters*

# C++ Function Object Adapters

- **bind1st, bind2nd**:
  - convert a binary function into a unary function by saving one of the arguments
- **not1, not2**:
  - logically negate the return value of a function
- Among others

# Using **bind2nd** and **not1**

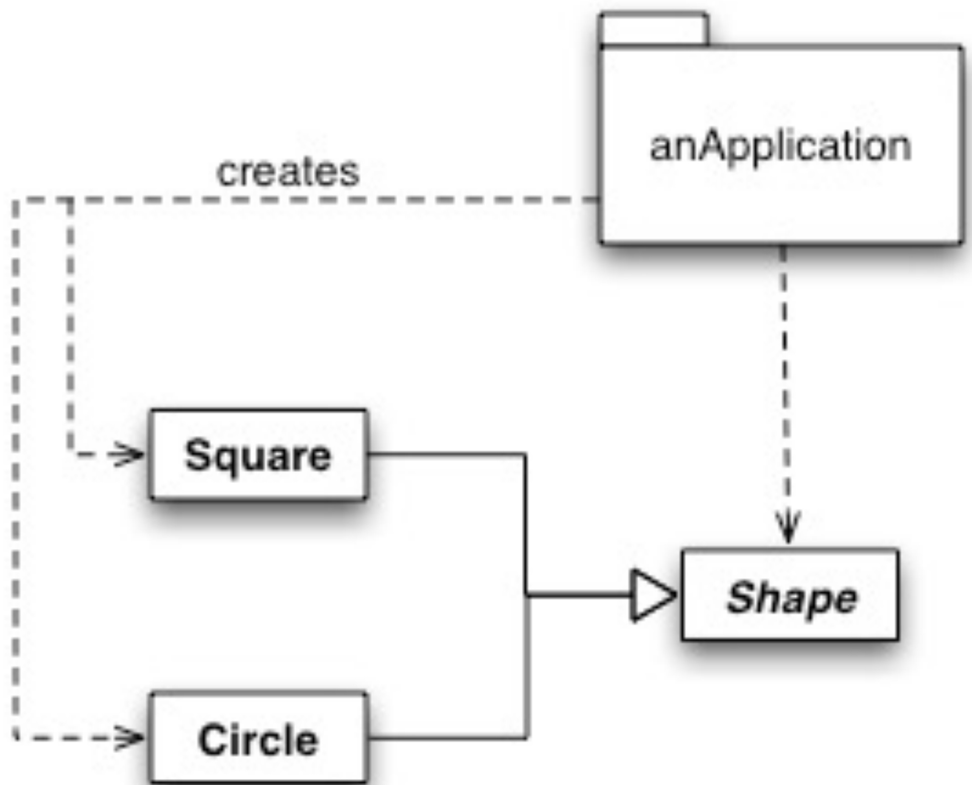
```
int main() {
    // Add 5 to some integers
    int a[] = {10, 25, 40};
    transform(a, a+3, a, bind2nd(minus<int>(), 5));
    copy(a, a+3, ostream_iterator<int>(cout, " "));
    cout << endl;   // Printed: 5 20 35

    // See if the result is even or not
    bool b[3];
    transform(a,a+3,b,not1(bind2nd(modulus<int>(),2)));
    cout << boolalpha;  // Print "true" instead of "1"
    copy(b, b+3, ostream_iterator<bool>(cout, " "));
    cout << endl;   // false true false
}
```

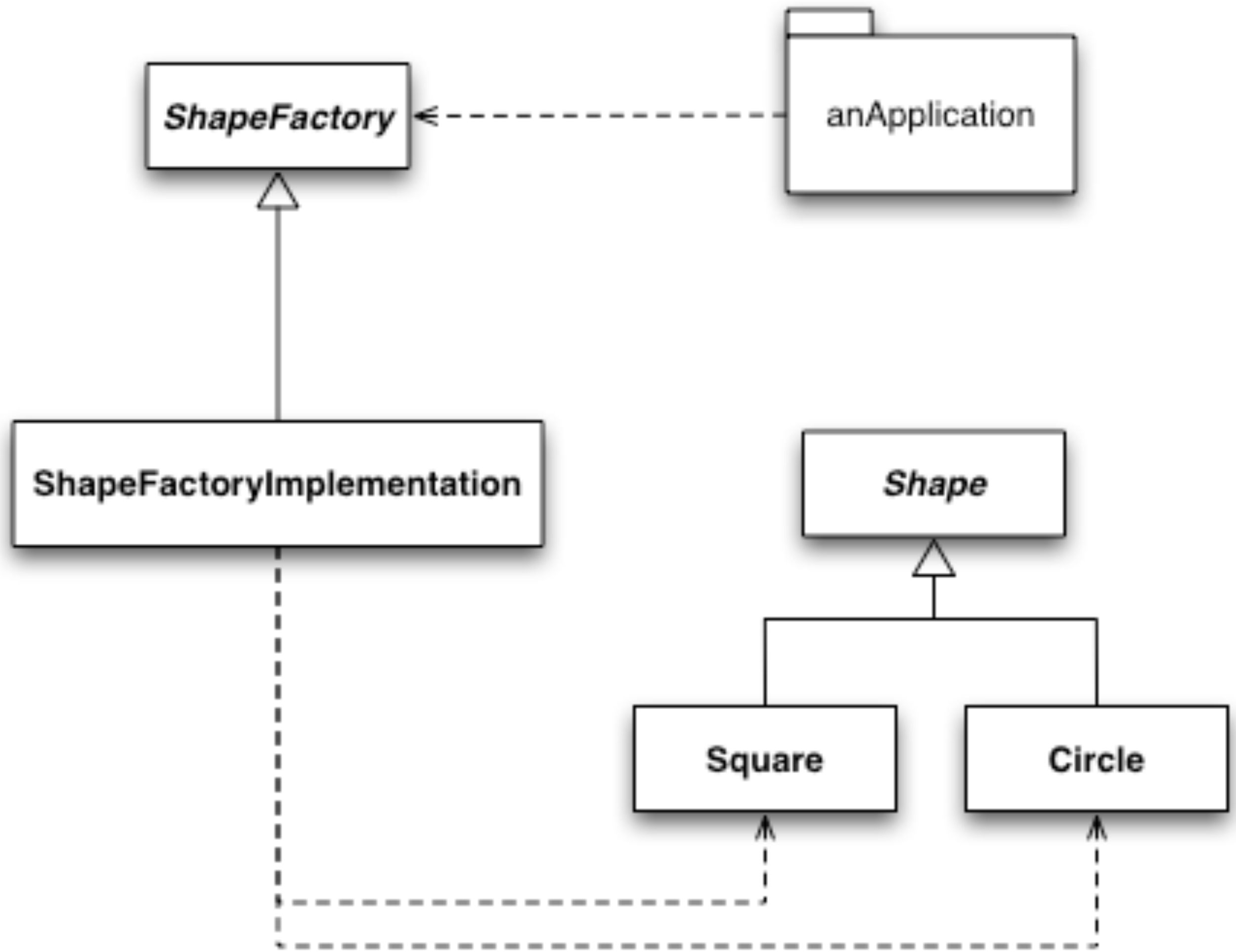
# A Related Principle

- *Separate object <u>creation</u> from object <u>use</u>*
- Client contexts can then use such objects *polymorphically*
  - by programming to an interface only
- Isolating object creation into a single module is Good Design

# Violating the Principles



# A Better Approach



# Factory Method Description

- Summary:
  - Lets a class defer object instantiation to concrete classes polymorphically.
- Problem:
  - A module uses an abstraction, so you want to follow the DIP and not depend on concrete details. Client modules shouldn't need to know *which* concrete class to instantiate.
- Solution:
  - Define an interface for creating a family of objects, but let concrete *subclasses* decide which class to instantiate.

# Factory Method Sketch


```
// Client has been given a Creator object
Product aProduct = aCreator.factoryMethod();
```

# Opposing Forces

- Objects are most easily created with a **new** expression, using the *concrete* class
- But this introduces a *dependency* on a concrete class, losing the flexibility of "programming to an interface, not an implementation"
    - and also losing the flexibility of separating object use from object creation
    - the using module may not have all the details needed for creation

# Balancing the Forces

- Factory Method balances these forces by encapsulating object creation
- Users call a method that "does the right thing"
- But one size does not fit all…

# Variations On Factory Method

- Plain Factory Method
  - just a function
  - no need for inheritance
- Class Factory Method
- Clone Method
  - an "Object Factory Method"

# Plain Factory Method

```
// Separate creator class
final class Creator {
    public static Product create() {
        return new Product();
    }
}

// Non-polymorphic:
class Product {
    // Non-public constructor
    Product(){/* whatever */}
}
```

# Class Factory Method

- The *class* is the creator
- The factory method is *static*
- Example:
  - **valueOf** methods:

```
Integer n = Integer.valueOf(s);
```

# Clone Method

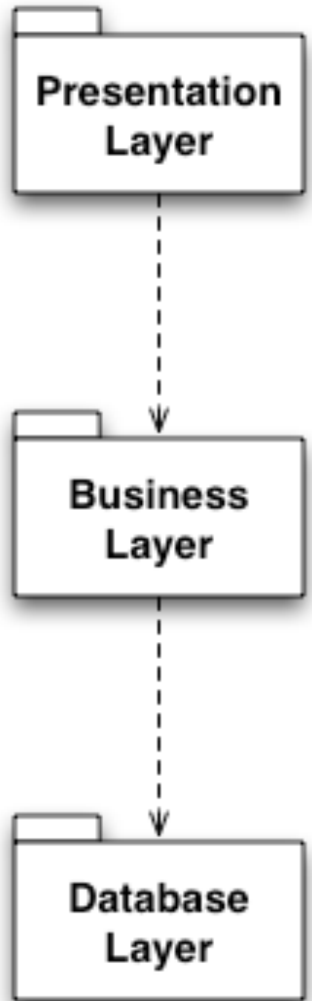
- An *object* is the creator
- The factory method is therefore *non-static*
- Example:
  - standard **clone( )** overrides:

```
Foo f2 = f.clone();
```

# Another Perspective

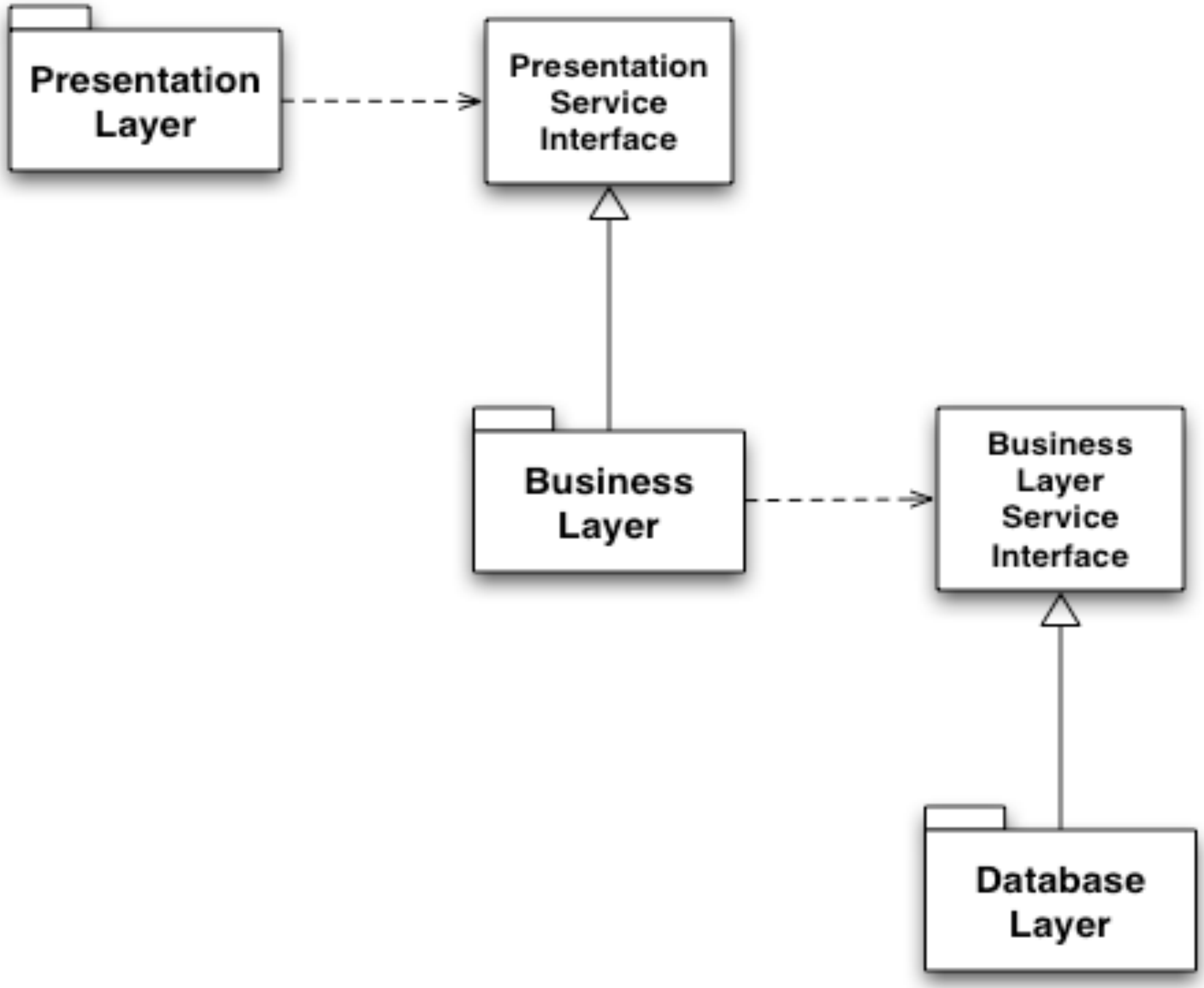
- *Dependency Inversion Principle*
- *High-level* components should not depend ("know about") *lower-level* components
  - that's why client modules should not explicitly create concrete objects
- All components should depend on *abstractions* as much as possible

# Violating the Principle




Classic
3-tier
Architecture

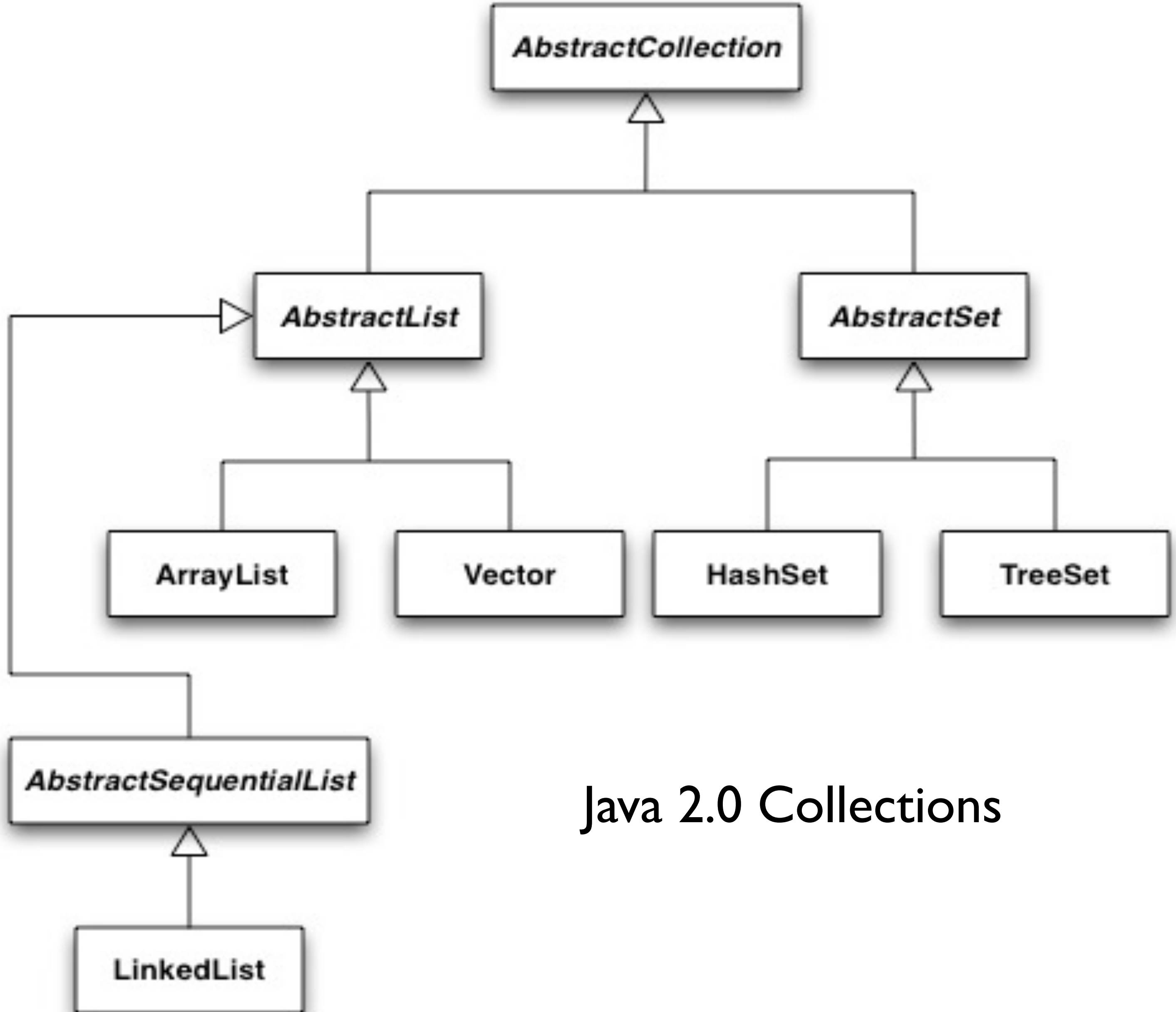
# A Better Design



# Dependency Rules of Thumb

## *"Little Principles"*

- No variable in an abstraction should hold an *explicit pointer* to a concrete class
  - Use *top-level* pointers polymorphically
- No class should derive from a concrete class
  - Only derive from *abstract* classes
- No method should override an *implemented* method of any of its base classes
  - Only override *abstract* methods
- These rules can't be followed *all* the time!
  - The key is: How *volatile* is the lower-level module?




Java 2.0 Collections

# Resource Management

- Factory Method is about *initialization*
  - resources other than memory can be allocated
- How do you ensure resource *deallocation*?

# Disposal Method

- Encapsulates the details of object disposal by providing an explicit method for cleanup
- **Disposal Method** complements Factory Method by resolving issues Factory Method leaves dangling
- Two Variations:
  - Factory Disposal Method
  - Self-Disposal Method

# Factory Disposal Method

```
final class Creator {
    public static Product create() {
        return new Product();
    }
    public static void dispose(Product p) {
        /* whatever */
    }
}
…
    Creator::dispose(p);
```

# Self-Disposal Method

```
final class Creator {
    public static Product create() {
        return new Product();
    }
    public void dispose() {
        /* whatever */
    }
}
…
    p.dispose();
```

# C++ Variation

- Deterministic destruction can automate resource deallocation:
  - constructors allocate a resource
  - destructors deallocate the resource
- Destructors execute automatically
- RAII Idiom
  - "Resource Acquisition Is Initialization"
- Similar functionality in C# via **using**

# RAII in C++

```
{
    ifstream f("myfile");
    string line;
    while (getline(f,line))
        cout << line << endl;
} // stream closes automatically
```

# Using C++0x's **shared_ptr**

```
class Foo {
public:
    Foo(){}
    ~Foo() {
        cout << "destroying a Foo\n";
   }
};

int main() {
    vector<shared_ptr<Foo> > v;
    v.push_back(shared_ptr<Foo>(new Foo));
    v.push_back(shared_ptr<Foo>(new Foo));
    v.push_back(shared_ptr<Foo>(new Foo));
}
```

# Using a Custom Deleter

```
int main() {
    FILE* f2 = fopen("deleter.cpp", "r");
    shared_ptr<FILE> theFile(f2, &fclose);
    /* … */
}
```

# Multi-Step Resource Management

- Composite resources usually need to be handled as transactions
  - if an exception occurs at any time during allocation, previously competed allocation need to be backed out
- Gnarly with **try**-blocks
  - see next slide

```
void g() {  //  3-part transaction
    risky_op1();
    try {
        risky_op2();
    }
    catch (Exception x) {
        undo_risky_op1();
        throw x;  // Rethrow exception
    }

    try {
        risky_op3();
        writeln("f succeeded");
    }
    catch (Exception x) {
        undo_risky_op2();
        undo_risky_op1();
        throw x;
    }
}
```

# Scope Guards in D

```
void g() {
    risky_op1();
    scope(failure) undo_risky_op1();
    risky_op2();
    scope(failure) undo_risky_op2();
    risky_op3();
    writeln("g succeeded");
}
```

# Extending a Class

- Typically done via *inheritance*
  - an example of *code reuse*
- Comes with a **price**:
  - dependency on a *concrete class*
  - inheritance is a *compile-time* mechanism
    - adding functionality statically can lead to *class explosion*
  - you may want *runtime extension*

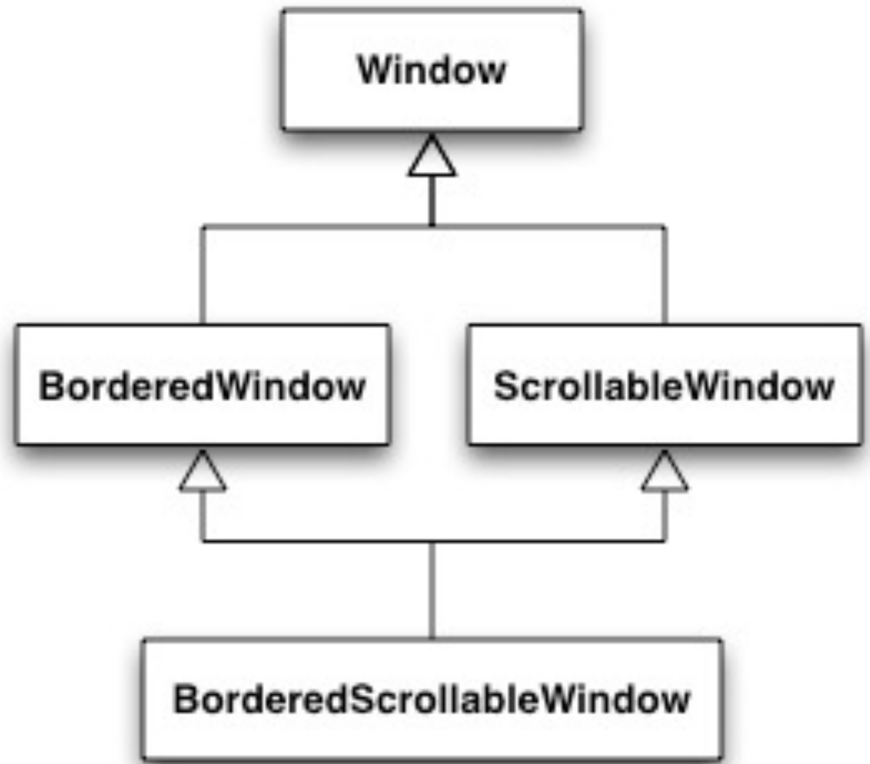
# OO Design 101 Redux

- Consider a GUI type named **Window**
  - Unadorned, but functional
- Now suppose we want some more full-featured windows
  - Bordered, scrollable, etc.
- How do we design this?

# OO Design 101 Redux

- A **BorderedWindow** is most assuredly a **Window**
  - Certainly *sounds* like an “is-a”
- Ditto **ScrollableWindow**
  - Sort of obvious, right?
  - Let’s see…

# A "Simple" Window Hierarchy



# Counting Classes

- Note that the number of classes is quite predictable:
  - 1 (= C(2,0)) for the root
  - 2 (= C(2,1)) for the single-featured subclasses
    - 2 features total, choosing 1 at a time
  - 1 (=C(2,2)) for the leaf
    - Combines all features
- Total of 4

# Evaluating Our Design

- Ignore details of multiple inheritance…
  - We can always work around that
- Any other problems?

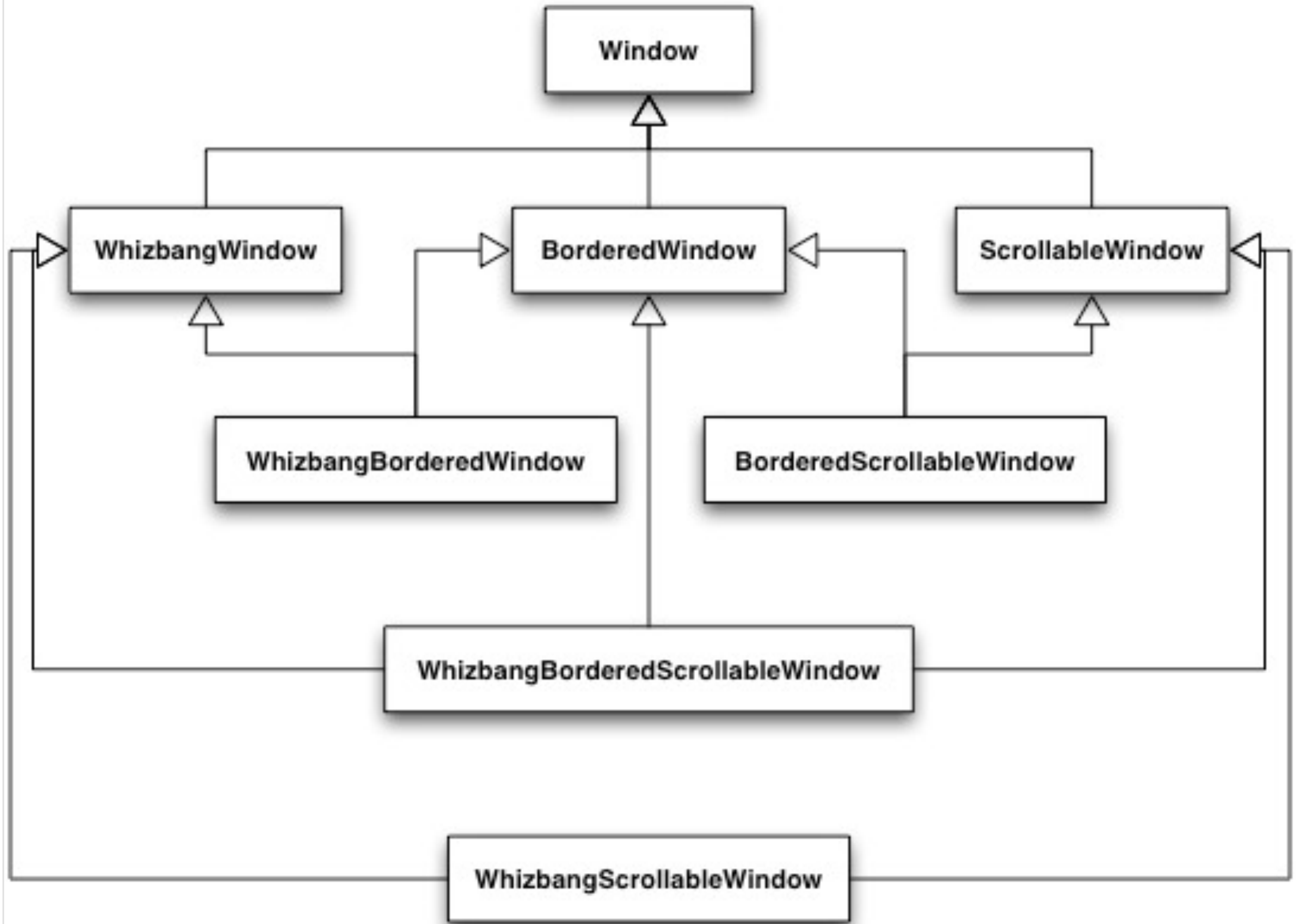
# Problem #1

- The subclasses have operations that the **Window** superclass doesn't
  - **scroll**, for example
  - Not completely an "is-a"
  - But it isn't terribly unusual for a subclass to add operations; no biggie
- We could put these methods in **Window**
  - But they'd be no-ops in the subclasses that don't use them
  - Someone isn't *encapsulating variation*!

# Problem #2

- What if we need to add another important, independent windowing feature?
  - **WhizbangWindow**
- What impact does this have on the hierarchy?

# Hierarchical "Progress"



# Definitely Counting Classes

- 1 (= C(3,0)) for the root
- 3 (= C(3,1)) for the first row
  - Single-featured
- 3 (= C(3,2)) for the second “row”
  - Double-featured
- 1 (= C(3,3)) for the leaf
  - All three
- Total of 8

# Looking Ahead

- $C(n,0) + C(n,n-1) + \ldots + C(n,1) + C(n,0)$
- Equals $2^n$
- Can anyone say “combinatorial explosion”?

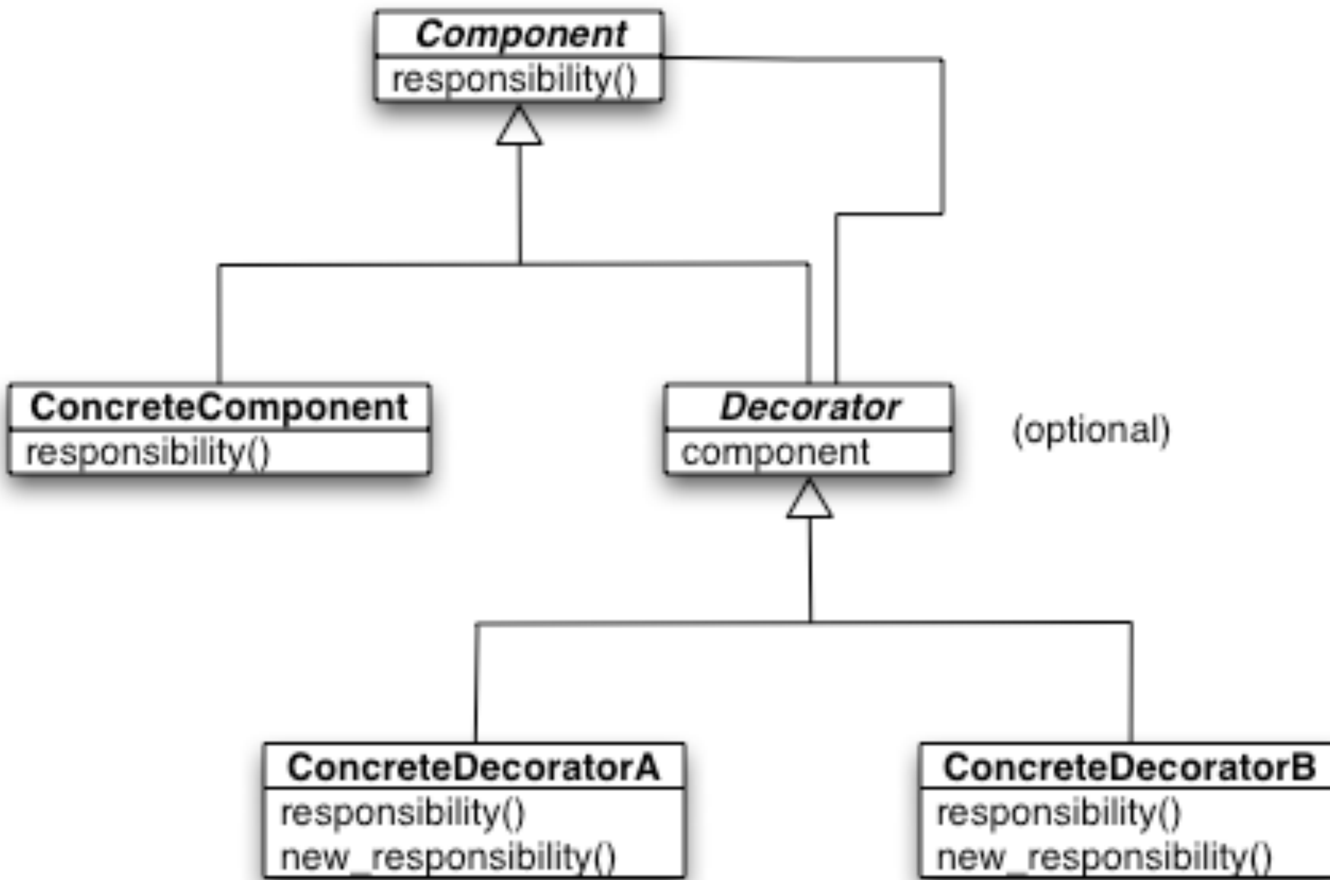
# The Open-Closed Principle

- *Classes should be open for extension, but closed to modification*
- In other words, you should be able to add to or modify a class's functionality *without changing its code*
  - Otherwise users depend on volatile code
- How?

# The Decorator Pattern

- Uses *composition* in place of inheritance
- A decorator wraps an object *polymorphically*
- It adds or modifies functionality
  - calling back to the original object as needed
- Decorators can be created and combined at *runtime*

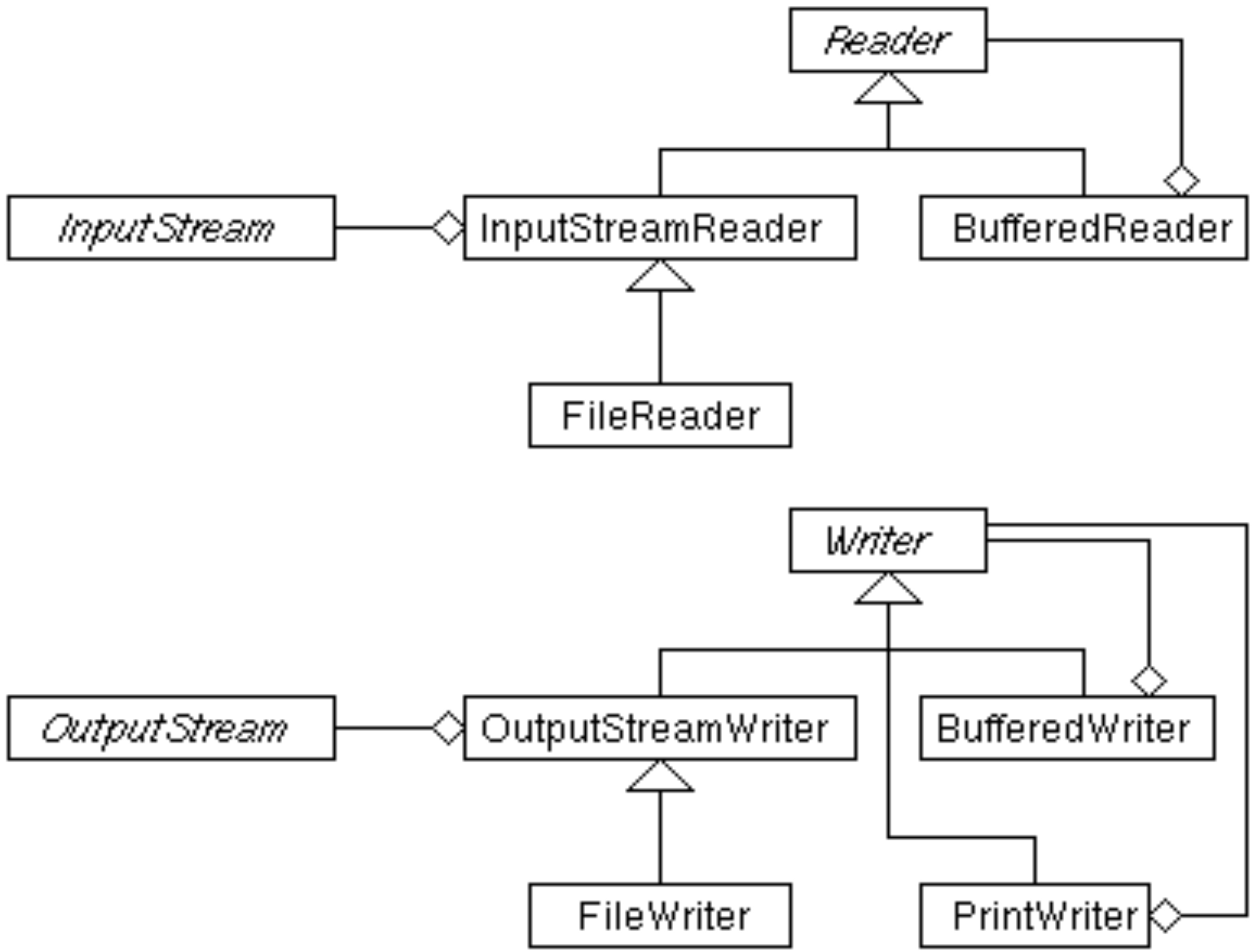
# Decorator Class Sketch



# Decorator Object Sketch


Decorator objects ultimately call back to an original concrete component. They can be used to implement *before-after-around* methods. They can be composed at *runtime*.

# Decorator in java.io



# A Companion Principle

- *Prefer Composition to Inheritance*
- More flexible
- Often simpler
- Inheritance is for *static*, "is-a" relationships

# C++ Example

- From class homework

# Coupling

- Another way to express these ideas is the old adage:
  *Minimize coupling between related entities*
- Or to paraphrase Einstein:
  *Objects that interact should have as little coupling as possible, but no less*
- Finding that coupling "sweet spot" takes a little finesse
  - and some abstractions :-)

# (Observer)

# (Near the end)

- shu-ha-ri

# Bibliography


- Bertrand Meyer
- Agile Software Development
- Head-First Design Patterns
- Factory and Disposal Methods, Kevlin Henny