Object-Oriented Design and Programming

Andy Podgurski
EECS Department
Case Western Reserve University

Characteristics of OOD/OOP

- Three main characteristics:
  - *Encapsulation*
  - *Inheritance*
  - *Polymorphism*
- These are supported by OOPLs.
- Each is actually a *design pattern*.

Encapsulation

- The fundamental principle of OOD is to *design for change*.
- The primary means of accomplishing this is to *encapsulate what varies*.
- Interfaces *shield* clients from implementation changes.

Examples of Changeable Design Decisions

- *Data representation*
  - array or linked list
  - big endian or little endian byte order
  - ASCII or Unicode
- *Algorithms*
  - heapsort or quicksort
  - top-down or bottom-up parsing
  - round-robin or priority scheduling
- *Hardware or software platform*
  - RISC or CISC
  - Windows or UNIX
  - Python or Java
- *User interface*
  - command language or GUI
  - English or Chinese
  - GUI components or HTML with JavaScript

Inheritance

- A class B can *inherit* part of its interface and/or implementation from another class A.
- B is said to be *derived* from A.
- B is called a *subclass* or a *derived class* of A.
- A is called a *superclass* or a *base class* of B.
- Inheritance is *transitive*: if B is a subclass of A and if C is a subclass of B, then C is a subclass of A.

Inheritance cont.

- If B is derived directly from A, then B is called a *direct subclass or child* of A and A is called a *direct superclass* or *parent* of B.
- A subclass can *augment* the attributes and methods of its parent class.
- A subclass can also *override* a method of its parent class by providing a new implementation.

Example: Inheritance in Java

```
class Employee {
        public Employee(String n, double s, Day d) { … }
        public void printInfo() { … }
        public int hireYear() { … }
        public void raiseSalary(double byPercent) {
                salary *= 1 + byPercent / 100;
        }
        private String name;
        private double salary;
        private Day hireDay;
}

class Manager extends Employee {
        public Manager(String n, double s, Day d) { … }
        public String raiseSalary(double byPercent) { // Overrides parent class method
                // add 1/2% bonus for every year of service
                Day today = new Day();
                double bonus = 0.5 * (today.getYear() – hireYear());
                super.raiseSalary(byPercent + bonus);
        }
        public void setSecretaryName(String n) { … }
        public String getSecretaryName() { … }
        private String secretaryName;
}
```

From Core Java 2, Volume 1 by C.S. Horstman and G. Cornell

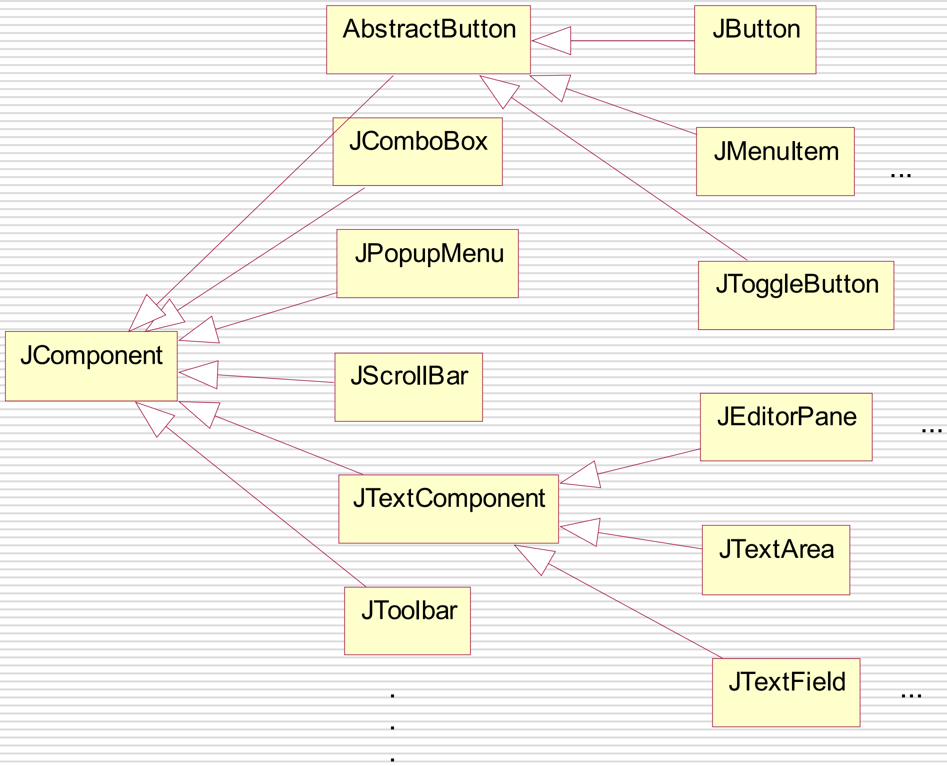
Inheritance and the "is a" Relationship

- Inheritance corresponds to the *is a* relationship.
- If B is a subclass of A, then every instance of B *is an* instance of A.
- That is, a subclass is a *specialization* of its superclasses.

Examples of the “is a” Relationship

- A *Human is a Primate*.
- A *Primate is a Mammal*.
- A *Mammal is an Animal*.
- A *DVDdrive is a Device*.
- A *RedBlackTree is a Dictionary*.
- A *Button is a Component*.
- A *SCARA_Robot is a Robot*.

Example: Inheritance Hierarchy



Subset of Java Swing Class Hierarchy

Uses of Inheritance

- Class *extension*
- *Classification* of objects (*modeling*)
- *Reuse*
  - of implementation code
  - of interfaces
- Class *restriction* (risky)
- Enabling *polymorphism*

Misuse of Inheritance

- Commonly occurs when inheritance is used to provide part of the *implementation* of a class
- Arises when derived class inherits *spurious* attributes and operations
- Avoided by checking if the *"is-a" relation* holds between derived class and its parent

Example: Misused Inheritance

```
public class Stack extends Array { ... }

s = new Stack(10);

s[5] = "abacadabra";  // Violates stack protocol.
```

A stack *is not* an Array.

This problem can be avoided by *encapsulating* the implementation class (Array) using *object composition*.

In C++, the problem can be addressed with *private inheritance*, e.g.,

```
class Stack<class T> : private Array<class T> { ...  }
```

Multiple Inheritance

- In some OOPLs, a class is permitted to inherit from *multiple parents*.
- The derived class inherits the attributes and methods of *all* of its parent classes.
- This is called *multiple inheritance*.

Example: Multiple Inheritance in C++

```
class Professor : public UniversityEmployee {
public:
    list<Course*> getCourses();
    list<Student*> getAdvisees();
    list<Grant*> getGrants();
...
}
class Administrator : public UniversityEmployee {
public:
    list<UniversityEmployee*> getManagees();
    AdministrativeUnit& getAdministrativeUnit();
...
}
class DepartmentChair : public Professor, public Administrator { ...
}
```

Ambiguities with Multiple Inheritance

- Multiple inheritance is trickier than it looks, especially in C++.
- Different kinds of *ambiguities* that can occur.
- One of these is a *name conflict* involving members of base classes.
- Another ambiguity arises when there are *multiple paths* in an inheritance graph from a base class to a derived class.

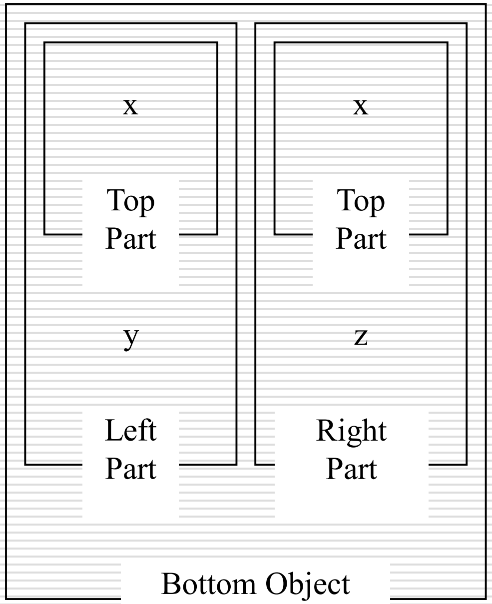
Example: Name Conflict

```
class Base1 {
public:
      void f();
      void g();
...
};
class Base2 {
public:
      void f();
      void h();
...
};
class Derived : public Base1, public Base2 {
...
};

d.f()               // compile-time error
d.Base1::f();       // unambiguous
d.Base2::f();       // unambiguous
```

Example: Ambiguity Due to Multiple Inheritance Paths

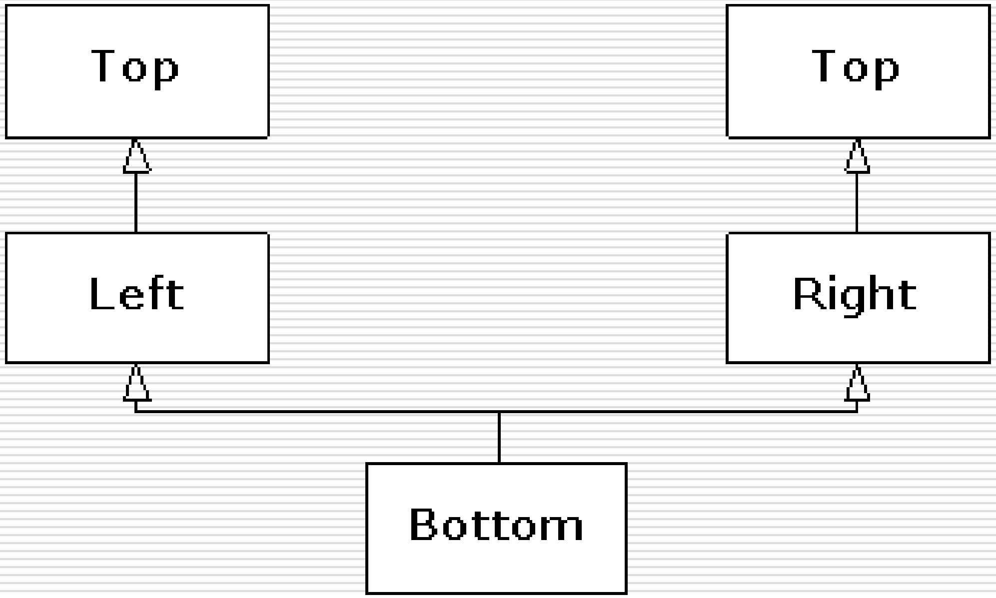
```
class Top {
protected:
    int x;
...
public:
    void m();
};
class Left : public Top {
protected:
    int y;
...
};
class Right : public Top {
protected:
    int z;
...
};
class Bottom : public Left, public Right
      {
...
};
```

```
Bottom::someMethod() {
    x = 0;  // ambiguous
};

...

Bottom b;
b.m();  // ambiguous
```



From *C++ Programming Style* by T. Cargill

Example cont. (2): Multiple Paths to Class *Bottom* in Inheritance Diagram

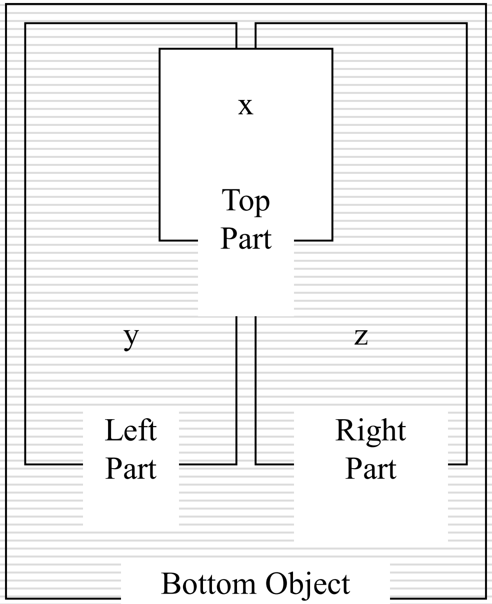


Example cont. (3): Resolving the Ambiguity

```
Bottom::someMethod() {
    Left::x = 0;            // OK; "::" is scope resolution
};                          // operator in C++

...

Bottom b;
b.Right::m();               //OK
```

Example cont. (4): Virtual Base Class in C++

```
class Top {
protected:
    int x;
...
public:
    void m();
};

class Left : public virtual Top {
protected:
    int y;
...
};

class Right : public virtual Top {
protected:
    int z;
}

class Bottom : public Left, public Right {
...
};
```



Interfaces and Types

- An *interface* is a set of operation signatures*.
  - An object's interface is the complete set of requests to which it responds.
  - An interface can *contain* another interface as a subset.
- A *type* is a named interface.
- One object may have *multiple types*.
  - Each corresponds to a *subset* of its set of operation signatures.
  - Example: multiple inheritance

*Method signature: method name and list of parameter types

Interfaces and Types cont.

- Different kinds of objects can also *share* a type.
- This means they share *part* of their interfaces.
- A type T is a *subtype* of another type T' (its *supertype*) if T contains the operations of T'.
  - Example: inheritance

Guidelines for Defining Interfaces [Wirfs-Brock]

- *Group* responsibilities used by the same clients.
- *Maximize* the conceptual cohesiveness of interfaces.
- *Minimize* the number of interfaces per class.

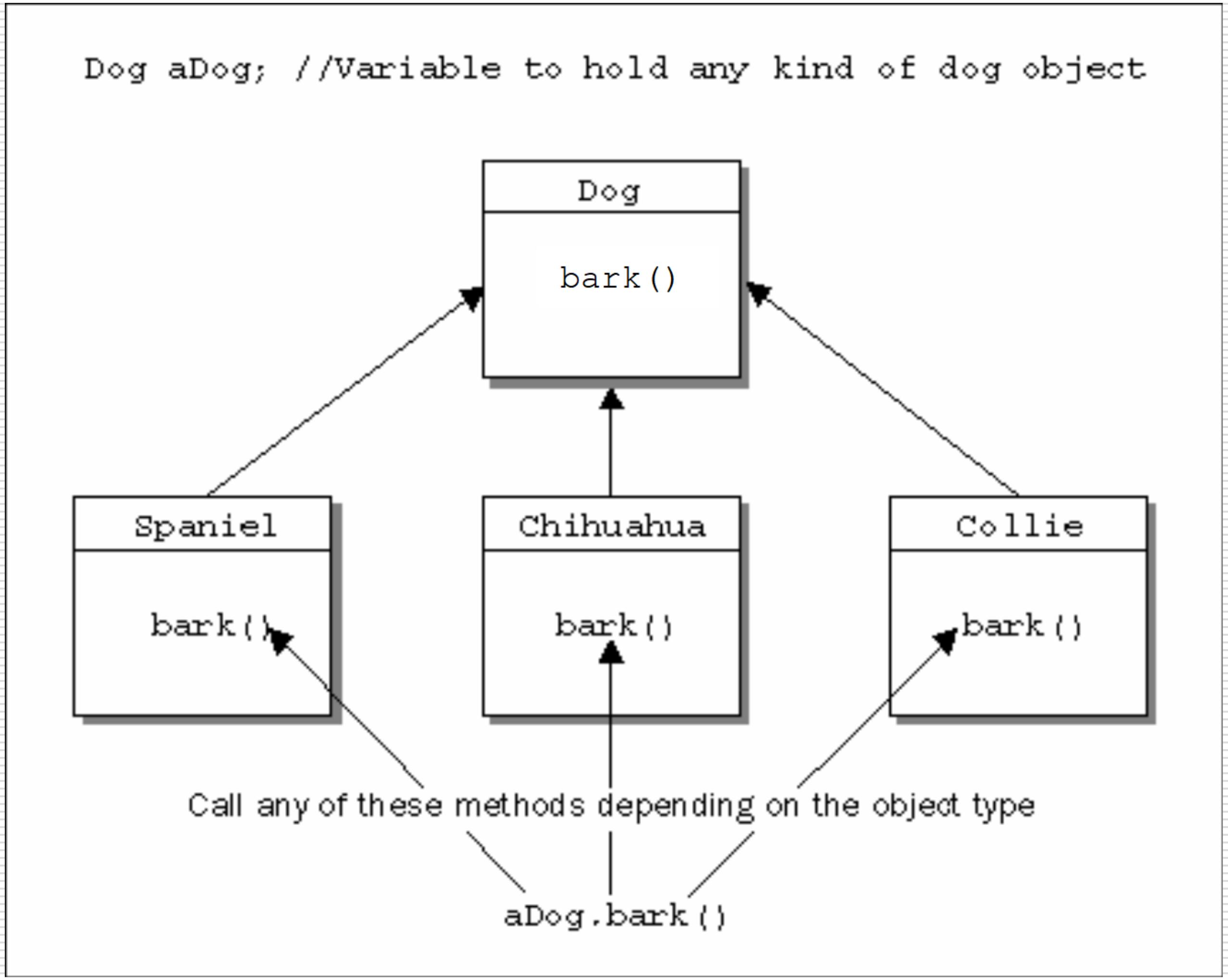
Polymorphism and Dynamic Binding

- OOPLs permit writing code that can interact with any object supporting a given interface.
  - The *same code* will work with objects that implement the interface *differently*.
  - This ability to substitute objects of different classes is called *polymorphism*.
- The object and operation used to respond to a request may be *found at runtime*.
  - This is called *dynamic binding* (of message to method).

Polymorphism and Dynamic Binding cont.

- In statically typed languages like Java and C++ the targets of polymorphic requests must implement a *specified type* (interface).
- This is called *subtype polymorphism*.

Example: Polymorphic Call



From *Beginning Java* by Ivor Horton

Example: Polymorphism in Java

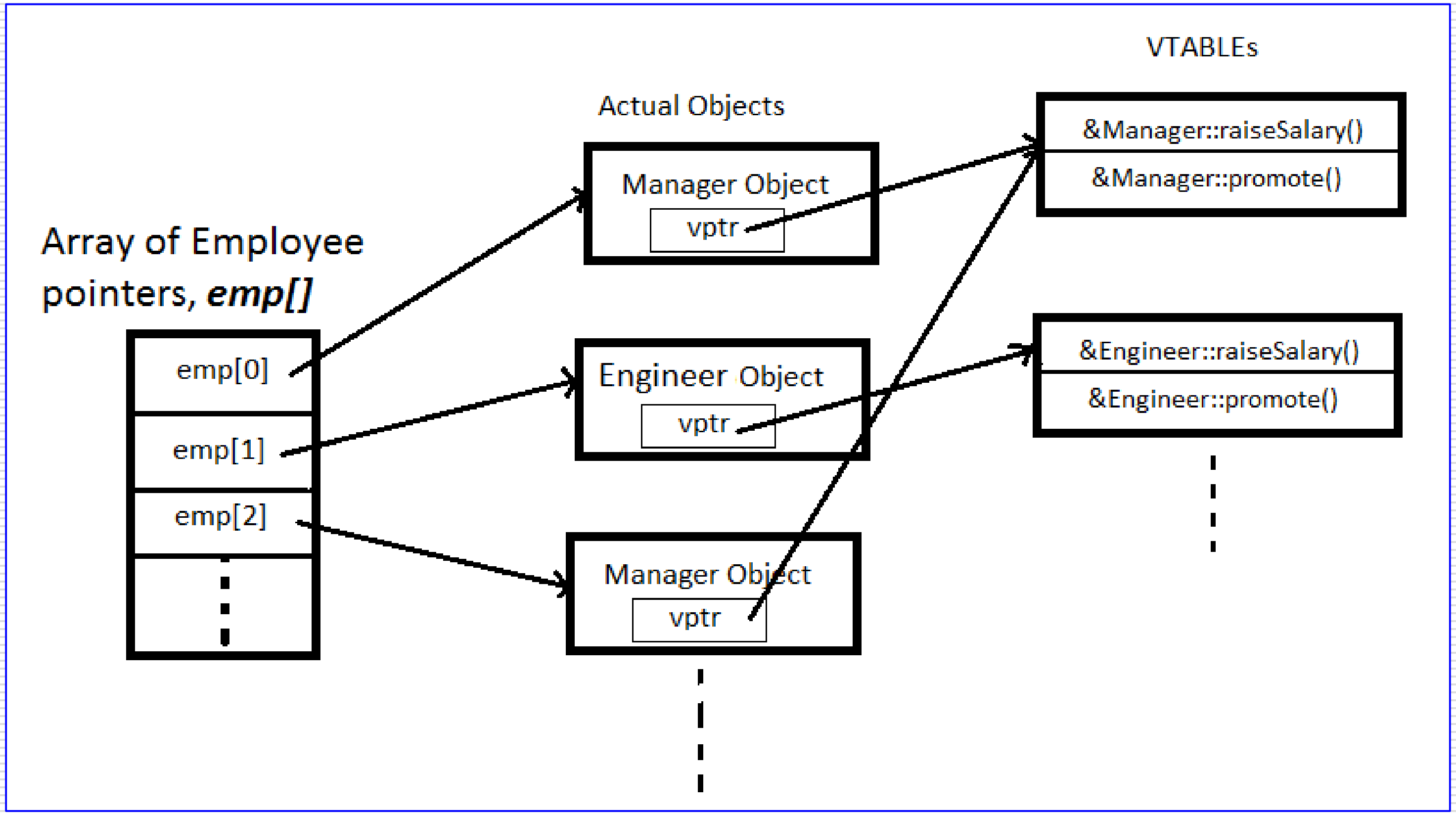
```
abstract class Phrases {
        public void sayHello() { };
        public static void greetGroup(Phrases[] greetings) { // Polymorphic method
                for (Phrases p : greetings)
                                p.sayHello();
        }
}
class English extends Phrases {
        public void sayHello() { System.out.println("Hello"); }
}
class Spanish extends Phrases {
        public void sayHello() { System.out.println("Hola"); }
}
class Norwegian extends Phrases {
        public void sayHello() { System.out.println("Hei"); }
}
public class Main {
        public static void main(String[] args) {
                Phrases[] languages = new Phrases[3];
                languages[0] = new English();
                languages[1] = new Spanish();
                languages[2] = new Norwegian();
                Phrases.greetGroup(languages);  // Call to polymorphic method
        }
}
```

Adapted from www.ao.net/~juang/IntroJava

Example: Polymorphism in C++

```
class Shape {
public:
	virtual double GetArea() = 0;
	virtual void Draw() = 0;
};
class Circle : public Shape {
public:
	double GetArea();
	void Draw();
	double GetCircumference();
	double GetRadius();
...
};
class Square : public Shape {
public:
	double GetArea();
	void Draw();
	double GetSide();
...
}
...
void Canvas::DrawShapes(Shape* shapes[], int n) { // Polymorphic method
	for (i=0; i < n; i++)
		shapes[i]->Draw();
}
```

Dynamic Binding in C++ through Virtual Method Table



Advantages of Polymorphism

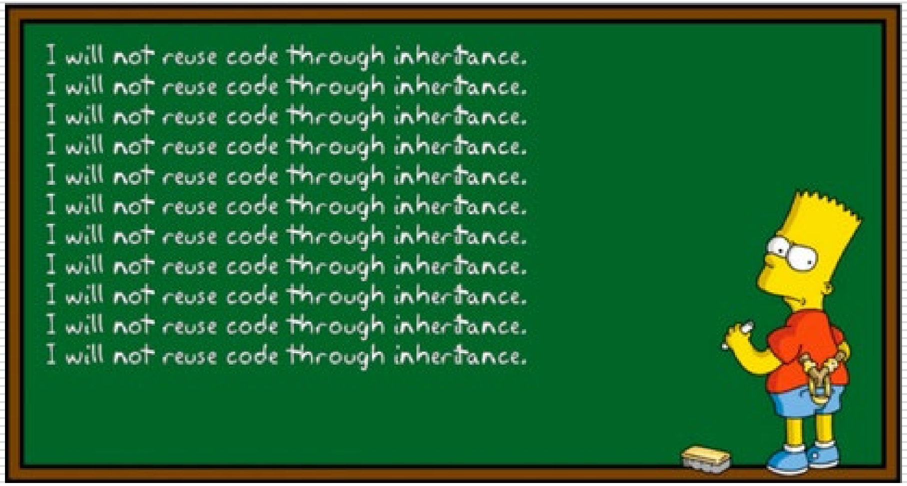
- It *simplifies* the definition of clients.
  - Why?
- It *decouples* objects from each other.
  - How?
- It lets them *vary their relationships* with each other at *runtime*.
- It permits existing code to be *used with new classes* of objects.

Interface Inheritance vs. Implementation Inheritance

- In *ordinary class inheritance*, a subclass typically inherits its parent's interface and implementation.
- *Implementation inheritance* allows a subclass to reuse its parent's code and representation.
- *Interface inheritance* (or *subtyping*) describes when
  - One type of object can be used *in place of another*
  - Class implementation is *not* inherited

Problems with Implementation Inheritance

- Implementation inheritance *breaks encapsulation*.
  - A change to its parent's implementation *forces* a subclass to change.
- Also, implementation inherited from a parent class *can't be changed at runtime*.



http://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html

Inheritance can break encapsulation

```
public class InstrumentedHashSet<E>
                              extends HashSet<E> {
  private int addCount = 0;  // count # insertions
  public InstrumentedHashSet(Collection<? extends E> c){
     super(c);
  }
  public boolean add(E o) {
     addCount++;
     return super.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size();
     return super.addAll(c);
  }
  public int getAddCount() { return addCount; }
}
```

If addAll uses add internally,
double-count will occur.

Zach Tatlock, http://slideplayer.com/slide/10740541/

Programming to an Interface

- An important principle of OOD [Gamma et al]:
  - *Program to an interface, not an implementation.*
- This implies that variables *should not* be declared to be of particular concrete classes.
- Instead, objects should be manipulated *solely in terms of interfaces* they support, so clients remain unaware of the:
  - Specific types of objects they use
  - Classes that implement these objects
- It is desirable to *"factor out"* operations that are common to a set of classes.
- These should be declared in a *separate interface*, which the classes implement.

Abstract Bases Classes

- One way to define an interface that may be implemented by multiple classes is to use an *abstract base class*:
  - This is a class that *defers* some or all of its implementation to *subclasses*.
  - It declares one or more *abstract operations*, for which it provides no implementation.
- *Polymorphic code* can be written in terms of abstract base class variables.
- It is best if an abstract base class *reveals no implementation details*.
- A *pure abstract base class* has only abstract operations.
- A class that is not abstract is called *concrete*.

Example: Pure Abstract Base Class in C++

```
class CharacterDevice {
public:
    virtual int open() = 0;
    virtual int close(const char *) = 0;
    virtual int read(const char *, int) = 0;
    virtual int write(const char *, int) = 0;
    virtual int ioctl(int ...) = 0;
...
};
```

From *The C++ Programming Language* by Bjarne Stroustrup

Interface Types

- Java and C# permit the declaration of *interface types*, e.g.,

```
public interface List extends Collection {
      public boolean add(Object o);
      public boolean contains(Object o);
      public Object get(int index);
      public boolean isEmpty();
      public Iterator iterator();
      public boolean remove(Object o);
      public int size();
…
}
public class LinkedList implements List { … };
…
List nameList = new LinkedList();
nameList.add("Iggy");
```

Implementation of Multiple Interfaces

- A class can implement multiple interfaces, e.g.,

```
public class DrawableScalableRectangle extends
    DrawableRectangle implements Drawable, Scalable { ...
};
```

From *Java in a Nutshell* by D. Flanagan

Object Composition

- *Object composition* is an alternative to implementation inheritance:
  - It involves implementing an object's functionality by assembling or composing other objects.
  - The references to the latter objects are made *private attributes* of the object being implemented.
  - The *has-a relationship* holds between the containing object and the contained ones.

- Object composition is an example of black-box reuse
  - It *does not break encapsulation*, because the composed objects are accessed solely through their interfaces.
  - Hence object composition creates *less coupling* then implementation inheritance.

Object Composition cont.

- Object composition is defined *dynamically*.
  - It permits the behavior of an object to be *altered at runtime*, by replacing references.
- Using it instead of implementation inheritance yields *smaller class hierarchies*.
- Designs based on object-composition tend to have *more objects* than ones based on implementation inheritance.
  - Object-composition produces systems whose behavior depends on the *interrelationships* between objects.
- Another principle of object-oriented design [Gamma et al]:
  - *Favor object composition over class inheritance.*
- Reuse by inheritance can make it easier to create new components that can be composed with old ones.

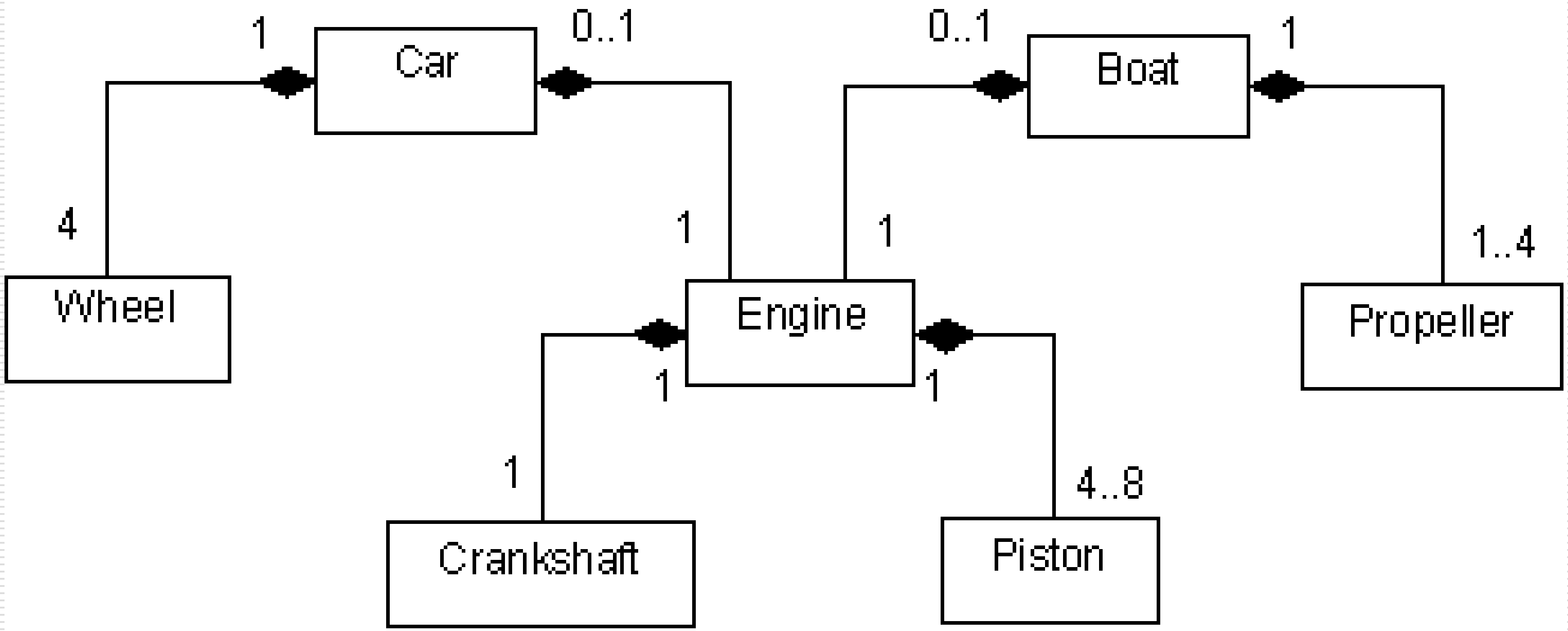
Example: Object Composition

```
public class Job {
    private String role;
    private long salary;
    private int id;

    public String getRole() { return role; }

    public void setRole(String role) {
        this.role = role;
    }
    public long getSalary() { return salary; }

    public void setSalary(long salary) {
        this.salary = salary;
    }
    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
}
```

```
public class Person {

    //composition has-a relationship
    private Job job;

    public Person(){
        this.job=new Job();
        job.setSalary(1000L);
    }
    public long getSalary() {
        return job.getSalary();
    }

}
```

www.journaldev.com/1325/composition-in-java-example

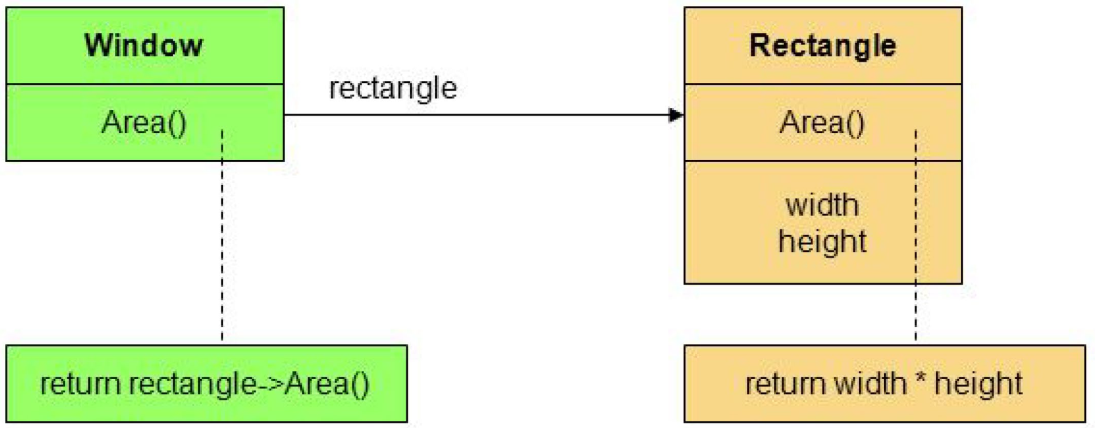
Example: Object Composition in UML



www.jot.fm/issues/issue_2004_11/column5/

Example: Delegation

- In *delegation*, an object delegates a request to its *delegate*, which it holds a reference to.
- This is analogous to a subclass deferring requests to its parent class.
- Delegation makes it easy to *compose* or *change* behaviors at runtime, e.g., by replacing *Rectangle* with *Circle*.



From *Design Patterns* by Gamma *et al*

Parameterized Types

- These are types whose definitions take *other types as parameters*.
- They are *instantiated* to produce concrete types.
- This involves *substituting concrete types* for type parameters.
- Parameterized types are called *templates* in C++ and *generics* in Java and C#.
- They provide another means of composing behavior.

Example: Sorting

- To parameterize sorting by the operation used to compare elements, we can make comparison:
  - An operation implemented by subclasses
  - The responsibility of an object passed to the sorting routine
  - An argument of a C++ template or of a Java or C# generic

Example: Generic Sorting Method in Java

```
public <E extends Comparable<E>> void selectionSort(E[] a) {
        for (int i = 0; i < a.length - 1; i++) {
            // find index of smallest element
            int smallest = i;
            for (int j = i + 1; j < a.length; j++) {
                if (a[j].compareTo(a[smallest])<=0) {
                    smallest = j;
                }
            }
            swap(a, i, smallest);  // swap smallest to front
        }
    }
```

stackoverflow.com/questions/18284636/implementing-comprable-method-in-a-generic-way-in-java-for-sorting

Parametric Polymorphism

- The form of polymorphism supported by parameterized types
- Facilitates compile-time *type checking*
  - Note that a type parameter *cannot* be changed at runtime

Generic Stack in C++

```
template <class TYPE>
class Stack {
public:
        Stack(): max_len(1000), top(EMPTY) {
                s = new TYPE[1000];
        }
        Stack(int size) : max_len(size), top(EMPTY) {
                s = new TYPE[size];
        }
        ~Stack() { delete [] s; }
        void reset() { top = EMPTY; }
        void push(TYPE c) { s[++top] = c; }
        TYPE pop() { return s[top--]; }
        TYPE top_of() const { return s[top]; }
        Boolean empty() const {
                return Boolean(top == EMPTY);
        }
        Boolean full() const {
                return Boolean(top == max_len - 1);
        }
private:
        enum            { EMPTY = -1 };
        TYPE*           s;
        int             max_len;
        int             top;
};
```

From C++ for C Programmers by Ira Pohl

C++ Generic Stack Example cont.

To define a member function outside of its class declaration, syntax like the following must be used:

```
template <class TYPE>
TYPE Stack<TYPE>::top_of() const {
        return s[top];
}
```

To instantiate different kinds of stacks, we write:

```
Stack<char>                stk_ch;
Stack<char*>               stk_str(200);
Stack<Complex>             stk_cmplx(100);
```

We might use the Stack class as follows:

```
// Reversing an array of strings
void reverse(char* str[], int n) {

        Stack<char*> stk(n);

        for (int k = 0; k < n; ++k)
                stk.push(str[k]);
        for (k = 0; k < n; ++k)
                str[k] = stk.pop();

}
```

Example: Java Generic List

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

From java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Example: Multiple Type Parameters in Java

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements
Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()    { return key; }
    public V getValue() { return value; }
```

The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new
OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new
OrderedPair<String, String>("hello", "world");
```

docs.oracle.com/javase/tutorial/java/generics/types.html

Overloading

- *Operator overloading* means giving a new meaning to a predefined operator of a programming language.
- The new meaning should be a *natural extension* of the predefined meaning.
  - Example: the multiplication operator "*" might be overloaded to support *matrix multiplication*, as in

    C = A * B;

    where A, B, C are matrices.
- *Function overloading* means giving multiple, related meanings to the same function name, e.g.,

```
class String {
public:
    String();
    String(const String&);
    String(const char*);
…
};
```

The Refinement Phase of OOD

- Activities:
  - Analyzing and improving inheritance hierarchies
  - Analyzing patterns of collaboration to identify subsystems
  - Designing methods
- Each activity may give rise to new classes and interfaces.

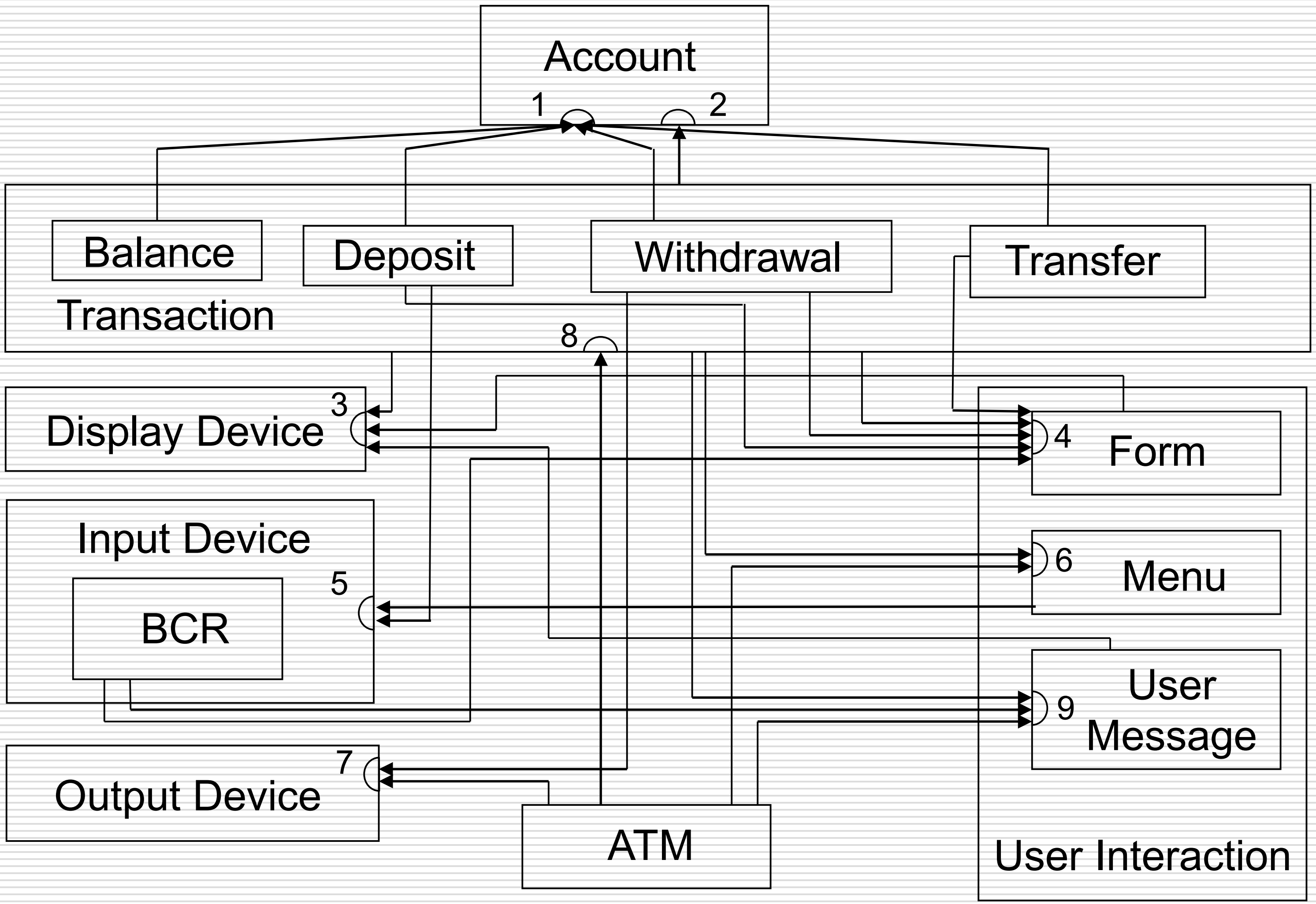
Guidelines for Constructing Inheritance Hierarchies

- Create "Is-a" hierarchies.
- *Factor* common responsibilities as high as possible.
  - Why?
- *Eliminate* classes/interfaces that do not add functionality.
- *Verify* that derived classes/interfaces support at least the responsibilities of their parents.
  - Why?

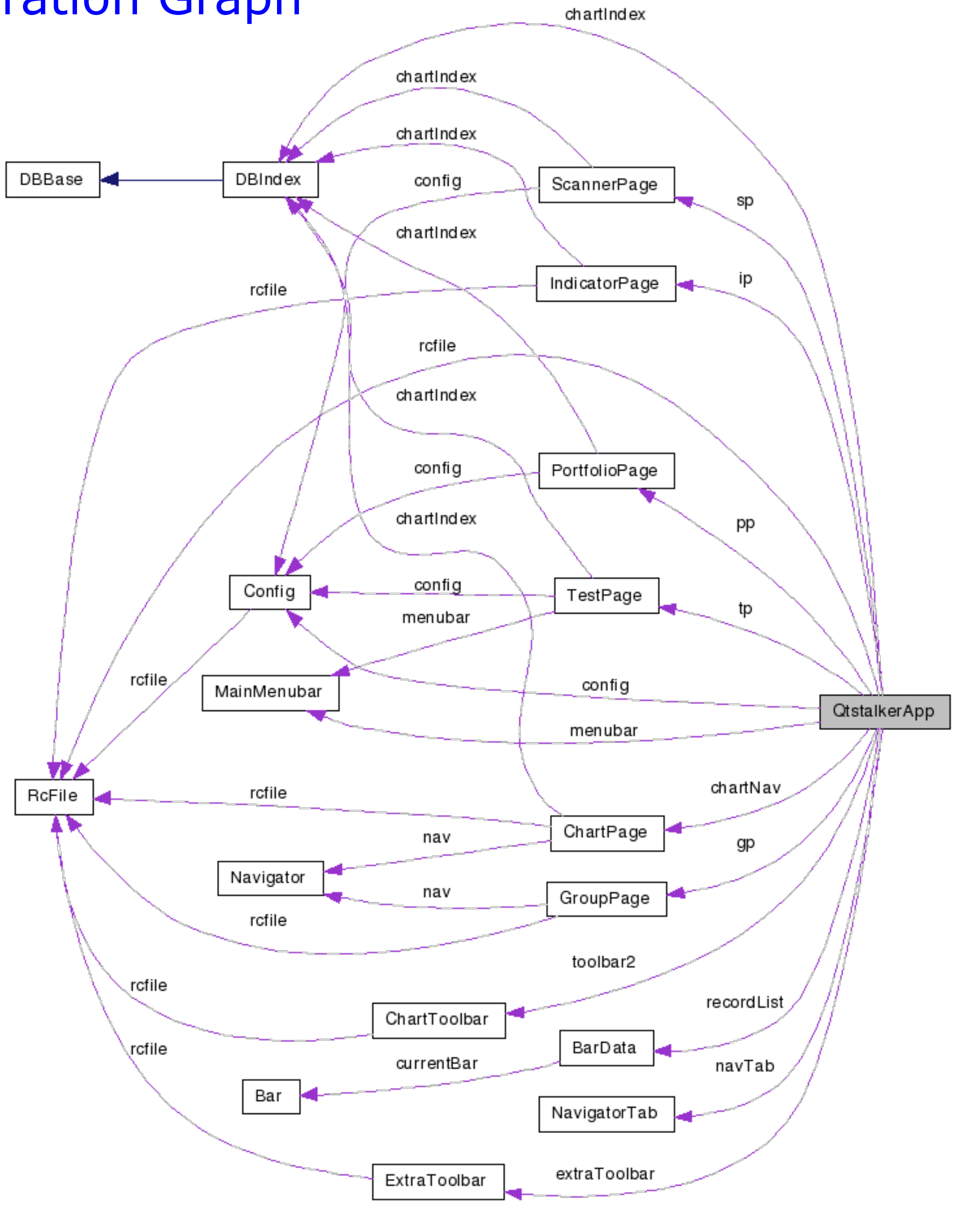
Identifying Subsystems

- A *subsystem* is a group of classes that collaborate to support a set of interfaces.
  - Unlike subclasses, the classes of a subsystem *fulfil different responsibilities*.
    - Example: *PrintingSubsystem*
      - Classes *PrintServer* and *Printer*
      - Subclasses *InkjetPrinter* and *LaserPrinter*
  - Like a class, a subsystem itself fulfills a set of responsibilities.
  - These may be partitioned into interfaces or *contracts*.
- A subsystem corresponds to *strongly coupled classes* in a *collaboration graph*.

Example: Initial ATM Collaboration Graph [Wirfs-Brock et al.]



Example Collaboration Graph



qtstalker.sourceforge.net/doc/class_qtstalker_app.html

Subsystem Interfaces

- A subsystem's interfaces are identified by examining its classes that provide services to *external clients*.
- A subsystem *delegates* each of its interfaces to a class or subsystem within it.
- A subsystem may be documented with a *card* similar to a CRC card.
  - This records the subsystem's name, its interfaces, and the class to which each interface is delegated.

Example: Drawing Subsystem of Drawing Editor

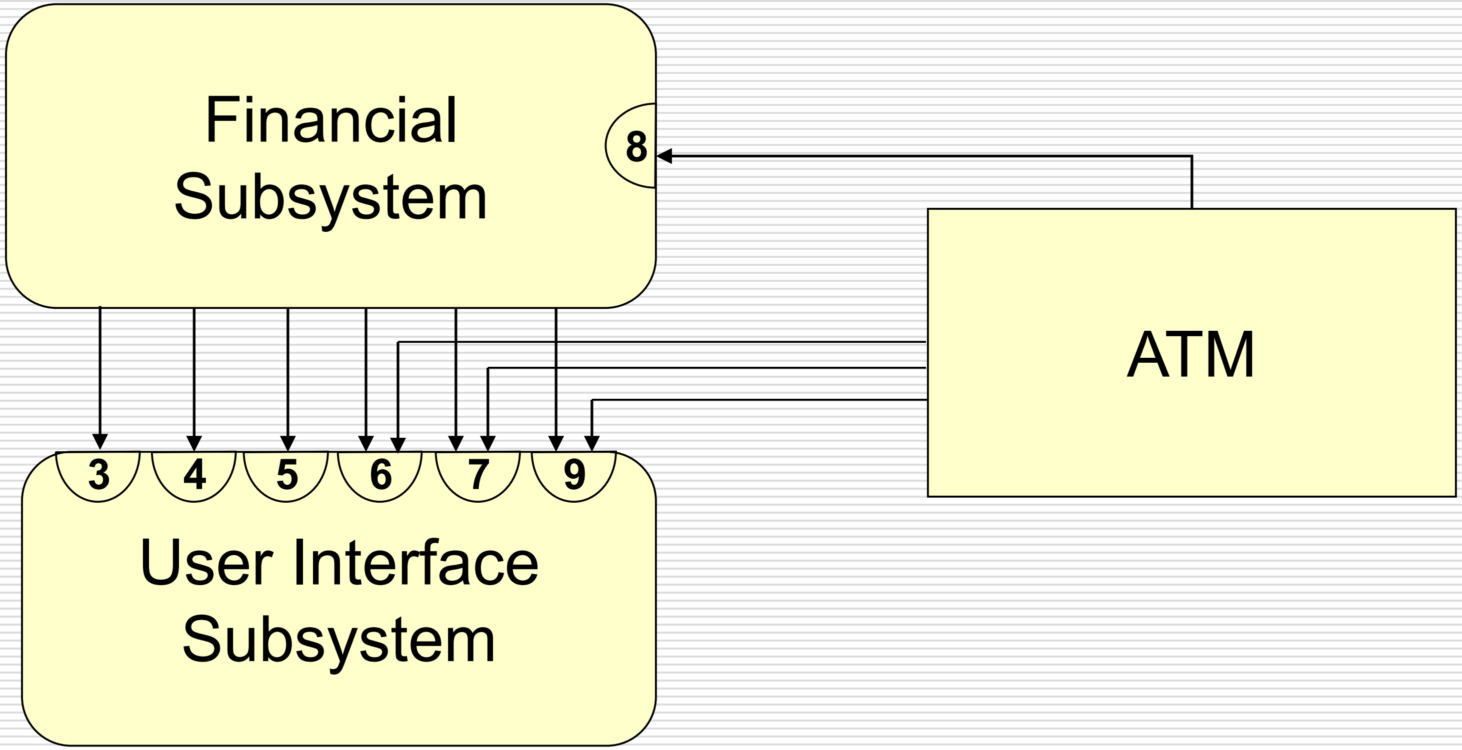
Subsystem: Drawing Subsystem	
Contracts:	**Delegations:**
Display itself.	Drawing
Maintain its elements.	Drawing Element
Modify an attribute of a drawing element.	Control Point
Test the location of a drawing element.	Drawing Element
Modify the geometry of a drawing element.	Drawing Element

From Designing Object-Oriented Software by R. Wirfs-Brock, et al

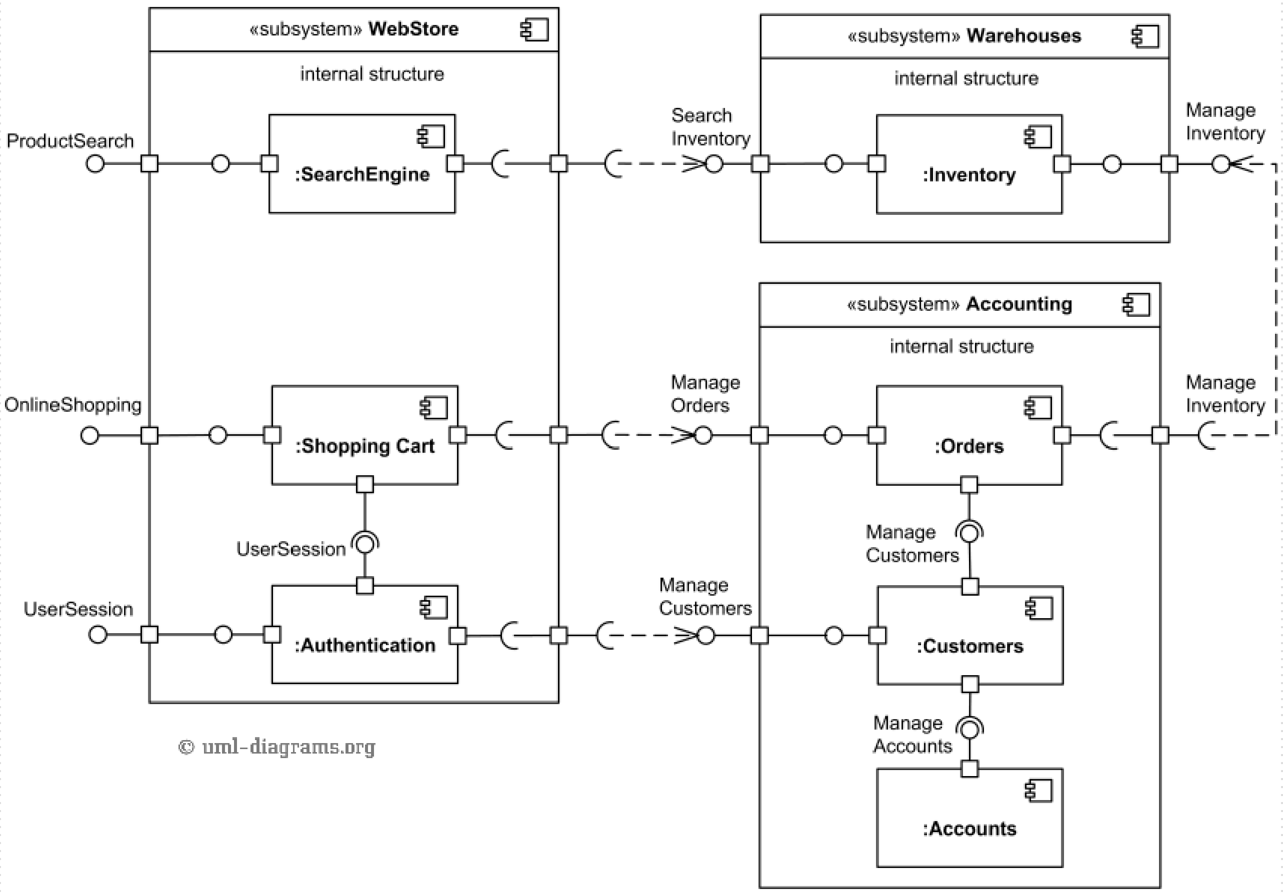
Simplifying Interactions

- Once subsystems are identified, it is possible to simplify patterns of collaboration.
  - This *reduces coupling* between components.
- Guidelines:
  - Minimize the number of collaborations a class has with other classes or subsystems.
  - Minimize the number of classes and subsystems to which a subsystem delegates.
  - Minimize the number of interfaces supported by a class or subsystem.
- It is desirable to *centralize* the *communications* into a subsystem.
  - One (possibly new) *façade class* becomes the principal communications intermediary.

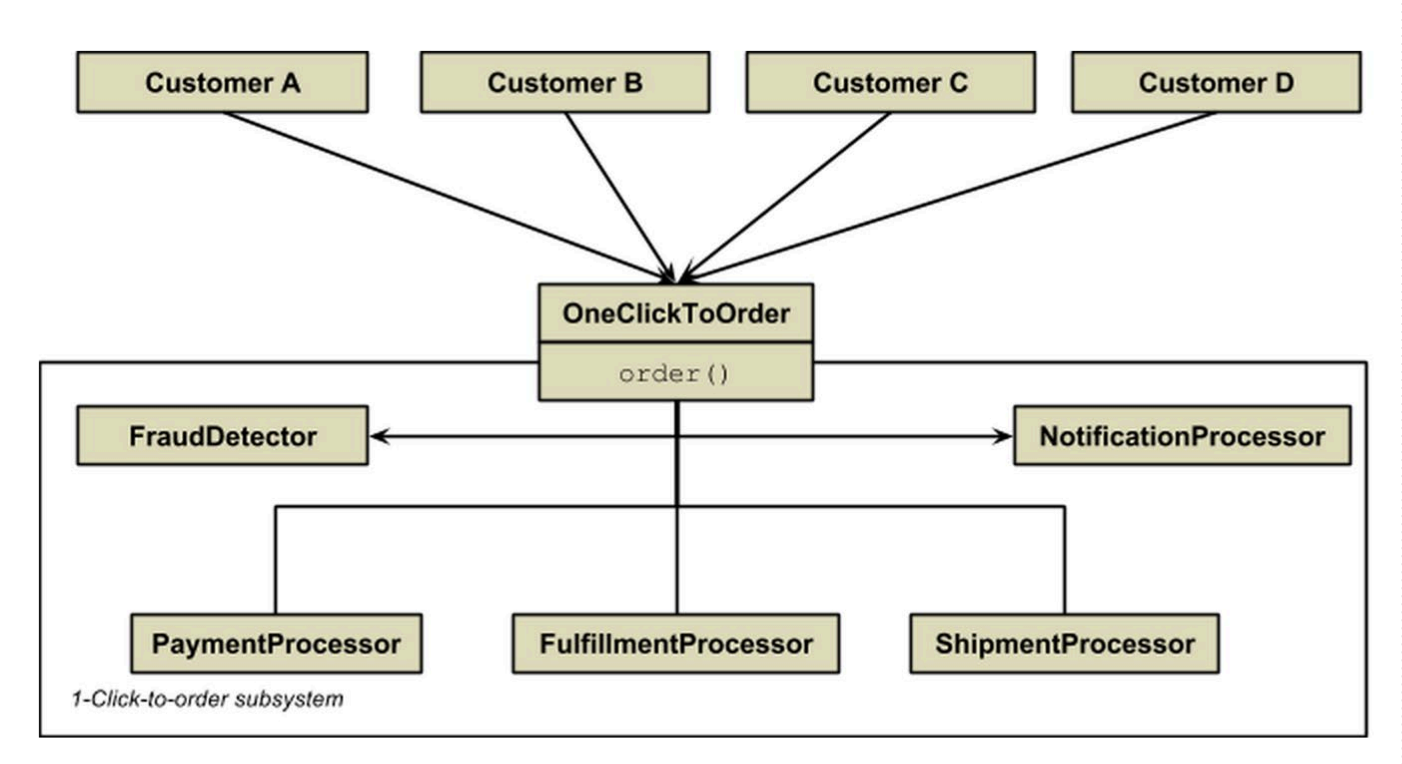
Example: Simplified ATM Design



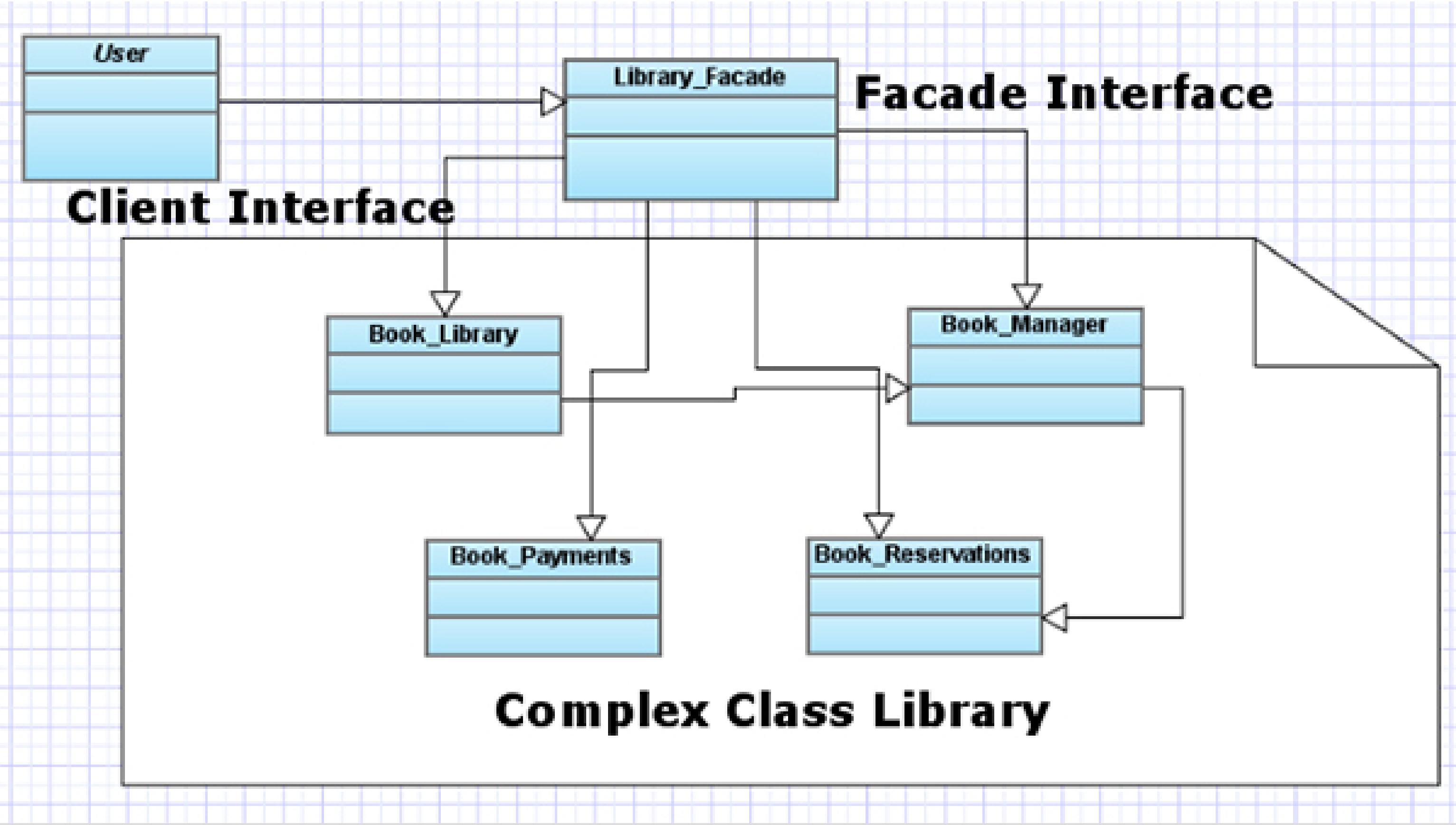
Example Subsystems



Example Façade Class



Example Façade Class (2)



www.sitepoint.com/manage-complexity-with-the-facade-pattern/

Detailed OOD

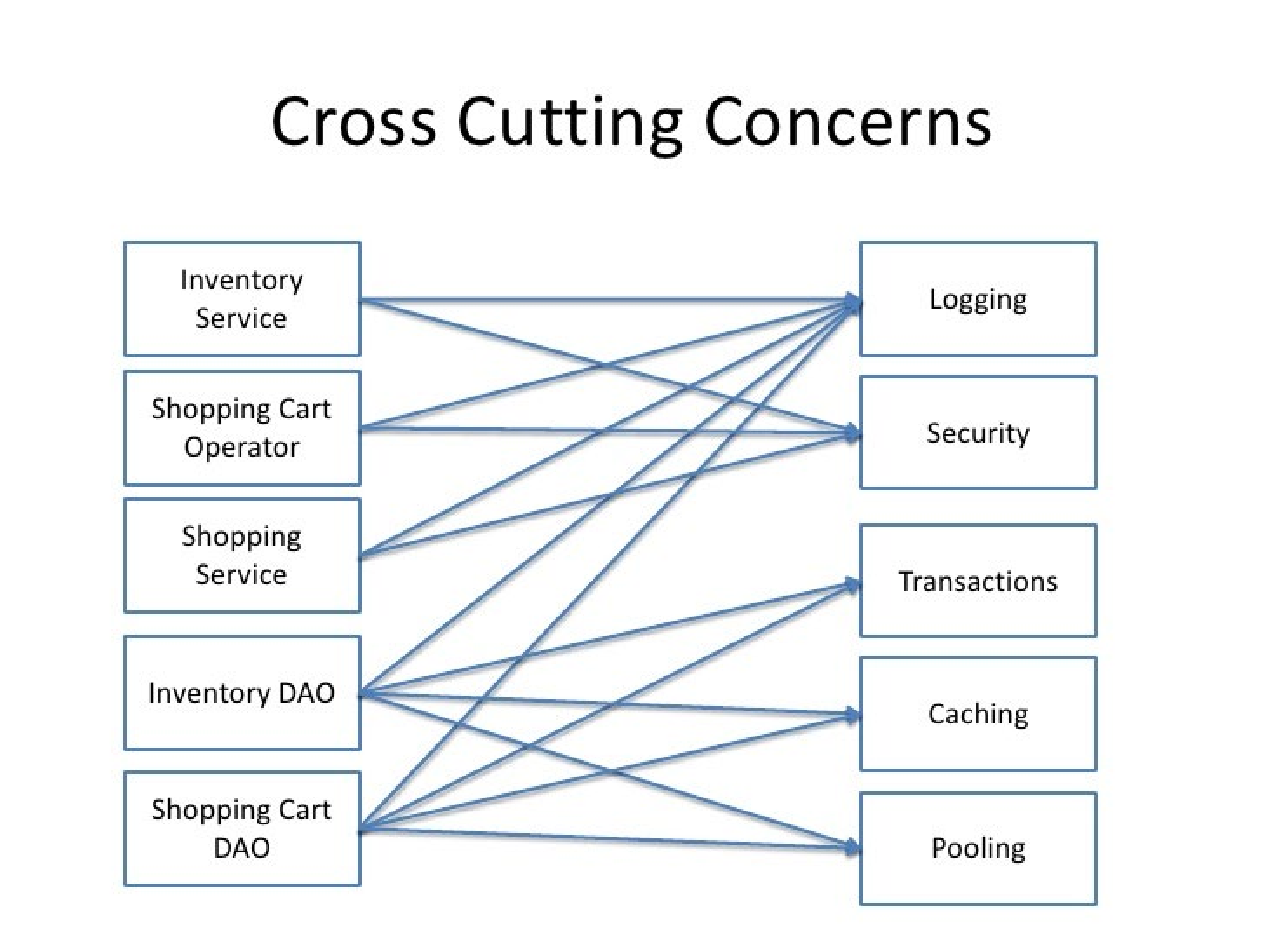
- Define *method signatures* for each class and interface.
- Write a *design specification* for each
  - Class
  - Subsystem
  - Interface
  - Method

Design Specifications

- For each class, specify:
  - Its name
  - Whether it is abstract or concrete
  - Its immediate superclasses
  - References to design diagrams that include it
  - Its purpose, in detail
  - Each interface it supports (including private ones)
  - Notes about algorithms, hardware constraints, error handling, etc.
- For each interface, specify its associated responsibilities.
  - For each responsibility, specify the signatures of the methods that will implement it.
    - For each method, specify its requirements.

Cross-Cutting Concerns: A Limitation of Object-Oriented Design

- *Separation of concerns* is the idea that each module in a design should address a different "concern"
  - Relationships to encapsulation, coupling?
- *Cross-cutting concerns* are "aspects" of a design that inherently affect *multiple modules*
- They may cause *scattering* (code duplication) and/or *tangling* (high coupling)
- Cross-cutting concerns motivate the ideas of *aspect-oriented programming* (*AOP*)



www.slideshare.net/rohitsghatol/aspect-oriented-prog-with-aspectj-spring-aop

Aspect-Oriented Programming

- Supports *separation* of cross-cutting concerns
- Permits addition of behavior, called *advice*, to existing code, without actually changing it
- Places where advice is to be applied, called *joint points*, are indicated by *pointcut* specification
  - E.g., in Spring:

```
@Pointcut("execution(public String org.baeldung.dao.FooDao.findById(Long))")
```

http://www.baeldung.com/spring-aop-pointcut-tutorial

Example: “Before” Advice

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* 
com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

https://docs.spring.io/spring/docs/2.0.x/reference/aop.html

Sources

- *Design Patterns* by Gamma, Helm, Johnson, and Vlissides
- *The Object Primer* by Scott Ambler
- *Designing Object-Oriented Software* by R. Wirfs-Brock, *et al*.