Overview of Object-Oriented Design

Object Oriented Programming

- Computer programs solve problems of the real world
- In the real world, entities (objects) interact with each other to perform a task
  - Entities have features
  - Entities ask other entities to do something
- Why not have programs that perform their tasks using entities a.k.a. **Objects**
  - Objects have features, i.e. attributes
  - Objects ask other objects to perform operations
- The premise of OOP
  - Easier to match real world problems to software world
  - Ability to create objects to mimic real life
  - Achieve modularity with Object abstraction

Objects are data abstractions

- Internal representation
  - Attributes/features/fields
  - What properties does it have
- Operations
  - What are ways to interact with them
  - Defines behaviors but hides implementation

Here are our 3 dog objects =>

Duffy	Fluffy	Aslan
-lying	-sitting	-lying
-bulldog	-poodle	-terrier
-blue	-red	-none
+bark()	+bark()	+bark()
+run()	+run()	+run()

Objects and classes

- An **object** is an entity in a software system which represent instances of real-world and system entities
  - has a **state** and a defined set of operations which operate on that state
  - The **state** is represented as a set of object attributes.
  - The **operations** associated with the object provide services to other objects (clients) which request these services when some computation is required.
- Objects are created according to some **object class** definition.
  - An object class definition serves as a template for objects.
  - It includes declarations of all the attributes and services which should be associated with an object of that class.

Class and Objects: implement & use

1. How to define a dog blueprint in a program?
   - Define your own abstract data type => CLASS
   - **Class** definition is a **type** definition
2. How to create dog objects ?
   - Once a class is defined, a user can define **variables** of that type
   - Use constructor method
3. How to use objects in a program?
   - Ask the object to perform an operation
   - Manipulate/**interact** with an **object** via its public methods

```
public class Dog{
   private String breed;
   private String posture;
   ……
   public void bark(){……}
   public void sit(){…}
   public Dog() {…}
}
```

```
Dog fluffy =new Dog();

Dog aslan =new Dog();
```

```
fluffy.sit();

aslan.bark();
```

Interacting with Objects

- An object performs an operation when it receives a request (or **message**) from a **client**.
- Requests are the *only* way to get an object to execute an operation.
- Operations are the *only* way to change an object's internal data.
- Hence, the object's internal state is encapsulated;
  - it cannot be accessed directly
  - its representation is invisible from outside the object.

Object-oriented development

- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to fulfil the requirements.
- OOP is concerned with realizing an OOD using an OO programming language such as Java, C++, C#.

Analysis vs Design

- Analysis
  - What needs to be done?
  - Not how they are need to be!
- Design
  - How the problem could be solved?
- Programming
  - Bring the design into concrete existence
  - Realization of the design

Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- Hiding information inside objects means that changes made to an object do not affect other objects in an <u>unpredictable</u> way
- For some systems, there may be an obvious mapping from real world entities to system objects

Information Hiding

- Keep internal representation **private**
  - Correct behavior may be compromised if someone manipulate them directly
- Information hiding
  - don't need to know what the internal representation is to use an object in a program
  - Analogy: don't need to know how a function is implemented to be able to call that function

Violating invariants
Missing actions when attribute change

```
public class Dog{
   public  int age; //!!!!!
   private String breed;
   private String collar;
   …
   public void setAge(int age){
    //change maturity
    //using age and breed
   }
   private void becomeAdult(){
     … //update diet, exercise
   }
   private void becomeSenior(){
     …//update exercise, vet freq.
   }
}
```

Information Hiding

- Keep internal representation **private**
  - Correct behavior may be compromised if someone manipulate them directly
- Information hiding
  - don't need to know what the internal representation is to use an object in a program
  - Analogy: don't need to know how a function is implemented to be able to call that function

```
//user program fails after the change
Dog duffy=new Dog();
duffy.posture="running";
```

```
public class Dog{
    public  String posture;
    private String breed;
    private String collar;
  …}
}
//CHANGED TO…
public class Dog{
    public  String pose;
    private String breed;
    private String collar;
  …}
}
```

Hide information & use methods

- Keep internal representation **private**
- Use set and get methods to read and write to attributes.

- **Setter** method check validity and consistency of the intented attribute values
- **Getter** methods hide internal representation
  - Attribute name
  - Attribute type and structure

```
public class Dog{
 private String posture;
 private  String breed;
 private String collar;
   /*… other methods */
 public String getBreed(){…}
 public String getPosture(){…}
 public String getCollar(){…}

 public void setCollar(String clr){
  if(breed.equals(“tiny”)
      System.out.println(“no collar
      available for a tiny breed”);
  else collar=clr;}

}
//checking dependencies
```

Topics to be reviewed

- Object identification
- Generalizations/inheritance
  - Liskov Substitution Principle
- Composition vs Inheritance
- Open-closed principle
- Modularity
  - Cohesion
  - Coupling

Object Identification

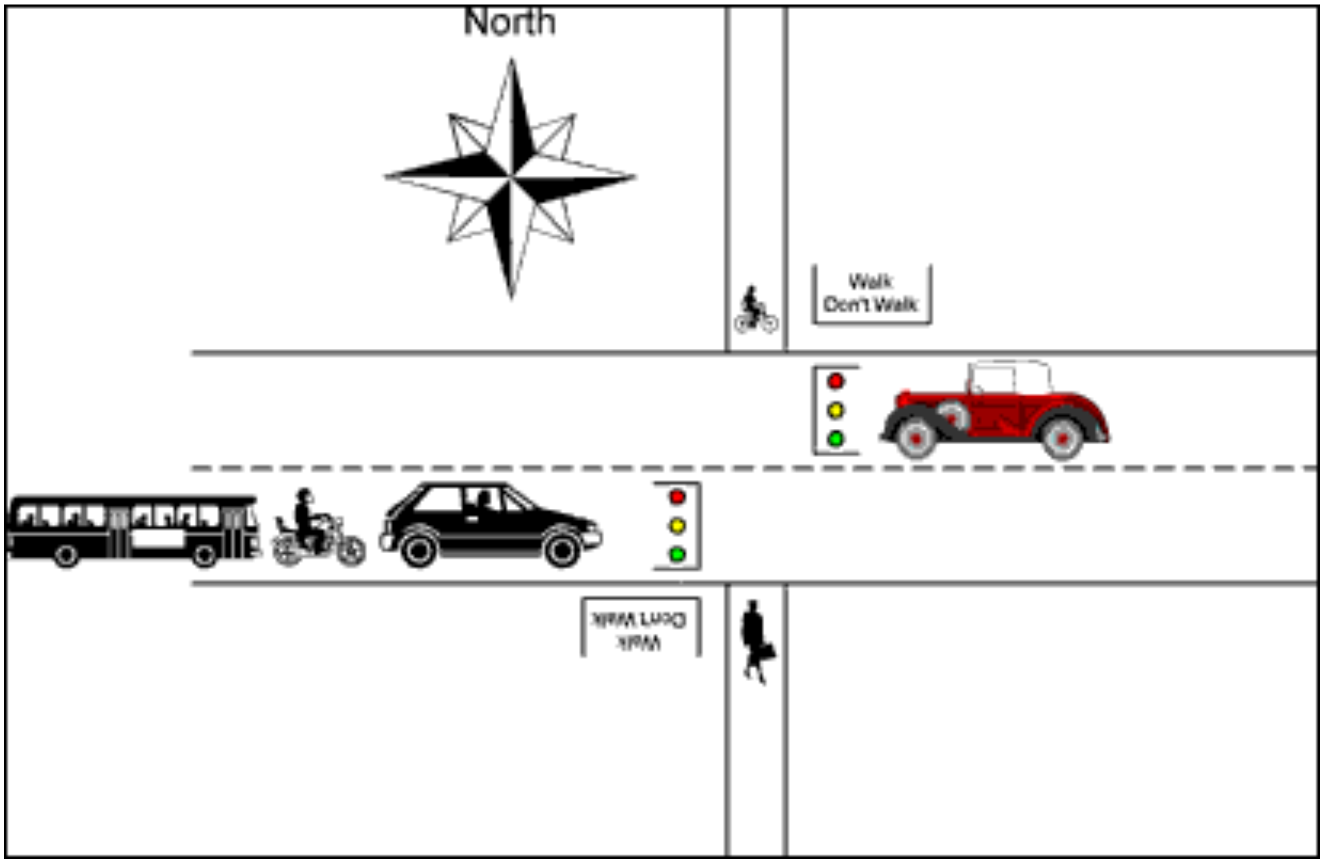
- Identifying objects is the most difficult part of object oriented design.
- No 'magic formula' for object identification.
  - It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

- Use a grammatical approach on the description of the system
  - nouns are candidate objects
- Base the identification on tangible things in the application domain.
- Identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.
- There will always be less obvious ones
  - Abstractions that does not match the real world

Object Identification

- Example: Simulation of traffic flow at an Intersection



Simulation of Traffic flow (Cont’d)

- Cars move in one of two directions.
- Traffic flows through ‘green’ lights.
- Pedestrians cross only at the crosswalk, on a ‘walking man’ sign.
- Traffic is stopped by a ‘red’ light, pedestrians by a ‘red standing man’ sign.
- This is a simulation of a traffic flow without a user interface
- What are the object classes?

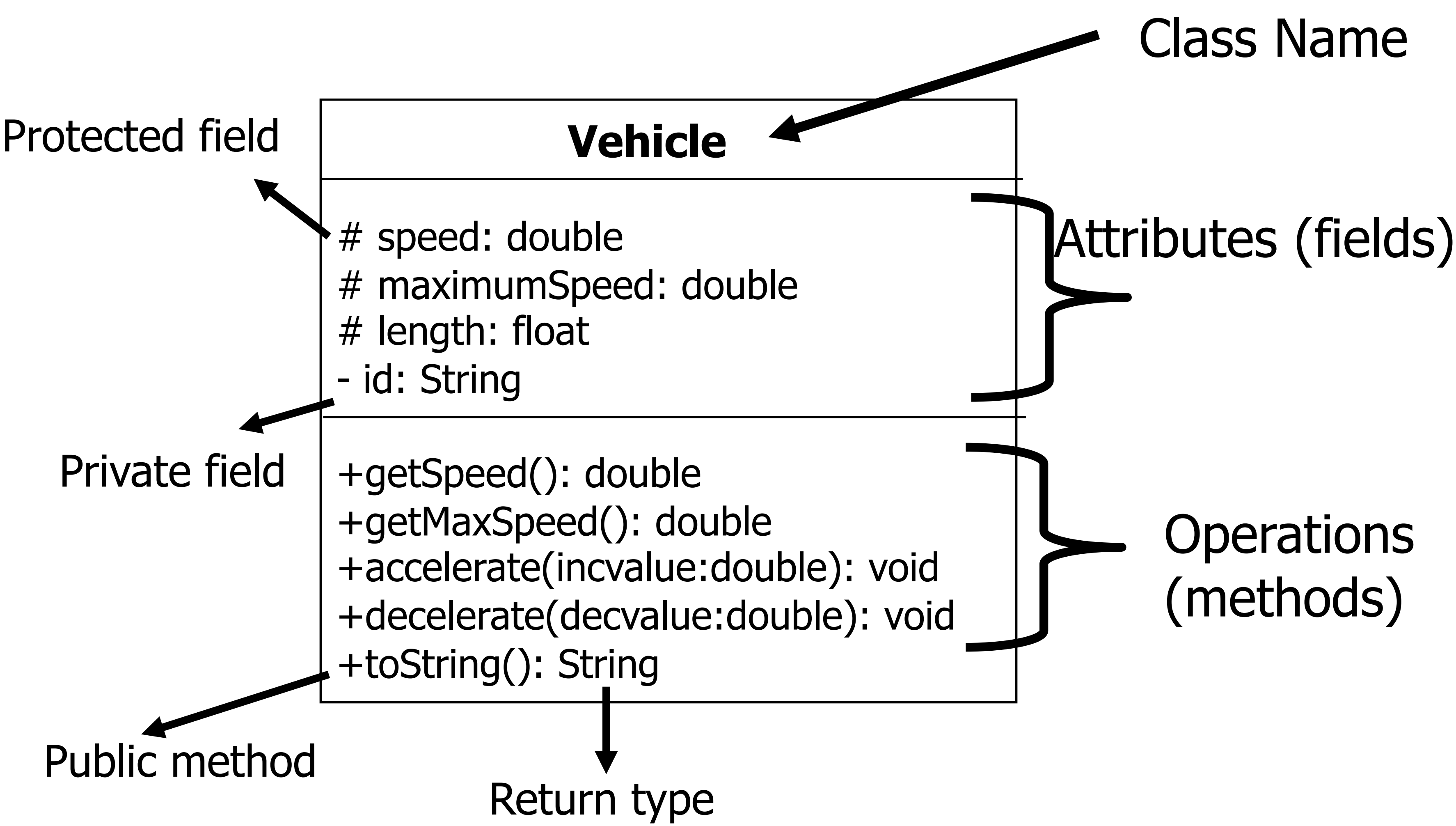
Simulation of a traffic flow (Cont'd)

- Object Classes
  - Vehicle
  - Pedestrian
    - essentially the same thing, an object that crosses the road.
    - Bicycle, truck, taxi…
  - Traffic Light (red, green, yellow)
  - Traffic Sign for Pedestrians (walking man and stopping man)
    - same class as Traffic Light but parameterized differently
    - concrete subclasses of an abstract Signal superclass
  - Road/Intersection

Simulation of a traffic flow (Cont’d)

- Not so obvious classes:
  - Timer /Clock
  - Mediator/Control
    - manages the different lights.
    - making sure that a ‘walking man’ signal is not put out when the traffic light is still ‘green’
  - Injector
    - feeds traffic and pedestrians into the intersection in a pseudo-random fashion

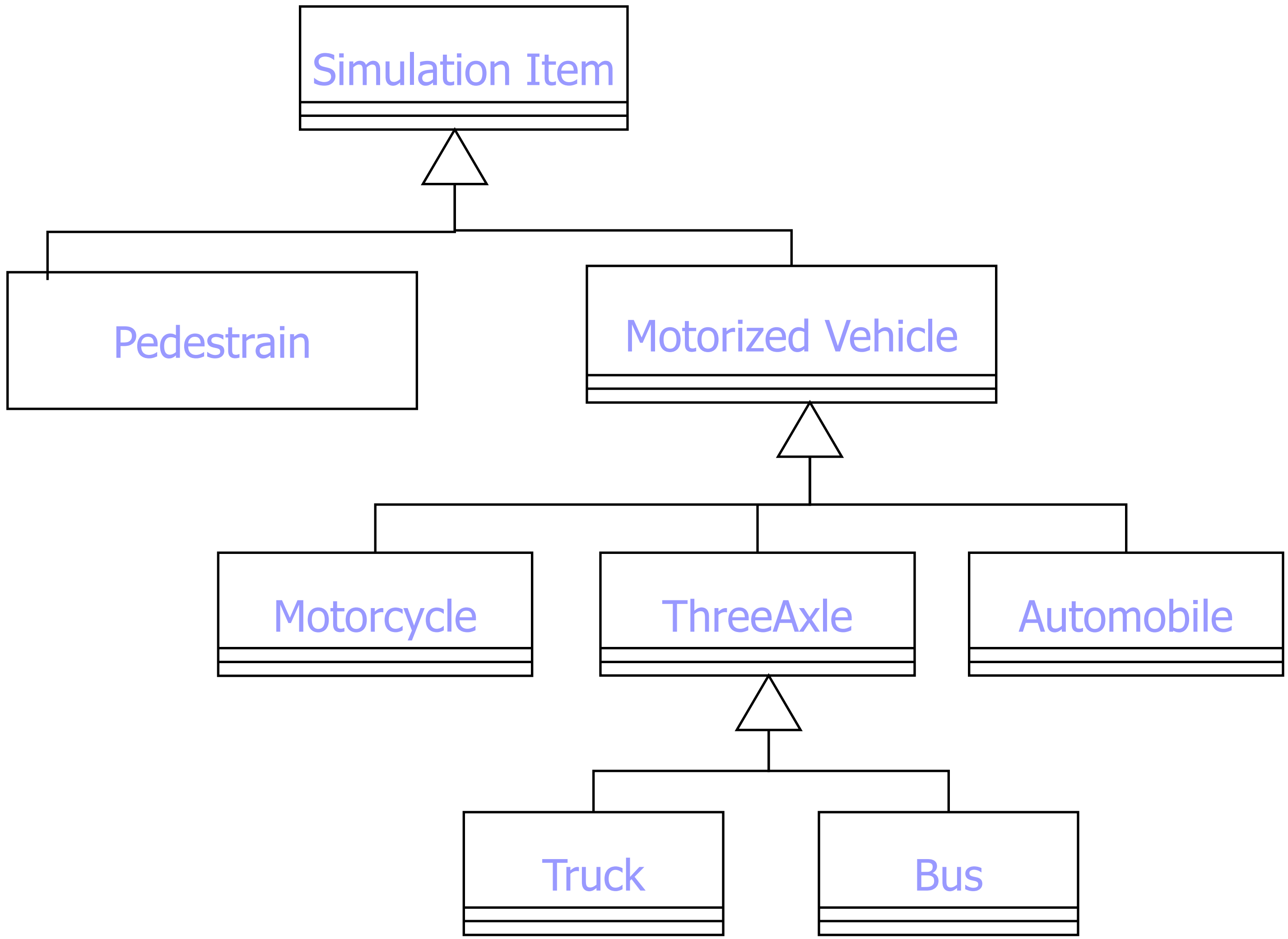
Quick Reference: Class Representation



Generalization and inheritance

- Classes may be arranged in a class hierarchy where one class (a superclass) is a generalization of one or more other classes (subclasses)
- A subclass inherits the attributes and operations from its super class and may add new methods or attributes of its own.
- Generalization is implemented as inheritance in OO programming languages.

A Generalization Hierarchy



Advantages of inheritance

- It is an abstraction mechanism which may be used to classify entities.
- It is a reuse mechanism at both the design and the programming level.
- The inheritance graph is a source of organizational knowledge about domains and systems.

Problems with inheritance

- Object classes are not self-contained. They cannot be understood without reference to their superclasses.
- Creates interdependencies among classes that complicate maintenance
  - can I modify the private attributes without affecting the subclasses?

Alternatives

- Composition (object composition)
- Delegation
  - Extreme composition
- Inheritance vs parameterized types
  - Templates in C++ and generics in Java2
- **Inheritance is still necessary**
  - You cannot always get all the necessary functionality by assembling existing components

Composition

- New functionality obtained by composing objects
- Runtime dependency via acquiring object references
- A black-box reuse
- No overgrown class hierarchy
- Disadvantage: More objects and their interrelationships
- **Use when it makes the design simple**

Reuse mechanisms

Inheritance

- Whitebox: Subclass reuses details of its base and extends with new functionality
- Defined at compile time
- Straightforward to use
- Breaks encapsulation- superclass details are exposed to subclass
- Reuse can be difficult in some context – may require rewrite of base or carrying extra baggage

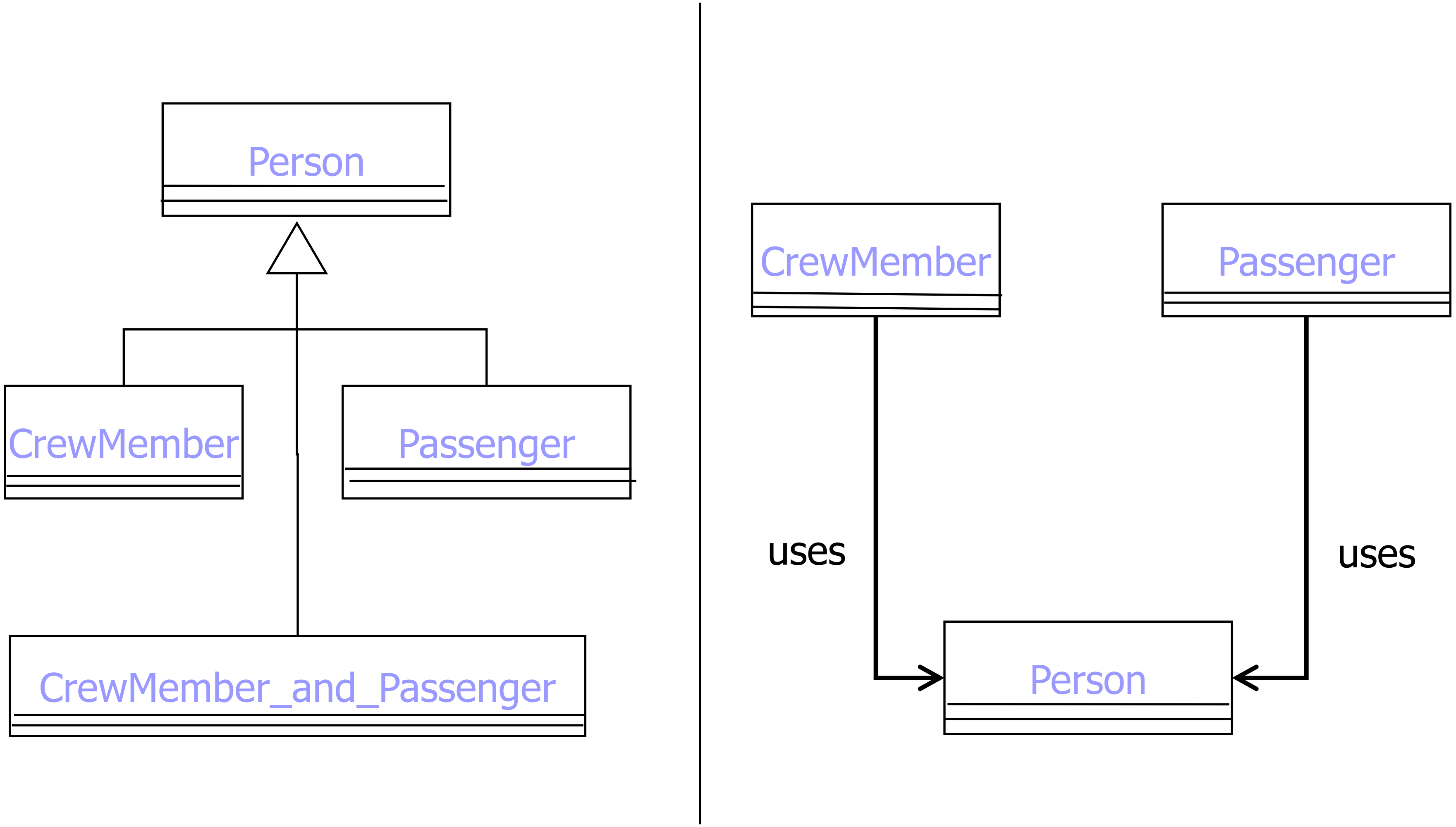
Composition

- Blackbox: new functionality obtained by bringing objects together
- Defined at runtime by getting reference of other objects
- must program to interfaces

Strong composition:Delegation

- Two objects are involved in handling a request: a request receiving object delegates operations to its delegatee
- When inheritance is not appropriate use delegation
- Inheritance: is-a-kind-of relation
- Delegation: is-a-role-played-by relation
- Implementation
  - Instead of extending a base class, create a *delegator* class have a reference to the base class
  - *Delegator* uses the base class to fulfill a particular role

Inheritance vs Delegation



- is-a and has-a relations
  - Manager is a Person
  - Manager is an Employee

  - What would be an Inheritance solution?
  - A Composition solution?

Delegation

- Advantages
  - Easy to compose behaviors at runtime
  - Easy to change the way the objects are composed
    - Dependency Injection
- Disadvantages
  - Runtime inefficiency
  - Useful only when it simplifies

Reusable OOD Practices -1

1) Favor object composition over inheritance

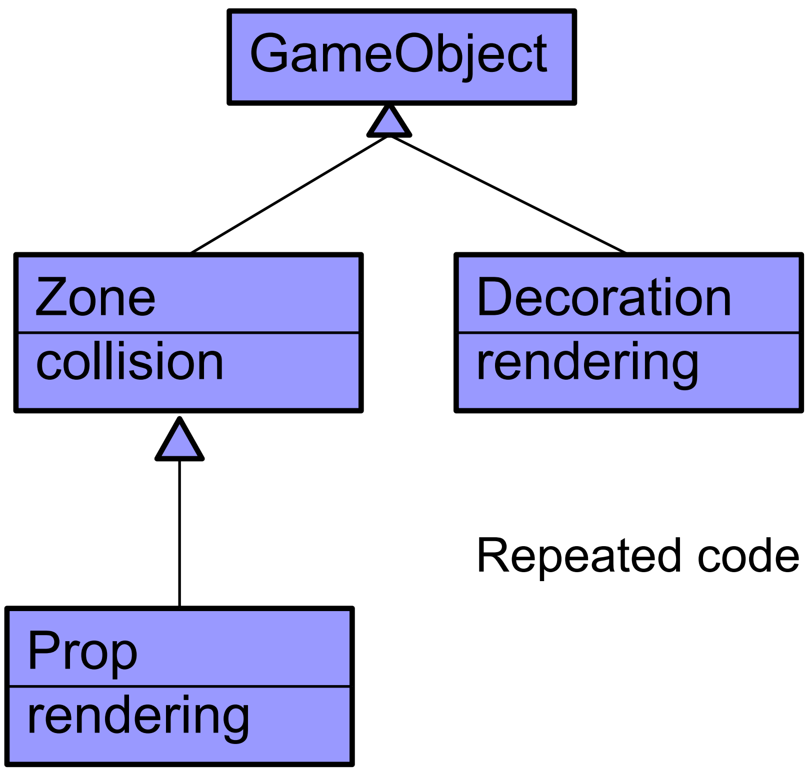
- More flexibility
- Robust to changes
- dependency injection helps

Example: Game scene

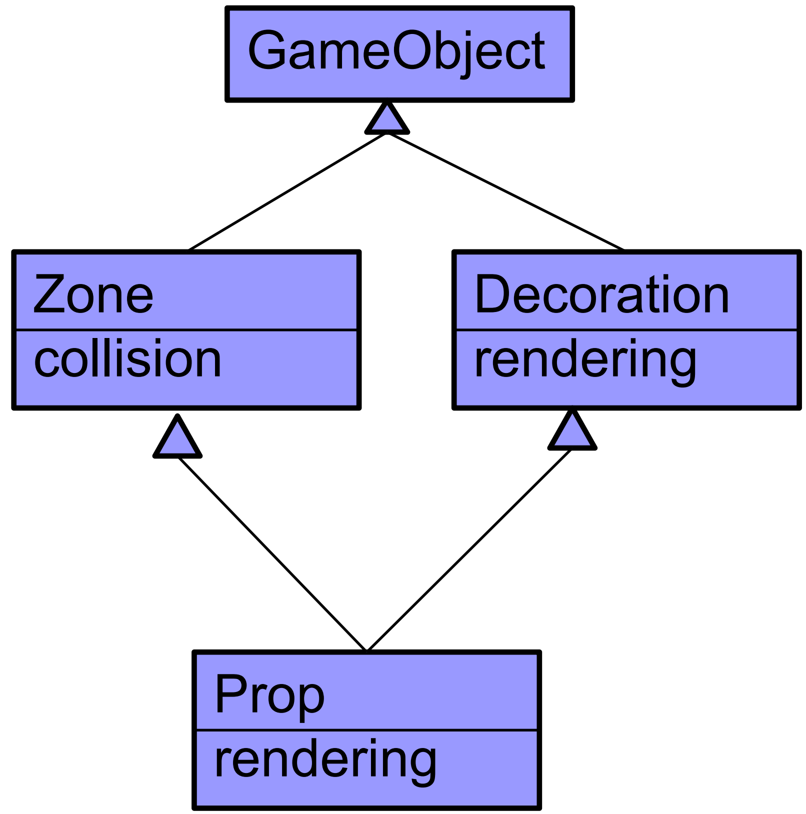
- A scene consists of Players and..
- Decorations are things in the world the player sees but doesn't interact with.
  - bushes, debris and other visual detail.
- Props are like decorations but can be touched.
  - boxes, boulders, and trees.
- Zones are invisible but interactive.
  - the opposite of decorations
- We have GameObject class that has common features like position, orientation

https://gameprogrammingpatterns.com/component.html

GameObject hierarchy



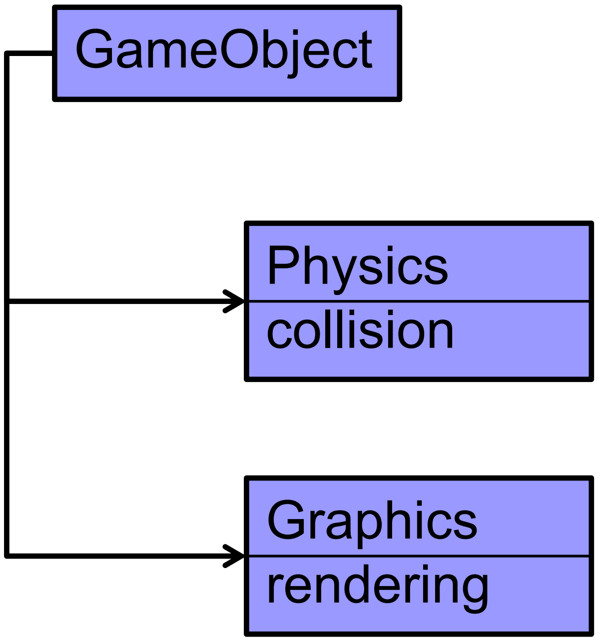
Repeated code



NEVER make a diamond!

GameObjects

- Lets break things up and use composition




```
GameObject Zone = new
    GameObject(
            new Physics());

GameObject Prop= new
    GameObject(
        new Physics(),
        new Rendering);
```

Principle2 -Motivating Problem

- Car class and its 2 operations/behaviors:
  - Brake and accelerate

- Simple car decelerates with a constant
- Sports car uses ABS
- EV uses regenerative breake system

Attempt -1

```
class  Car{
  public void break(){
      switch(model){
      case Simple:
        //reduce speed 5 unit/sec
      case Sportcar:
       //….with ABS
      case EV:
        //regenerative breaking
   }
//other members
```

```
public void accelerate(){
      switch(model){
      case Simple:
                //speed up  7u/s
      case EV:
         //smart speed up
      case Sports: //….
}
```

- as ugly as it gets,
- useless when it comes to extensibility
- throws out any hope of it being reusable.

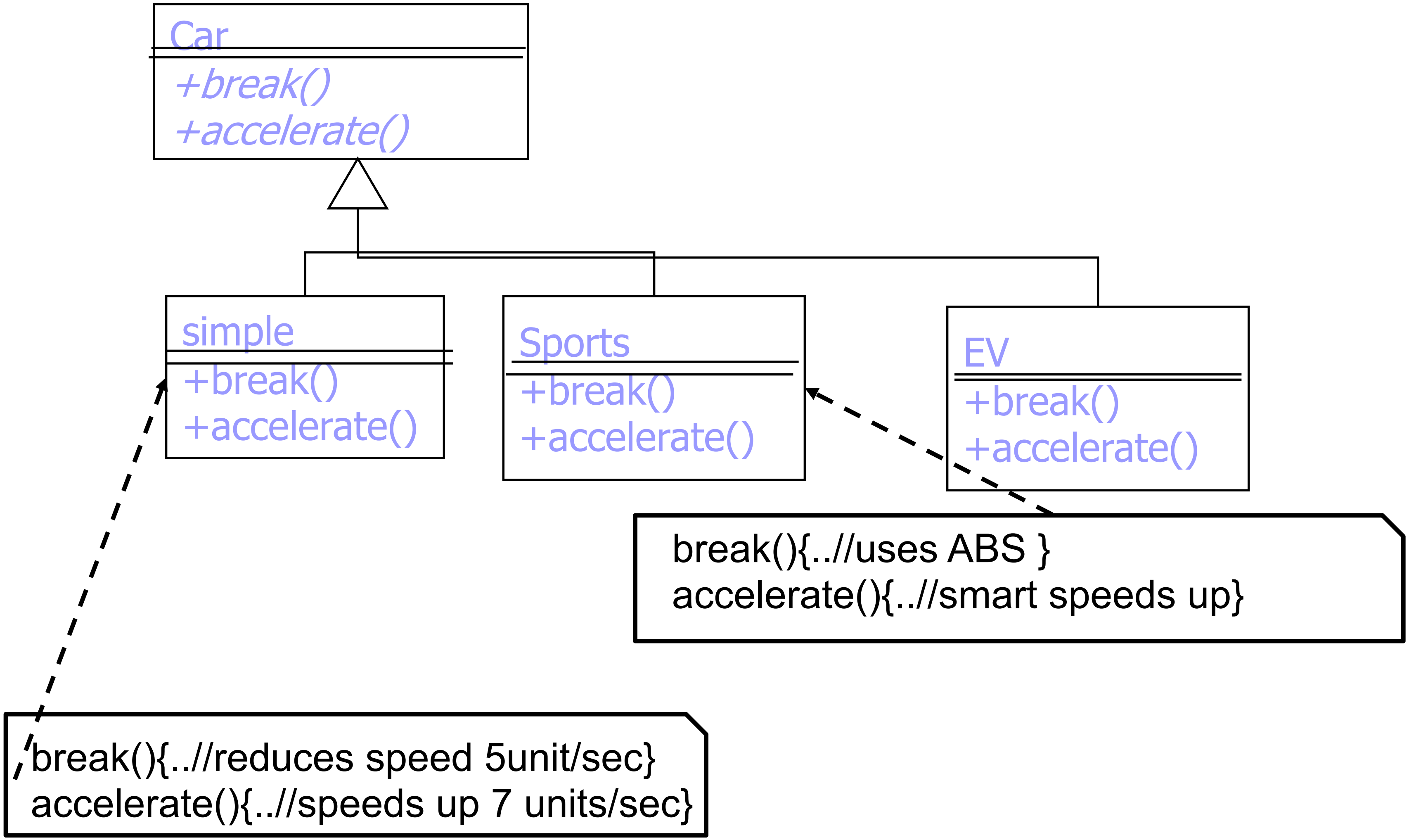
Why is it undesirable?

- tightly coupling functionality that varies to the object
- difficult to manage over time
  - every time you think of another case, you get to let this beast of a switch statement grow and grow and grow.

Principle2 -Motivating Problem

- Car class and its 2 operations/behaviors:
  - Brake and accelerate
- Attempt - 2:
  - These behaviors change frequently between models, so implement these behaviors in subclasses: overriding
  - For each new model, override!

Attempt- 2:


Car
+break()
+accelerate()
simple
+break()
+accelerate()
Sports
+break()
+accelerate()
EV
+break()
+accelerate()
break(){..//uses ABS }
accelerate(){..//smart speeds up}
break(){..//reduces speed 5unit/sec}
accelerate(){..//speeds up 7 units/sec}

Principle2 -Motivating Problem

- Car class and its 2 operations/behaviors:
  - Brake and accelerate
- Attempt- 2:
  - These behaviors change frequently between models, so implement these behaviors in subclasses: overriding
  - For each new model, override
    - Beware: Code duplication across models
    - The work of managing these behaviors increases greatly as the number of models increases

Alternative?

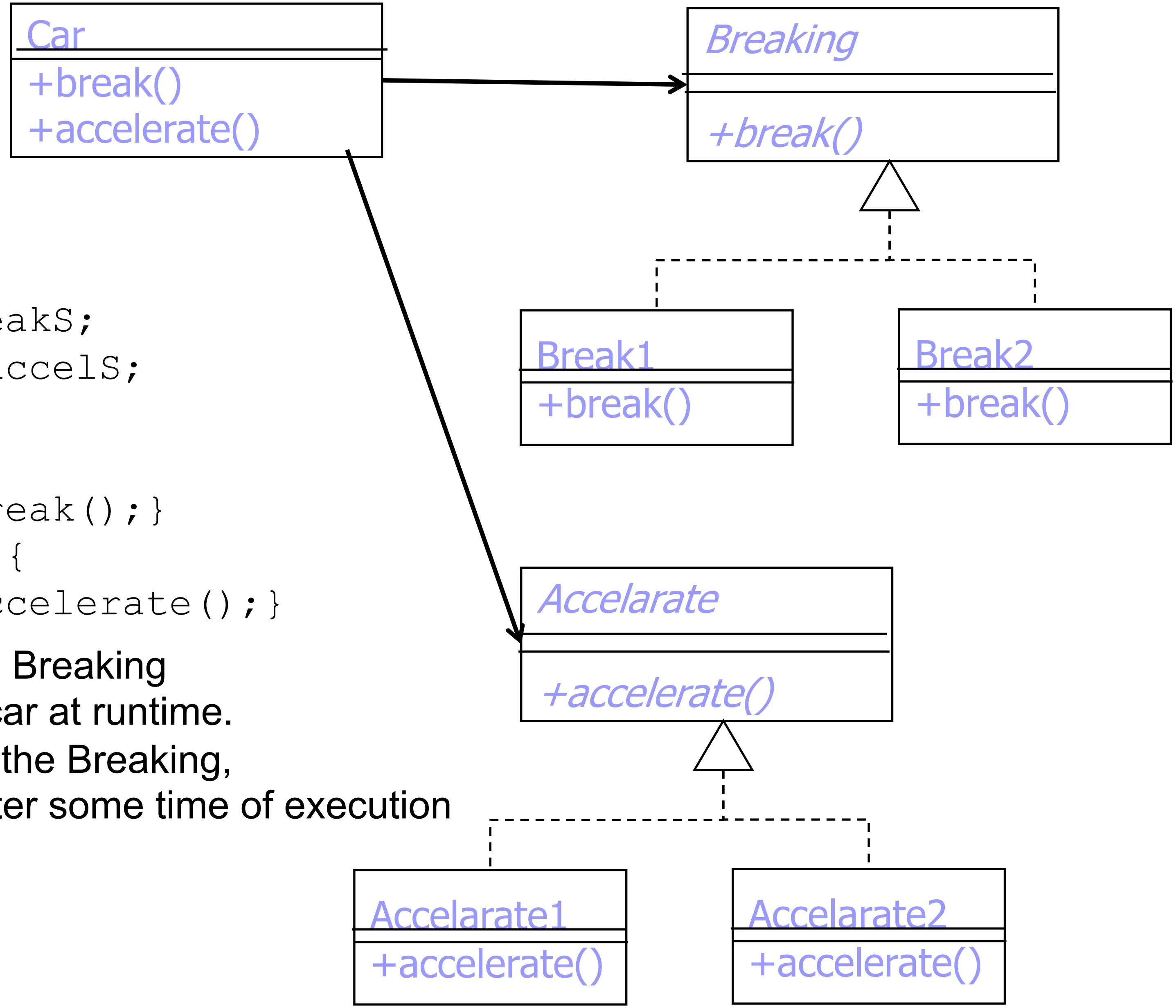
- Design principle:

  Encapsulate what varies
- What is varying?

Alternative?

- Design principle:

  Encapsulate what varies
- What is varying? A function realization
  - Put it in a class
  - Associate the appropriate brake & accelerate for each model
- Choose a suitable one for each car object
  - Delegation instead of inheritance




```
class Car{
Breaking breakS;
Accelerate accelS;
…
break(){
    breakS.break();}
accelerate(){
    accelS.accelerate();}
```

You can attach a Breaking after creating a car at runtime.
You can change the Breaking, e.g. use ABS, after some time of execution

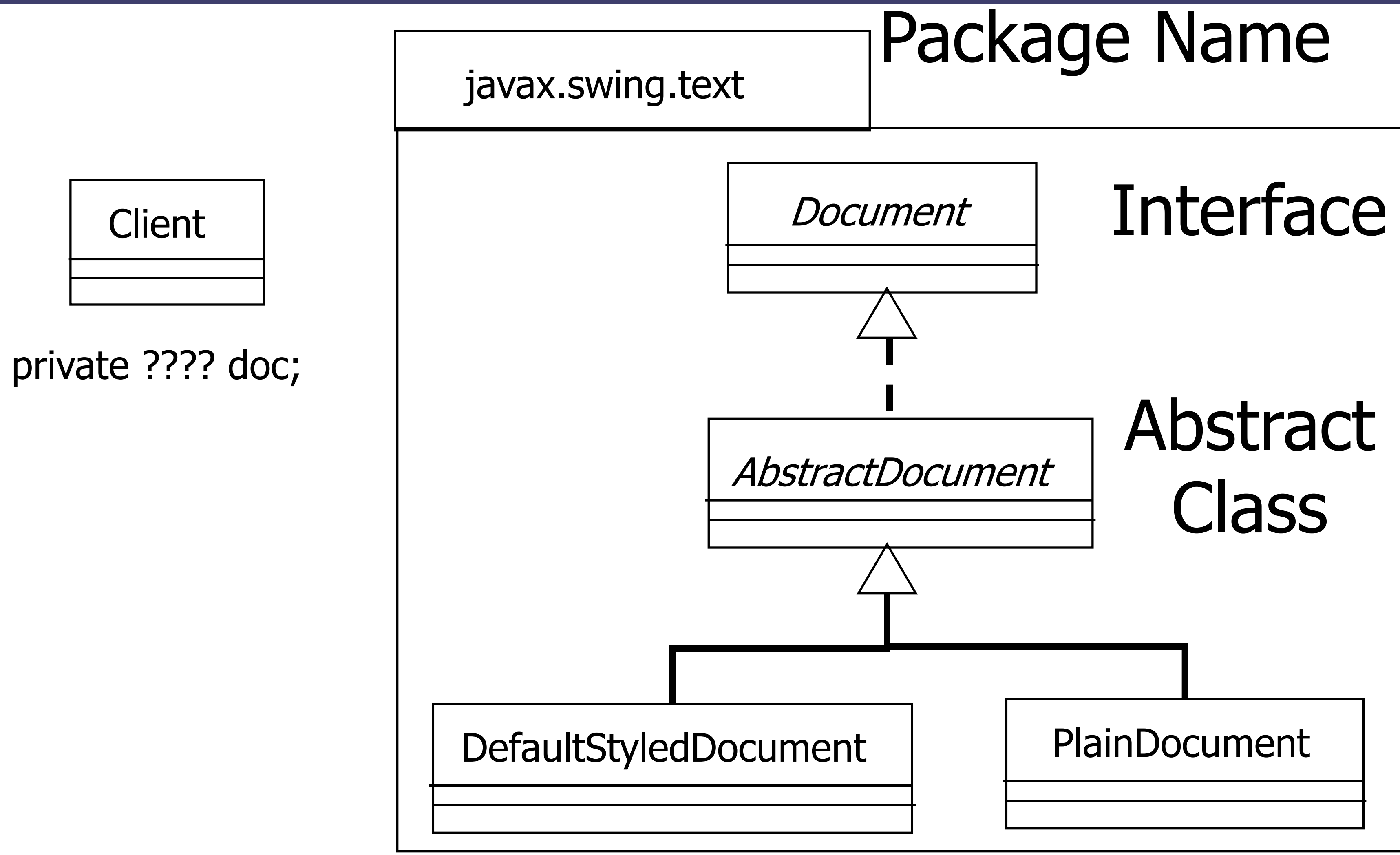
Reusable OOD Practices - 2

1) Favor composition over inheritance
2) **Encapsulate what varies**
   - Variation in its own class
   - Use composition and dependency injection to build the structure
     - E.g. break and acceleration are pluggable to Car

Interface and Abstract Classes

- Interface
  - When you need to hide from the clients the class of an object that provides a service
- Abstract Class
  - When you need to design a set of related classes that provide similar functionality
- Use both when you need both
  - public interface and a package private abstract class

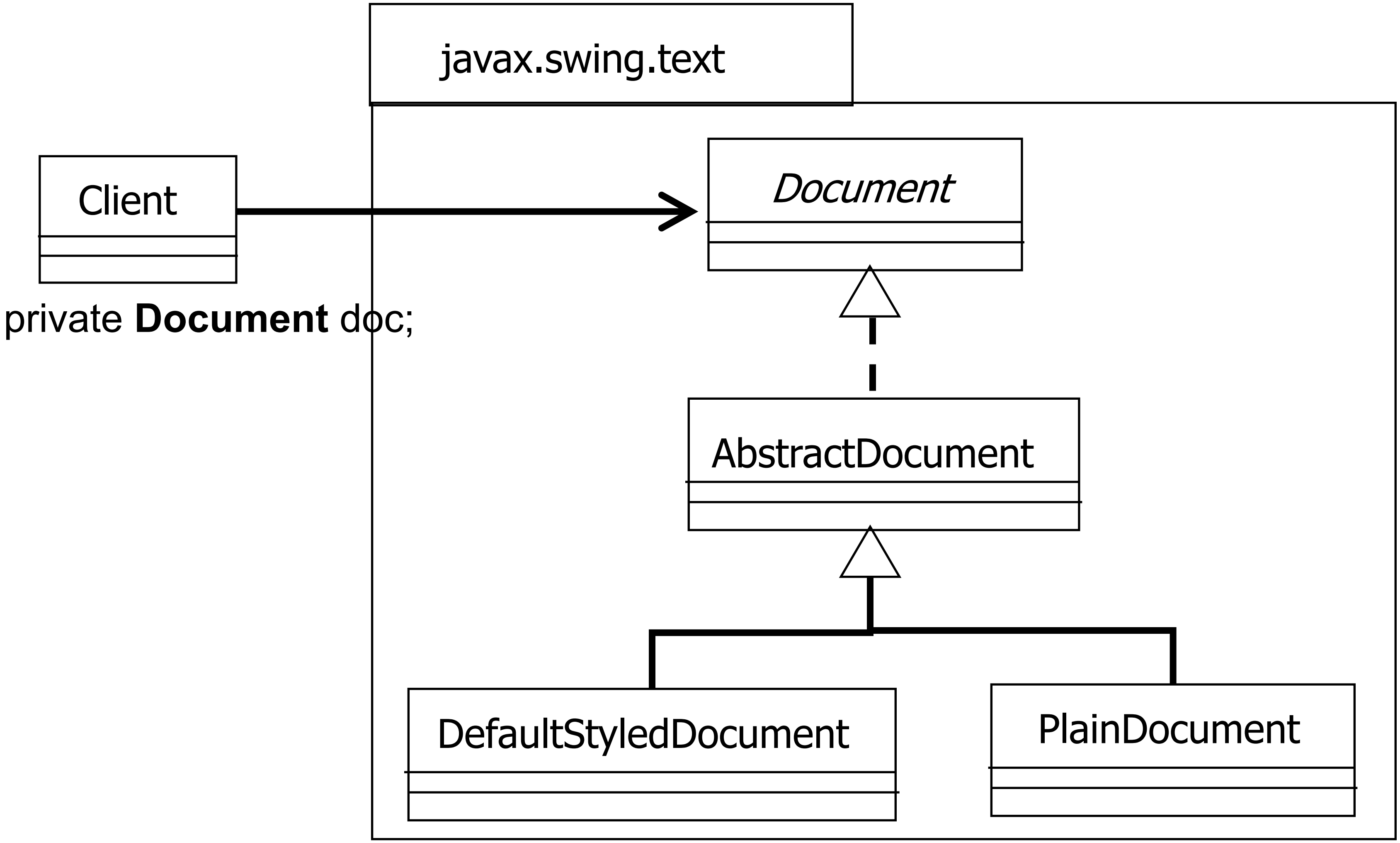
Example


Package Name
javax.swing.text
Client
private ???? doc;
Document
Interface
AbstractDocument
Abstract Class
DefaultStyledDocument
PlainDocument

Interface – Virtual class

- Very important in reusable design
- In compositions, objects will use the interfaces and interface methods
  - E.g. Document interface , insert(), remove()
    - No matter if the document is text only or styled
- What operations can I ask a Document to do is defined in the Interface.

Example


javax.swing.text
Client
Document
private **Document** doc;
AbstractDocument
DefaultStyledDocument
PlainDocument

Reusable OOD Practices - 3

1) Favor composition over inheritance
2) Encapsulate what varies
3) **Program to an interface not implementation**
   - favor List over ArrayList in a client code
   - favor Map over HashMap in a client code
   - Fewer implementation dependency

```
class Editor{
  private Document doc;  …}
```

   - Helps dependency injection

Reusable OOD practices

1) **Favor composition over inheritance**
2) **Encapsulate what varies**
3) **Implement to interface**

Helps with

- Open closed principle
- Dependency injection
- Single responsibility
- True subtyping

SOLID recap

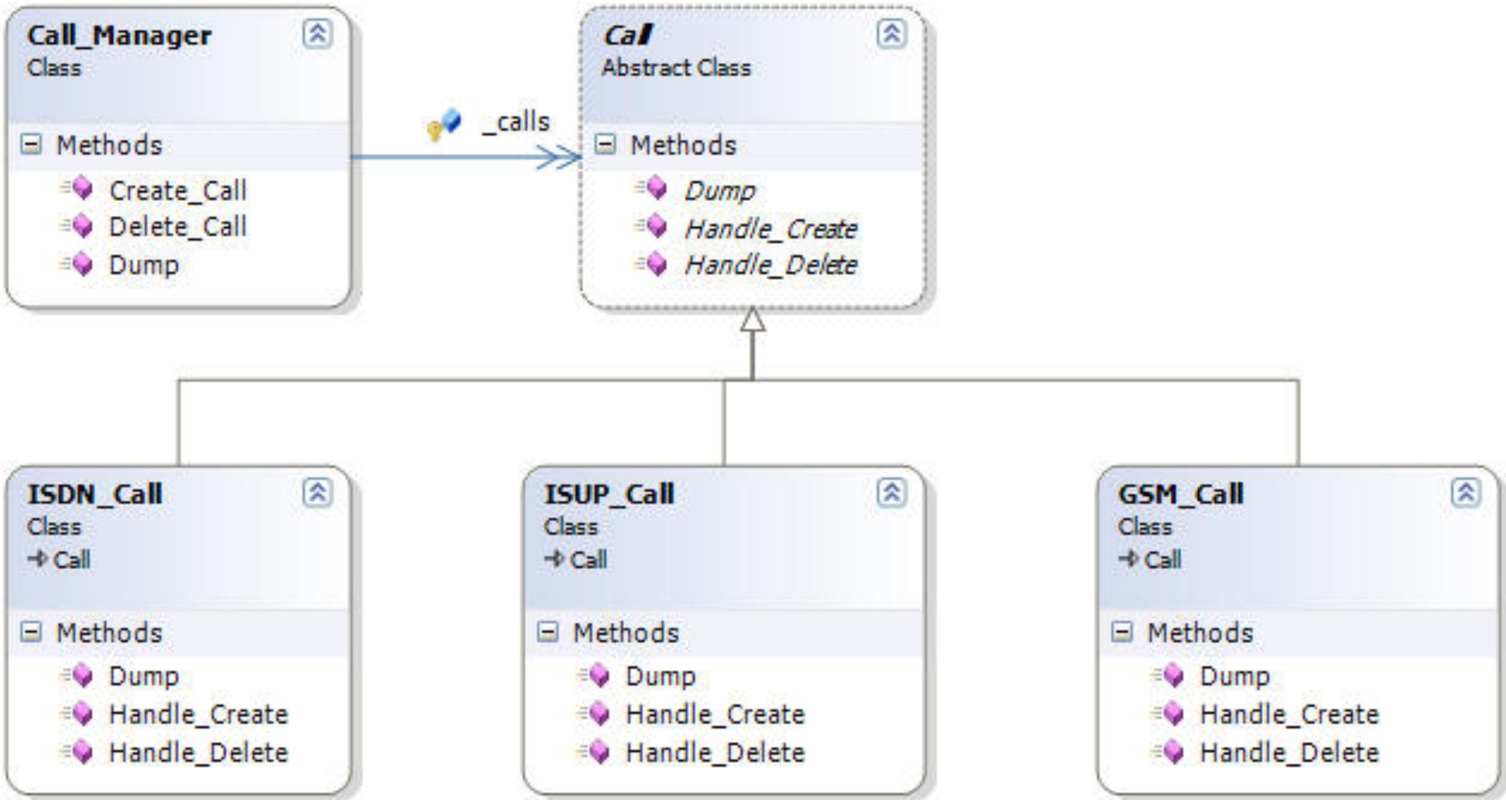
- Single responsibility
- Open closed principle
- Liskov's substitution principle
  - True subtyping
- Interface segregation
- Dependency injection

Open-Close Principle (OCP)

- Modules should be both:
  - ***open: for extension***
  - ***closed: the module is closed to modification in ways that affect clients***
    - closed wrt X = clients are not affected if X changes

- Recall Car and Strategies
  - Can extend with new breaking techniques
  - No modification to Car class

Example1:



- The Call_Manager design is closed for modification.
- Addition of a new call type requires writing a new class that inherits from Call. No changes are needed in the Call_Manager.

Example2:

```
void DrawAllShapes(
        ShapePointer list[], int n){
 int i;
 for (i=0; i<n; i++){
   struct Shape* s = list[i];
   switch (s->itsType){
    case square:
          DrawSquare((struct Square*)s);
           break;
    case circle:
           DrawCircle((struct Circle*)s);
           break;
}}}
```

- The function DrawAllShapes does not conform to the open-closed principle because it cannot be closed against new kinds of shapes.

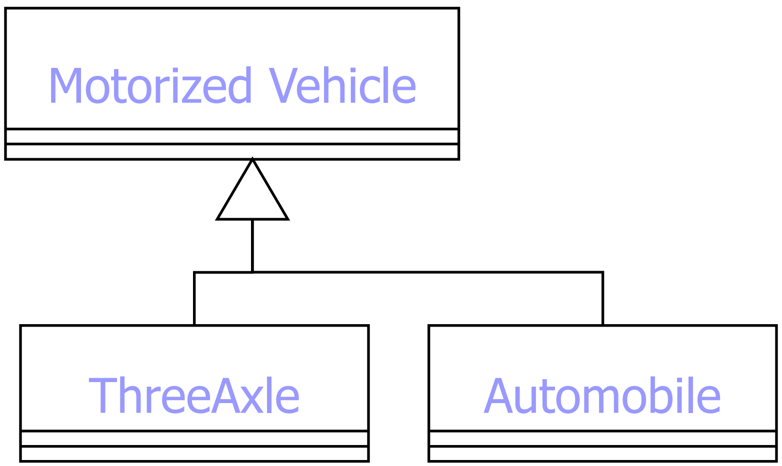
Design Principles help OCP

- Implement to interface –big help
  - Open for modification by extending with subclasses
  - Client code is closed for modification since interface stays the same even after the new extension
- Favor composition over inheritance
  - Composition enables plug-ins
- Encapsulate what varies
  - Change is isolated
    - See the car example

Principles & Dependency injection

1) Favor composition over inheritance
   - Inheritance creates hard bindings
   - Composition enables different configurations with plug-ins
2) Encapsulate what varies
3) Implement to interface –big help
   - Client code uses interface type
   - Client code is configured with concrete subclass – injection

True subtyping (LSP)



- Inheritance relation defines subtypes.
- Class = Type

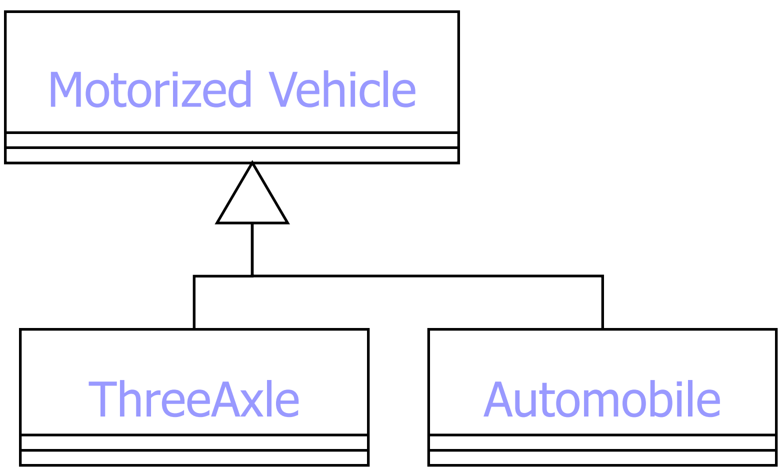
Subclass = Subtype

Be careful when you use generalization.

True subtyping

- **The Liskov Substitution Principle:** (True subtyping)

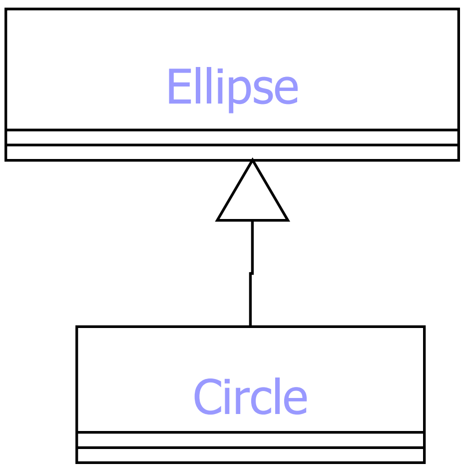
  Let T and S be two types. If the behavior of any program does not change when you replace every T with S, then S is a subtype of T



Apply to this example:

True subtyping

- The Liskov Substitution Principle: LetT and S be two types. If the behavior of any program does not change when you replace every T with S, then S is a subtype of T



Can I replace every Ellipse with Circle?

```
public void foo(Ellipse e){
…
Point x, y;
X=new Point(3,4)
Y=new Point(1,1)
…
e.setFoci(x,y);
..
e.getArea();
}
….
foo(new Ellipse());
foo(new Circle())
```

Can I replace every Ellipse with Circle?

```
public class Circle extends Ellipse{
  private Point c;
  public void setCenters(Point x, Point y)
     throws Exceptions{
    if !(x.equals y) throw new Exception();
   c=x;
  }
 public void setCenters(Point x, Point y){
   //assuming x is c, ignore y
  c=x;}
  public void setCenter(Point x) {c=x;}
}
```

```
public void foo(Ellipse e){
  …
  Point x, y;
  x=new Point(3,4)
  y=new Point(1,1)
  …
  try{
   e.setcenters(x,y);}
  catch(Excepion e){…}
  ..
   e.getArea();
   e.getCenter2();
}
public static void main(String a[]){
  foo(new Ellipse());
  foo(new Circle());}
```

Implement to Interface for LSP

- **Implement to interface** ensures that different classes can be substituted for one another as long as they implement the same interface.
- This aligns with the Liskov Substitution Principle, which states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

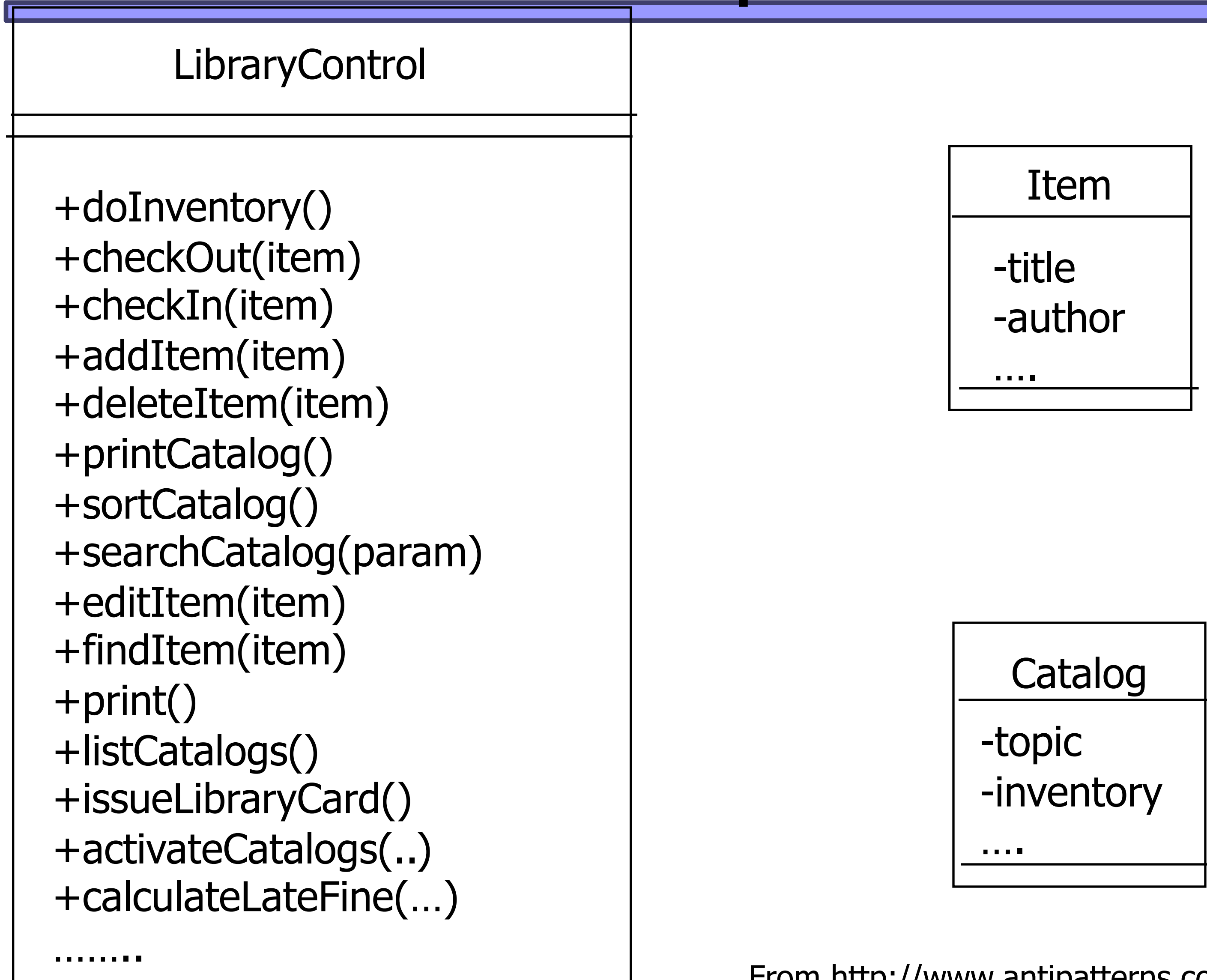
Modularity-Cohesion and Coupling

- Terms of Structural programming
  - Still valid for object oriented design
- Cohesion
  - A module (object) has one a single well-defined purpose
  - “the act or state of sticking together tightly” by Merriam Webster Dictionary
- Coupling
  - Dependencies between modules (i.e. objects)
- Goal: high cohesion and low coupling

Cohesion

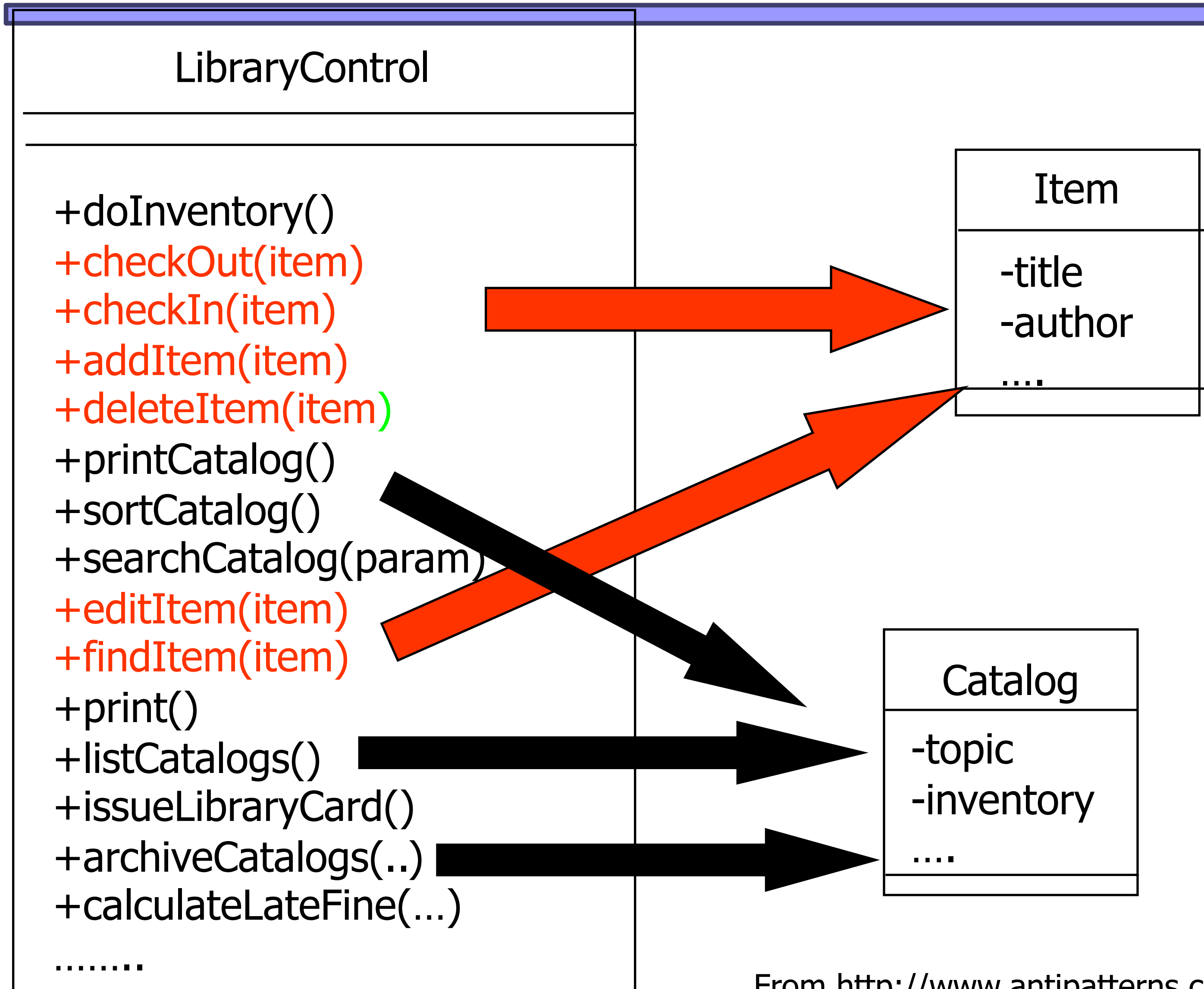
- A measure of the internal quality of an object
- Measures how well the contents of an object support a single well-defined purpose
- If you increase information or the number and type of behaviors in an object, you complicate its design.
  - A lack of cohesion means that a class is performing several unrelated tasks
- Goal: design a class that performs a set of closely related actions (high cohesion)

Cohesion Example



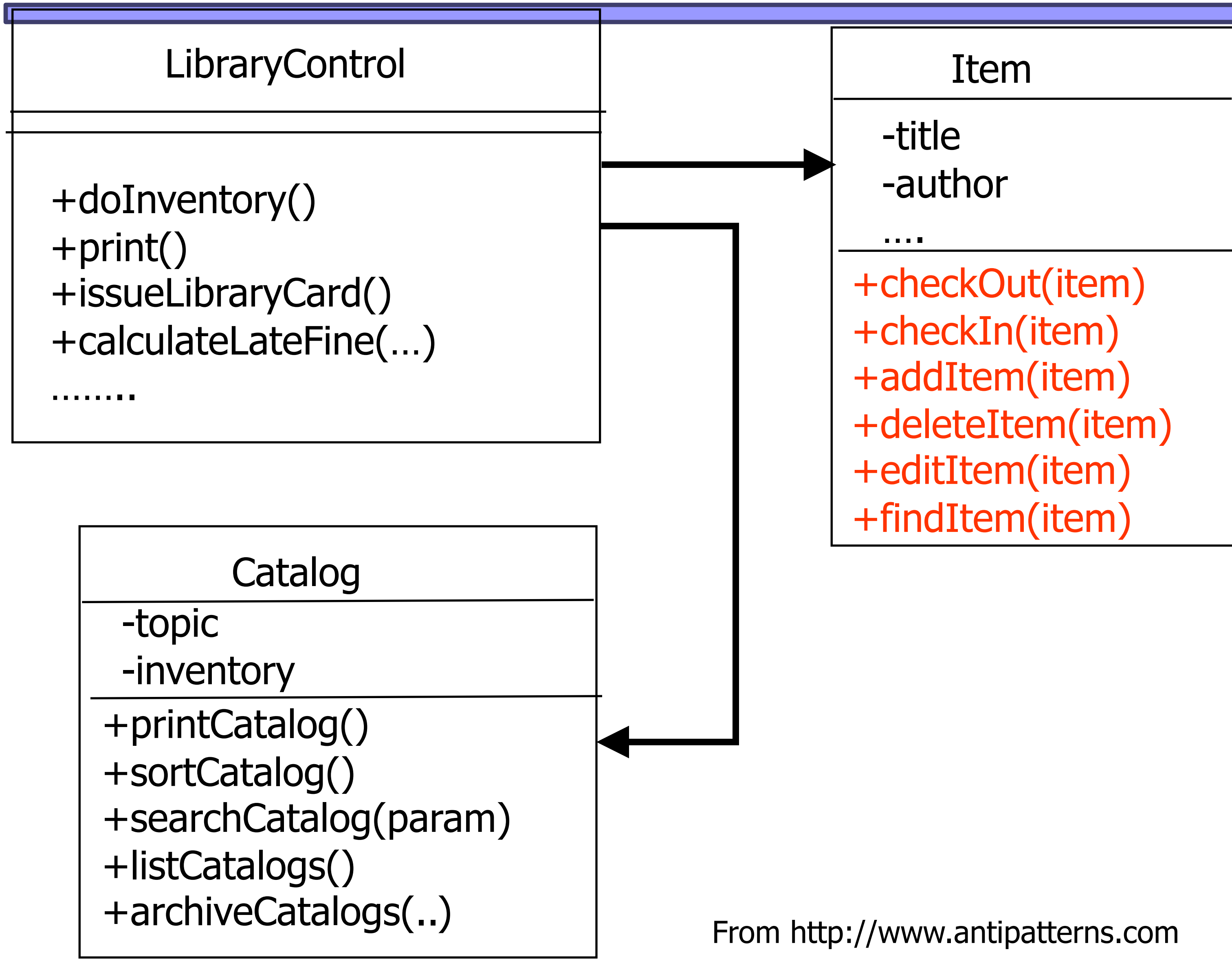
From http://www.antipatterns.com

Low Cohesion Example



From http://www.antipatterns.com

Better Cohesion Example



From http://www.antipatterns.com

Low Cohesion Example (2): Method Level Cohesion

```
public class CoffeeCup {
 public final static int ADD = 0;
 public final static int RELEASE_SIP = 1;
 public final static int SPILL = 2;
 private int innerCoffee;
 public int modify(int action, int amount) {
  int returnValue = 0;
  switch (action) {
   case ADD: // add amount of coffee
        innerCoffee += amount;
        returnValue=0; break;
   case RELEASE_SIP: // remove the amount of coffee passed as amount
        int sip = amount;
        if (innerCoffee < amount) { sip = innerCoffee; }
        innerCoffee -= sip; // return removed amount
        returnValue = sip; break;
  case SPILL: // set innerCoffee to 0
       amount int all = innerCoffee; innerCoffee = 0; // return all coffee
       returnValue = all;
  default: break; }
  return returnValue; }
}
```

Not cohesive:
Too many different tasks for a method

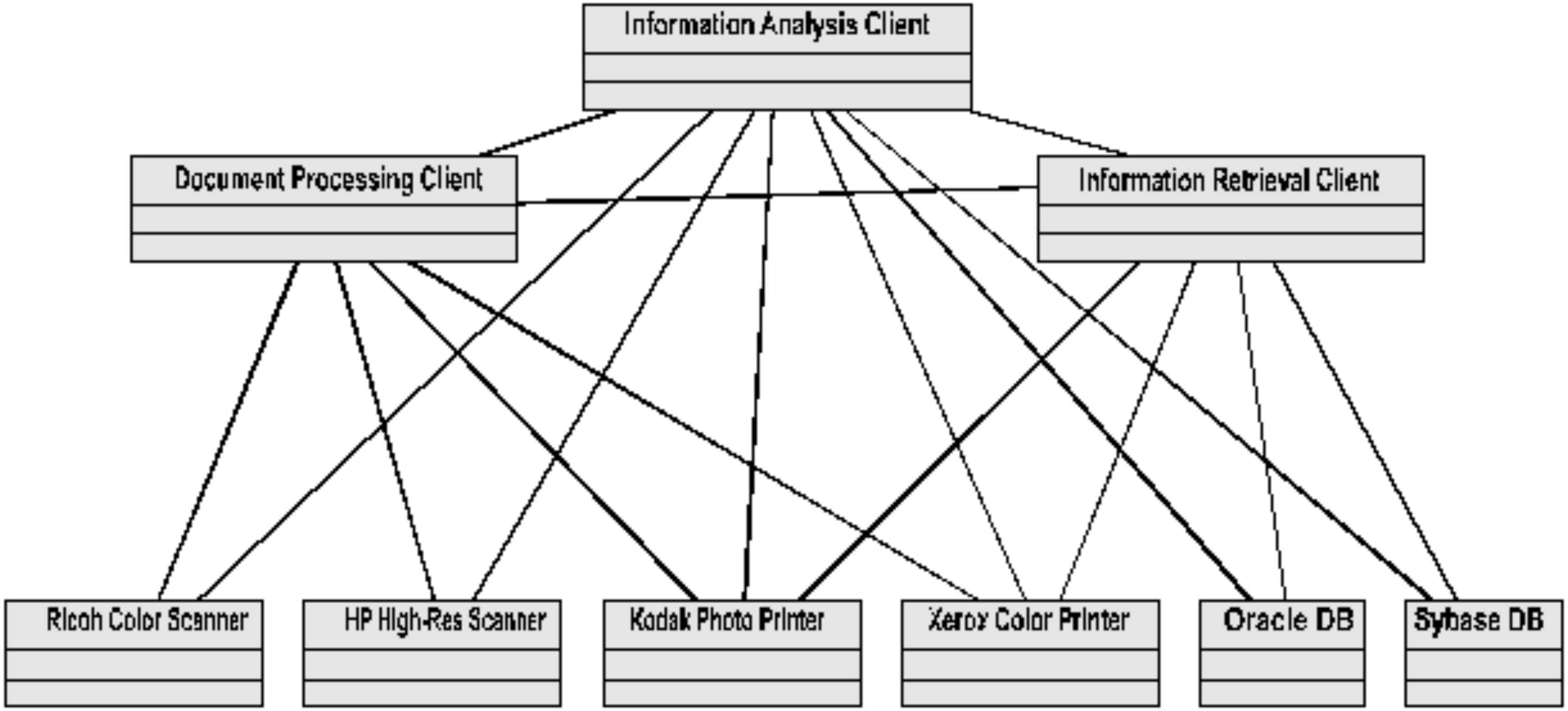
Coupling

- An external measure of object quality
- Measures the complexity of the dependencies between objects in terms of the volume of communication and knowledge that objects have of one another
- Goal: reduce unnecessary dependencies and make necessary dependencies coherent

Coupling

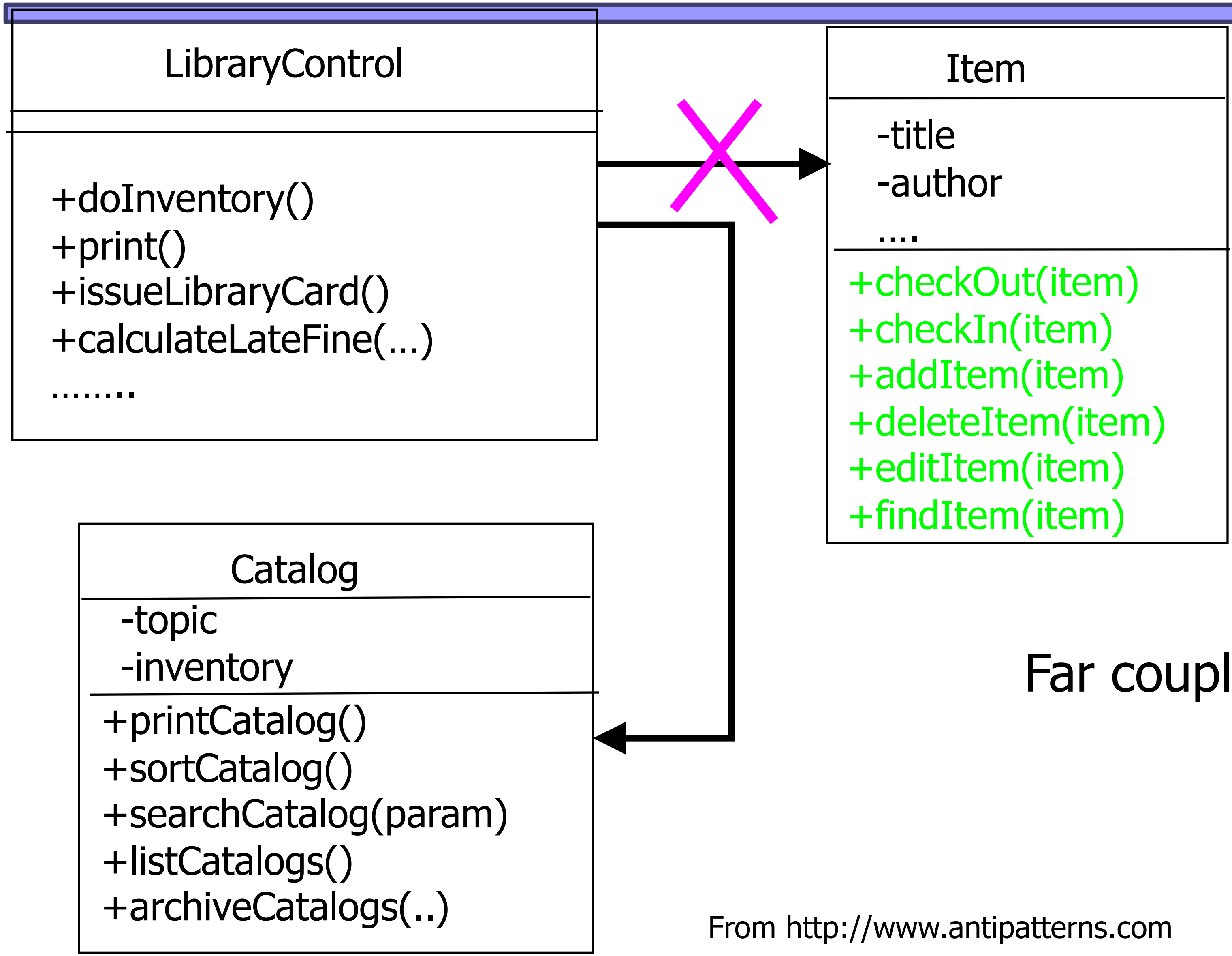
- A class with high coupling is undesirable since
  - Changes in related classes force local changes
  - The class is harder to understand in isolation
  - The class is harder to reuse because its use requires the inclusion of all classes it is dependent upon.

High Coupling Example



From http://www.antipatterns.com

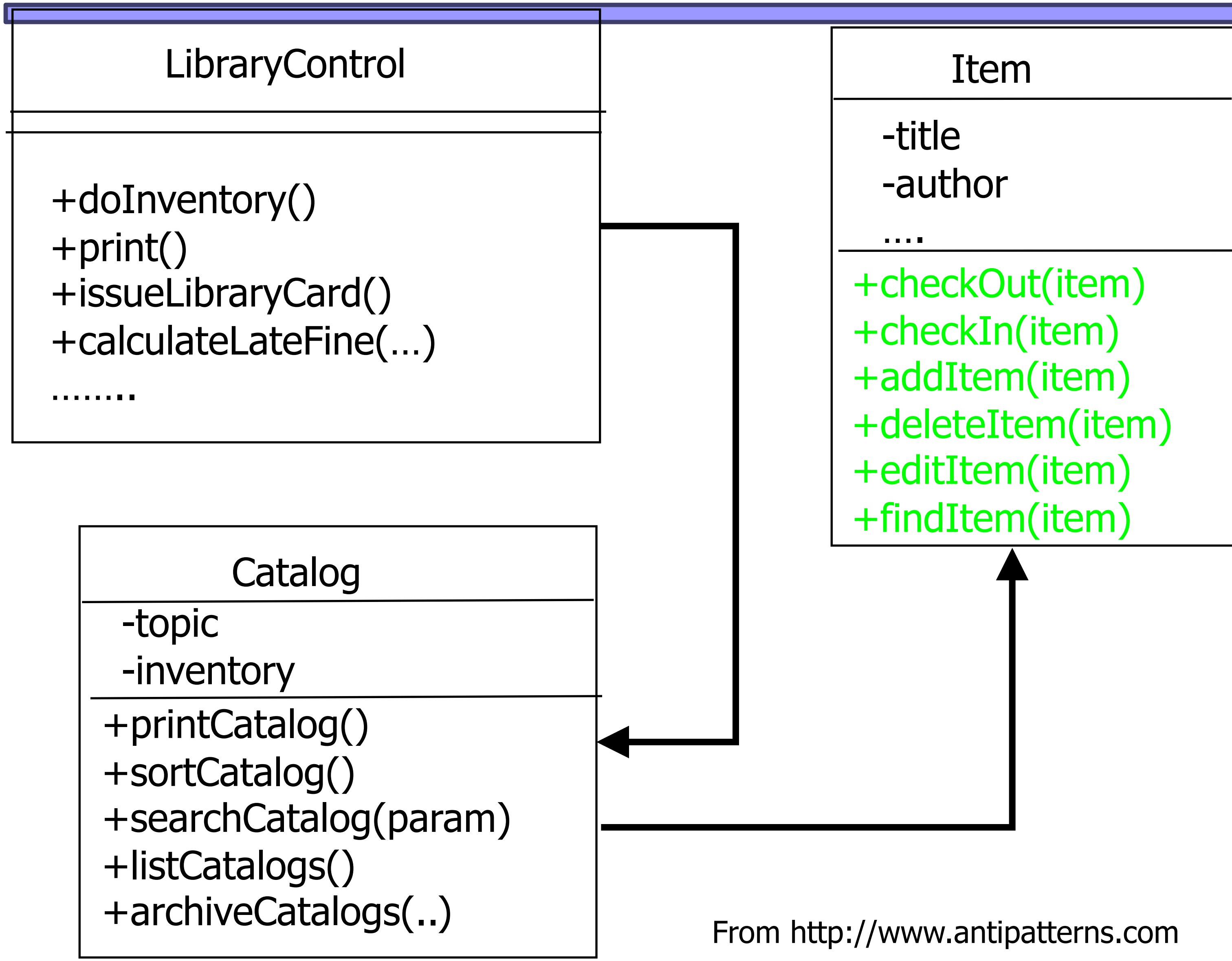
Library Example:Removing coupling



Far coupling

From http://www.antipatterns.com

Library Example



From http://www.antipatterns.com