UNIVERSITÄT
BERN

8. Refactoring and Design Patterns

Birds-eye view

Beware of misplaced responsibilities
— cluttered code impacts extensibility.

Roadmap

- Some Principles
- What is Refactoring?
- The Law of Demeter
- Common Code Smells
- Design Patterns
- The Singleton Pattern

Literature

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
- Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Mass., 1995.
- Sherman R. Alpert, Kyle Brown and Bobby Woolf, *The Design Patterns Smalltalk Companion*, Addison Wesley, 1998.

Roadmap

- **Some Principles**
- What is Refactoring?
- The Law of Demeter
- Common Code Smells
- Design Patterns
- The Singleton Pattern

The Open-Closed Principle

- Software entities should be *open for extension* but *closed for modifications*.
  - Design classes and packages so their functionality can be extended without modifying the source code

The Object Manifesto

- Delegation and encapsulation:
  - "Don't do anything you can push off to someone else."
  - "Don't let anyone else play with you."

– *Joseph Pelrine*

The Programmer Manifesto

- Once and only once
  - Avoiding writing the same code more than once

- Don't ask, tell!

```
MyWindow>>displayObject: aGrObject
   aGrObject isSquare ifTrue: […]
   aGrObject isCircle ifTrue: […]
   …
```

```
MyWindow>>displayObject: aGrObject
   aGrObject displayOn: self
```

Good Signs of OO Thinking

- Short methods
  - Simple method logic
- Few instance variables
- Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects
  - No manager objects

Some Principles

- **The Dependency Inversion Principle**
  - Depend on abstractions, not concrete implementations
    - *Write to an interface, not a class*
- **The Interface Segregation Principle**
  - Many small interfaces are better than one “fat” one
- **The Acyclic Dependencies Principle**
  - Dependencies between package must not form cycles.
    - *Break cycles by forming new packages*

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF

Packages, Modules and other

- **The Common Closure Principle**
  - Classes that change together, belong together
    - *Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.*

- **The Common Reuse Principle**
  - Classes that aren't reused together don't belong together
    - *The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.*

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF

Roadmap

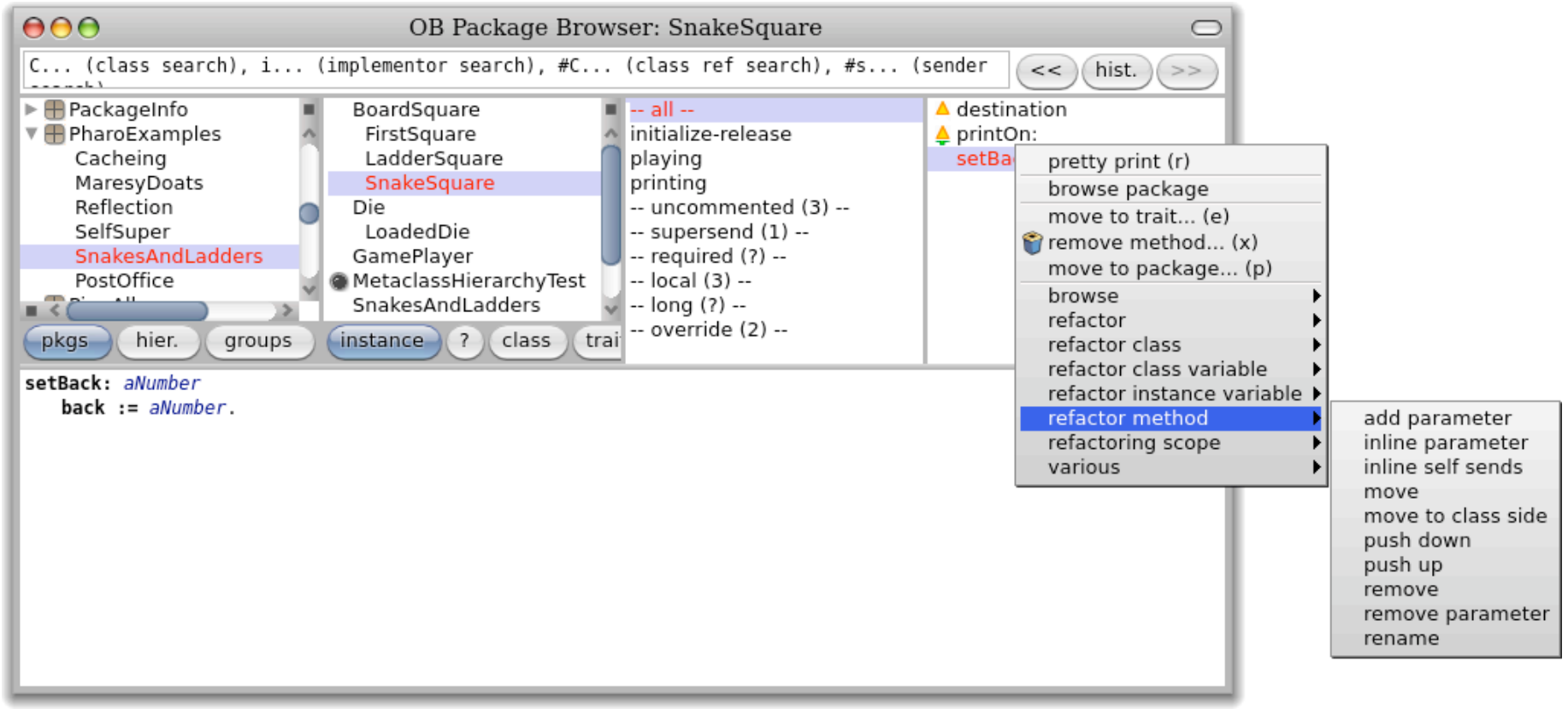
- Some Principles
- **What is Refactoring?**
- The Law of Demeter
- Common Code Smells
- Design Patterns
- The Singleton Pattern

What is Refactoring?

> The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*.

— Fowler, et al., Refactoring, 1999.

Refactoring in the Package Browser



Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Why Refactor?

"Grow, don't build software"

— Fred Brooks

- The reality:
  - Extremely difficult to get the design "right" the first time
  - Hard to fully understand the problem domain
  - Hard to understand user requirements, even if the user does!
  - Hard to know how the system will evolve in five years
  - Original design is often inadequate
  - System becomes brittle over time, and more difficult to change

- Refactoring helps you to
  - Manipulate code in a safe environment (behavior preserving)
  - Recreate a situation where evolution is possible
  - Understand existing code

Rename Method — manual steps

- Do it yourself approach:
  - Check that no method with the new name already exists in any subclass or superclass.
  - Browse all the implementers (method definitions)
  - Browse all the senders (method invocations)
  - Edit and rename all implementers
  - Edit and rename all senders
  - Remove all implementers
  - Test
- Automated refactoring is better !

Rename Method

- Rename Method (method, new name)
- Preconditions
  - No method with the new name already exists in any subclass or superclass.
  - No methods with same signature as method outside the inheritance hierarchy of method
- PostConditions
  - method has new name
  - relevant methods in the inheritance hierarchy have new name
  - invocations of changed method are updated to new name
- Other Considerations
  - Typed/Dynamically Typed Languages => Scope of the renaming

Roadmap

- Some Principles
- What is Refactoring?
- **The Law of Demeter**
- Common Code Smells
- Design Patterns
- The Singleton Pattern

The Law of Demeter

- ***"Do not talk to strangers"***
  - You should only send messages to:
    - *an argument passed to you*
    - *an object you create*
    - *self, super*
    - *your class*

- Don't send messages to objects returned from other message sends

en.wikipedia.org/wiki/Law_Of_Demeter

Law of Demeter by Example

```
NodeManager>>declareNewNode: aNode
   |nodeDescription|

   (aNode isValid) "OK – passed as an argument to me"
      ifTrue: [ aNode certified].

   nodeDescription := NodeDescription for: aNode.
   nodeDescription localTime.  "OK – I created it"

   self addNodeDescription: nodeDescription.
                           "OK – I can talk to myself"

   nodeDescription data      "Wrong! I should not know"
      at: self creatorKey    "that data is a dictionary"
      put: self creator
```

The Dark Side of the Law of Demeter

To avoid giving direct access to instance variables, you may need to introduce many delegating methods …

Traits can help — see final lecture

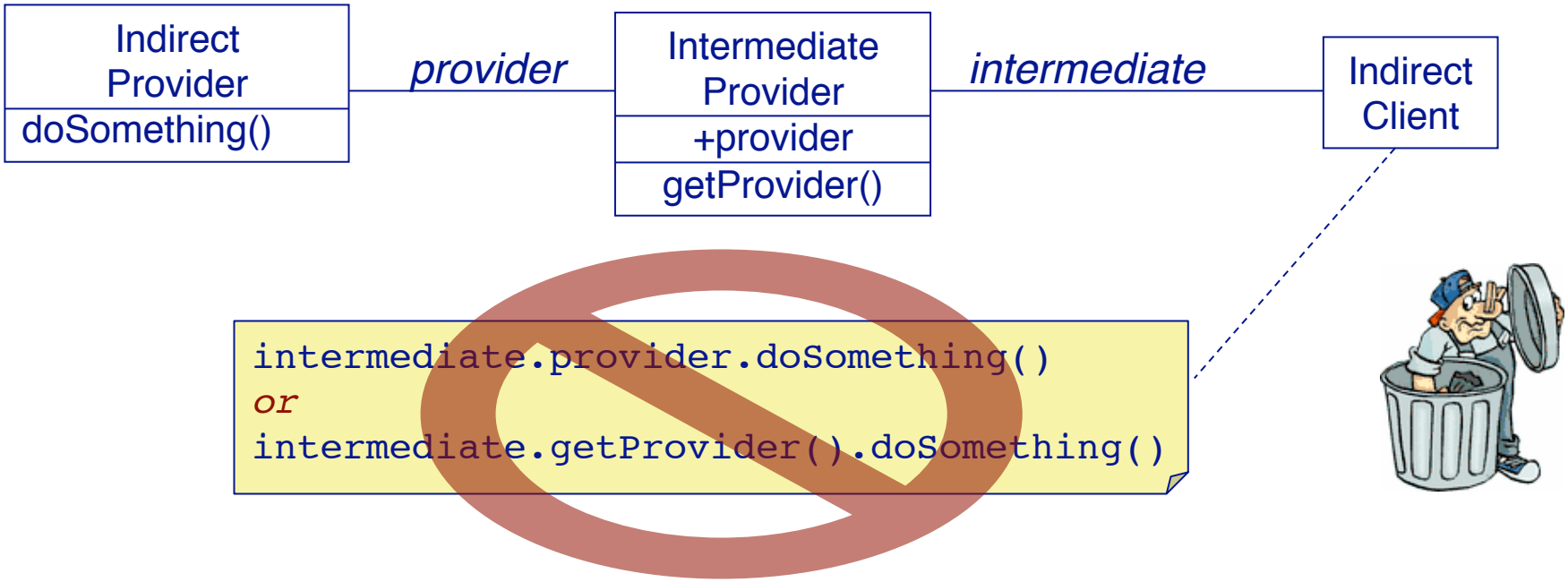
```
Class A
   instVar: myCollection

A>>do: aBlock
   myCollection do: aBlock

A>>collect: aBlock
   ^ myCollection collect: aBlock

A>>select: aBlock
   ^ myCollection select: aBlock

A>>detect: aBlock
   ^ myCollection detect: aBlock

A>>isEmpty
   ^ myCollection isEmpty
```

Curing Navigation Code

- Iteratively move behaviour close to data
  - Use Extract Method and Move Method

See: *Object-Oriented Reengineering Patterns*

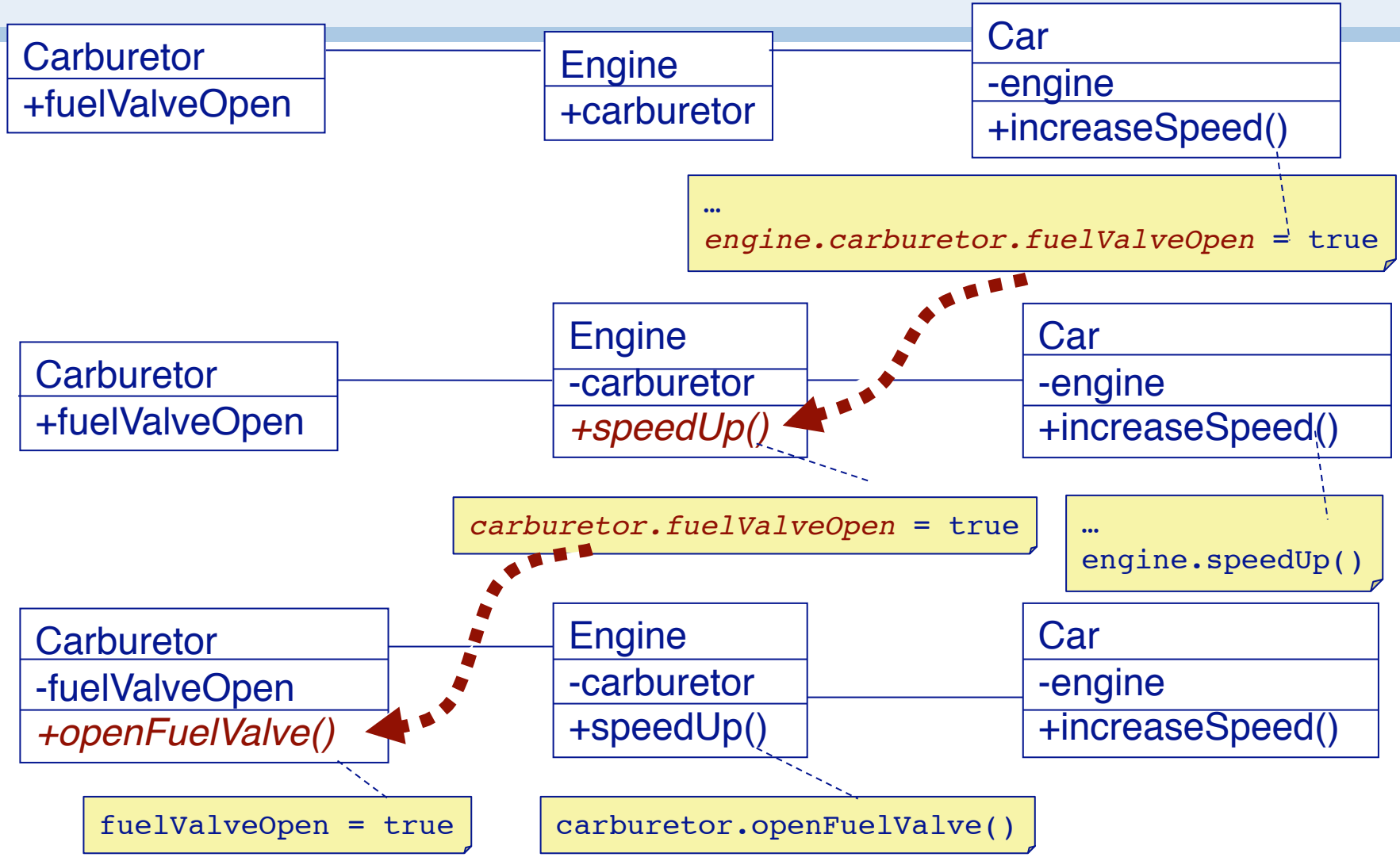
The Law of Demeter, violated



Law of Demeter: A method "M" of an object "O" should invoke only the methods of the following kinds of objects.

1. *itself*
2. *its parameters*
3. *any object it creates /instantiates*
4. *its direct component objects*

Eliminate Navigation Code



Roadmap

- Some Principles
- What is Refactoring?
- The Law of Demeter
- **Common Code Smells**
- Design Patterns
- The Singleton Pattern

Misplaced Methods

- Avoid implementing a method which neither accesses instance variables nor accesses any state
  - Probably belongs in one of the classes of the objects it does send to

```
MyClass>>pop: anOrderedCollection
   anOrderedCollection removeFirst.
```

Klimas, et al., *Smalltalk with Style*

Code Smells

"If it stinks, change it"

— *Grandma Beck*

- Duplicated Code
  - *Missing inheritance or delegation*
- Long Method
  - *Inadequate decomposition*
- Large Class / God Class
  - *Too many responsibilities*
- Long Parameter List
  - *Object is missing*
- Type Tests
  - *Missing polymorphism*
- Shotgun Surgery
  - *Small changes affect too many objects*

Code Smells

- Feature Envy
  - *Method needing too much information from another object*
- Data Clumps
  - *Data always used together (x,y -> point)*
- Parallel Inheritance Hierarchies
  - *Changes in one hierarchy require change in another hierarchy*
- Lazy Class
  - *Does too little*
- Middle Man
  - *Class with too many delegating methods*
- Temporary Field
  - *Attributes only used partially under certain circumstances*
- Data Classes
  - *Only accessors*

Curing Long Methods

- Long methods
  - Decompose into smaller methods
  - Self sends should *read like a script*
  - Comments are good delimiters
  - A method is the *smallest unit of overriding*

```
self setUp; run; tearDown.
```

Curing Duplicated Code

- In the same class
  - Extract Method
- Between two sibling subclasses
  - Extract Method
  - Push identical methods up to common superclass
  - Form Template Method
- Between unrelated class
  - Create common superclass
  - Move to Component
  - Extract Component (e.g., Strategy)

Curing God Class

- God Class
  - Incrementally redistribute responsibilities to existing (or extracted) collaborating classes
  - Find logical sub-components
    - *Set of related working methods/instance variables*
  - Move methods and instance variables into components
  - Extract component
  - Extract Subclass
    - *If not using all the instance variables*

See: *Object-Oriented Reengineering Patterns*

Curing Type Tests

- Missing Polymorphism
  - Tell, don't ask!
  - Shift case bodies to (new) methods of object being tested
  - Self type checks:
    - *Introduce hook methods and new subclasses*
  - Client type checks
    - *Introduce "tell" method into client hierarchy*
  - Possibly introduce State / Strategy or Null Object Design Patterns

See: *Object-Oriented Reengineering Patterns*

Transformation

```
switch (a.class)
case B: a.init(); ((B) a).x();
case C: a.init(); ((C)) a).y();
Case D: ((D) a).z()
```

Roadmap

- Some Principles
- What is Refactoring?
- The Law of Demeter
- Common Code Smells
- **Design Patterns**
- The Singleton Pattern

Design Patterns

- Design Patterns document *recurrent solutions* to *design problems*
  - They have *names*
    - *Composite, Visitor, Observer...*
  - They are not components!
  - Design Patterns entail *tradeoffs*
  - Will be implemented in different ways in different contexts

Why Design Patterns?

- Smart
  - Elegant solutions that a novice would not think of
- Generic
  - Independent of specific system type, language
- Well-proven
  - Successfully tested in several systems
- Simple
  - Combine them for more complex solutions

- ***Caveat***
  - Not everything that is called a “pattern” fulfils these criteria!

Alert!!! Patterns are invading!

- Design Patterns are not “good” just because they are patterns
  - It is just as important to understand when *not* to use a Design Pattern
  - Every Design Pattern has tradeoffs
  - Most Design Patterns will make your design *more complicated*
    - *More classes, more indirections, more messages*
  - Don’t use Design Patterns unless you really need them!

About Pattern Implementation

- Do not confuse *structure* and *intent!*
  - Design Patterns document a *possible* implementation
    - *Not a definitive one*
  - Design Patterns are about *intent* and *tradeoffs*

Common Design Patterns

Pattern	***Intent***
Adapter	Convert the interface of a class into another interface clients expect.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Composite	Compose objects into part-whole hierarchies so that clients can treat individual objects and compositions uniformly.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps so they can be redefined by subclasses.

What tradeoffs do these patterns introduce?

Roadmap

- Some Principles
- What is Refactoring?
- The Law of Demeter
- Common Code Smells
- Design Patterns
- **The Singleton Pattern**

The Singleton Pattern

Intent:

> Ensure that a class has only one instance, and provide a global point of access to it

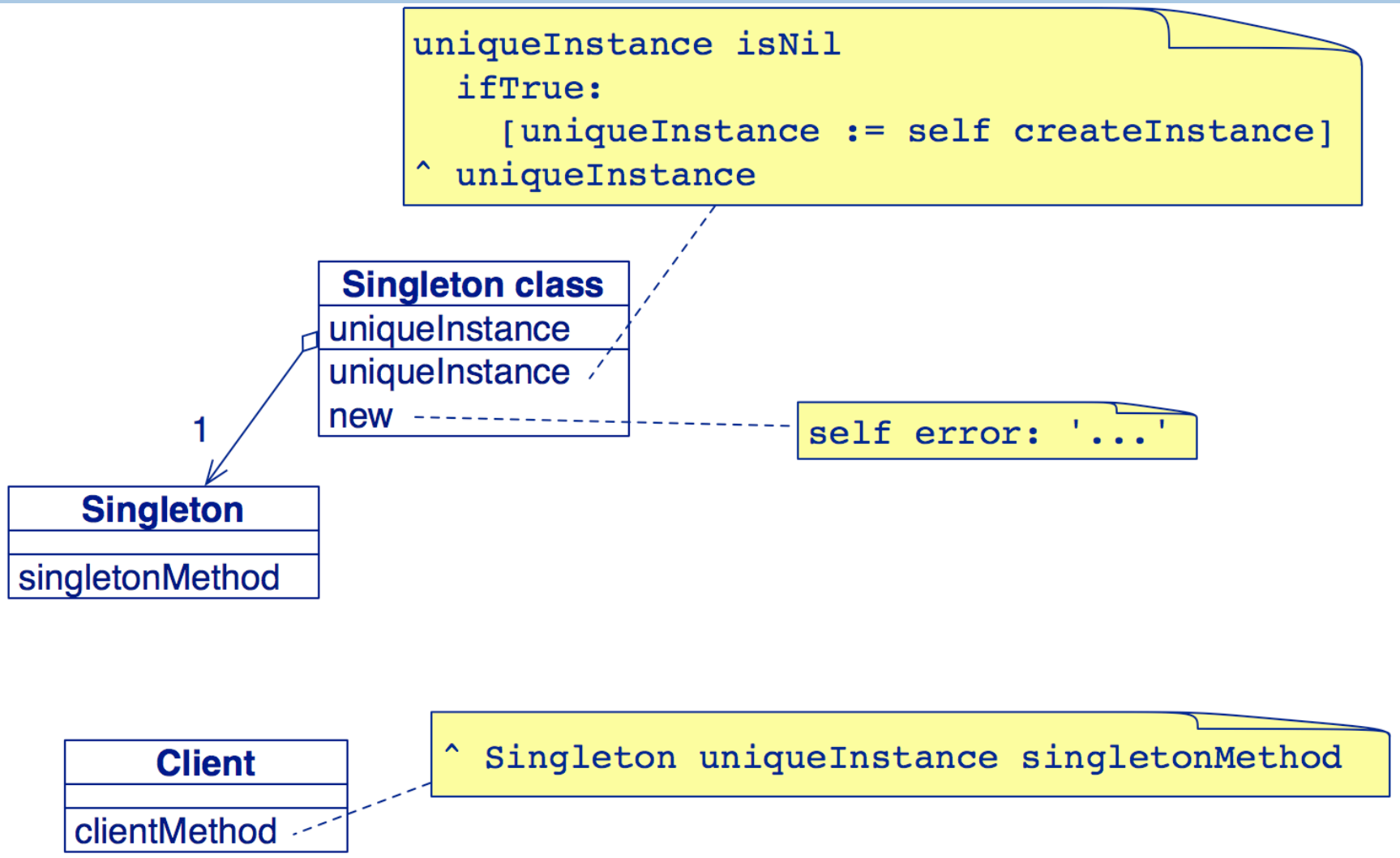
Problem:

> We want a class with a unique instance.

Solution:

> Give the class the responsibility to initialize and provide access to the unique instance. Forbid creation of new instances.

Singleton Structure



Singleton Example: SystemChangeNotifier

```
Object subclass: #SystemChangeNotifier
   instanceVariableNames: 'eventSource silenceLevel'
   classVariableNames: 'UniqueInstance'
   poolDictionaries: ''
   category: 'System-Change Notification'

SystemChangeNotifier class
   instanceVariableNames: ''

SystemChangeNotifier class>>new
   ^self error: self instanceCreationErrorString

SystemChangeNotifier class>>uniqueInstance
   UniqueInstance ifNil: [UniqueInstance := self createInstance].
   ^UniqueInstance
```

NB: This example uses a class variable rather than a class instance variable.

Singleton Example:

```
ChangeSet class
   instanceVariableNames: 'current'

ChangeSet class>>new
   ^ self basicNewChangeSet: ChangeSet defaultName

ChangeSet class>>newChanges: aChangeSet
   SystemChangeNotifier uniqueInstance noMoreNotificationsFor: current.
   current isolationSet: nil.
   current := aChangeSet.
   …

ChangeSet class>>current
   ^ current
```

Here we have many instances, but only a Singleton is active.
A class instance variable is used. `new` is not forbidden.

Implementation Issues

- Class variable
  - One singleton for a complete hierarchy
- Class instance variable
  - One singleton per class

Implementation Issues

- Singletons may be accessed via a global variable
  - E.g., NotificationManager uniqueInstance notifier

```
SessionModel>>startupWindowSystem
    Notifier initializeWindowHandles.
    ...
    oldWindows := Notifier windows.
    Notifier initialize.
    ...
    ^oldWindows
```

- Global Variable vs. Class Method Access
  - Global Variable Access is dangerous: if we reassign Notifier we lose all references to the current window.
  - Class Method Access is better because it provides a single access point.

Implementation Issues

- Persistent Singleton:
  - only one instance exists and its identity does not change
    - *E.g., notification manager*
- Transient Singleton:
  - only one instance exists at any time, but that instance changes
    - *E.g., current session*
- Single Active Instance Singleton:
  - a single instance is active at any point in time, but other dormant instances may also exist.
    - *E.g., active project*

Singleton is about time, not access

Access using new — not a good idea

- The intent (uniqueness) is not clear anymore!
  - `new` is normally used to return *newly created instances*. The programmer does not expect this:

```
Singleton class>>new
    ^self uniqueInstance
```

```
|screen1 screen2|
screen1 := Screen new.
screen2 := Screen uniqueInstance
```

What you should know!

- *How does the Open-Closed Principle apply to OOP?*
- *What are signs that an object has clearly-defined responsibilities?*
- *How can you recognize misplaced methods?*
- *How should you refactor long methods?*
- *How can you eliminate duplicated code between unrelated classes?*
- *Why are type tests a code smell?*
- *When do design patterns themselves turn into code smells?*
- *Why is it a bad idea to use global variables to store Singleton instances?*

Can you answer these questions?

- *How do the Common Closure and Common Reuse Principles alter the usual notion of cohesion?*
- *How does refactoring differ from reengineering?*
- *Can refactoring be fully automated?*
- *In what situations does the Law of Demeter not apply?*
- *How do design patterns make use of delegation?*
- *Why are Long Parameter Lists a code smell?*
- *Are `isNil` tests a code smell? What design pattern could help you eliminate them?*
- *Is the Smalltalk SystemDictionary a good example of a Singleton?*

License

http://creativecommons.org/licenses/by-sa/3.0/