Génie Logiciel et Gestion de Projets

Basics of Good OO Designs

Good OO Design and Thinking

Good OO Design

- Minimize number of messages in the protocol of the class.
- Use public only when necessary.
- Minimize number of accessors.
- Factorize as much as possible
- Short, clear methods
- No dense methods

Good OO Thinking

- Objects should have clear responsibilities
  ! Try to state the purpose of the class in one sentence
- No super-intelligent objects
- No manager objects
- Not too many instance variables

Behaviour needs to be distributed more or less evenly

Inheritance Relation

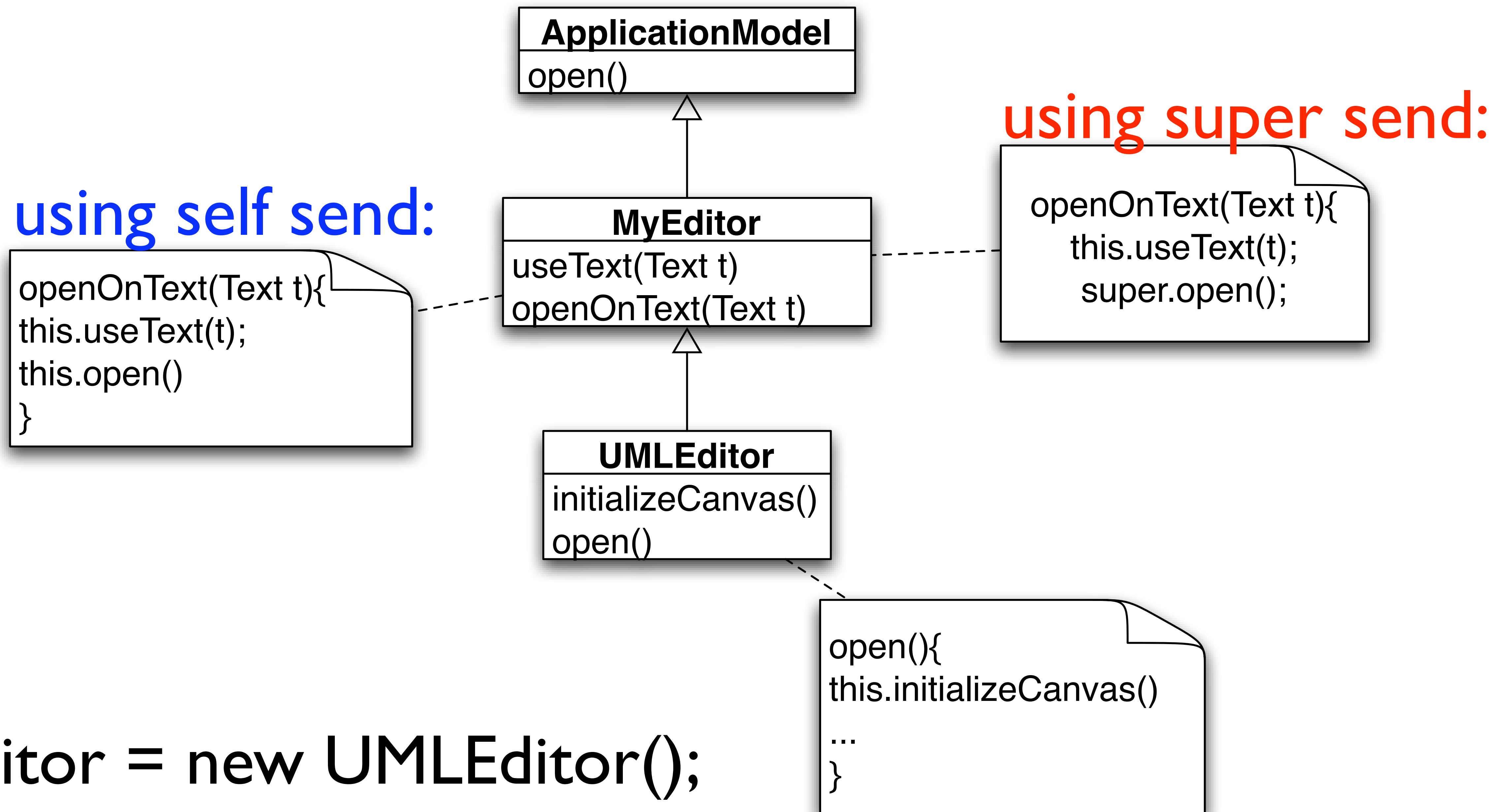
Put this one above your bed and use it each time you want to create a subclass:

When class B inherits from class A:

B "is-a" A

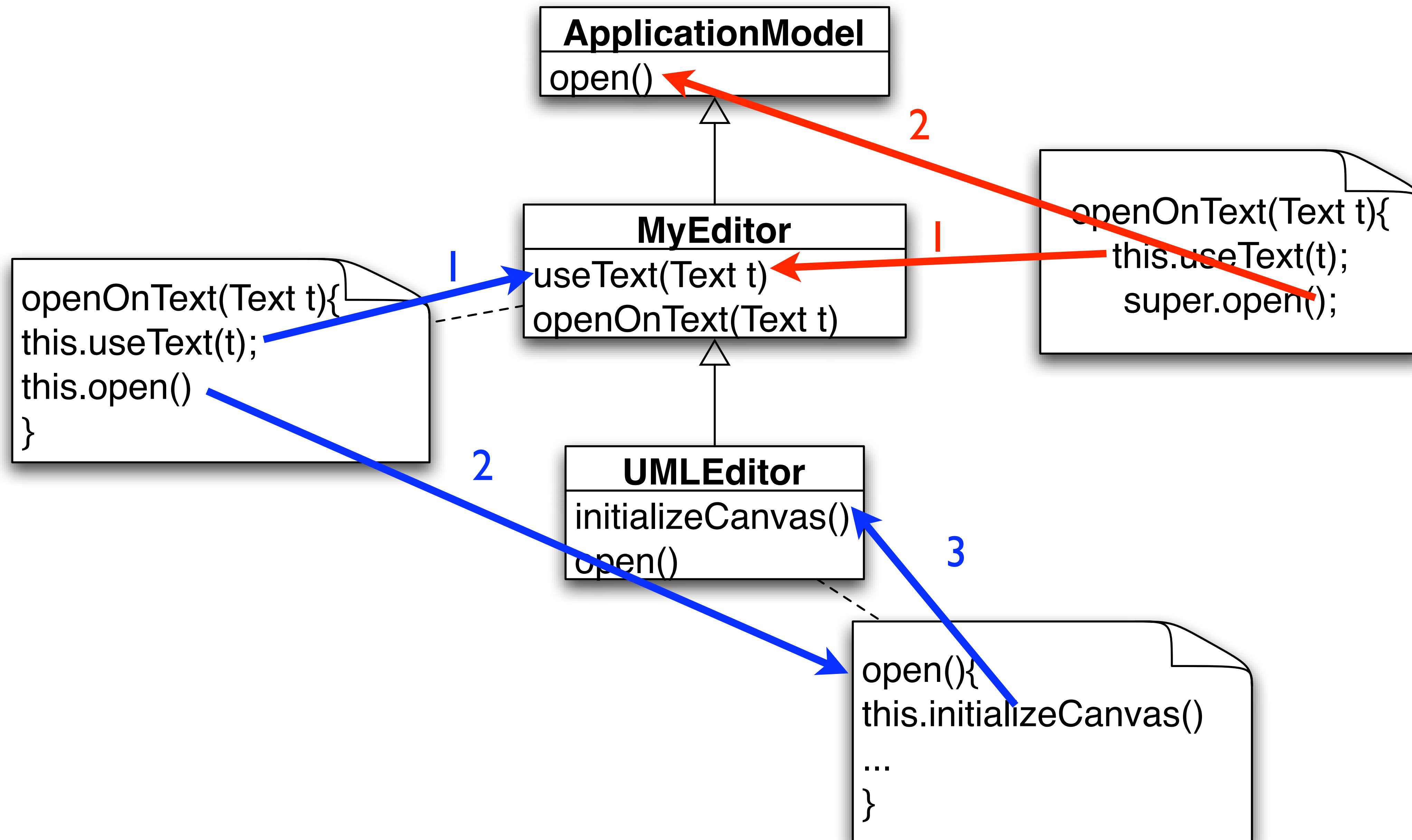
A subclass is a more specialized version of the superclass

this versus super




```
editor = new UMLEditor();
editor.openOnText(t);
```

this versus super



editor = new UMLEditor();
editor.openOnText(t);

Discussion

- Using **super** in the previous example:
  - `MyEditor` can no longer specialize open.
  - Subclasses of `MyEditor` can no longer specialize open.
- Advice:
  - Use self sends to access one's own behaviour.
  - Use **super** to access overridden methods.

```
public class MyClass{
  public void myMethod(T1 arg1, T2 arg2){
  ...
  super.myMethod(..., ...)
  }
}
```

=

Roadmap

Let’s see a number of techniques/concepts to apply when you implement software

- when you gain some experience you will also start to apply them up-front at design level

- Standards
- Law of Demeter
- Coupling and Cohesion

Coding standards

Coding Standards

- improve communication

- Naming standards
- Formatting standards

Naming Standards

- Names should mean something
- Can introduce standard naming conventions

- What are good names to use for methods, classes and variables??
  - yes, in some sense it's kind of boring...
  - yet extremely important!

Naming Methods

- Name of a method:
  - specifies what a method does, not how
- Example:

```
myDoor.open()
```

vs

```
myDoor.putPressureOnHandleThenPullW
ithRotation()
```

- Consistent use of capitals
  - best lowercase

Example 1

```
class ParagraphEditor{
 public void highlight(Rectangle
                    aRectangle){
  aRectangle.reverse()
  ...}
...}
```

- Clients do not know how that highlighting is the same as reversing; they just want to highlight a particular selection
  - What vs. How!

Example 2

```
public class Array {
  public Element linearSearchFor(Element anElement){
   ...
}}

public class Set{
 public Element hashedSearchFor(Element anElement){
  ...
}}
```

- Not a good idea for various reasons:
  - named after how the search is implemented
  - not polymorphic
- Solution

```
public Element includes(Element anElement){...}
```

Example 3

```
public class MyClass{

 public void setType(Value aValue){
   “compute and store the variable type”
...
}
```

- Not precise, not good.

```
public class MyClass{

 public void computeAndStoreType(Value aValue){
   “compute and store the variable type”
...
}
```

- More precise, gives idea about functionality
- Even better: `computeAndStoreTypeFrom`

Standard Method Names

- If there is already a standard name for what you are implementing, use that name:
  - polymorphism,
  - ease of understanding.

- Let’s have a look at:
  - methods that change the state of the receiver,
  - methods that change the state of an argument,
  - methods that return the value of a receiver.

Change State of Receiver

- Method name is verb phrase:
  - `translateBy(amount)`
  - `add(anElement)`

Change State of Argument

- Verb phrase ending with proposition like on or to:
  - `displayOn(aCanvas)`
  - `printOn(aStream)`

Return Value from Receiver

- Method name is noun phrase or adjective, a description rather than a command:
  - `translatedBy(amount)`
  - `size`
  - `topLeft`

More Standard Method Names

- There are other special-purpose methods that follow specific names:
  - accessing methods,
  - testing methods,
  - convertor methods.

Accessing Methods

- Accessing methods: methods for reading and writing into instance variables
  - typically come in pairs
- In Java, C++: prefix with either *get* or *set* :
  - `Player TicTacToe::getWinner()`
  - `TicTacToe::setWinner(Player p)`
- In Smalltalk: name of variable, with and without :   :
  - TicTacToe>>winner
  - TicTacToe>>winner: aPlayer

Testing Methods

- Method that returns a *boolean*.
- Prefix testing method with *is*.
  (or with other forms of *be* or *has*, e.g.: *is*, *was*, *will*, *has*)

```
isNil
isControlWanted
isEmpty
hasBorder
```

Converting Methods

- Often you want to return the receiver in a different format
- Prepend *as* to the name of the convertor method, followed by the type of the return:

```
asSet
asFloat
asComposedText
```

Naming Classes

- Start classnames with capitals.
- Use simple names that convey meaning.
  - Use one or two words.
  - Capitalize between words.

```
Number
Collection
VisualCollection
GregorianCalendar
```

Qualified Subclass Names

- If you create a subclass (e.g. specialized version of superclass) you can prepend an adjective to the name of the most important superclass.

  `OrderedCollection`
  (subclass of collection in which elements are ordered)

  `CloneFigureCommand, CompositeCommand, ConnectionCommand`
  (subclasses of Command, that implement more specific commands)

Standard Class Names

- If there is already a standard for what you are implementing use (part of) that name
- Let’s have a look at:
  - Abstract classes
  - Design Pattern classes

Abstract Classes

- Prefix abstract classes with Abstract

```
AbstractCollection
AbstractVisualComponent
```

Design Pattern Names

- Can use (part of) names of classes that play a role in a design pattern.
- For example, the Visitor design pattern talks about Visitor classes. If you implement a visitor pattern, you might use Visitor in the name of the class.
  - E.g. `NodeVisitor`
- For other ones:
  - `NodeSingleton, NodeFactory, ...`
- But not always necessary (prefer good domain name)

Formatting Standards

- Several standards exist.
  - Depend most on taste.
- Just pick one
  - and apply it *consistently*!!
- If you use a development environment that provides an option to reformat your code, use it!
  - e.g. Source -> Format in Eclipse

Coupling and Cohesion

Coupling and Cohesion

- *Cohesion* of a single module/component is the degree to which its responsibilities form a meaningful unit.
  - The higher the better.
- *Coupling* between modules/components is their degree of mutual interdependence.
  - The weaker the better.

Cohesion and Coupling Heuristics

- It is good practice:
  - For two classes to either be not dependent on one another, or for one class to be only dependent on the interface of another class.
  - To keep attributes and the related methods together in one class.
  - For a class to capture one and only one abstraction - unrelated information to be kept in separate classes.
  - To distribute the system intelligence as uniformly as possible

Kinds of Class Coupling

- X inherits from Y.
- X has an attribute of class Y.
- X has a template attribute with a parameter of class Y.
- X has a method with an argument of class Y.
- X knows of a global variable of class Y.
- X knows of a method containing a local variable of class Y.
- X is friend of Y (in C++).

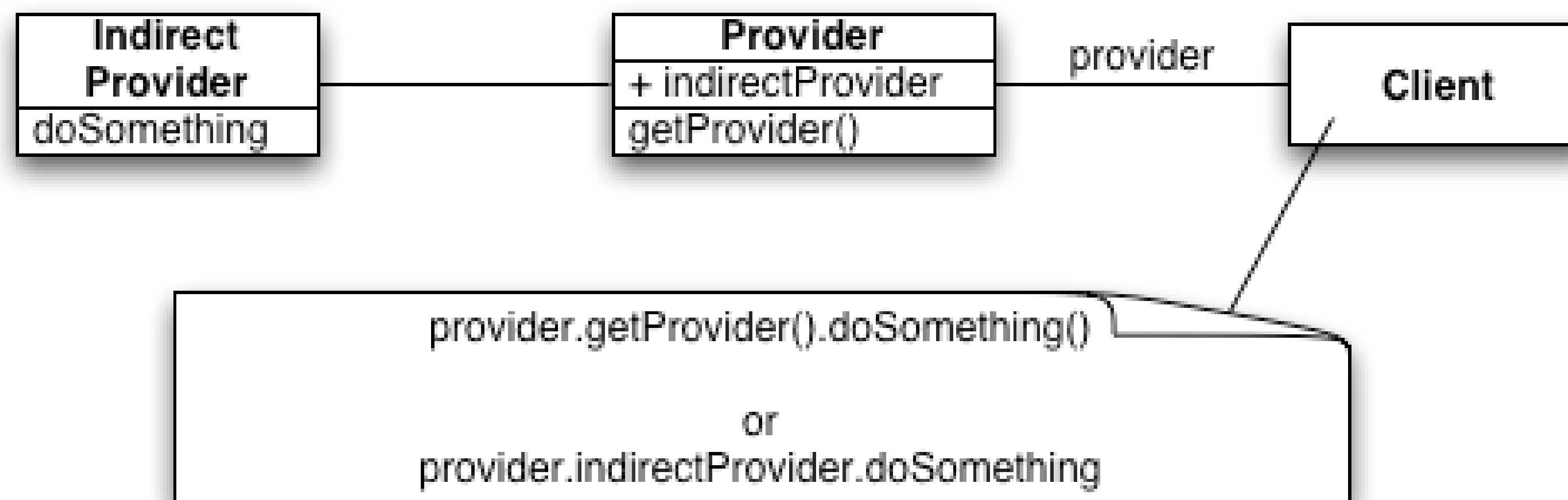
Trade-off

- Need to find the right balance between coupling and cohesion.
- Example: Making a subclass increases coupling (bad), but increases cohesion (good, when done right).
  - So adding tons of classes each overriding a single method might not be a good idea, even if the subclass semantics is right.
- Plays well together with encapsulation.

Law of Demeter

Law of Demeter

- The core of the problem illustrated:



- Client knows how Provider is implemented.
  - knows that it uses an IndirectProvider,
  - so Client and IndirectProvider are strongly coupled!

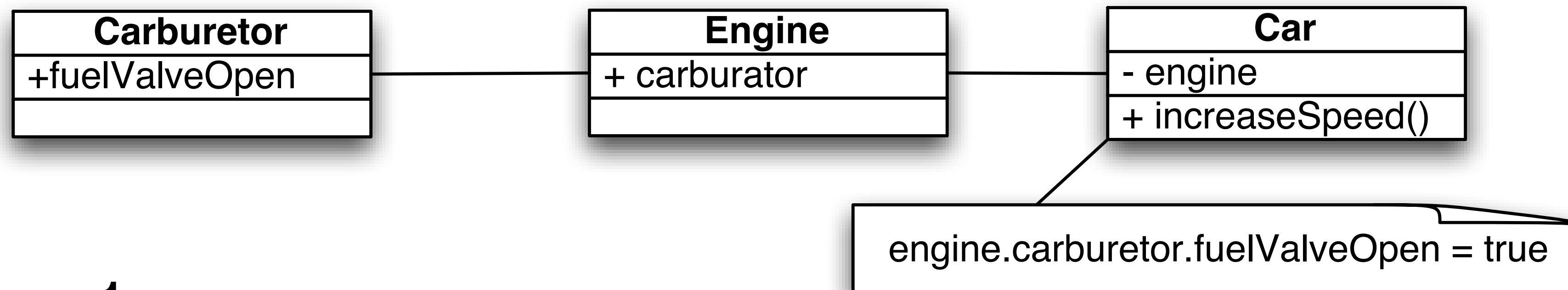
The Law of Demeter

- You are only allowed to send messages to:
  - an argument passed to you,
  - an object you create,
  - self, super.
- Avoid global variables!
- Avoid objects returned from message sends other than self!

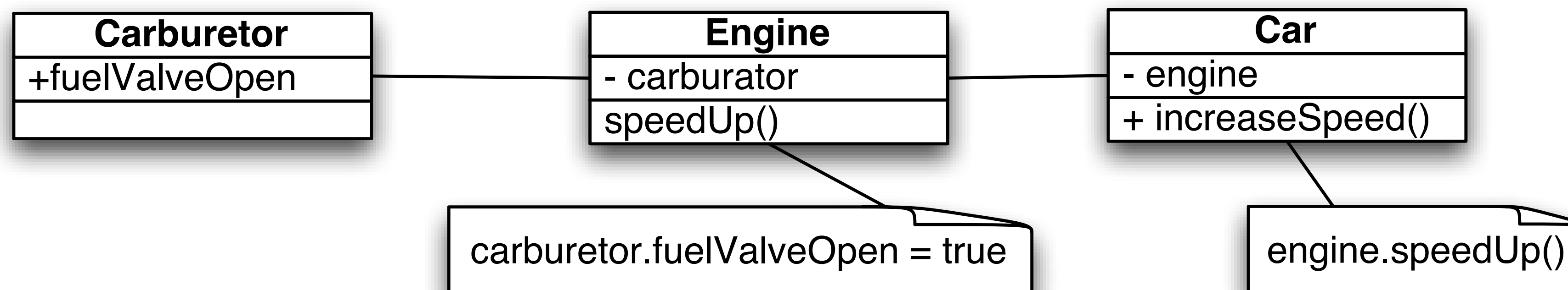
Examples of Correct Messages

```
public class MyClass{

  private Parameter instVarOne;

  public void someMethod(aParameter){
      Thing thing;

      this.foo;
      super.someMethod(aParameter);
      instVarOne.foo();
      aParameter.foo();
      thing = new Thing();
      thing.foo();
}
}
```

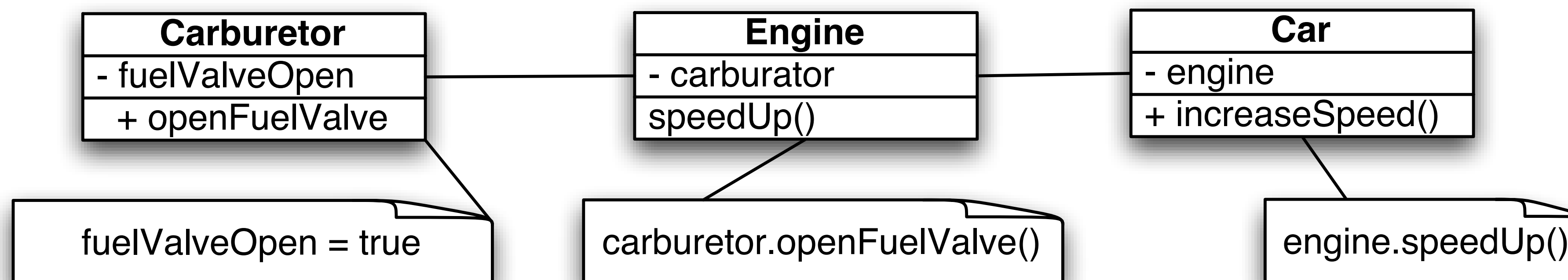
Getting Rid of Code breaking Demeter



Step 1



Step 2



Good OO Method Design

About Methods...

- Disadvantages of many methods:
  - methods take time to look up and call,
  - flow of control can be difficult to follow.
- On the other hand:
  - reading is much improved,
  - performance tuning is simpler (caches, ...),
  - easier to maintain,
  - more reuse possibilities (granularity is method).

Short Methods

- Methods should:
  - perform only one identifiable task
  - be short
  - only contain comment when necessary
    - let the implementation be the comment!

Motivating Short Methods

- Suppose we have a class with 2 big methods
  - subclasses inherit these two methods,
  - so they can only redefine 2 methods,
  - very coarse-grain reuse.
- Suppose the same class has 20 small methods
  - subclasses inherit 20 small methods
  - so they can redefine on a much finer level of granularity
- Of course: methods should make sense...

Finding Big Methods

- Tools based on metrics
  - Calculate LOC metric (Lines Of Code)
- In Eclipse: integrated in a plug-in Metrics
  - Go to http://metrics.sourceforge.net/
  - install plug-in,
  - run MLOC, etc.
  - You know that I will...

Good OO Class Design

Responsibility

- We need to make sure that each class has the right behaviour
  - implements good methods
    - short, with good names, readable, usable
  - has the right responsibilities
    - not too many, not too few
- Let's have a look at a couple of good practices that, when applied together, will help you to create classes with the right responsibilities

Choosing Alternatives

- Case (switch) statements in OO code are a sign of a bad design:
  - lack of polymorphism: procedural way to implement a choice between alternatives.

Example

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
    {
    switch (aCommand){
       case EAknSoftkeyExit:
        case EAknSoftkeyBack:
        case EEikCmdExit:{Exit();  break;}
          // Play command is selected
       case EVideoCmdAppPlay:{DoPlayL(); break; }
         // Stop command is selected
       case EVideoCmdAppStop:{DoStopL(); break; }
         // Pause command is selected
       case EVideoCmdAppPause:{DoPauseL(); break; }
        // DocPlay command is selected
       case EVideoCmdAppDocPlay:{DoDocPlayL(); break; }
         // File info command is selected
       case EVideoCmdAppDocFileInfo:{DoGetFileInfoL();
               break; }
......
```

Replace case statement by polymorphism

REFACTORING

Methods Returning Multiple Values

- Methods can only have a single return value
  - ever felt the need to return multiple values?
- Example

```
public class Text{
    public ?? fontAndColourOfSelection(Position
                                       position)
  “ Want to return both font and colour!
    How to do this ?! “
```

Different solutions:
- Multiple methods
- Temporary values
- New class

REFACTORING

Duplicated Code

- Makes the system harder to understand and to maintain
  - One of the real big problems in maintenance is duplicated code
  - Typically hard to find afterwards
- Occurs a lot
  - Range of code duplication: roughly 10% to 25% !
  - 19% in X Window System
  - 68% of Java Buffer Library (JDK 1.4.1)

Problems with Duplication

- Errors get spread
  - fixes do not...
- Evolution of code is not reflected everywhere
  - some places are forgotten and do not get updated
- Code bloat: code gets much bigger
  - since no sharing

Where can we find Duplication?

- In the same class:
  - several methods that repeat a number of instructions
- Between siblings:
  - two classes that share a common superclass
  - methods in siblings can repeat a number of instructions
- Between unrelated classes:
  - classes not in a hierarchy can still repeat the same sets of instructions

Guardian Code

- It happens regularly that the body of a method should only be executed when a certain condition is met
  - typically nill checks for arguments, etc.
- Schematically the method typically looks like this:

```
MyMethod(...){
  ....
      if(guardian condition){
          ...
       }
....
}
```

Guardian Code Example

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId == R_VIDEO_MENU )
          {
          // Check whether the database has been created or not
          if ( iEngine->GetEngineState() != EPPlaying )
              {
              // The video clip is not being played
              aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
              aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
              }

          // If there is no item in the list box, hide the play, docplay
          // and file info menu items
          if ( !iAppContainer->GetNumOfItemsInListBox() )
              {
              aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
              aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
              aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
              }
        }
}
```

Guardian statement

http://www.forum.nokia.com/info/sw.nokia.com/id/ee56cba2-0b78-45c9-831f-69c5007652fe/Video_Example_v1_0.zip.html

Replace case statement by polymorphism

REFACTORING

Conclusion

- Make sure your classes have the right behaviour:
  - apply Law of Demeter, have short methods, use clear names, apply consistent formatting, distribute behaviour, no case statements, ...
- Refactor until code looks nice
  - understandable at a single glance,
  - tested.

References

- Kent Beck. Smalltalk Best Practice Patterns. Prentice Hall. 1996. ISBN 013476904X
- http://sis36.berkeley.edu/projects/streek/agile/bad-smells-in-code.html
- Joshua Bloch. Effective Java. Addison-Wesley. 2001. ISBN 0201310058 Sample Chapters @ http://java.sun.com/docs/books/effective/chapters.html
- Scott Meyers. Effective C++, Third Edition. Addison-Wesley. 2005
- Scott Meyers. More Effective C++. Addison-Wesley. 1996.