CS 535 Object-Oriented Programming & Design
Fall Semester, 2001
Doc 21 Some Design Heuristics

Meyer's Goals for Modular software 2
Decomposability 2
Composability 3
Understandability 4
Continuity 5
Protection 6
Design Heuristics 7
Some Basic Design Heuristics 7
Extremes - God class and Proliferation of Classes 13
Keep it Small 22
Common Minimal Public Interface 23
Relationships between Objects 24
Six different Ways to Implement Uses 25
Heuristics for the Uses Relationship 27
Containment Relationship 29
Narrow and Deep Containment Hierarchies 30
No Talking between Fields 31

References

Object-Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988

Object-Oriented Design Heuristics, Riel, Addison-Wesley, 1996,

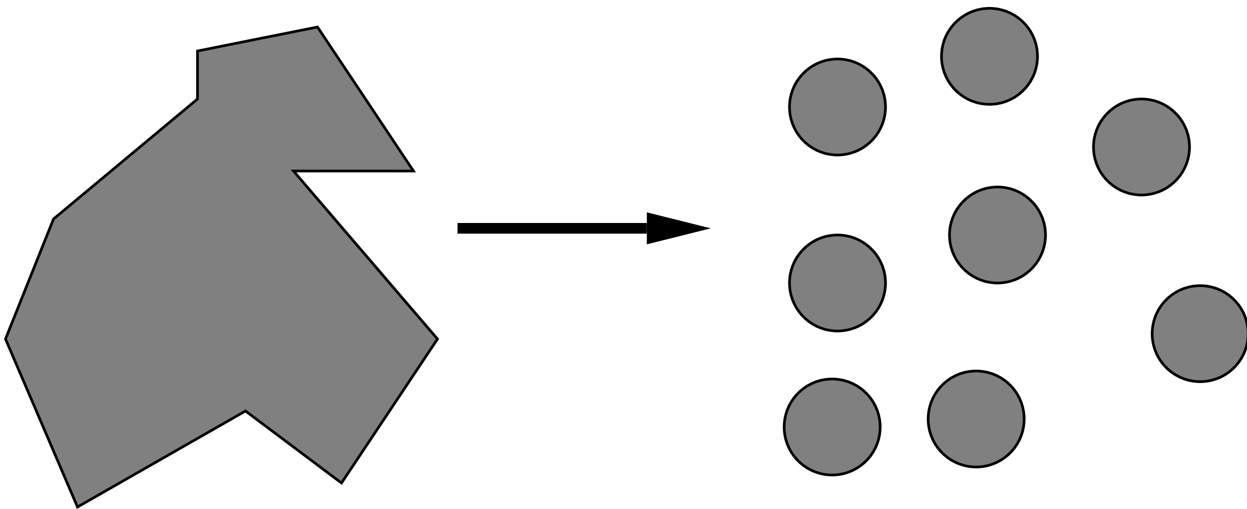
Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Meyer's Goals for Modular software
Decomposability

Decompose problem into smaller subproblems that can be solved separately

Example: Top-Down Design

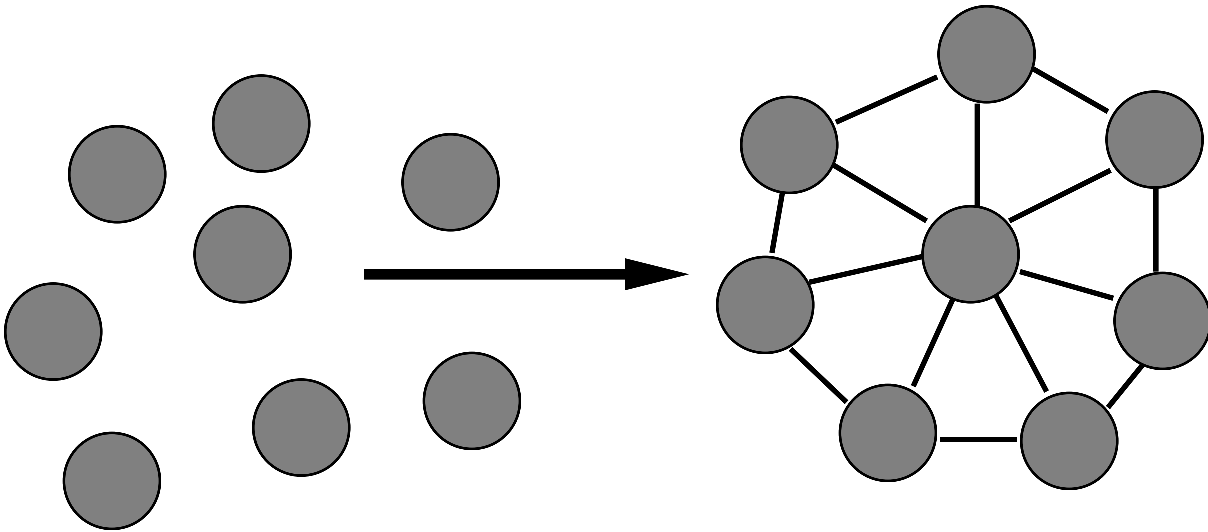
Counter-example: Initialization Module



Meyer's Goals for Modular software
Composability

Freely combine modules to produce new systems

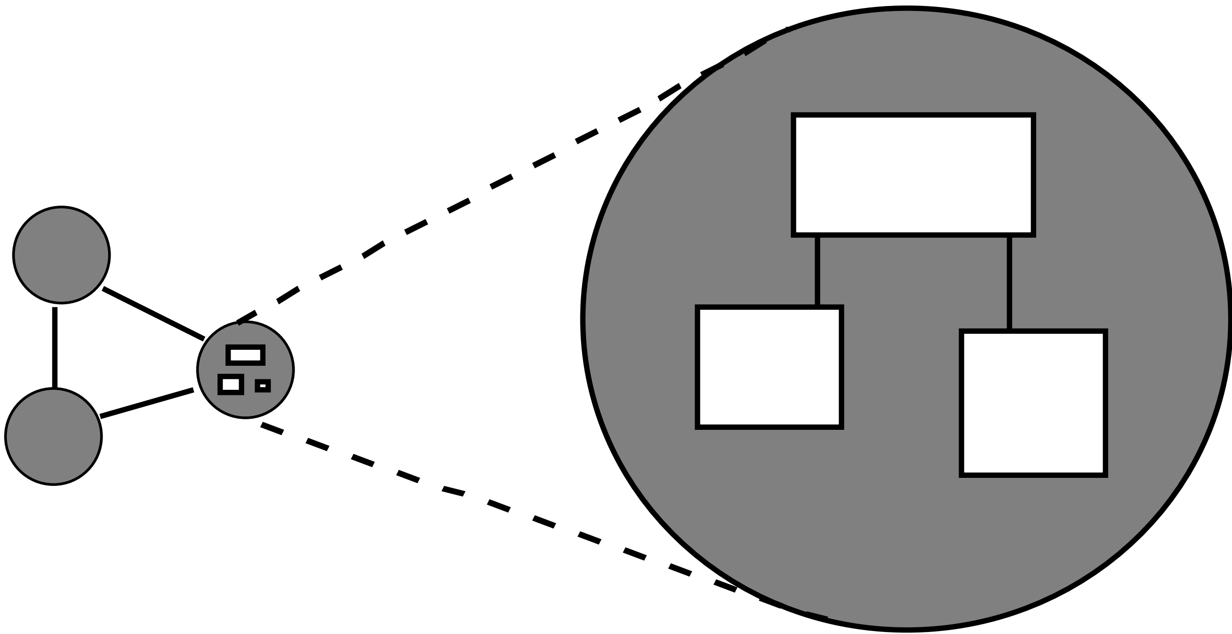
Examples: Math libraries
Unix command & pipes



Meyer's Goals for Modular software
Understandability

Individual modules understandable by human reader

Counter-example: Sequential Dependencies



Meyer's Goals for Modular software
Continuity

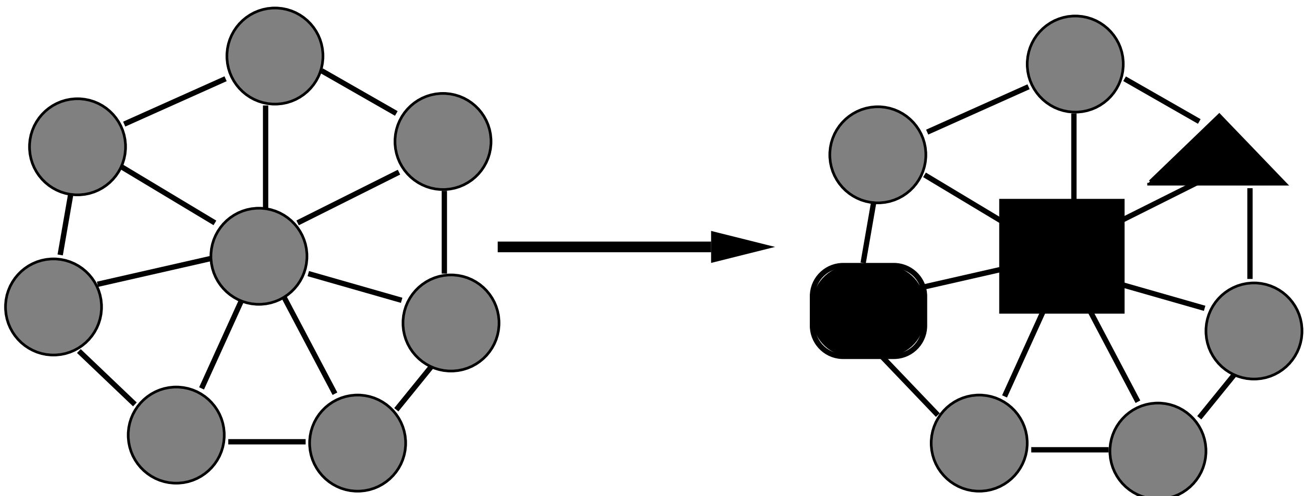
Small change in specification results in:

Changes in only a few modules

Does not affect the architecture

Example: Symbolic Constants

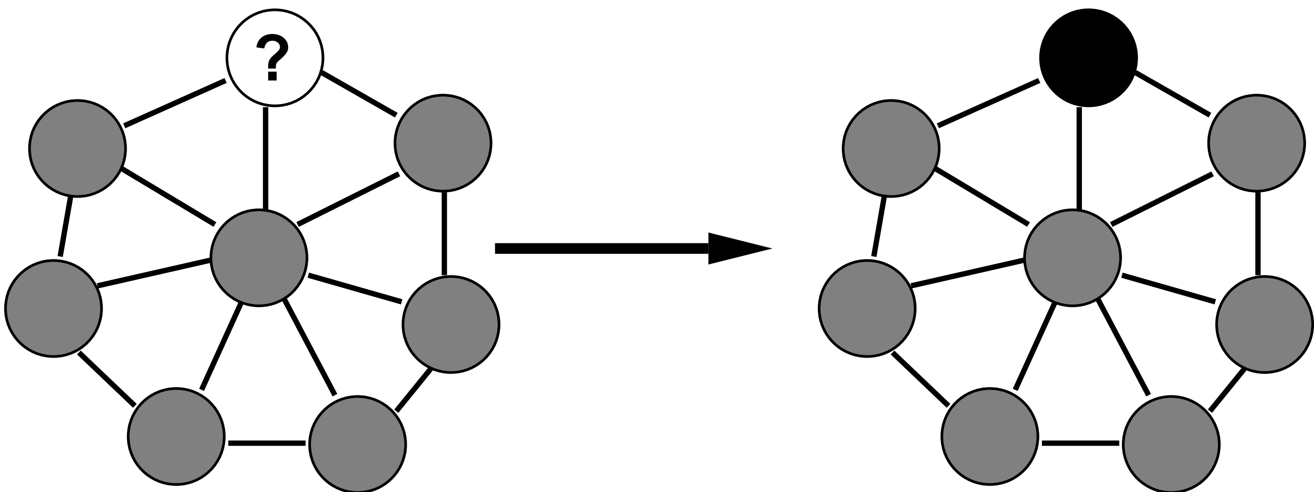
const MaxSize = 100



Meyer's Goals for Modular software
Protection

Effects of an abnormal run-time condition is confined to a few modules

Example: Validating input at source



Design Heuristics
Some Basic Design Heuristics

2.1 All data should be hidden within its class

2.9 Keep related data and behavior in one place

3.3 Beware of classes that have many accessor methods defined in their public interfaces. Having many implies that related data and behavior are not being kept in one place

All data should be hidden within its class

Each time you wish to make an instance variable public ask:

What am I trying to do with this data and why is it not being done in the class for me?

Is the data in the correct class?

Public instance variables verses accessor methods

Public instance variables

- Expose the class implementation
- Possible in C++ & Java

Accessor methods

- Do not have to expose the implementation
- By programmer convention often do expose implementation

Example

Assume we have a customer class with an id

```
Smalltalk.CS535 defineClass: #Customer
   instanceVariableNames: 'name phone id '

id
   ^id
```

Now we decide to use Customer as a model

Which do we use:

```
id
   ^id isNil
      ifTrue: [id := 0 asValue]
      ifFalse: [id]

id
   ^id value
```

Keep related data and behavior in one place

Data and behavior are part of the same abstraction

If they are not in the same place we have to violate previous heuristic

Keeping multiple copies of the data in different places is not keeping it in one place

Beware of many public accessor methods in a class

Having many implies that related data and behavior are not being kept in one place

Example Heating Problem

Example from Booch and used in Riel's text

Room has
- Temperature sensor
- Occupancy sensor
- Desired Temperature setting

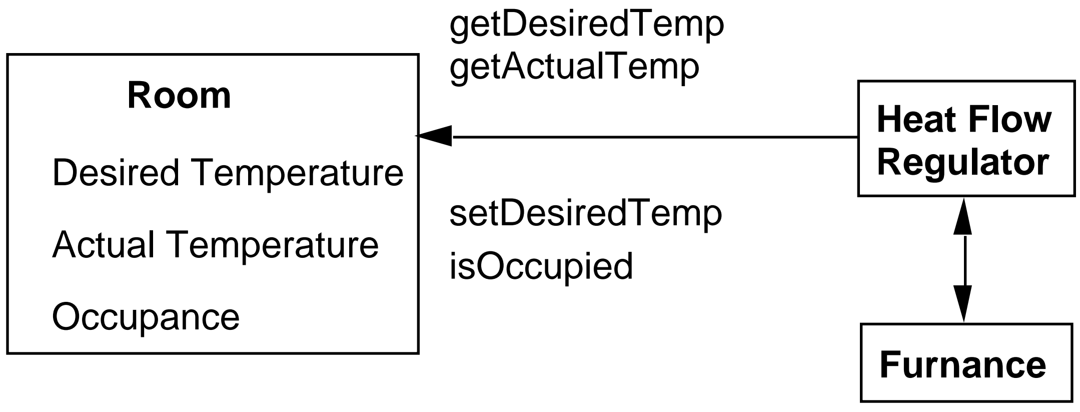
Heat Flow Regulator
- Responsible for sensing when any room needs heat

If a room needs heat regulator:
- Turns on furnace
- Waits for water to heat up
- Tells room heat is available

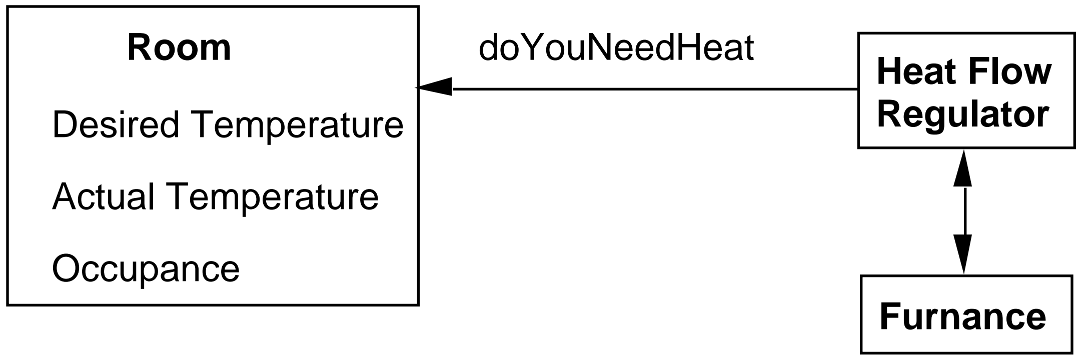
If room is occupied add heat when temperature is less than desired temperature

If room is not occupied add heat when temperature drops 5 degrees below the desired temperature

Solution 1



Solution 2



Extremes - God class and Proliferation of Classes

God class

- Performs most of the work in the system
- Leaves minor tasks to others
- Replaces main of procedural programming

Proliferation of Classes

- Lots and lots little classes

God Class Heuristics

2.8 A class should capture one and only one key abstraction

3.1 Distribute system intelligence horizontally as uniformly as possible

3.2 Do not create god classes/objects in your systems

2.10 Spin off nonrelated information into another class

3.4 Beware of classes that have too much noncommunicating behavior

4.6 Most of the methods defined in a class should be using most of the data members most of the time

A class should capture one and only one key abstraction

A god class does many things

Sometimes it is hard to determine what it is

If you have too many classes, some of them may not represent an abstraction

One sentence description of the class

- If it needs the word "and" the class may be doing too much
- Class Foo does X, Y and Z

Distribute system intelligence horizontally as uniformly as possible

Top-level classes in a design should share the work

God classes tend to do most of the work

Do not create god classes/objects in your systems

Beware of classes whose names contains:

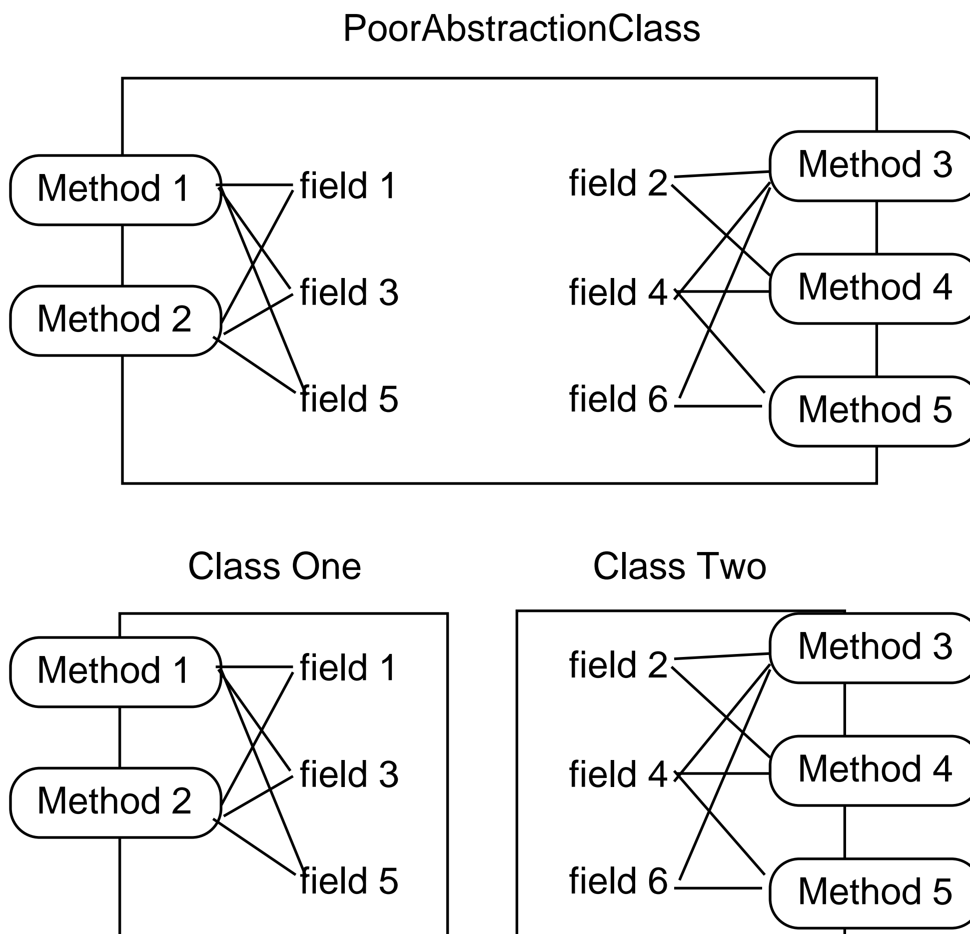
- Driver
- Manager
- System
- Subsystem

God classes tend to be big and complex

2.10 Spin off nonrelated information into another class

3.4 Beware of classes that have too much noncommunicating behavior

4.6 Most of the methods defined in a class should be using most of the data members most of the time



Proliferation of Classes Heuristics

2.11 Be sure that abstractions that you model are classes and not simply the roles objects play

5.15 Do not turn objects of a class into subclasses of the class

3.7 Eliminate irrelevant classes from your design.

3.8 Eliminate classes that are outside the system.

3.9 Do not turn an operation into a class

Model classes and roles objects play

Do not turn objects of a class into subclasses of the class

Is RentalItem a class or a role for an Item object?

Is OverDueRentalItem a class or a role for a RentalItem object

Eliminate irrelevant classes from your design

Irrelevant class
- A class with no meaningful behavior in the domain
- Has only gets, sets and print operations

Either there is missing behavior for the class or the data in the class should be made attributes of another class

Eliminate classes that are outside the system

Actors are an example of things outside the system

Actor is a role that users play with respect to the system

Manager, Clerk, Customer

Actor classes often occur in analysis models, but are found to be irrelevant at design time

Blender Example

Company X built a product registration system for:

Blenders, toasters, TVs, and other consumer equipment

Should Blender be a class?

Is has actions like whip, chop, puree, etc

Do not turn an operation into a class

Beware of a class whose name is a verb or derived from a verb

Check to see if such classes have more than one meaningful behavior

May need to move the behavior to a different class

Example

IOHandler Class

Methods:

    writeFile:
    readFile:

Keep it Small

2.3 Minimize the number of messages in the protocol of a class

2.5 Do not put implementation details such as common-code private functions into the public interface of a class.

2.6 Do not clutter the public interface of a class with things that users of the class are not able to use or are not interested in using.

Common Minimal Public Interface

2.4 Implement a minimal public interface that all classes understand

This can be applied a global Object class, but also applies to classes in a project or domain

Java's Object

clone()
   Creates and returns a copy of this object.
equals(Object obj)
   Indicates if another object is "equal to" this one.
finalize()
   Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
getClass()
   Returns the runtime class of an object.
hashCode()
   Returns a hash code value for the object.
toString()
   Returns a string representation of the object.

Thread Related Methods

notify()	wait(long timeout)
notifyAll()	wait(long timeout, int nanos)
wait()	

VisualWorks 5i4

The Object class has 201 instance methods

Operations, Classes, Methods

Relationships between Objects

Type of Relations:

Type	Relation between
Uses	(Object)
Containment	(Object)
Inheritance	(Class)
Association	(Object)

Uses

Object A **uses** object B if A sends a message to B

Assume that A and B objects of different classes

A is the sender, B is the receiver

Containment

Class A contains class B when A has a field of type B

That is an object of type A will have an object of type B inside it

Six different Ways to Implement Uses

How does the sender access the receiver ?

1. **Containment**

The receiver is a field in the sender

```
class Sender {
 Receiver here;

 public void method() {
   here.sendAMessage();
 }
}
```

2. **Argument of a method**

The receiver is an argument in one of the sender's methods

```
class Sender {
 public void method(Receiver here) {
   here.sendAMessage();
 }
}
```

3. **Ask someone else**

The sender asks someone else to give them the receiver

```
class Sender {
 public void method() {
    Receiver here = someoneElse.getReceiver();
    here.sendAMessage();
 }
}
```

4. **Creation**

The sender creates the receiver

```
class Sender {
 public void method() {
    Receiver here = new Receiver();
    here.sendAMessage();
 }
}
```

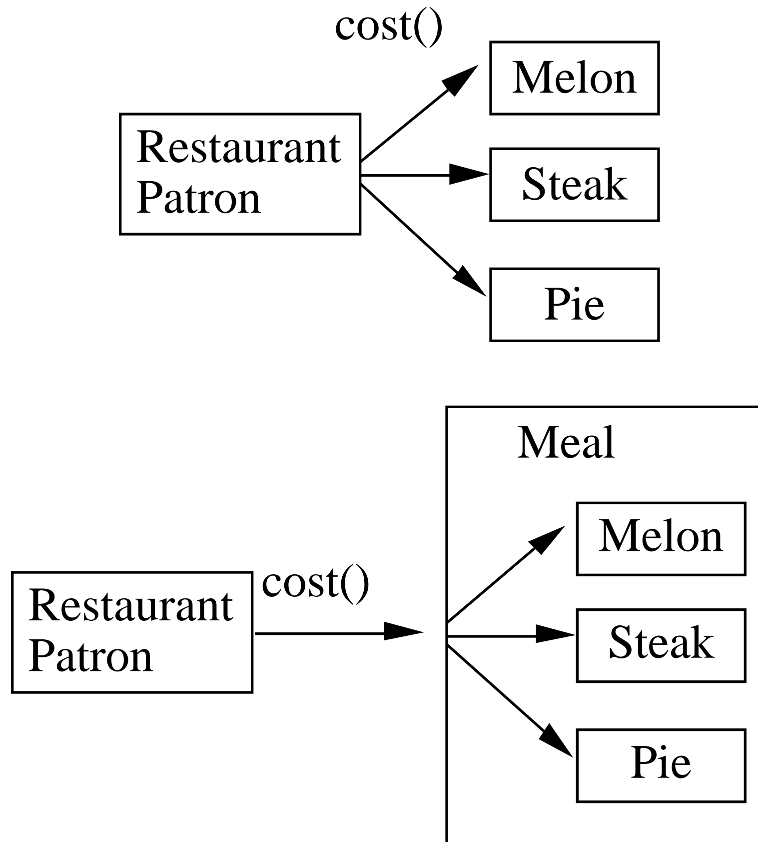
5 Referential Attribute (Used for Association)

6. **Global**

The receiver is global to the sender

Heuristics for the Uses Relationship

4.1 Minimize the number of classes with another class collaborates

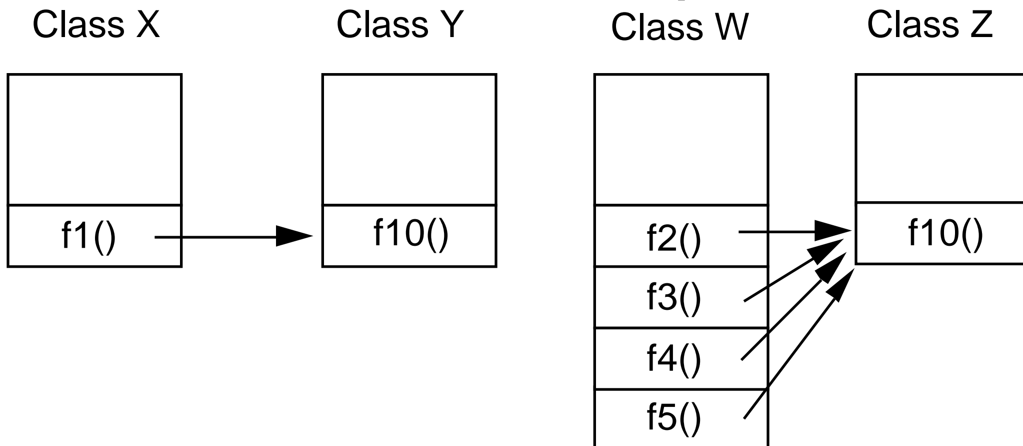


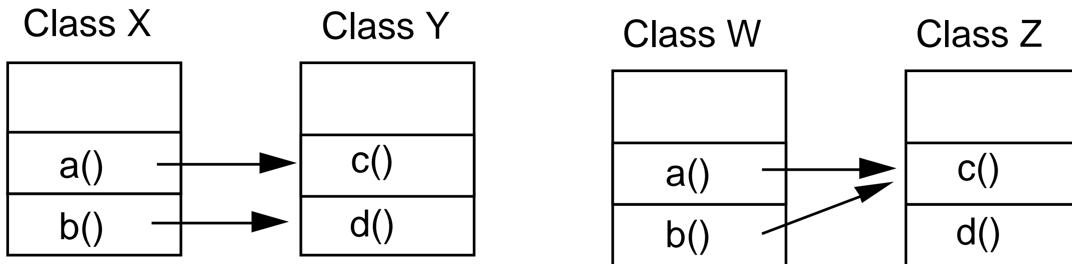
4.2 Minimize the number of message sends between a class and its collaborator

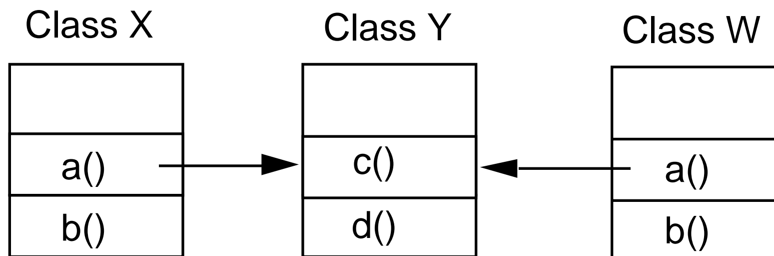
4.3 Minimize the number of different messages a class sends to another class.

4.4 Minimize the product of the number of methods in a class and the number of different messages they send.

Which is more complex?







Containment Relationship

4.5 If class A contains objects of class B, then A should be sending messages to its fields of type B.

This heuristic prohibits:

- Orphaned fields (ones that are never used)
- Fields that are only accessed in get/set methods
  - The one exception to 4.5 is container classes

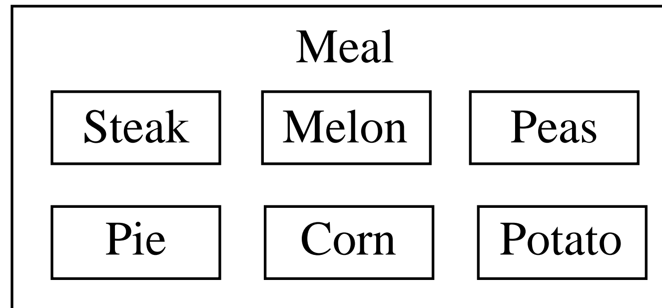
```
class Foo {
  Bar data;

  public Bar getData()  {
    return data;
  }

  public void setData( Bar newData) {
    data = newData;
  }
}
```

4.6 Most of the methods defined on a class should be using most of the fields in the class most of the time

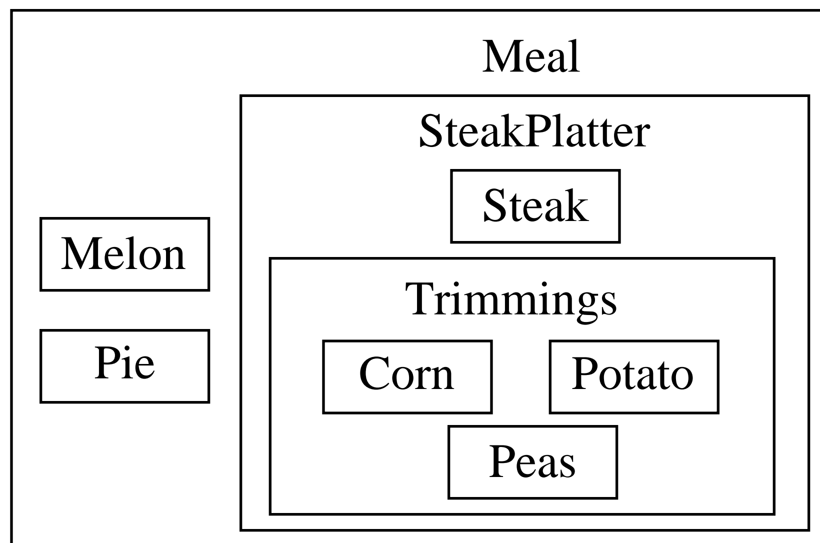
4.7 Classes should not contain more objects than a developer can fit in his or her short-term memory. A common value for this number is 6

Narrow and Deep Containment Hierarchies

Combining fields into new classes can reduce the number of fields in a class



A broad and shallow Containment Hierarchy

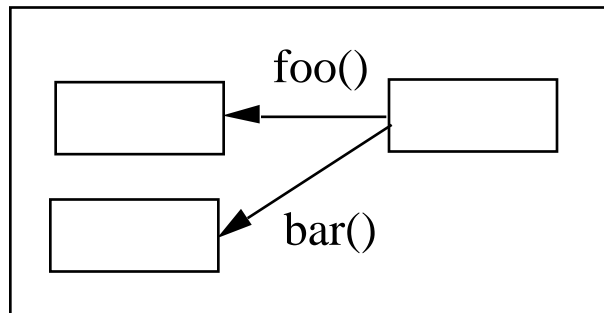


A narrow and deep Containment Hierarchy

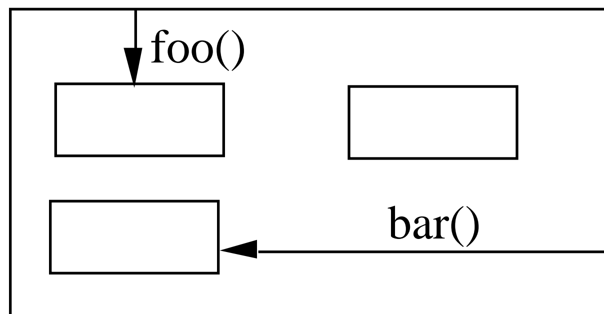
4.8 Distribute system intelligence vertically down narrow and deep containment hierarchies.

No Talking between Fields

4.14 Objects that are contained in the same containing class should not have a uses relationship between them.



Contained Objects with uses relationships



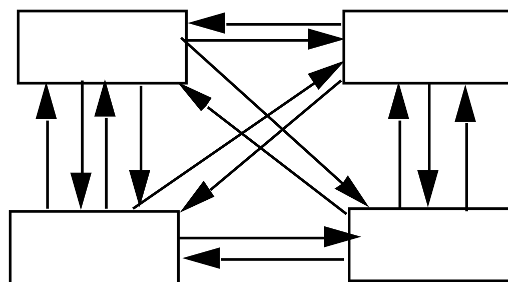
The containing class should send messages to the contained objects

4.13 A class must know what it contains, but it should not know who contains it.

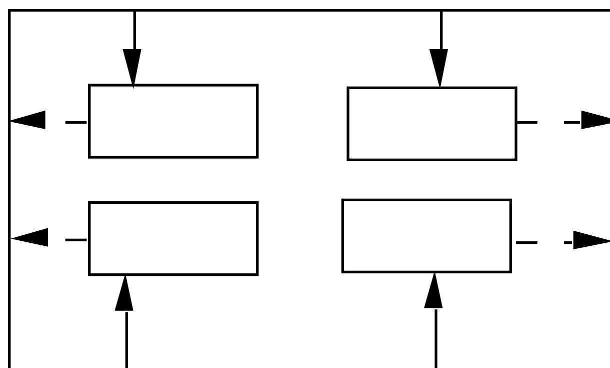
If a number of classes depend on each other in a complex way, you can violate 4.13 to reduce the number of classes they interact with.

Wrap the classes in a containing class.

Each contained class sends a message to the containing class, which broadcasts the message to the proper contained objects



Complex interactions



Replacing complex interaction with containing class