
Some Object-Oriented Design Principles

Naming Conventions

- Some of these have standard names, and some I just made up.

Uniform Access Principle

- The interface for getting information from an object should be the same regardless of whether the information is:
 - stored, vs.
 - computed
- Related idea: Information hiding

UAP Corollary

- Don't allow public access to data members.

Centralization Principle

- Don't implement the same functionality in more than one place.
- Corollary: Don't cut-and-paste code.

Robert C. Martin

- Robert Martin, in his articles, and recent book "Agile Software Development" has thought deeply about the principles needed to permit a software design to evolve rapidly (as implied by "agile").
- We will mention several of his principles here. They are not limited to agile processes.
- These ideas also tie in with "Design Patterns", as originated by Gamma, et al.
- In some cases, one principle may be seen to generalize another.

Design Smells: The Odors of Rotting Software (Robert C. Martin)

- Software is “rotting” when it starts to exhibit any of the following “odors”:
 - **Rigidity:** The system is hard to change, because every change forces many other changes.
 - **Fragility:** Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
 - **Immobility:** It is hard to disentangle the system into components that can be reused in other systems [or in a new iteration of the same system].
 - **Viscosity:** Doing things right is harder than doing them wrong.

Design Smells, continued

- **Needless Complexity:** The design contains infrastructure that adds no direct benefit.
- **Needless Repetition:** The design contains repeating structures that could be unified under a single abstraction.
- **Opacity:** Code is hard to read and understand. It does not express its intent well.
- These ideas are also related to “anti-patterns” discussed later.

Single-Responsibility Principle (SRP)

(Robert C. Martin)

- "A class should have only one [responsibility, i.e.] reason to change."
- This seems to be the extreme of saying that a class should not have too many responsibilities: It should only have *one*.
- Note, however, that latter would say that the *objects* of the class only have one reason to change, not the *class* itself. Isn't this what is really meant in the SRP?

Example of the SRP

- A Rectangle might be used in two different applications:
 - A geometric reasoning application
 - A graphical (drawing) application
- The reasoning application may need to do operations such as change the size of the rectangle.
- The graphical application may need to draw it (*and* change its size).
- These dual responsibilities should somehow reside in separate classes. [But obviously, there must be a connection, because there is only one rectangle in the second case.]

Rectangle, continued

- What are some ways in which the SRP could be followed in the Rectangle example?

Possible approaches

- Inheritance
- Delegation
- Multiple interfaces for a single implementation class

Martin's approach (stereotypes added)

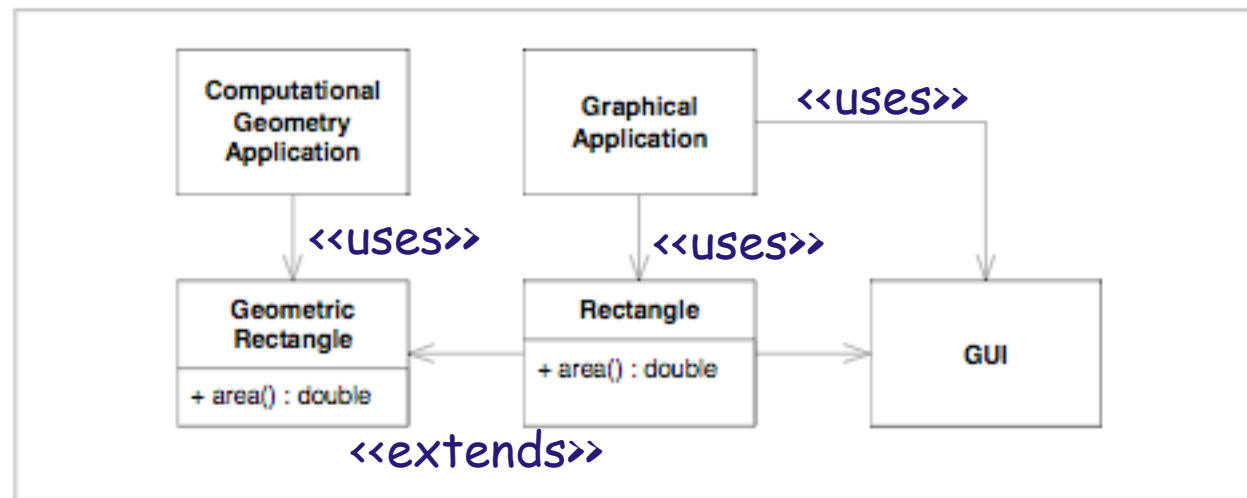


Figure 9-2
Separated Responsibilities

Another Martin example

Listing 9-1

Modem.java -- SRP Violation

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

Resolution using Interfaces

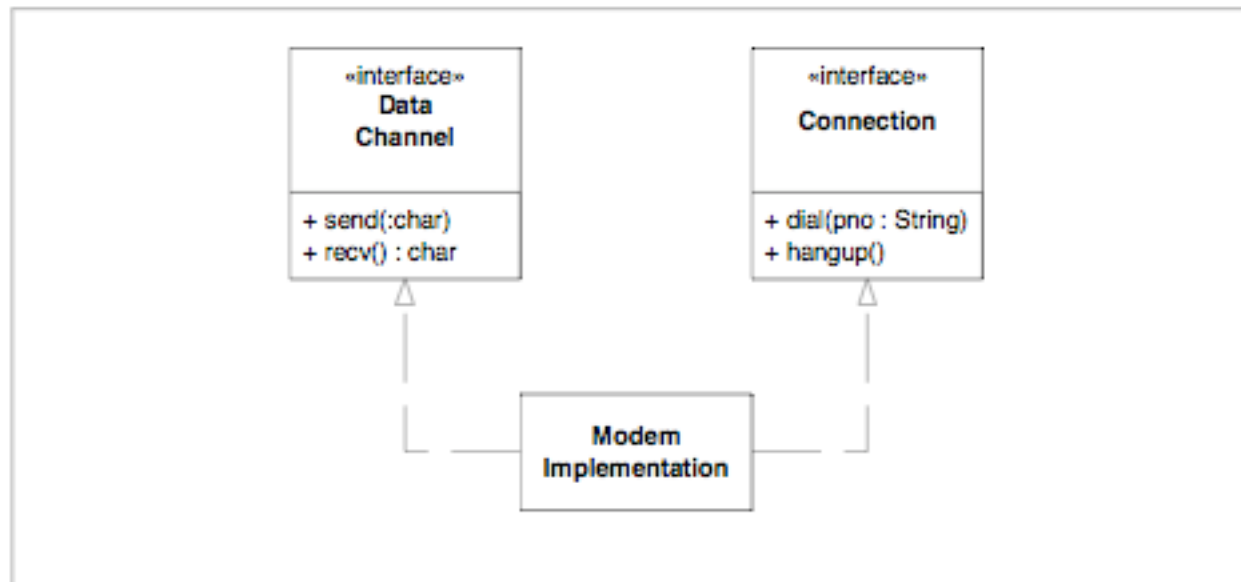


Figure 9-3
Separated Modem Interface

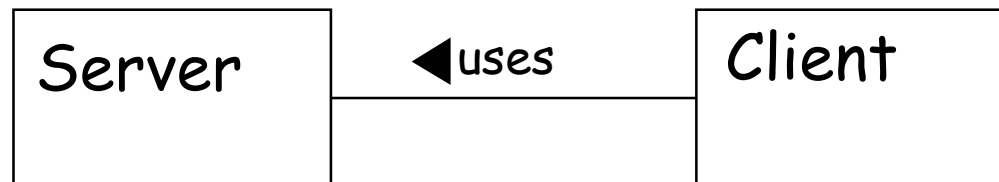
Open/Closed Principle (OCP)

(Robert C. Martin)

- "Classes should be both 'open' and 'closed'."
 - **Open**: in the sense that the class can be **extended** through inheritance.
 - **Closed**: in the sense that the functionality of a class, once set, should not be modified retroactively.
- In other words, add functionality by **inheriting from**, not rewriting, existing code.

OCP Generic Example

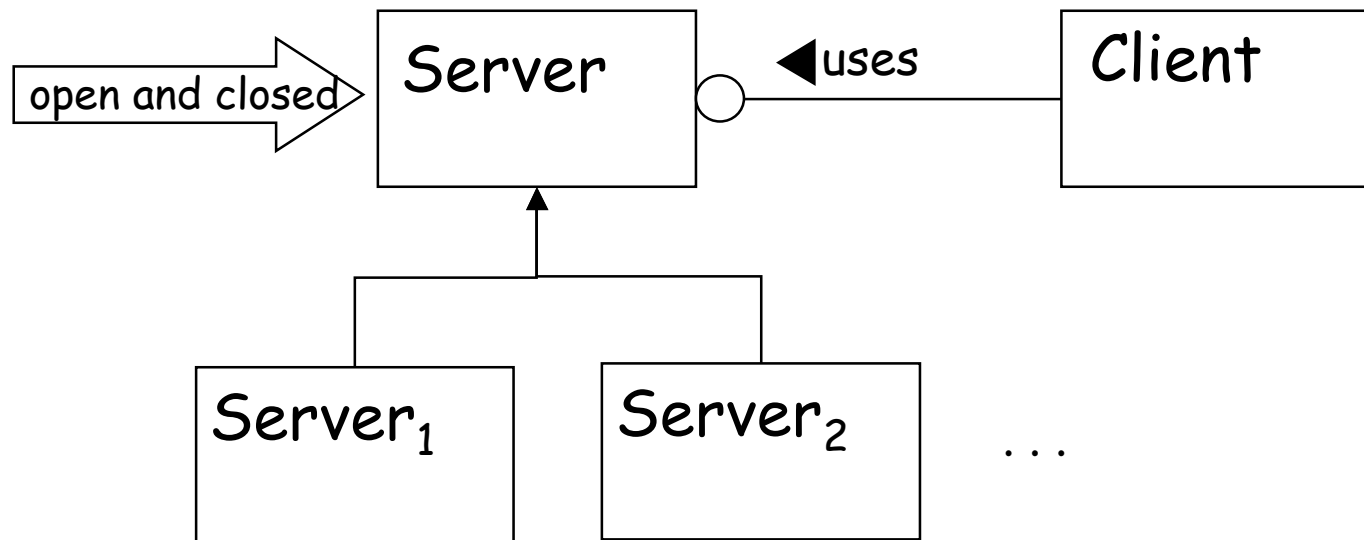
- Having a **client** class that can interact with a fixed server class tends to **violate OCP**:



A Client cannot use a different server, Server', without a rewrite (i.e. is not closed)

OCP Generic Example

- The correction is to define an abstract Server class, or **interface**, and derive concrete servers from it.



Another Common Type of OCP Violation

Listing 1

Procedural Solution to the Square/Circle Problem
enum ShapeType {circle, square};

```
struct Shape
{
    ShapeType itsType;
};

struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};
```

Why is this a violation?

Listing 1 (Continued)

Procedural Solution to the Square/Circle Problem

```
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

//
// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;

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;
        }
    }
}
```

Liskov Substitution Principle (LSP)

As popularly stated:

A member of a derived class must also make sense when used as a member of the base class.

For example, if a method has an object of a class as an argument, the same method should be able to work with an object of a derived class.

As originally stated:

1. INTRODUCTION

What does it mean for one type to be a subtype of another? We argue that this is a semantic question having to do with the behavior of the objects of the two types: the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell.

For example, in strongly typed object-oriented languages such as Simula 67[Dahl,



B. Liskov and J. Wing, A behavioral notion of subtyping, ACM TOPLAS, 16, 6 (Nov. 1994), 1811-1841.

(Prof. Barbara Liskov, M.I.T.)

Clarification

- LSP applies to **behavioral**, rather than structural, properties of objects

Behavioral vs. Structural Sub-Typing

- Consider two classes:
 - Array
 - Stack
- Which is reasonably a sub-class of the other, if either?

LSP Corollary

(Robert C. Martin)

- Functions that use *pointers* or *references* to base classes must be able to use objects of derived classes **without differentiation** of whether the referenced object is base vs. derived.

Impact of LSP

- We should not have to make special-cases of behaviors of derived methods.

This would be an indication that they are not behaving like members of the parent class; maybe rethink this membership.

Thought Questions

- Suppose we have a class `RealValuedFunction` and a sub-class `DifferentiableFunction`. How would these be tied together or differentiated?
- Should a class `Bird` have a method `flyDistance()`?
- Suppose we have a class `Year`. Should we have a subclass `LeapYear`?

Thought Questions

- Should class Square be derived from class Rectangle; or vice-versa?
- Should class Circle be derived from class Ellipse; or vice-versa?
- Suppose we have a class FacultyMember, which has methods
 hire()
 fire()

Should we have a sub-class TenuredFacultyMember?

Thought Questions

- Suppose we have two kinds of Employee:
Agent
LotAttendant

Should both of these be classes extending a common base class: Employee?

- For more discussion of this kind:

<http://alistair.cockburn.us/crystal/articles/cdos/constructivedesconstructionofsubtyping.htm>

Martin's Rectangle & Square, v1

```
class Rectangle
{
public:
    void SetWidth(double w)    {itsWidth=w;}
    void SetHeight(double h)  {itsHeight=w;}
    double GetHeight() const  {return itsHeight;}
    double GetWidth() const   {return itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};
```

```
void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}
```

```
void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

LSP violation in v1

```
void f(Rectangle& r)
{
    r.SetWidth(32); // calls Rectangle::SetWidth
}
```

If a Square is passed for r,
the width is set, but not the height.
The resulting shape might not be a square any
longer, even though it is called one.
That is, the "square invariant" is violated.
[This is an issue in C++, but not Java.]

```
void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

v2 to "fix" v1 problems, using virtual methods in C++

```
class Rectangle
{
public:
    virtual void SetWidth(double w)    {itsWidth=w;}
    virtual void SetHeight(double h)  {itsHeight=h;}
    double      GetHeight() const     {return itsHeight;}
    double      GetWidth() const      {return itsWidth;}
private:
    double itsHeight;
    double itsWidth;
};

class Square : public Rectangle
{
public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

The OCP was violated.
We had to recode Rectangle.

An anomaly with v2

```
void g(Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.GetWidth() * r.GetHeight() == 20);
}
```

The assertion will always succeed for a Rectangle, but will fail for a Square, a violation of LSP.

Dependency-Inversion Principle (DIP)

(Robert C. Martin)

- Details should depend on abstractions; Abstractions should not depend on details, but at most on other abstractions.
- High-level modules should not depend on low-level modules; both should depend on abstractions.
- In other words, don't let low-level modules "call the shots" for high-level ones. The high-level ones are where policies should be set.
- Succinctly: *Specify the interface first*, then implement.

Exercise

- Design a simple class structure for switches (e.g. button, toggle, knife, ...) and devices (e.g. lamp, motor, radio, ...).
- A switch is used to turn a device on and off.
- Every device has a switch.
- Different kinds of switches should be usable with a given type of device.

- Where are abstract classes vs. interfaces appropriate?

- Extend to the case where devices are operated by relays, e.g. for noise or power isolation purposes.

More Named Principles (1)

(cf. <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>)

- **Note that not all principles are without controversy.**
- **The Interface Segregation Principle:** Clients should not be forced to depend on unused aspects of interfaces.
 - Having multiple client-specific interfaces is better than having one general purpose (aka "all-singing, all-dancing") interface.
- **The Reuse/Release Equivalency Principle:** The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
- **The Common Reuse Principle:** Classes that aren't reused together should not be grouped together in the same package.
- **The Common Closure Principle:** Classes that tend to change together, belong together in the same package. [Is this different from Common Reuse?]

More Named Principles

(cf. <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>)

- **The Acyclic Dependencies Principle:** The dependency structure for released components (i.e. packages) should be a directed acyclic graph.
- **The Stable Dependencies Principle:** Dependencies between released categories must run in the direction of stability. The dependee should be more stable than the depender.
- **The Stable Abstractions Principle:** The more stable a class category is, the more it should consist of abstract classes. A completely stable category should consist of nothing but abstract classes.

Law of Demeter



Law of Demeter

- This “law” seems generally worthwhile.
- It is not without controversy and difficulties in understanding, cf.
<http://c2.com/cgi/wiki?LawOfDemeterIsHardToUnderstand>
- Therefore it should not be followed slavishly, but neither should it be ignored.

Law of Demeter (LoD)

colloquial version

- “Only talk to your immediate friends”.
- [not meaning *friends* in the sense of C++ classes and methods]
- so named by Prof. Karl Lieberherr, Northeastern University:
- LoD home page:
<http://www.cs.neu.edu/home/lieber/LoD.html>



Law of Demeter Principle as stated by its author

- Each unit [i.e. class, method] should only use a limited set of other units: only units "closely" related to the current unit.
- Main Motivation: Control information overload. We can only keep a limited set of items in our short-term memory.
- Secondary Motivation: maintainability: A class should not have to know much about distant classes.

Rumbaugh, Booch and the LoD

Rumbaugh: "Avoid traversing multiple links or methods. **A method should have limited knowledge of an object model.** A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class."

Booch: "The basic effect of applying this Law is the creation of **loosely coupled classes**, whose implementation secrets are **encapsulated**. Such classes are fairly unencumbered, meaning that **to understand the meaning of one class, you need not understand the details of many other classes.**"

LoD: More Specific OO Interpretation

- An object should only invoke methods of:
 - objects that are *declared* within the object's class
 - objects that are *parameters of the method*
 - *itself*
 - objects that it *creates*
- mnemonic DPIC ("depict")

Expressly Precluded by LoD

- Do not extract object B from an object A and perform an operation on it.
- Instead, **recast** what you want to do as an operation on A. That operation may call operations on B.

Example of Violating the Law of Demeter

- class Patron
{
void sendNotice();
}

- class Book {
Patron getBorrower();
};

```
library.getBook("Ulysses").getBorrower().sendOverdueNotice();
```

- class Library
{
map<Book> booksByTitle;

Book getBook(string Title);
};

The above statement is considered bad:
The client of library has to *know about* books and borrowers just to send this notice.

Remedying the Violation

- ```
class Library
{
 map<Book> booksByTitle;

 void sendOverdueNotice(string Title);
};
```

```
library.sendOverdueNotice("Ulysses");
```

# Also called the "one-dot" rule

---

---

Prefer

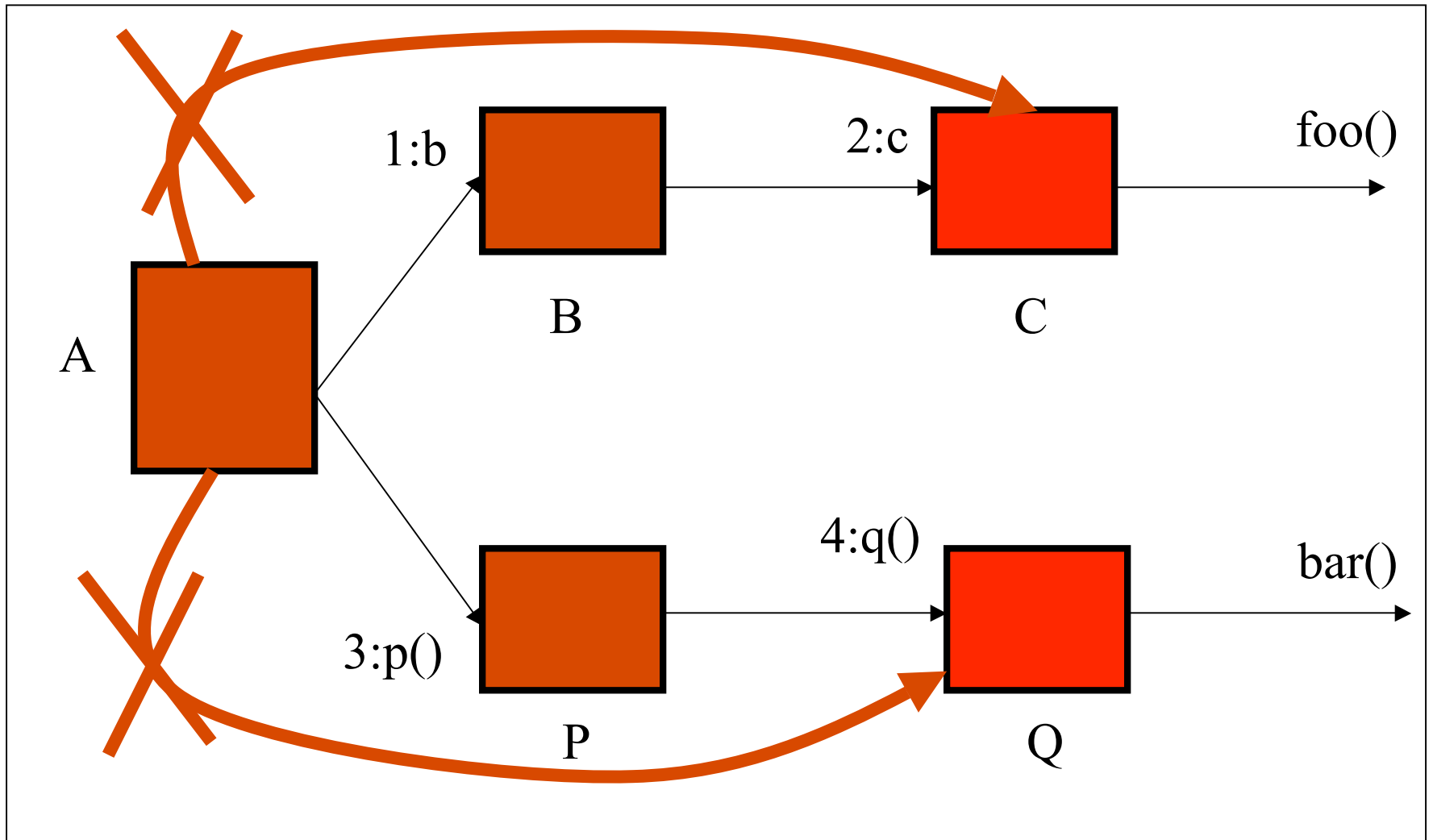
```
sendOverdueNotice("Ulysses");
```

to

```
library.getBook("Ulysses").getBorrower().sendOverdueNotice();
```

# Violations: Dataflow Diagram

(slide from Lieberherr)



# More on Following the LoD

---

---

- General idea: Keep the structure as loose as possible (known as "late binding" in conventional system design).
- Specific incarnations:
  - "Adaptive Plug-and-Play Components"  
(Liberherr)

---

---

Further Guidelines  
related to LoD  
will be seen when we  
discuss "Design Patterns"

# Related Hot Topic

---

---

- Aspect-Oriented Programming
  - Program implementation is sliced up by "aspects" which tend to **cut across** class boundaries.
  - Aspects are addressed separately in programming, the **woven** together with a weaver, using "join points".
- *Aspect-Oriented Programming*, Gregor Kiczales, et al., Xerox PARC, 1997

# Example of Aspects

---

---

- For a mail-order company
  - Ordering aspect
  - Customer service aspect
  - Shipping aspect
  - Inventory aspect
  - Pricing aspect
  - Marketing aspect
  - Accounting aspect
  - Shareholder aspect
- These aspects can largely be treated separately in an enterprise application, yet they have various *join points* at specific products, customers, etc.