ECOS Lab
Engineering COmplex and Smart systems

ICAR CNR

Consiglio Nazionale delle Ricerche

Soluzioni Generali a Problemi Ricorrenti nello Sviluppo di Software ad Oggetti:

Design Pattern

Luca Sabatucci

Programming Abstractions

Procedural Style vs Object-Oriented Style

- Procedure / Function
- Record
- Module
- Procedure Call

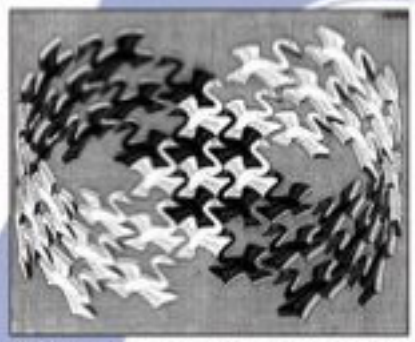
- Class/Object
- Abstract Class/Interface
- Attribute/Method
- Inheritance/Subclassing
  - Type, SubType, SuperType
- Dynamic Binding
- Polymorphism
  - Abstract operations

Design Pattern

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice


Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

Patterns derive from Experience

- Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder

- The value of design patterns is that of being the result of experience on the field gained over several years of trial-and-error attempts.

- Using a pattern consists in exploiting a well-proven solution, with general benefits on software quality

Description

- Design patterns are typically described via informal natural language
  - A name
  - The problem (why, when)
  - The solution (what, how)
  - The consequences (decisions, trade-offs)

GoF's Book description template:
Name, Classification, Intent, Motivation,
Applicability, Structure, Participants, Collaborations,
Consequences, Implementation, Sample Code,
Known Uses, Related Patterns

The GoF's Catalog

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

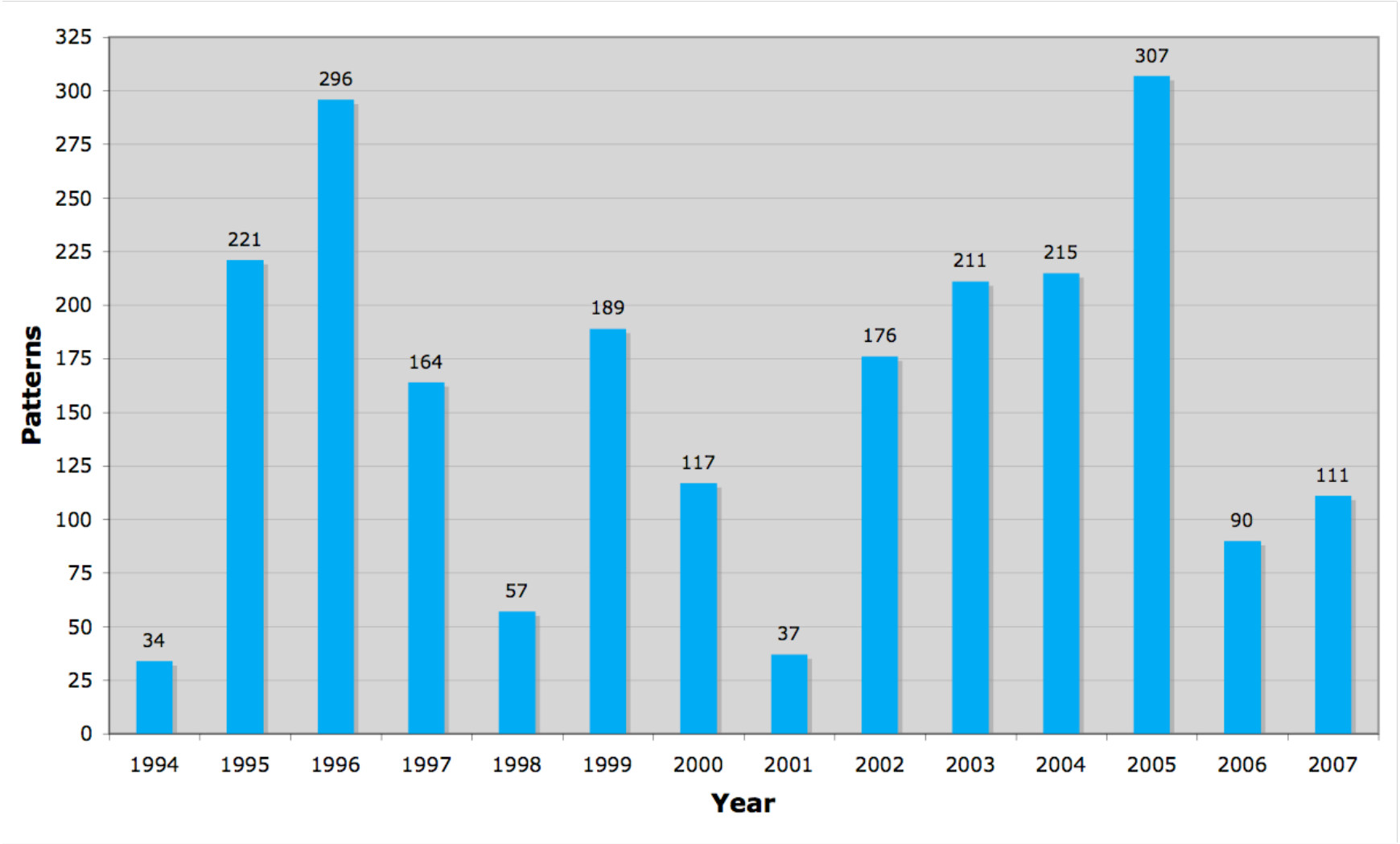


Figure 1: Number of Patterns Created, 1994 - 2007.

Table 1: Pattern Diversity by Technical Domain

Type	#Collections	#Patterns
User Interface	14	425
Programming Languages	14	243
Architecture	11	231
OO Design	33	161
Workflow	11	149
Systems	14	140
Communication	11	91
Database	5	54
Frameworks	4	51
Components	3	47
Parallelization	3	35
Security	2	16
Management	2	12
Concurrency	7	11
Networking	3	11
Information Integrity	1	10
Fault Tolerance	1	8

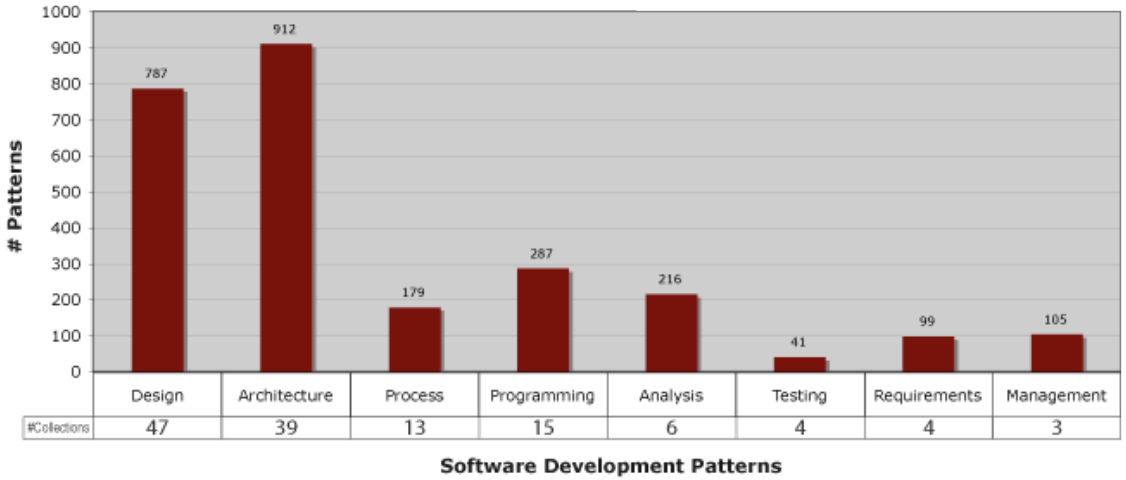


Figure 3: Types of Software Development Patterns.

Design Patterns
solve
Design Problems

Typical OO Design Problems

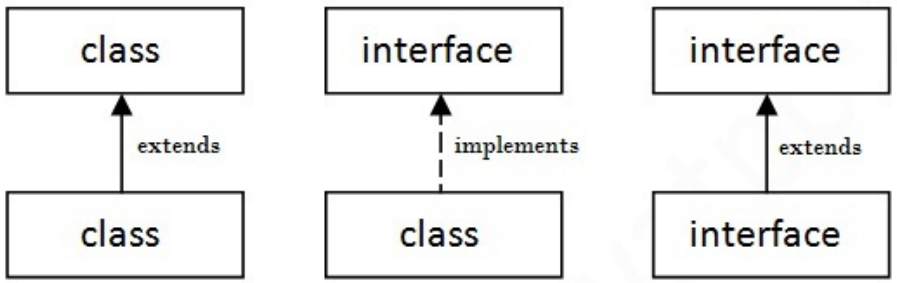
Class vs Type

- An object's class defines how the object is implemented.
  - the object's internal state
  - the implementation of its operations
- An object's type only refers to its interface
  - the set of requests to which it can respond
  - object supports the interface defined by the class

- An object can have many types
- Objects of different classes can have the same type

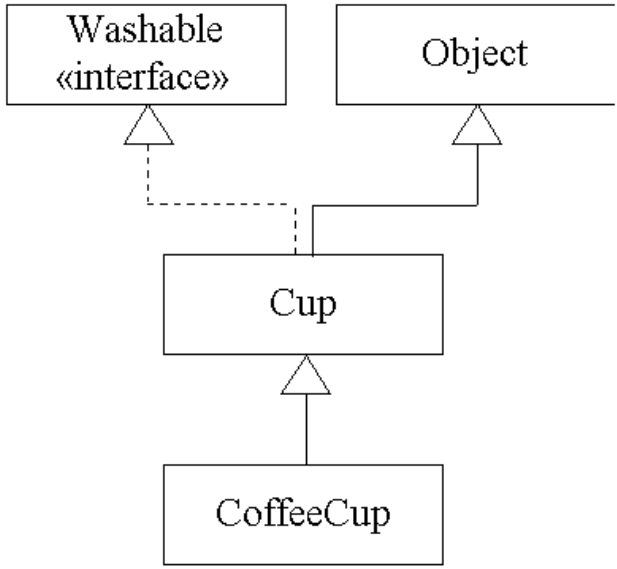
Class Inheritance
vs
Interface Inheritance

- Class inheritance defines an object's implementation in terms of another object's implementation (code sharing/reuse)
- Interface inheritance (or subtyping) describes when an object can be used in place of another



Implementation vs Interface

- Implementation Inheritance
  - just a mechanism for reusing functionality in parent classes.
- Interface Inheritance
  - defining families of objects with identical interfaces
    - classes derived from an abstract class will share its interface
    - adding/overriding operations
    - not hiding operations of the parent class.
  - all subclasses can respond to the same requests



- Committing to an interface
  - clients are unaware of the specific types of objects they use
  - clients are unaware of the classes that implement these objects
  - this greatly reduces implementation dependencies between subsystems

Inheritance vs Composition [for reusing functionality]

- Reusing by subclassing is “white-box reuse”
  - inheritance is defined statically at compile-time
  - provides default implementations for operations and lets subclasses override them
  - the internals of parent classes are often visible to subclasses (inheritance breaks encapsulation)

- Reusing by composition is “black-box reuse”
  - new functionality is obtained by assembling objects to get more complex functionality
    (objects shall respect each others' interfaces)
  - It is defined dynamically at run-time through objects acquiring references to other objects
    (object can be replaced at run-time by another with the same interface)
  - no internal details of objects are visible
  - fewer implementation dependencies

Inheritance vs Generics

- A third way to compose behavior in object-oriented systems
- This technique lets you define a type without specifying all the other types it uses
  - unspecified types are supplied as parameters at the point of use
  - Parametrized type cannot change at run-time

- Not all programming languages support Generics

The case of the SimUDuck app

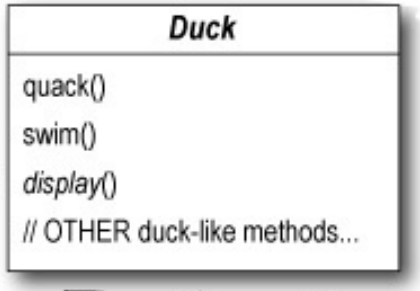
- Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*
- The game can show a large variety of duck species swimming and making quacking sounds
- The designers use standard OO techniques
- They created one Duck superclass from which all other duck types inherit.

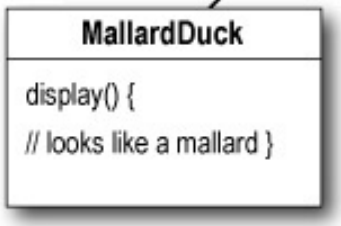
The display() method is abstract, since all duck subtypes look different.

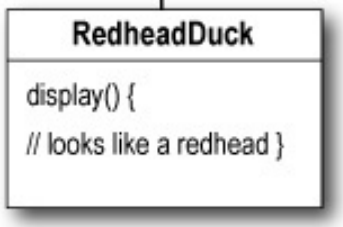
All ducks quack and swim, the superclass takes care of the implementation code.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.







Evolving the system

- In the last year, the company has been under increasing pressure from competitors.

- After a week long off-site brainstorming session, the company executives think it's time for a big innovation.

- They need something *really* impressive to show at the upcoming shareholders meeting ***next week***.

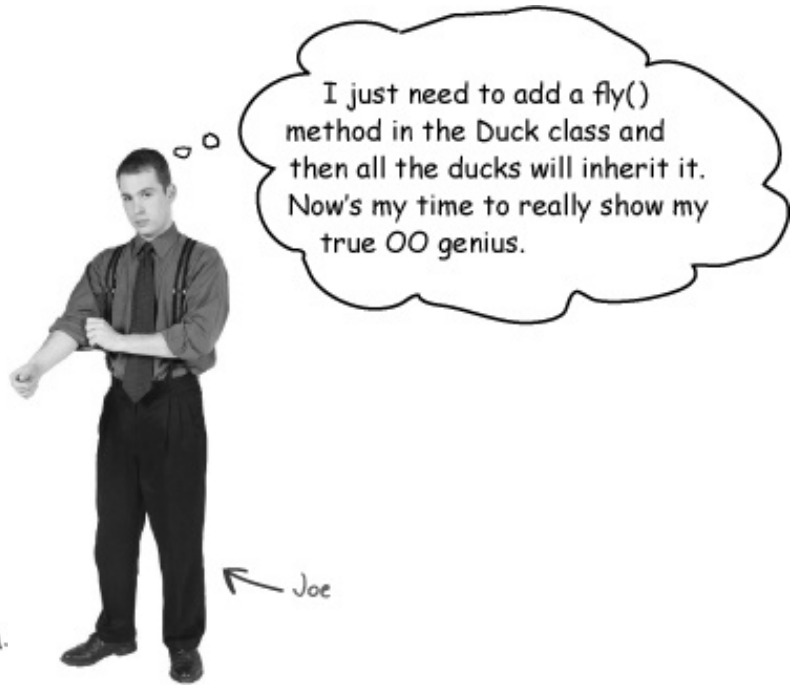
Now we need the ducks to FLY

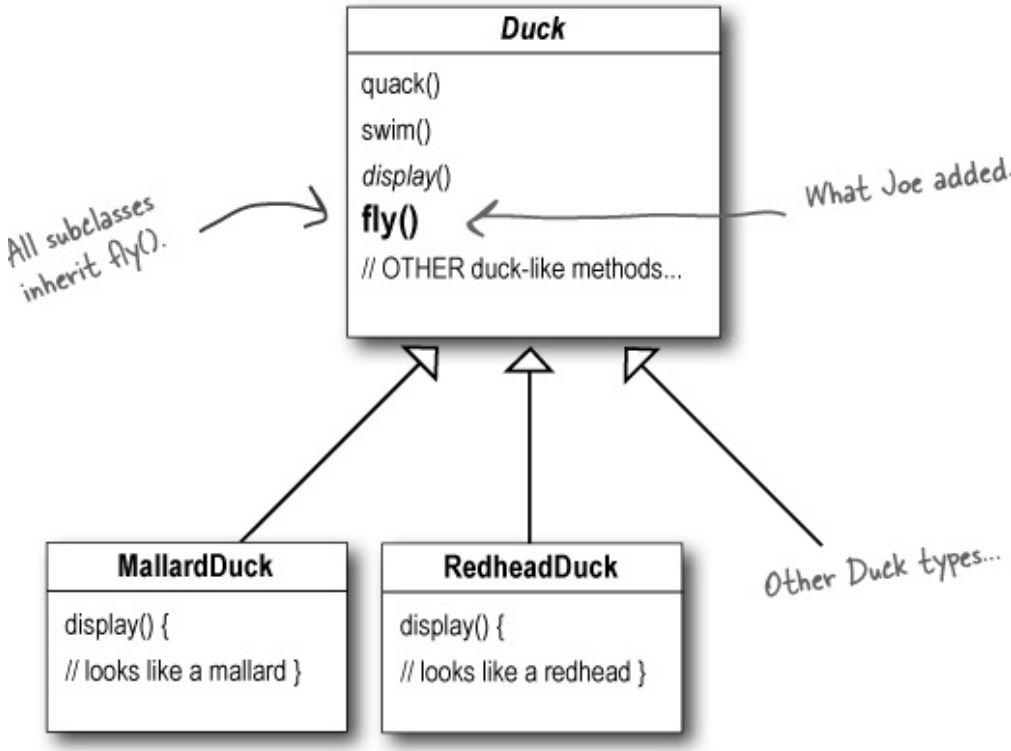
Evolution Maintainance

- The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors

- And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week
  - he's an OO programmer... *how hard can it be?*

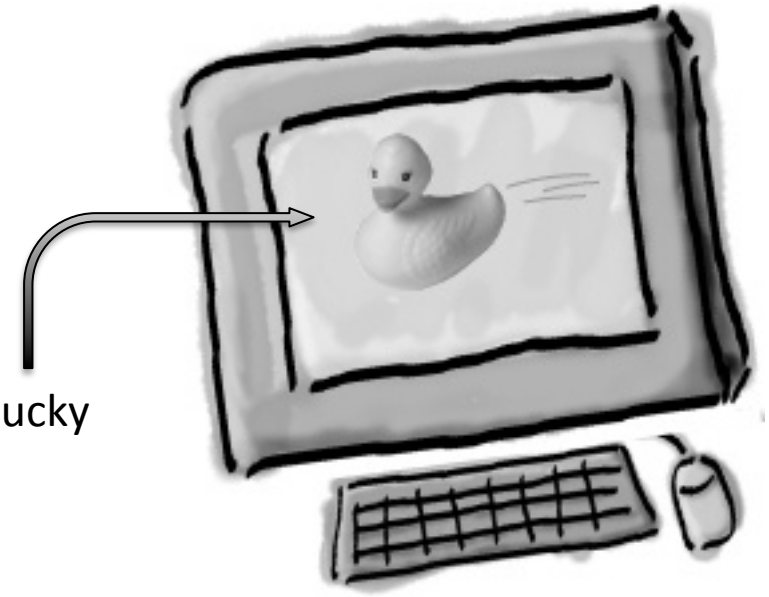
What we want.


I just need to add a fly() method in the Duck class and then all the ducks will inherit it. Now's my time to really show my true OO genius.
Joe


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

But something went horribly wrong...

rubber ducky

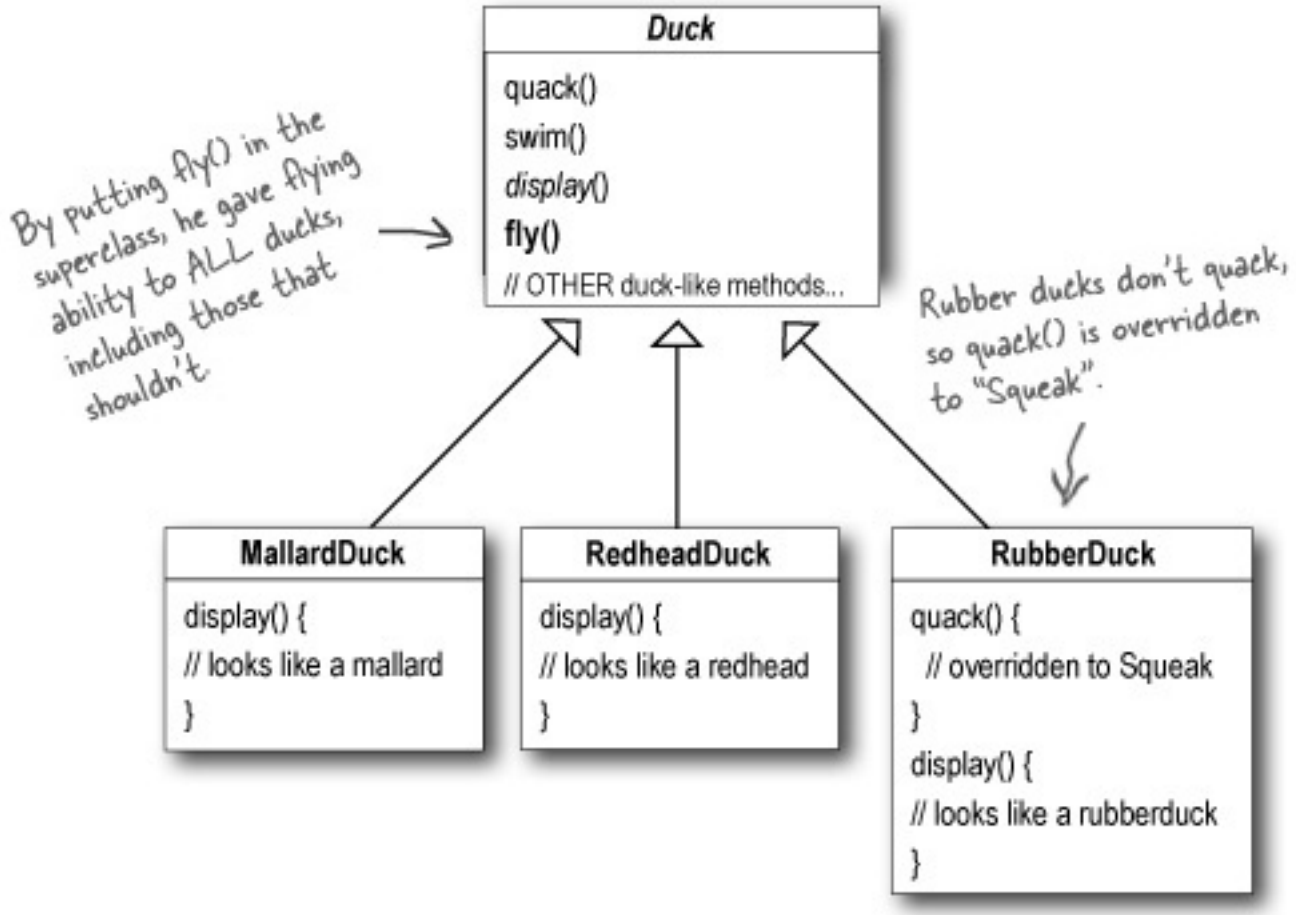


What happened?

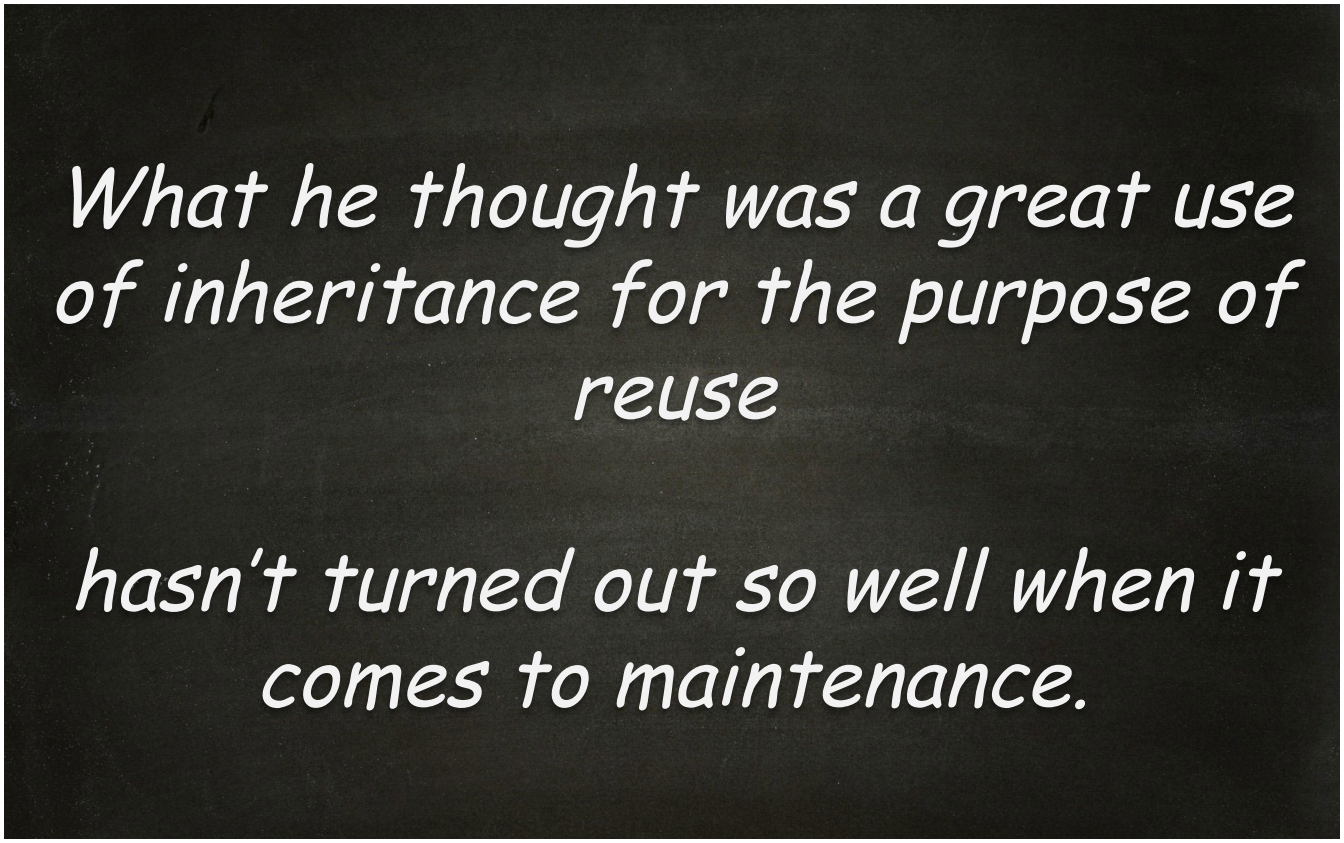
- Joe failed to notice that not *all* subclasses of Duck should *fly*

- When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses

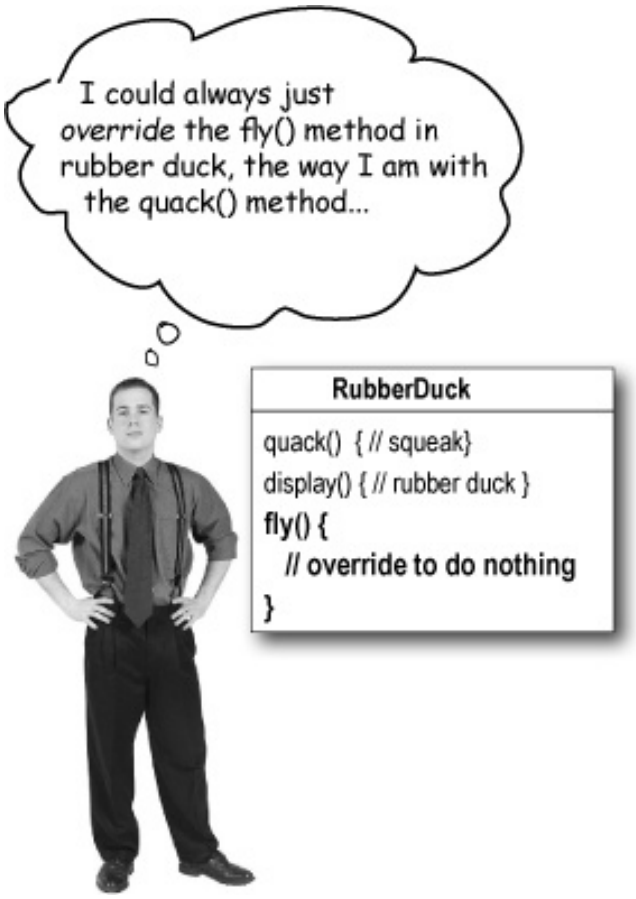
- He now has flying inanimate objects in the SimUDuck program



A localized update to the code caused a nonlocal side effect (flying rubber ducks)!


What he thought was a great use of inheritance for the purpose of reuse
hasn't turned out so well when it comes to maintenance.

Joe thinks about inheritance...



RubberDuck

```
quack()  { // squeak}
display() { // rubber duck }
fly() {
  // override to do nothing
}
```

DecoyDuck

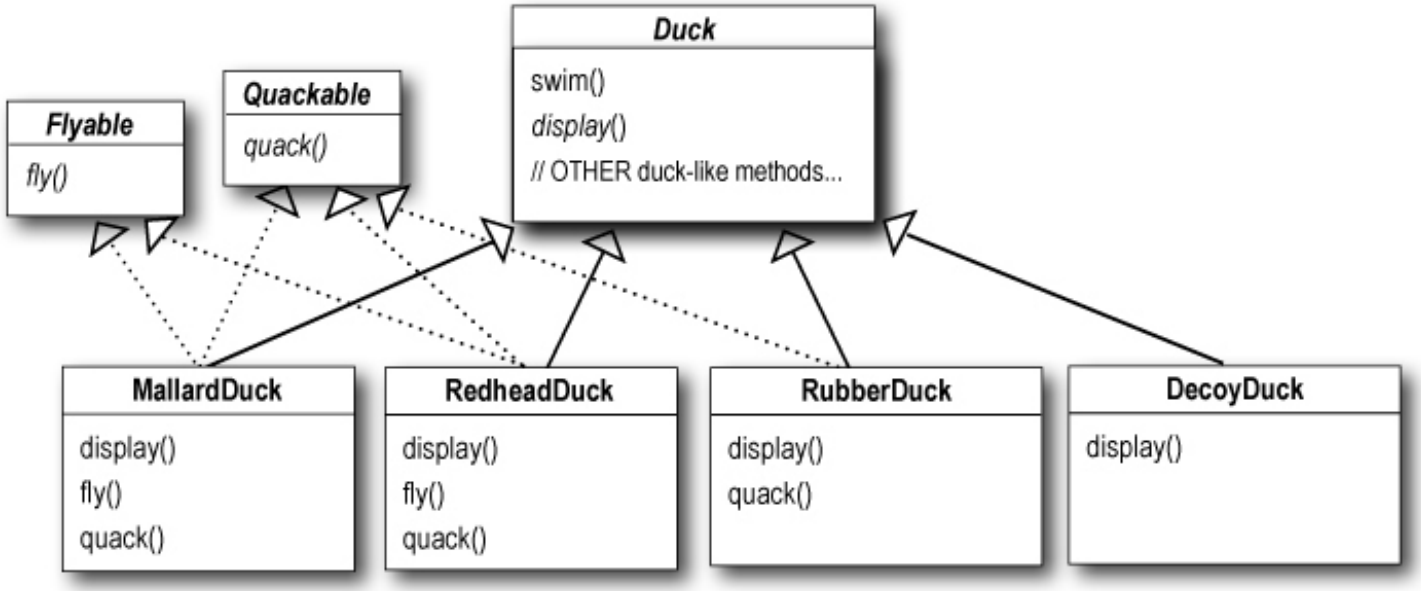
```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

- Joe realized that inheritance probably wasn't the answer,
  - Because executives now want to update the product every six months (in ways they haven't yet decided on).
  - He'll be forced to look at and possibly override fly() and quack() for every new Duck subclass that's ever added to the program... *forever.*
- So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.

How about an interface?

I could take the fly() out of the Duck superclass, and make a ***Flyable() interface*** with a fly() method. That way, only the ducks that are *supposed* to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.



- Having the subclasses implement Flyable and/or Quackable solves *part* of the problem (no inappropriately flying rubber ducks),

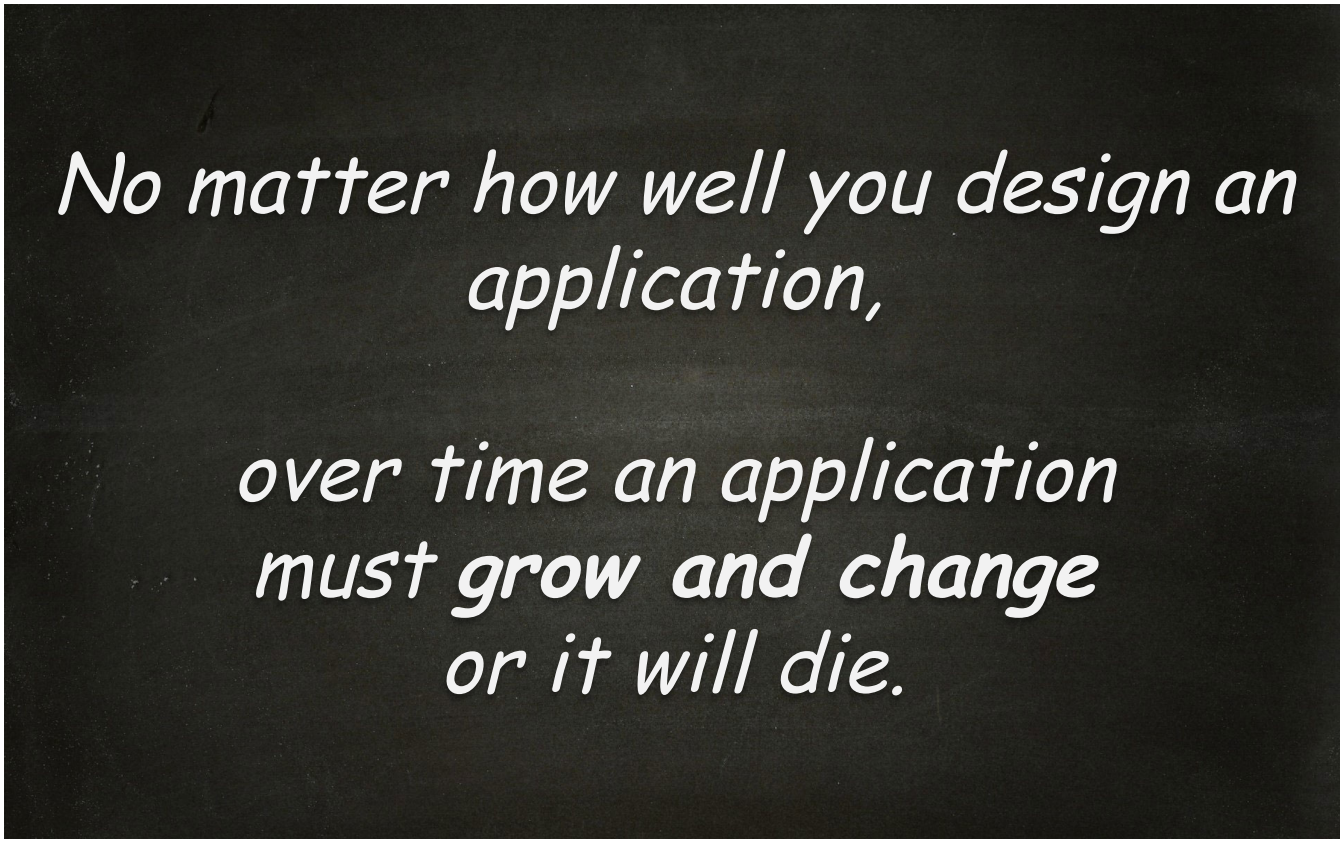
- It completely destroys code reuse for those behaviors,
  - so it just creates a *different* maintenance nightmare.

- And of course there might be more than one kind of flying behavior even among the ducks that *do* fly...

What's the one thing you can always count on in software development?

No matter where you work, what you're building, or what language you are programming in, what's the one true constant that will be with you always?

(use a mirror to see the answer)


No matter how well you design an application,
over time an application must grow and change or it will die.

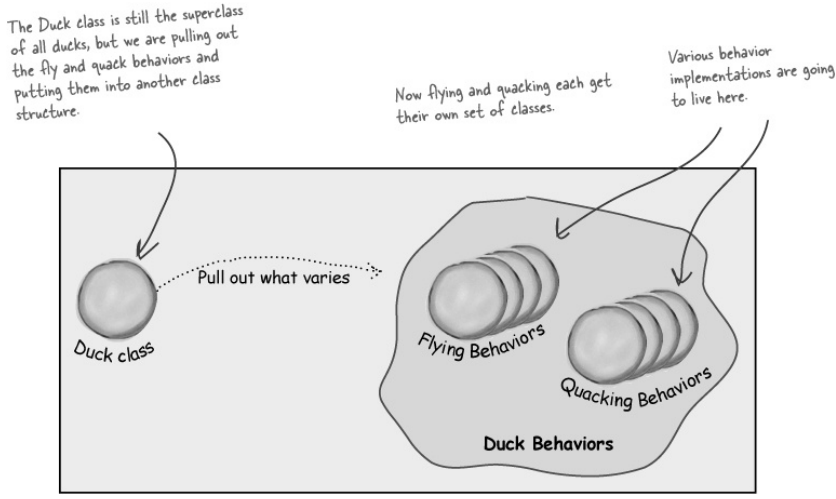
Identify the aspects of your application that vary and separate them from what stays the same.

- *Take what varies and "encapsulate" it so it won't affect the rest of your code*
  - *later you can alter or extend these parts without affecting the rest*

- *The result is fewer unintended consequences from code changes and more flexibility in your systems!*

Implementation Details

- We know that fly() and quack() are the parts of the Duck class that vary across ducks.
- We add two *sets* of classes (apart from the Duck class),
  - one for *fly* and
  - one for *quack*.
- Each set of classes will hold all the implementations of their respective behavior.
- For instance implementing:
  - *quacking*,
  - *squeaking*,
  - *silence*.

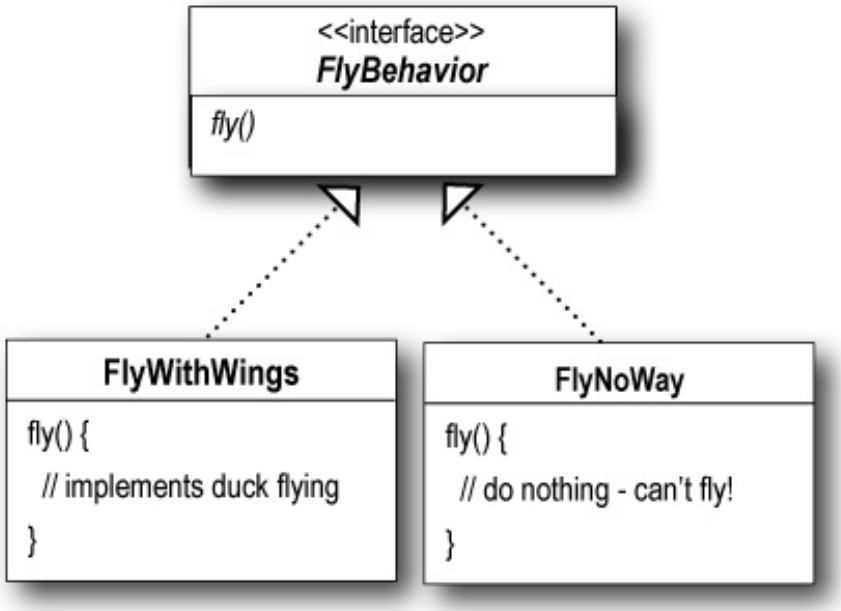


Implementation Details (II)

- Now we *assign* behaviors to the instances of Duck.
  - For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific *type* of flying behavior.
- We include behavior "setter" methods in the Duck classes so that we can *change* the MallardDuck's flying behavior *at runtime*.
- This solution allows changing the behavior of a duck dynamically

Program to an interface (*not an implementation)*

- If we use an interface to represent each behavior
  - then the Duck classes does not know implementation details for their own behaviors



This time, it is NOT the *Duck* classes that will implement the flying and quacking interfaces

- for instance, FlyBehavior and QuackBehavior will implement one of those interfaces.

Program to an interface” really means “Program to a supertype”

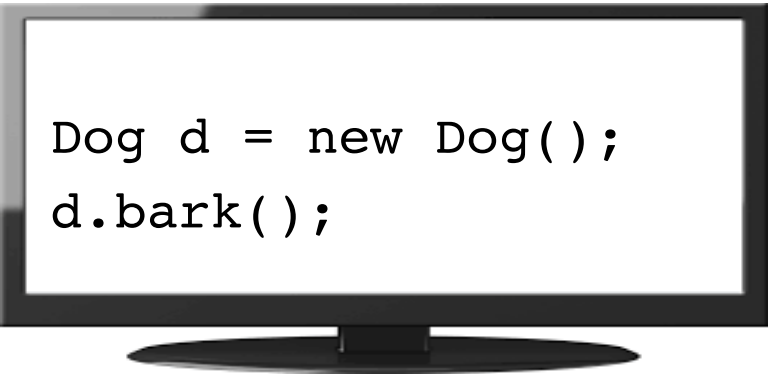
The word interface is *overloaded*

You can *program to an interface*, without having to actually use a Java *interface*

The point is to exploit ***polymorphism*** by programming to a supertype

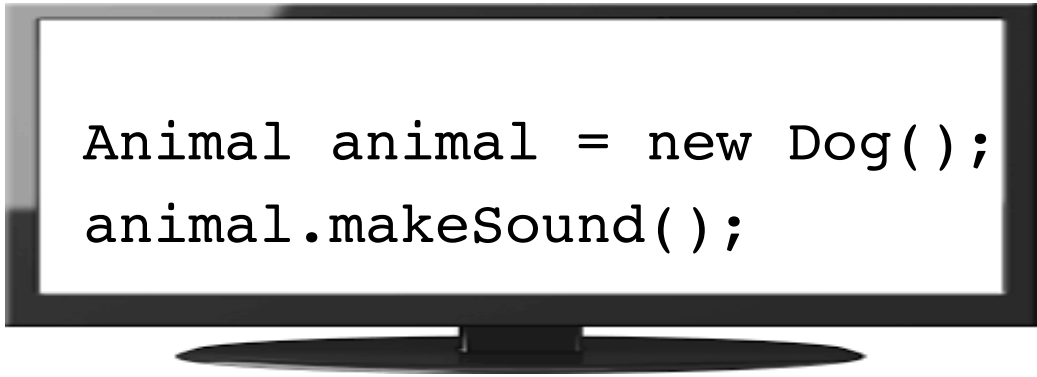
Implementation vs Supertype

- Programming to an **implementation** would be:



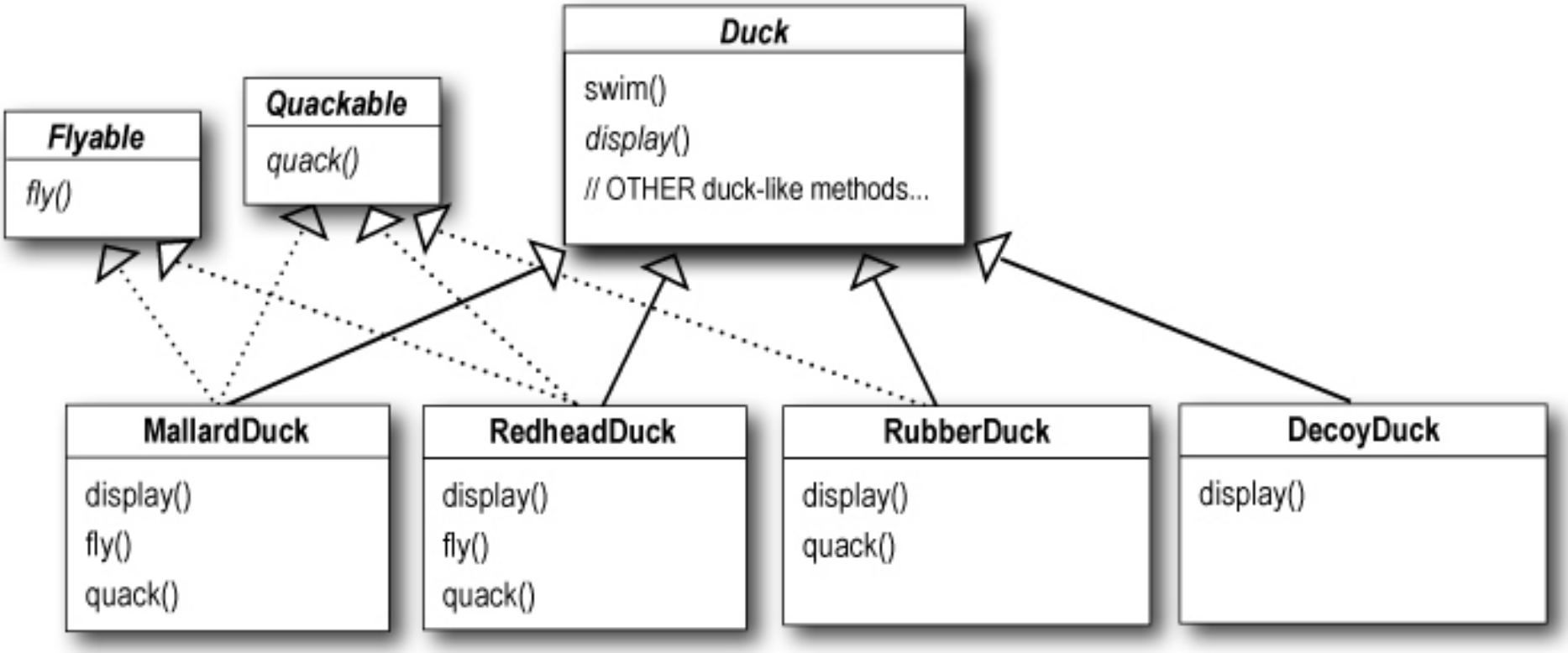
- Declaring the variable "d" as type Dog forces us to code to a concrete implementation.

- Programming to an **interface/supertype** would be:

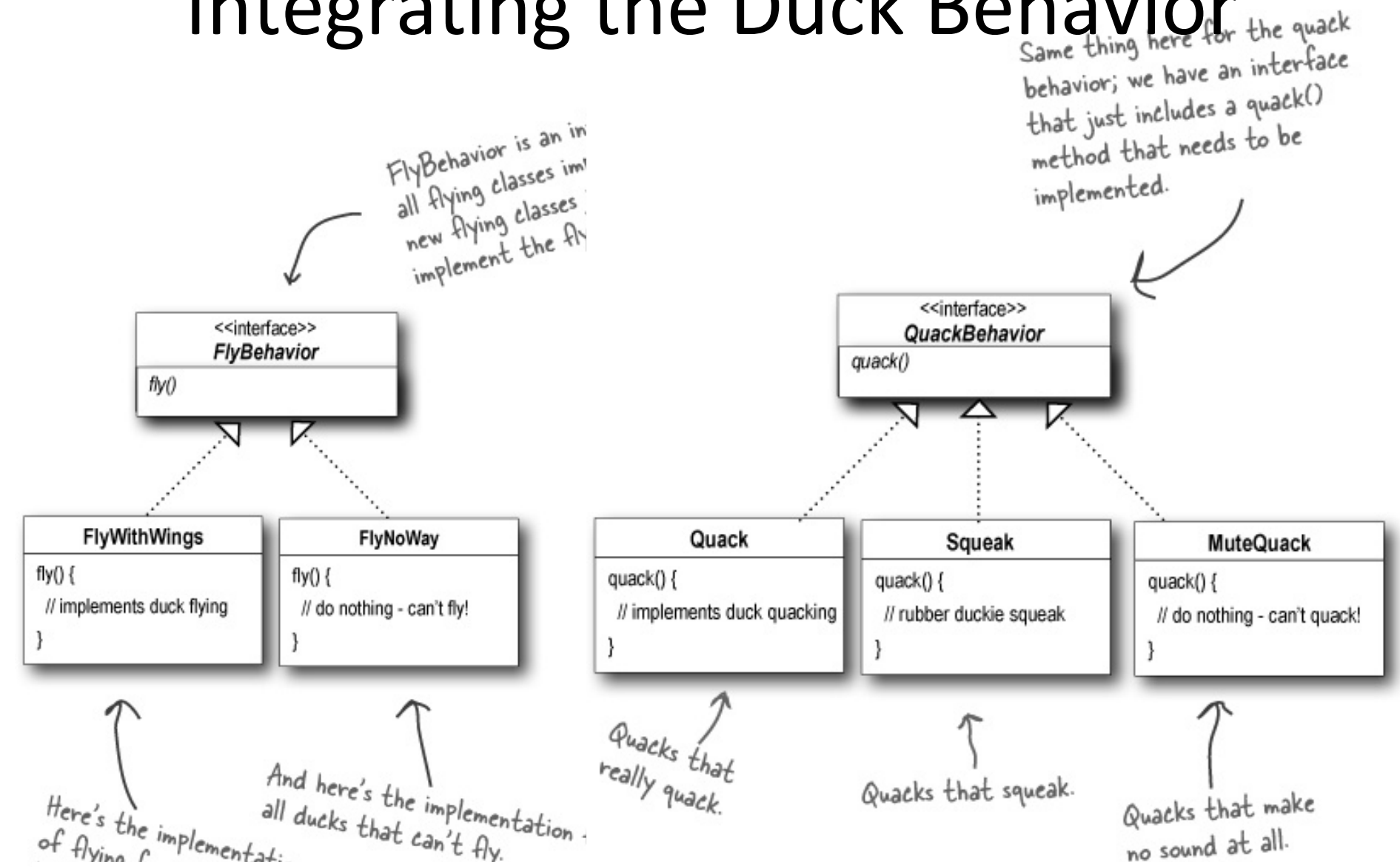


- We know it's a Dog, but we can now use the animal reference polymorphically.

Do you remember?

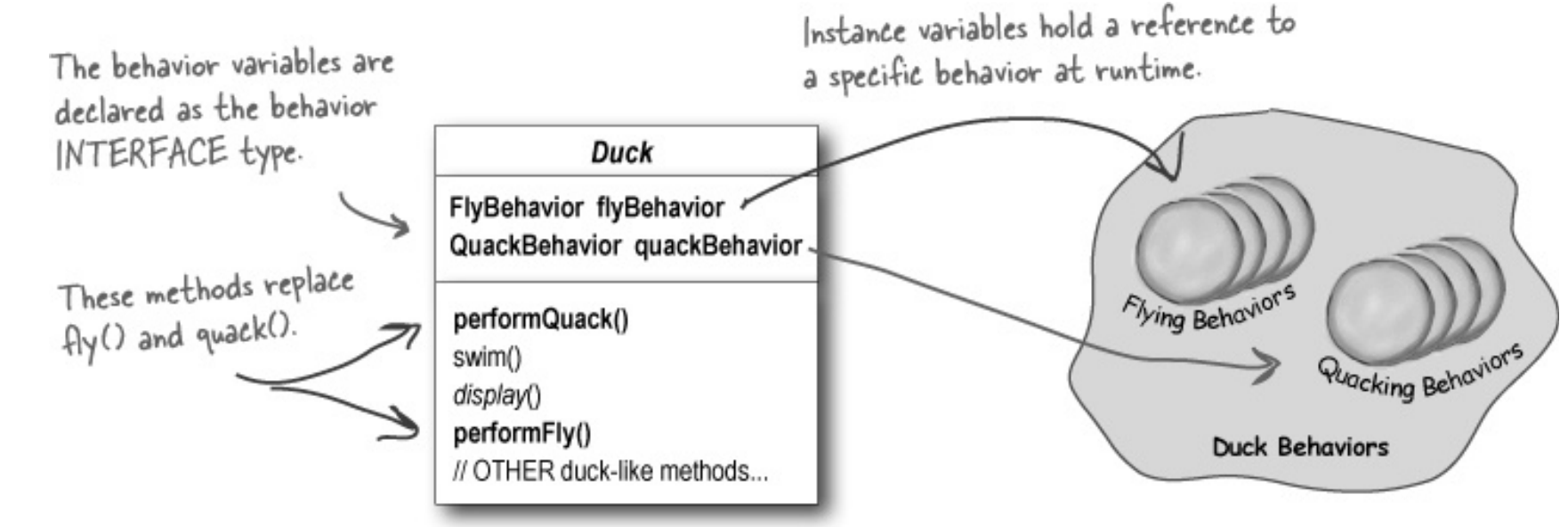


Integrating the Duck Behavior



The key is that a Duck will now ***delegate*** its behavior, instead of using methods defined in the Duck class (or subclass).

1. **First we'll add two instance variables**
2. **Now we implement performQuack()**




```
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
      quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

How the flyBehavior and quackBehavior instance variables are set

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

The constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack

However, it is also possible to set the behavior dynamically

Joe applied the Strategy pattern

Encapsulate what varies.

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t

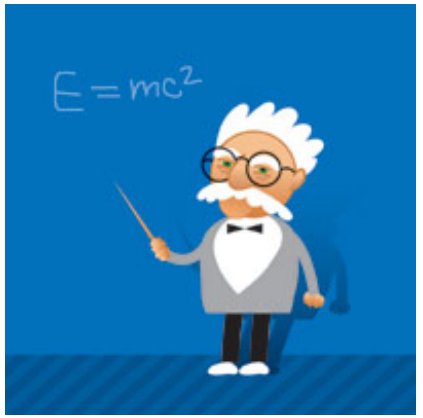
Program to an interface,
not an implementation.

Don't declare variables to
be instances of particular
concrete classes.
Instead, commit only to
an interface defined by
an abstract class.

HAS-A
can be better than
IS-A

Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task

The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is.



Design Patterns
solve
Design Problems

Determining Object Granularity

FACADE, FLYWEIGHT, ABSTRACT FACTORY, BUILDER, VISITOR, COMMAND

- How do we decide what should be an object?
- Objects can vary in size and number
- They can represent everything
- They may have no counterparts in the real world

Specifying Object Interfaces

MEMENTO, DECORATOR, PROXY, VISITOR

- Objects are known only through their interfaces
- Two objects having completely different implementations can have identical interfaces
- Defining interfaces means
  - identifying their key elements and
  - the kinds of data that get sent across an interface
- It is also important to specify relationships between interfaces

Specifying Object Implementation

CHAIN OF RESPONSIBILITY, COMPOSITE, COMMAND, OBSERVER, STATE, STRATEGY
ABSTRACT FACTORY, BUILDER, FACTORY METHOD, PROTOTYPE, SINGLETON
STATE, STRATEGY, VISITOR, MEDIATOR, BRIDGE

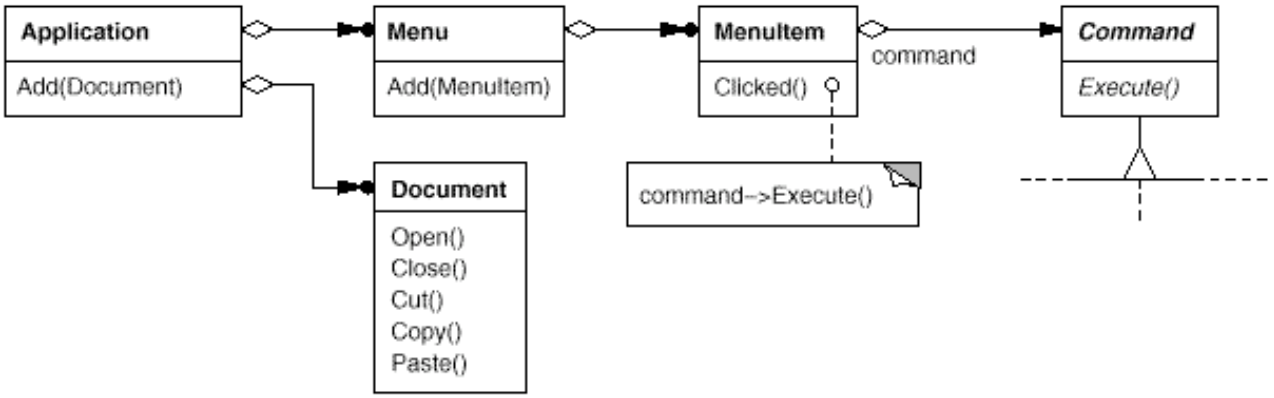
- Class vs Interface Inheritance
- Programming to an Interface, not an Implementation
- Inheritance versus Composition
- Delegation

EXAMPLES
OF DESIGN PATTERNS

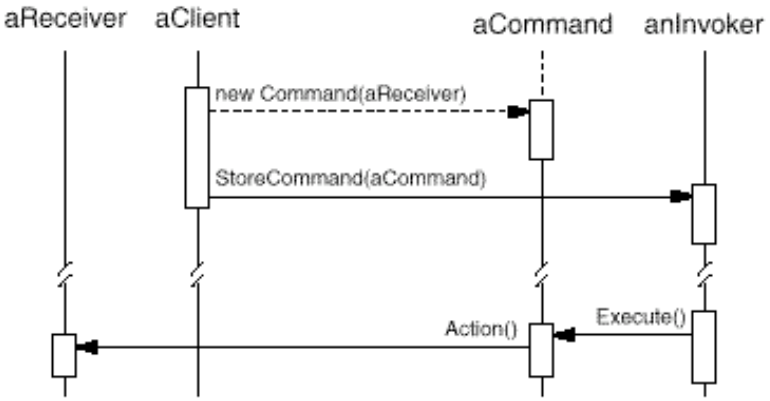
1. Command

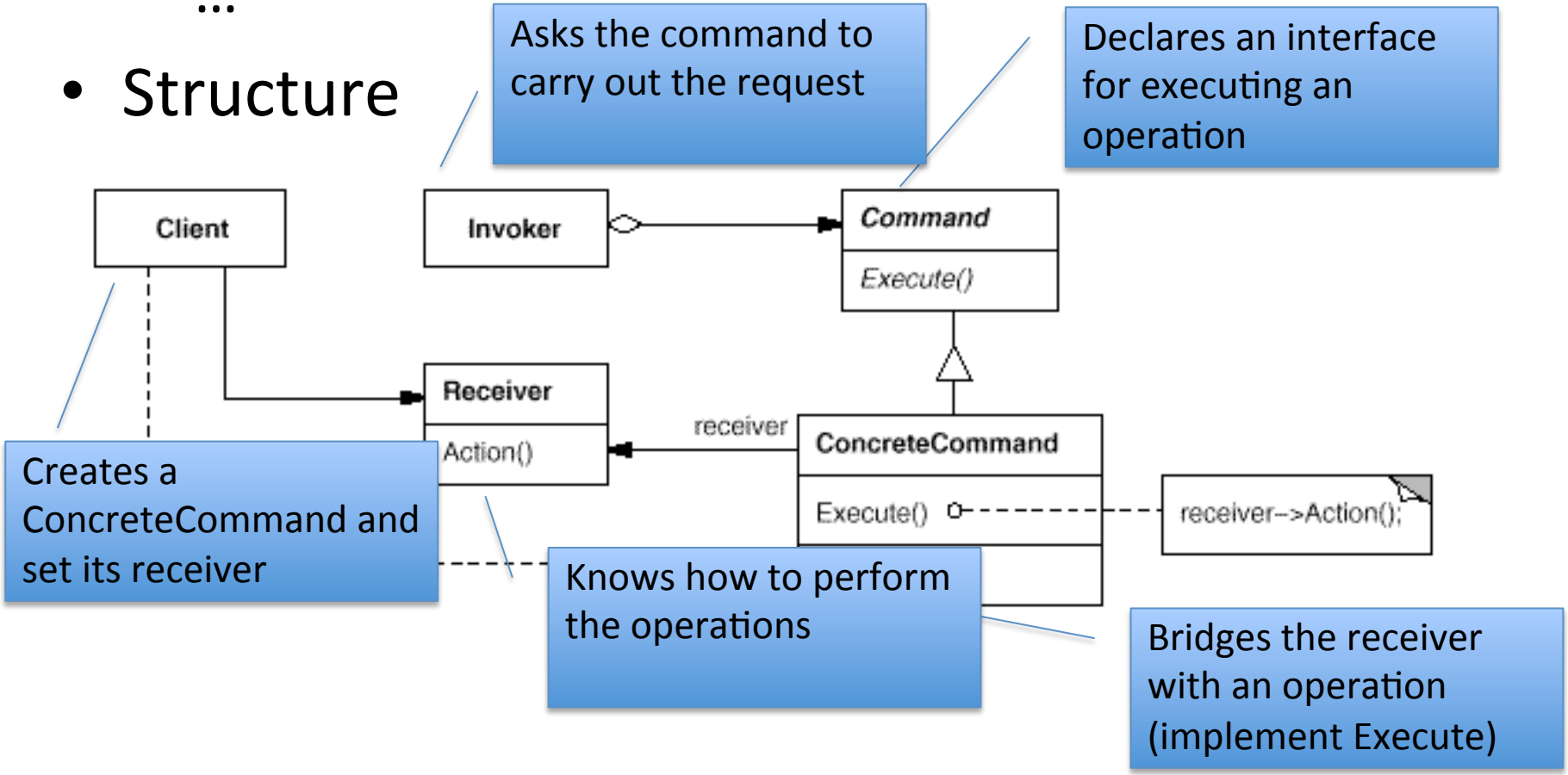
- Intent: encapsulate a request as an object,…

- Motivation: sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request



- Applicability
  - Callback
  - Undo/Redo
  - Logging
  - …
- Structure





- Consequences:
  - Decoupling
    - the object that invokes the operation (invoker)
    - from the one that knows how to perfom it (receiver)
  - Easy to add new Commands
  - Easy to create families of Commands
  - Macro-commands

Receiver

```
class Fan {
        public void startRotate() {
                System.out.println("Fan is rotating");
        }
        public void stopRotate() {
                System.out.println("Fan is not rotating");
        }
}

class Light {
        public void turnOn( ) {
                System.out.println("Light is on ");
        }
        public void turnOff( ) {
                System.out.println("Light is off");
        }
}
```

Command/Concrete Commands

```
public interface Command {
        public abstract void execute ( );
}
```

```
class LightOnCommand implements Command {
        private Light myLight;
        public LightOnCommand ( Light L) {
                myLight  =  L;
        }
        public void execute( ) {
                myLight . turnOn( );
        }
}
class LightOffCommand implements Command {
        private Light myLight;
        public LightOffCommand ( Light L)
{
                myLight  =  L;
        }
        public void execute( ) {
                myLight . turnOff( );
        }
}
```

```
class FanOnCommand implements Command {
        private Fan myFan;
        public FanOnCommand ( Fan F) {
                myFan  =  F;
        }
        public void execute( ) {
                myFan . startRotate( );
        }
}
class FanOffCommand implements Command {
        private Fan myFan;
        public FanOffCommand ( Fan F) {
                myFan  =  F;
        }
        public void execute( ) {
                myFan . stopRotate( );
        }
}
```

Invoker

```
class Switch {
        private Command UpCommand, DownCommand;
        public Switch( Command Up, Command Down) {
                UpCommand = Up;
                DownCommand = Down;
        }
        void flipUp( ) {
                        UpCommand . execute ( ) ;
        }
        void flipDown( ) {
                        DownCommand . execute ( );
        }
}
```

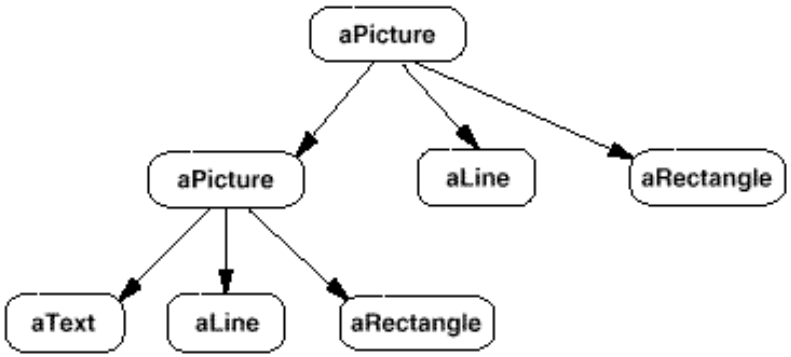
Client

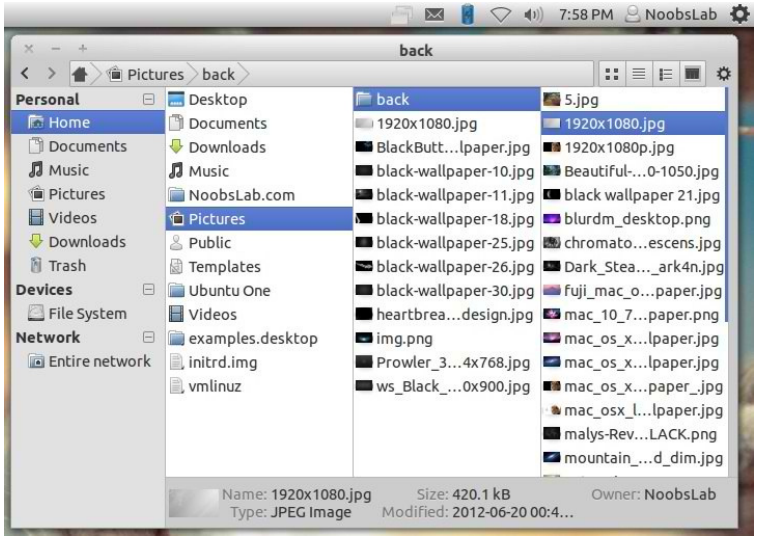
```
public static void main(String[] args) {
                        Light  testLight = new Light( );
                        LightOnCommand testLOC = new LightOnCommand(testLight);
                        LightOffCommand testLFC = new LightOffCommand(testLight);
                        Switch testSwitch = new Switch( testLOC,testLFC);
                        testSwitch.flipUp( );
                        testSwitch.flipDown( );

              Fan testFan = new Fan( );
                        FanOnCommand foc = new FanOnCommand(testFan);
                        FanOffCommand ffc = new FanOffCommand(testFan);
                        Switch ts = new Switch( foc,ffc);
                        ts.flipUp( );
                        ts.flipDown( );
                }
```

2. Composite

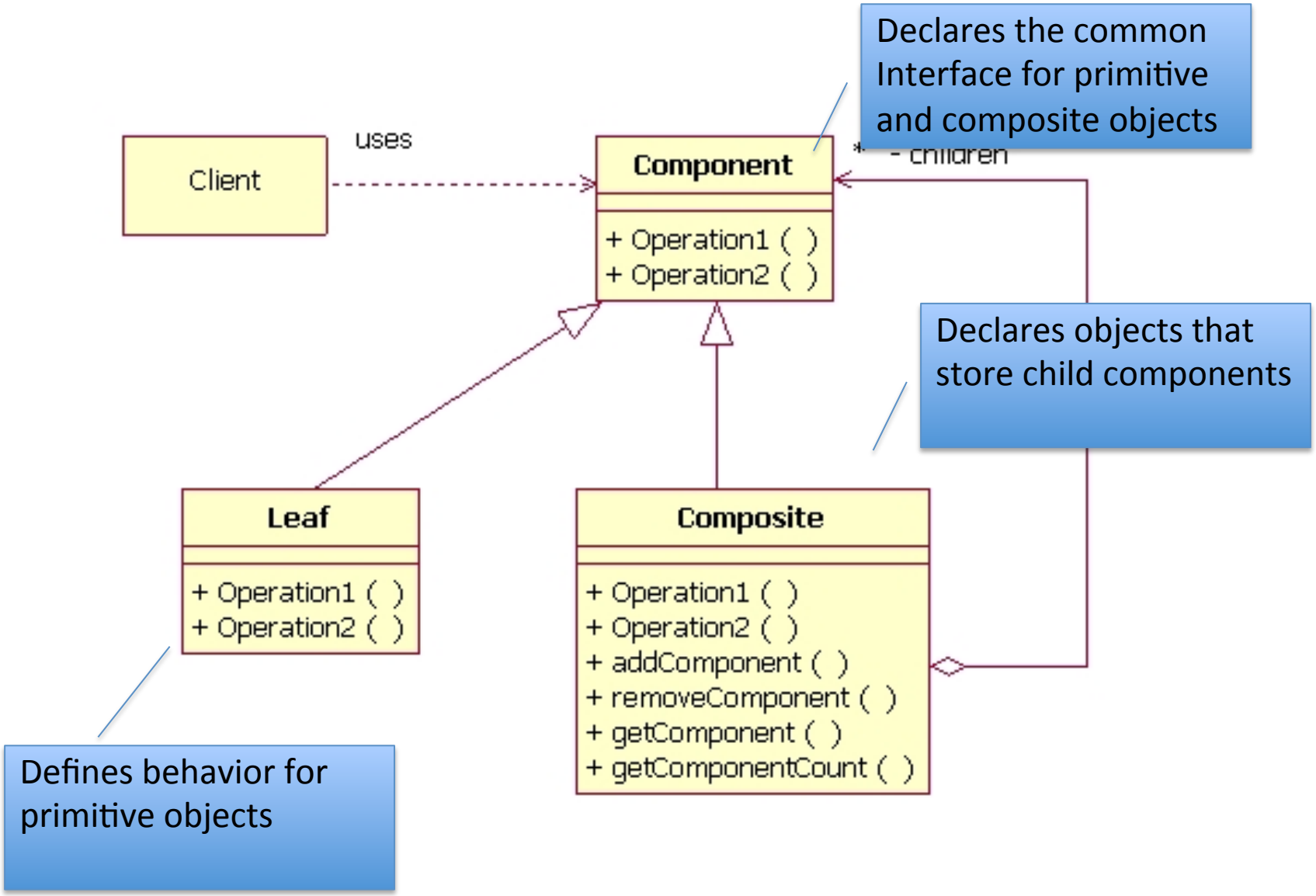
- Intent: compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

- Motivation: defining hierarchies of primitive and composite objects





Structure


Declares the common Interface for primitive and composite objects
Client
uses
Component
+ Operation1 ( )
+ Operation2 ( )
- children
Declares objects that store child components
Leaf
+ Operation1 ( )
+ Operation2 ( )
Composite
+ Operation1 ( )
+ Operation2 ( )
+ addComponent ( )
+ removeComponent ( )
+ getComponent ( )
+ getComponentCount ( )
Defines behavior for primitive objects

- Consequences
  - Defines class hierarchies consisting of primitive and (recursively) composite objects
  - Makes the client simple
    - Primitive and composite objects can be treated uniformly
  - Makes it easier to add new kind of components
  - Makes it harder to restrict the components of a composite

Component, Leaf and Composte

```
interface AbstractFile {
  public void ls();
}
```

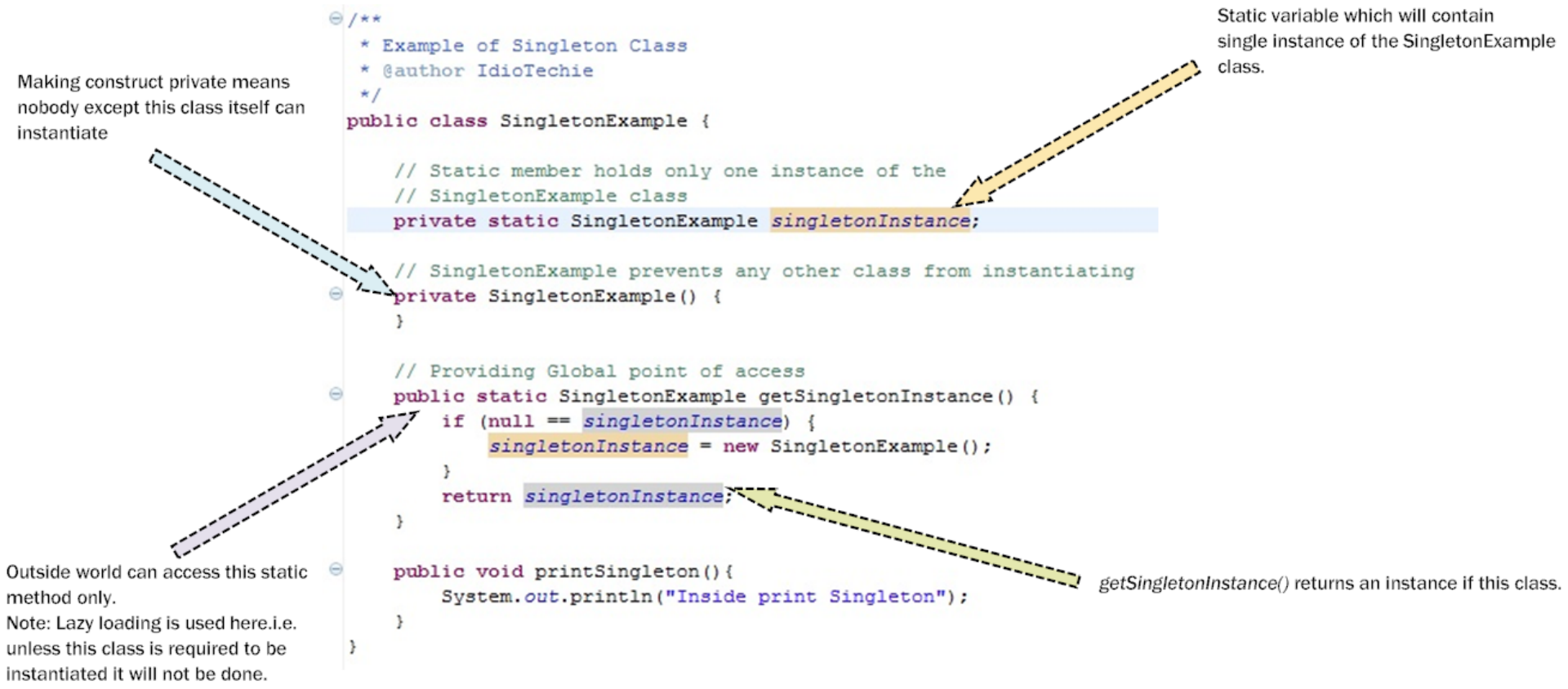
```
class File implements AbstractFile {
    private String m_name;

    public File(String name) {
        m_name = name;
    }
    public void ls() {
        System.out.println(m_name);
    }
}
```

```
class Directory implements AbstractFile {
 private String m_name;
    private ArrayList m_files = new ArrayList();

    public Directory(String name) {
        m_name = name;
    }
    public void add(Object obj) {
        m_files.add(obj);
    }
    public void ls() {
        System.out.println(m_name);
        for (int i = 0; i < m_files.size(); ++i)  {

            AbstractFile obj = (AbstractFile)
m_files.get(i);
            obj.ls();
        }
    }
}
```

3. Singleton

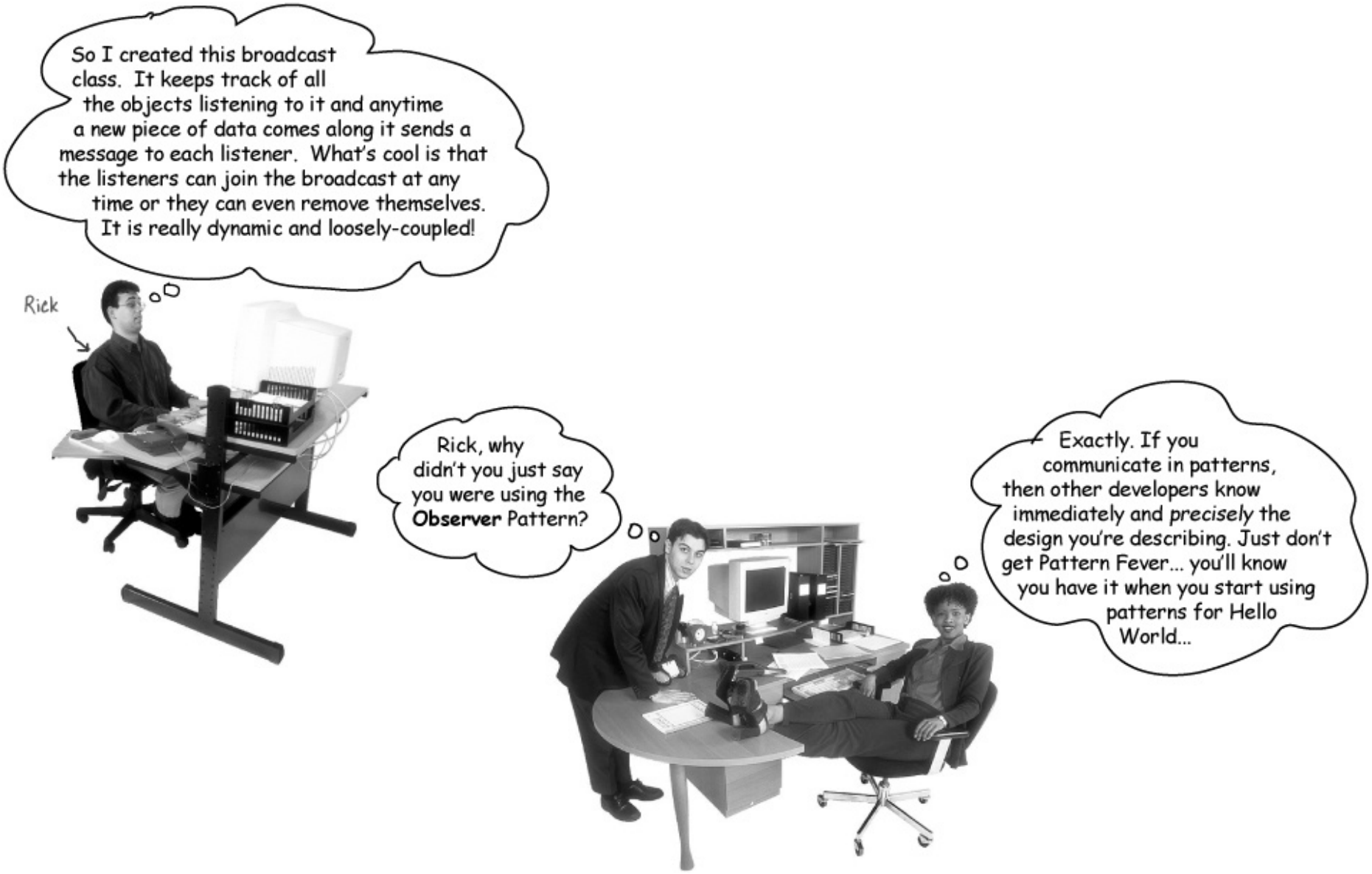
- Intent: ensure a class has one and only one instance, and provide a global point of access to it

- Motivation: It's important for some classes to have exactly one instance.
  - In a system, there should be only one printer spooler and only one file system.
  - A digital filter will have one A/D converter.
  - An accounting system will be dedicated to serving one company.

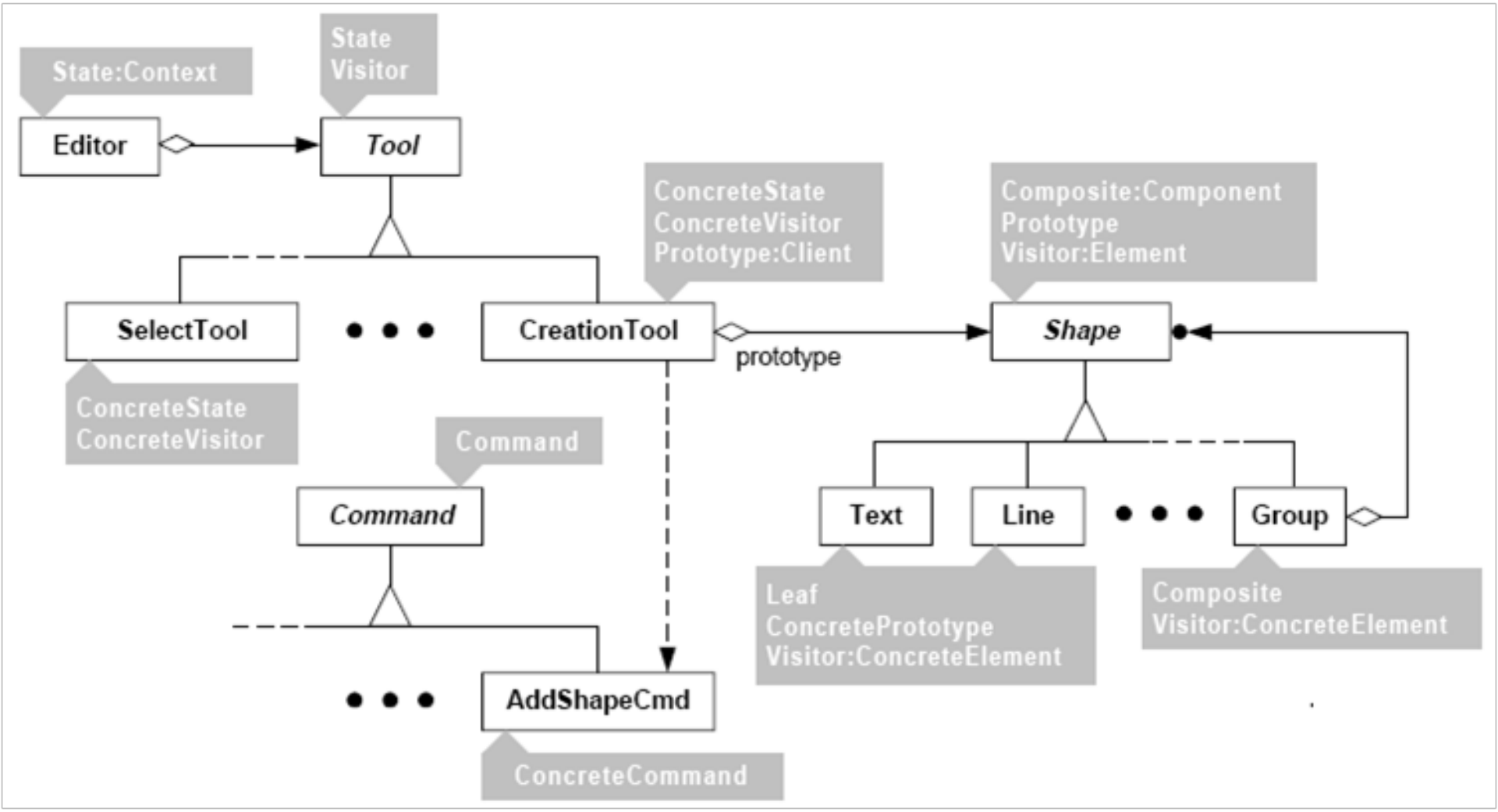



```java
/**
 * Example of Singleton Class
 * @author IdioTechie
 */
public class SingletonExample {

    // Static member holds only one instance of the
    // SingletonExample class
    private static SingletonExample singletonInstance;

    // SingletonExample prevents any other class from instantiating
    private SingletonExample() {
    }

    // Providing Global point of access
    public static SingletonExample getSingletonInstance() {
        if (null == singletonInstance) {
            singletonInstance = new SingletonExample();
        }
        return singletonInstance;
    }

    public void printSingleton(){
        System.out.println("Inside print Singleton");
    }
}
```

Patterns as a shared Vocabulary



Composing Design Patterns



Grazie per l'attenzione!

sabatucci@pa.icar.cnr.it

ECOS Lab
Engineering COmplex and Smart systems

ICAR CNR

http://ecos.pa.icar.cnr.it