(Object-Oriented)

DESIGN PRINCIPLES, BAD SMELLS AND REFACTORING

Material available at https://combemale.github.io
Version Nov., 2017

BENOIT COMBEMALE
PROFESSOR, UNIV. TOULOUSE, FRANCE

HTTP://COMBEMALE.FR
BENOIT.COMBEMALE@IRIT.FR
@BCOMBEMALE

Objectif du cours

- Connaitre les principes fondamentaux de la conception Objet

- Prendre conscience de l'intérêt de la conception Objet (mais aussi de sa difficulté) par rapport à n’importe quelle technologie de mise en oeuvre

- Savoir faire les bons choix de conception et comprendre leurs intérêts

SOLID: Principles of Class Design

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
  - ➡ a.k.a. Design by Contract
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single-Responsibility Principle (SRP)

- « A class should have one, and only one, reason to change »
- « There should never be more than one reason for a class to change »

- Principe qui semble facile et plein de bon sens
  - Pas si simple dans la vraie vie
  - Compromis à faire avec la complexité, les répétitions et l'opacité

Open-Closed Principle (OCP)

- « Software entities should be open for extension, but closed for modification », *B. Meyer (1988), quoted by R. Martin (1996)*
- « You should be able to extend a class behavior, without modifying it »

- Un code doit être "ouvert à l'extension",
  - l'évolution du logiciel doit se faire de façon incrémentale
- mais "fermé à la modification".
  - sans modifier une ligne de source existante

- Une approche de ce principe se fait par les design patterns "template method" et "strategy"

Heuristiques pour l'OCP

- Mettre toutes les données d'un objet privées
- Pas de variables globales!
- Une modification d'une donnée publique est toujours un risque d'ouvrir un module
  - Il peut y avoir un effet de bord;
  - Les erreurs peuvent être complexes à trouver et fixer
  - Les patchs peuvent créer des erreurs ailleurs
- RTTI (Run-Time Type Information) est une mauvaise pratique dangereuse

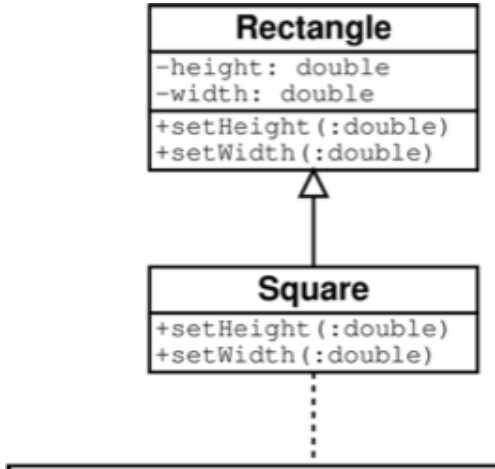
Liskov Substitution Principle (LSP)

- « All derived classes must be substitutable for their base classes », Barbara Liskov, 1988

- The "Design-by-Contract" formulation: « All derived classes must honour the contracts of their base classes », Bertrand Meyer

LSP: le problème du rectangle carré

- Les clients (users) d'un Rectangle s'attendent à que la mise à jour de la hauteur n'est pas d'impact sur la largeur (et vice versa)
- Le carré ne respecte pas cette attente
- Les programmes clients peuvent être en erreur




```
void Square::setHeight( double h ) {
  Rectangle::setHeight( h );
  Rectangle::setWidth( h );
}
void Square::setWidth( double w ) {
  setHeight( w );
}
```

LSP: violation de contrat

- Le contrat du rectangle:
  - Hauteur et largeur indépendantes. Il est possible de modifier l'une sans modifier l'autre
- Le carré viole ce contrat
- Les méthodes dérivées ne doivent pas attendre plus et fournir moins que les méthodes de la classe de base
  - Pré conditions ne sont pas plus forte
  - Post conditions ne sont pas plus faibles

LSP: résumé

- Les classes dérivées doivent être pleinement substituables à leurs classes de base
- Guide pour la conception et les choix d'abstraction
- Les bonnes abstractions ne sont pas toujours intuitives
- Violer le principe LSP peut casser le principe d'OCP
  - Besoin de RTTI et utilisation de if/switch
- L'héritage et le polymorphisme sont des outils puissants
  - À utiliser avec attention
- EST-UN est une relation avec une sémantique très particulière en conception objet

Rappel

- Héritage != Sous-typage
- Adaptation != Substitution
- `extends` != `implements`

Dependency Inversion Principle (DIP)

- « Details should depend on abstractions. Abstractions should not depend on details. », *Robert Martin*

- « High level modules should not depend upon low level modules. Both should depend on abstractions », *Robert Martin*

- Why dependency inversion?
  - In OO we have ways to invert the direction of dependencies, i.e. class inheritance and object polymorphism.

Résolution de dépendances

- Ancienne façon
  - Utilisation du ‘new’ pour créer les dépendances
- Registre de service
  - Utilisation d’un registre de service pour obtenir ses dépendances
- Injection de dépendances
  - Les dépendances sont fournies par l’environnement

➡ Chacune de ces solutions implique un certain couplage

Ancienne façon (classique)

```
public class Foo {

  private IBar bar;
  private IBaz baz;

  public Foo() {
     bar = new SomeBar();
     baz = new SomeBaz();
  }
}
```

- Contre
  - Dépendance entre votre classe et ses classes de dépendance
  - Votre classe doit connaître comment assembler les instances de ses dépendances
  - Très difficile de changer le code sans modifier le code source
  - Très difficile de tester quand vous devez utiliser des stubs ou des mocks.
- Pour
  - Facile à comprendre

Registre de service

```
public class Foo {

   private IBar bar;
   private IBaz baz;
   private IServiceLocator locator;

   public Foo(IServiceLocator locator_) {
      locator = locator_;
      bar = locator.Get(
         ServiceNames.BAR
     );
      baz = new SomeBaz(
        ServicesNames.BAZ
     );
   }
}
```

- Contre
  - Votre classe dépend du registre de service
  - Vous devez toujours obtenir le registre de services – soit statiquement ou via … une sorte de mécanisme d'injection de dépendances
- Pour
  - Facile à comprendre
  - Testable
  - Flexible
  - Extensible
  - Force une meilleure séparation entre interfaces et implémentations

Inversion de contrôle

```
public class Foo {
  private IBar bar;
  private IBaz baz;
  public Foo(IBar bar_, IBaz baz_) {
    bar = bar_;
    baz = baz_;
  }
}
```

- Contre
  - –Vous devez créer les dépendances et les passer pour créer votre schéma d'instance
- Pour
  - –Facile à comprendre
  - –Testable
  - –Flexible
  - –Extensible
  - –Force une meilleure séparation entre les interfaces et les implémentations
  - –Code propre, clair et simple à comprendre

Inversion de contrôle

- *Inversion Of Control, Dependency Injection, The Hollywood Principal, etc.*
- Instead of instantiating concrete class references in your class, depend on an abstraction and allow your concrete dependencies to be given to you.

- *A la place d'instancier une classe concrète (implémentation) dans votre classe, il vaut mieux dépendre d'une abstraction (Interface) et permettre que l'implémentation concrète vous soit fournie*

Types d’injection de dépendances

- Injection par *Setter*
  - Passe les dépendances par les setter/modifieurs de propriétés
- Injection par constructeur
  - Passe les dépendances par les constructeurs

Sans IoC

```
public class WithoutIoC
{
    private IDoSomething somethingDoer;

    public WithoutIoC()
    {
        somethingDoer = new SomethingSpecificDoer();
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Dépendances avec les classes concrètes

```
public class WithoutIoC
{
    private IDoSomething somethingDoer;

    public WithoutIoC()
    {
        somethingDoer = new SomethingSpecificDoer();
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Injection par constructeur

```
public class WithIoC
{
    private IDoSomething somethingDoer;

    public WithIoC(IDoSomething somethingDoer)
    {
        this.somethingDoer = somethingDoer;
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Injection par *setter*

```
public class WithIoCPropertySetter
{
    private IDoSomething somethingDoer;

    public IDoSomething SomethingDoer
    {
        set { somethingDoer = value; }
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

DIP: résumé

I. High-level modules should ***not*** depend on low-level modules.
   Both should depend on abstractions.
II. Abstractions should not depend on details.
   Details should depend on abstractions

R. Martin, 1996

- Une classe de base dans une hiérarchie d'héritage ne doit pas connaître ses sous classes

- *OCP définit l'objectif*
- *DIP définit les mécanismes pour atteindre ses objectifs*
- *LSP offre une garantie pour le DIP*

DIP: résumé

- L'inversion de dépendance permet de rendre un code indépendant de ses conditions d'initialisation, et des API précises ses données d'initialisation
- Utile dans les architectures multi couches

- Exemple:
  - Le framework Spring: http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework
  - le projet Pico (http://www.picocontainer.org/)
  - le projet Avalon

Interface Segregation Principle (ISP)

- « Make fine grained interfaces that are client specific », *Robert Martin*
- « Clients should not be forced to depend upon interfaces that they do not use », *Robert Martin*

- De nombreuses interfaces spécifiques aux besoins de la classe cliente sont meilleures qu'une interface aux objectifs généraux
- Les clients ne doivent pas dépendre d'interfaces qu'ils n'utilisent pas

Interface Segregation Principle (ISP)

- Un programme ne doit pas dépendre de méthodes qu'il n'utilise pas
- Principe également lié à la cohésion forte
- Conduit à la multiplication d'interfaces très spécifiques et petites.

Summary

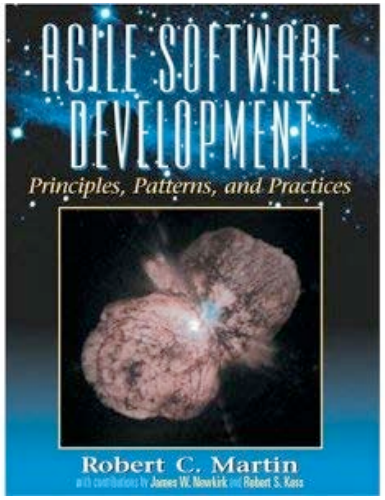
- Single Responsibility Principle (SRP)
  - ➡Une seule raison de changer
- Open-Closed Principle (OCP)
  - ➡Prévoir l'extension sans devoir modifier le code existant
- Liskov Substitution Principle (LSP)
  - ➡Les classes dérivées doivent pleinement se substituer à leur classe de base
- Dependency Inversion Principle (DIP)
  - ➡Dépendance vers les abstractions, pas vers des classes de mises en oeuvre
- Interface Segregation Principle (ISP)
  - ➡Éclater les interfaces pour contrôler les dépendances

References

- Website
  - –http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
  - –*Robert Martin's '10 Commandments':*

    http://groups.google.com/group/comp.object/browse_frm/thread/58808f0dc5c0306f/adee7e5bd99ab111?q=dependency+inversion+group:comp.object&rnum=11&hl=en#adee7e5bd99ab11

- Book:
  - – « *Agile Software Development, Principles, Patterns, and Practices* », *Robert C. Martin, Prentice Hall, 1st ed., 2002.*



Principes de conception Objet - Philosophie

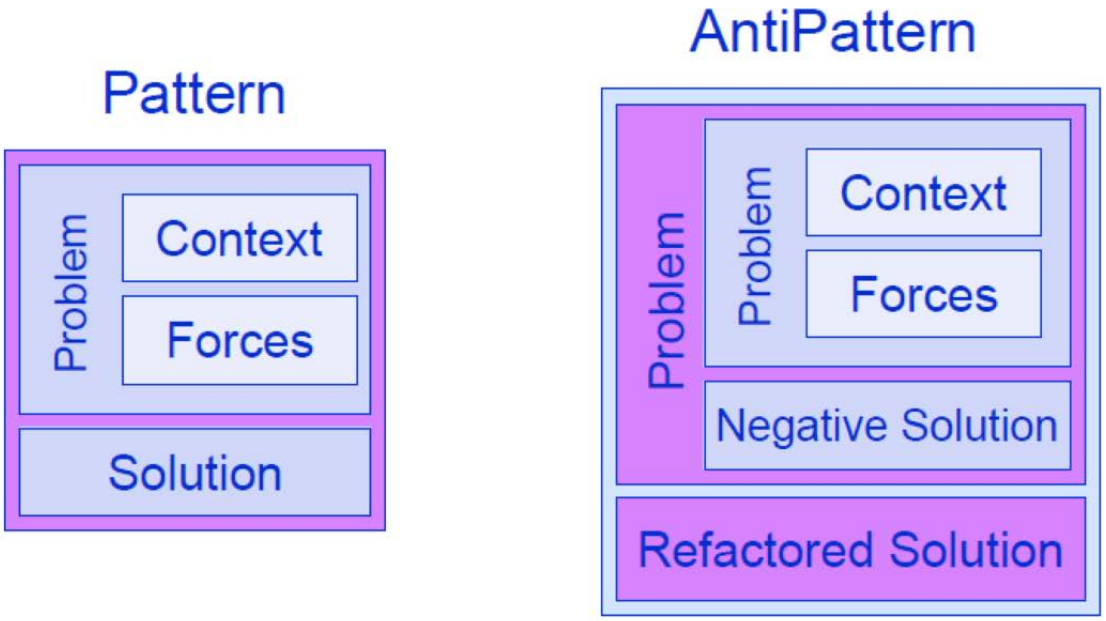
- Il vaut mieux programmer à l'aide d'interfaces plutôt que d'utiliser des classes.
- Les *getters* et les ***setters*** des objets fournissent un excellent moyen de configurer une application.
- La conception objet est plus importante que n'importe quelle technologie de mise en œuvre
- Un framework ne doit pas forcer les utilisateurs à catcher des exceptions qu'il ne pourra pas récupérer
- La testabilité est essentielle. Un framework d'entreprise doit aider les développeurs à tester leur application.

Conséquences d'une mauvaise conception

- Rigidité
  - Difficulté à changer des parties du code (*continuity*)
  - Réticence aux changements (tout changement devient politique)
- Fragilité
  - Des erreurs surviennent à des endroits inattendus (*protection*)
  - Même de légers changements peuvent causer des erreurs en cascade
- Immobilité
  - Le code est si emmêlé qu'il devient impossible de réutiliser
  - Nombreux codes dupliqués (sémantique ou syntaxique)
- Viscosité
  - Plus facile de faire une grosse verrue que de préserver la conception originale
  - "easy to do the wrong thing, but hard to do the right thing" (R.Martin)

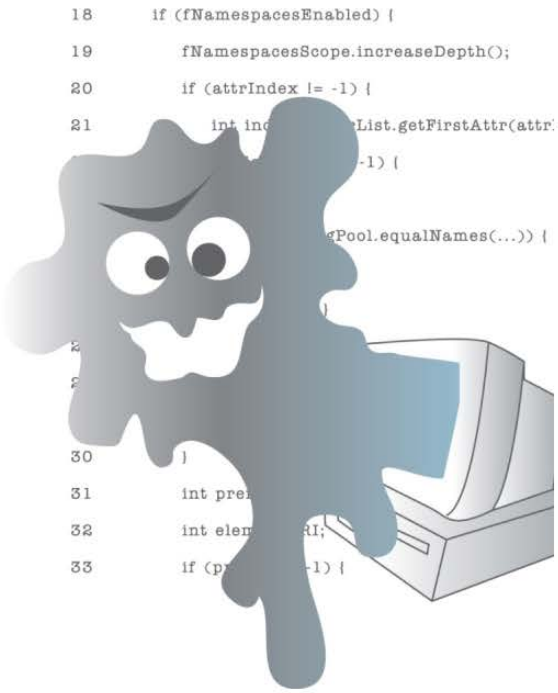
Défauts de conception

- Un **anti-patron** est un type spécial de patron de conception caractérisé par une solution refactorisée



Défauts de conception

- 2 exemples d'anti-patrons
  - Blob *(God Class)*



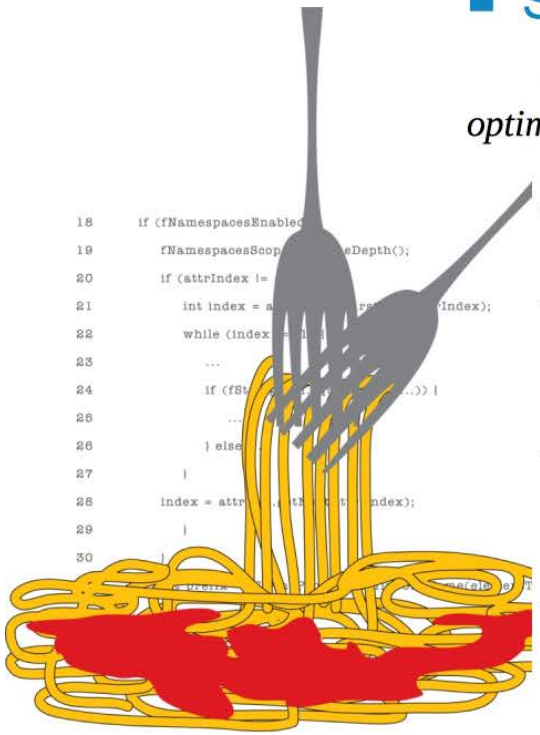
" Procedural-style design leads to one object with a lion's share of the responsibilities while most other objects only hold data or execute simple processes "

- Conception procédurale en programmation OO
- Large classe contrôleur
- Beaucoup d'attributs et méthodes avec une faible cohésion*
- Dépend de classes de données

** À quel point les méthodes sont étroitement liées aux attributs et aux méthodes de la classe.*

Défauts de conception

- 2 exemples d'anti-patrons



- Spaghetti Code

" Ad hoc software structure makes it difficult to extend and optimize code. "

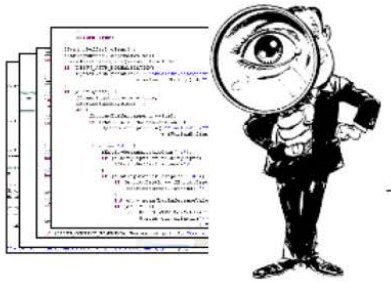
  - Conception procédurale en programmation OO
  - Manque de structure : pas d'héritage, pas de réutilisation, pas de polymorphisme
  - Noms des classes suggèrent une programmation procédurale
  - Longues méthodes sans paramètres avec une faible cohésion
  - Utilisation excessive de variables globales

En génie logiciel

un patron ≠ de

≠ d'un anti-patron

Défauts de conception

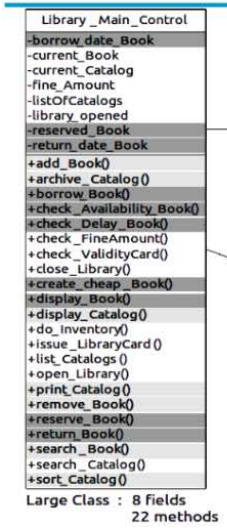


Défaut

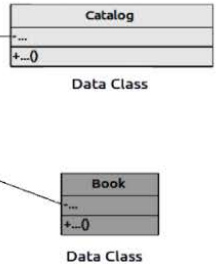
À détecter!

À corriger!

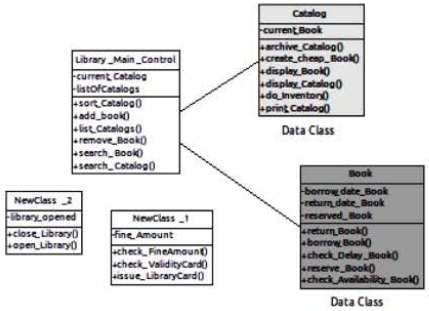
Anti-patron: Blob



Large Class : 8 fields
22 methods



Avant

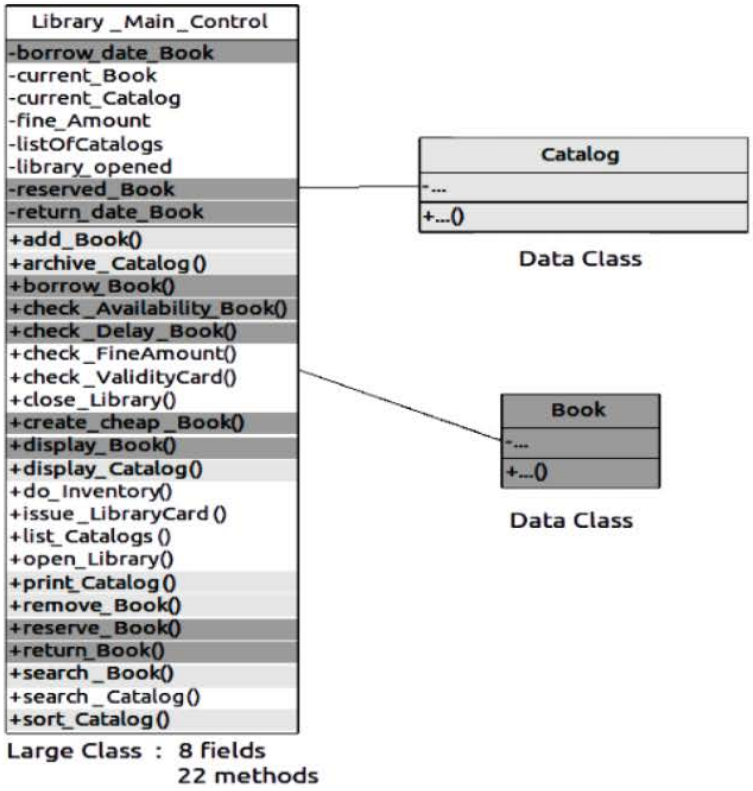


Après

Exemple extrait du livre "*AntiPatterns* in Project Management" de W. Brown *et al.* 1998

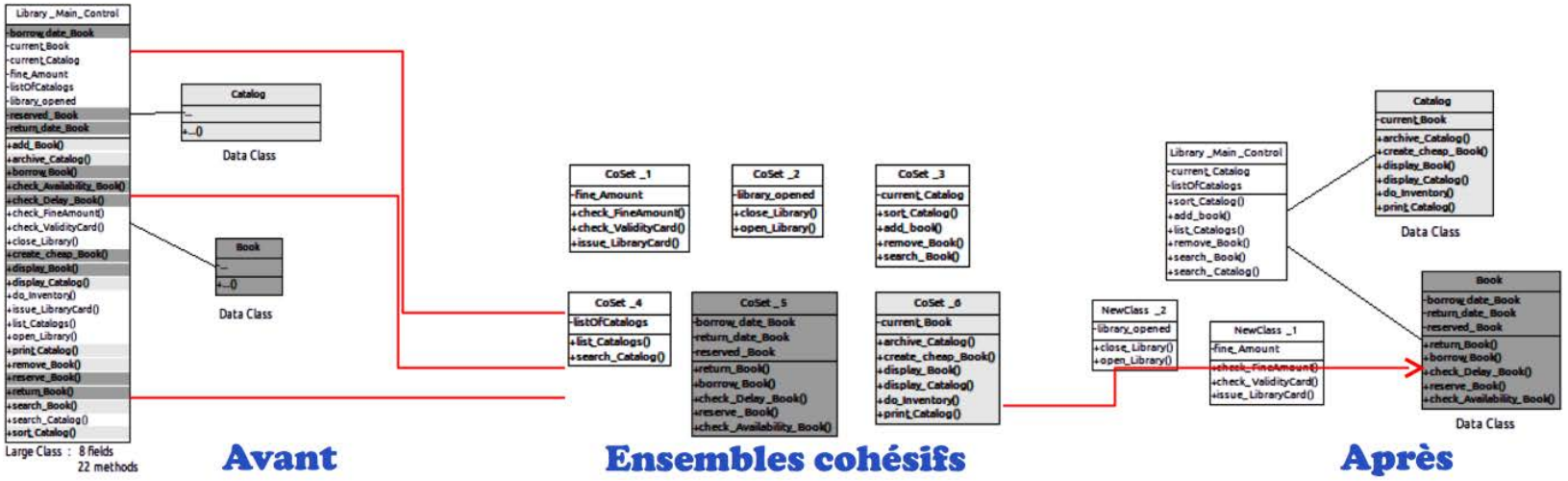
Détecter un anti-patron

- Blob
  - Identifier les larges classes
  - Identifier les classes de données



Corriger un anti-patron

- Blob
  - Identifier ou catégoriser les attributs et opérations liées
  - Rechercher des classes candidates pour les accueillir
  - Appliquer des techniques de conception objet (héritage, délégation, patrons, *etc.*)



Mauvaises odeurs

- Longue méthode
- Large classe
- Longue liste de paramètres
- Primitive obsession
- Groupe de données (Data clumps)
- Instructions Switch
- Champ temporaire
- Héritage refusé
- Classes alternatives avec des interfaces différentes
- Hiérarchies parallèles d'héritage
- Classe paresseuse
- Classe de données
- Code dupliqué
- Généralité spéculative
- Chaine de messages
- Middle man
- Feature envy
- Divergent change
- Shotgun surgery
- Classe de librairie incomplète
- Commentaires

- Code dupliqué
  - Structure de code dupliqué à différents endroits
- Longue méthode
  - À décomposer pour viser la clarté et facilité de maintenance
- Large classe
  - Classes qui essaient d'en faire trop. Présence de code dupliqué
- Longue liste de paramètres
  - Passer à la méthode juste ce dont elle a besoin
- Commentaires

- Divergent Change
  - Si une classe est modifiée de différentes manières pour différentes raisons, ça vaut la peine de diviser la classe de sorte que chaque partie soit associée à un type de changement particulier.
- Shotgun Surgery
  - Si un type de changement nécessite plusieurs petits changement de code dans différentes classes, tous ces bouts de code qui sont affectés devraient être mis ensemble dans une classe.
- Feature Envy
  - Une méthode d'une classe est plus intéressée par les attributs d'une autre classes que celles de sa propre classe. Peut-être que placer la méthode dans cette autre classe serait plus appropriée.

```
public class A {
 public void fooA() {
 }
```

```
public class B {
 A a = new A();
 public void foobarB() {
```

- Primitive Obsession
  - Parfois, plus intéressant de déplacer un type de données primitives vers une classe légère pour le rendre explicite et identifier les opérations à réaliser (ex : créer une classe date plutôt qu'utiliser un couple d'entiers).
- Instructions Switch
  - Tendent à créer de la duplication. Plusieurs instructions switch éparpillées à différents endroits. Utiliser des classes et du polymorphisme.
- Hiérarchies parallèles d'héritage
  - Deux hiérarchies parallèles existent et un changement dans une classe de la hiérarchie nécessite des changements dans l'autre hiérarchie.

```
public class B extends A {
}
```

```
public enum AEnum {
 B,
```

Trouver le défaut

```
class OwnershipTest...
   private void createUserInGroup() {
      GroupManager groupManager = new GroupManager();

      Group group = groupManager.create(TEST_GROUP, false,
                    GroupProfile.UNLIMITED_LICENSES, "",
                       GroupProfile.ONE_YEAR, null);

      user = userManager.create(USER_NAME, group, USER_NAME,
             "joshua", USER_NAME,LANGUAGE, false, false,
             new Date(), "blah", new Date());
}
```

Longue liste de paramètres

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }
}

public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
    }
}
```

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }
}
```

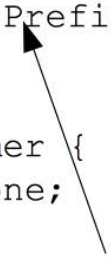
Feature Envy

Customer va rechercher dans les données de Phone
getPhoneNumber devrait être La class Phone.

```
public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
    }
}
```

Trouver le défaut

```
public class Phone {
  public String getAreaCode() {
     return 1 ;
  }
  public String getPrefix() {
     return 21 ;
  }
  public String getNumber() {
     return 1234 ;
  }

  public String toString() {
    return "(" + phone.getAreaCode() + ") "
        + phone.getPrefix() + "-" + phone.getNumber();
}

public class Customer {
  private Phone phone;

  public String getPhoneNumber() {
    return phone ;
  }
}
```



Correction

`Customer` compte sur `Phone` pour faire le formatage

```
public abstract class AbstractCollection implements collection
   public void addAll(AbstractCollection c) {
      if(c instanceof Set) {
           Set s = (Set)c;
           for(int i=0; i<s.size();i++)
                if(!contains(s.get(i)))
                     add(s.get(i));
      }
      else if(c instanceof List) {
             List l = (List)c;
             for(int i=0;i<l.size();i++)
                    if(!contains(l.get(i)))
                           add(l.get(i));
   }
```

Trouver le défaut

Instruction Switch

```
public abstract class AbstractCollection implements collection
   public void addAll(AbstractCollection c) {
      if(c instanceof Set) {
           Set s = (Set)c;
           for(int i=0; i<s.size();i++)
                if(!contains(s.get(i)))
                     add(s.get(i));
      }
      else if(c instanceof List){
             List l = (List)c;
             for(int i=0;i<l.size();i++)
                    if(!contains(l.get(i)))
                           add(l.get(i));
   }
```

Classes alternatives avec interfaces différentes

Code dupliqué

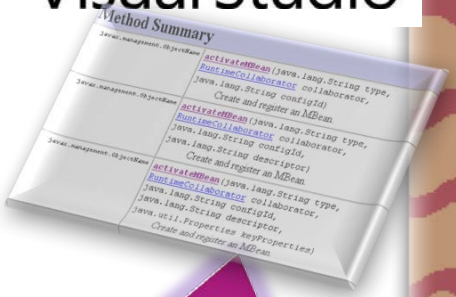
Trouver le défaut

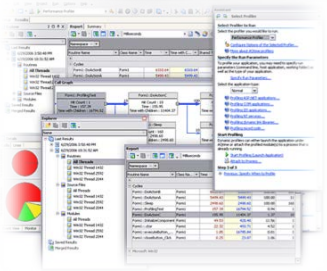
```
public abstract class AbstractCollection  {
    public void add(Object element) {
    }
}
```

```
public class Map extends AbstractCollection {
    // Do nothing because user must input key and value
    public void add(Object element) {
    }
}
```

Outils et Méthodes

Software Engineering





Documentation and Source Code

Documentation

- Source code: one of the best artefact for documenting a project
- Javadoc (JDK)
  - Automatic **generation** of HTML documentation
  - Using comments in java files
- Syntax

```
/**
 * This is a <b>doc</b> comment.
 * @see java.lang.Object
 * @todo fix {@underline this !}
 */
```

- Includes
  - class hierarchy, interfaces, packages
  - detailed summary of class, interface, methods, attributes
- Note
  - Add doc generation to your favorite **compile chain**

Package javax.swing

Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.

See:

Description

Interface Summary

Action	The `Action` interface provides a useful extension to the `ActionListener` interface in cases where the same functionality may be accessed by several contro
BoundedRangeModel	Defines the data model used by components like Sliders and ProgressBars.
ButtonModel	State Model for buttons.
CellEditor	This interface defines the methods any general editor should be able to implement.
ComboBoxEditor	The editor component used for JComboBox components.
ComboBoxModel	A data model for a combo box.
DesktopManager	DesktopManager objects are owned by a JDesktopPane object.
Icon	A small fixed size picture, typically used to decorate components.
JComboBox.KeySelectionManager	The interface that defines a `KeySelectionManager`.
ListCellRenderer	Identifies components that can be used as "rubber stamps" to paint the cells in a JList.
ListModel	This interface defines the methods components like JList use to get the value of each cell in a list and the length of the list.
ListSelectionModel	This interface represents the current state of the selection for any of the components that display a list of values with stable indices.
MenuElement	Any component that can be placed into a menu should implement this interface.
MutableComboBoxModel	A mutable version of `ComboBoxModel`.
Renderer	Defines the requirements for an object responsible for "rendering" (displaying) a value.
RootPaneContainer	This interface is implemented by components that have a single JRootPane child: JDialog, JFrame, JWindow, JApplet, JInternalFrame.
Scrollable	An interface that provides information to a scrolling container like JScrollPane.
ScrollPaneConstants	Constants used with the JScrollPane component.
SingleSelectionModel	A model that supports at most one indexed selection.
SpinnerModel	A model for a potentially unbounded sequence of object values.
SwingConstants	A collection of constants generally used for positioning and orienting components on the screen.
UIDefaults.ActiveValue	This class enables one to store an entry in the defaults table that's constructed each time it's looked up with one of the `getXXX(key)` methods.
UIDefaults.LazyValue	This class enables one to store an entry in the defaults table that isn't constructed until the first time it's looked up with one of the `getXXX(key)` methods.
WindowConstants	Constants used to control the window-closing operation.

public class **JFrame**
extends Frame
implements WindowConstants, Accessible, RootPaneContainer

An extended version of `java.awt.Frame` that adds support for the JFC/Swing component architecture. You can find task-o

The `JFrame` class is slightly incompatible with `Frame`. Like all other JFC/Swing top-level containers, a `JFrame` contains a `J` the AWT `Frame` case. For example, to add a child to an AWT frame you'd write:

```
frame.add(child);
```

However using `JFrame` you need to add the child to the `JFrame's` content pane instead:

```
frame.getContentPane().add(child);
```

The same is true for setting layout managers, removing components, listing children, and so on. All these methods should nc exception. The default content pane will have a BorderLayout manager set on it.

update

```
public void update(Graphics g)
```

Just calls `paint(g)`. This method was overridden to prevent an unnecessary call to clear the background.

Overrides:

`update` in class `Container`

Parameters:

`g` - the Graphics context in which to paint

See Also:

`Component.update(Graphics)`

Kornel Kisielewicz @epyoncf 12 Aug

ProTip: "//" is the speedup operator. Use // before the statement you want to speed up. Works in C++, Java and a few others!

Retweeted by Mathieu Acher

Collapse Reply Retweeted Favorite More

1,253 RETWEETS **295** FAVORITES

12:31 AM - 12 Aug 13 · Details

Coding Conventions

- Rules on the coding style :
  - Apache, Oracle and others template
    - e.g. http://www.oracle.com/technetwork/java/codeconv-138413.html
    - http://geosoft.no/development/javastyle.html
- Verification tools
  - CheckStyle, PMD, JackPot, Spoon Vsuite…
  - Some integrated into IDEs

Why Coding Standards are Important?

- Lead to greater **consistency** within your code and the code of your teammates
- Easier to **understand**
- Easier to **develop**
- Easier to **maintain**
- Reduces overall cost of application

Example

8. Private class variables should have underscore suffix.

```
class Person
{
  private String name_;
  ...
}
```

Apart from its name and its type, the *scope* of a variable is its most
higher significance than method variables, and should be treated w

A side effect of the underscore naming convention is that it nicely r

```
  void setName(String name)
  {
    name_ = name;
  }
```

Tools to Improve your Source code

- Formatting tools
  - Indenteurs (Jindent), beautifiers, stylers (JavaStyle), …
- « Bug fixing » tools
  - Spoon VSuite, Findbugs (sourceforge) …
- Quality report tools : code metrics
  - Number of Non Comment Code Source, Number of packages, Cyclomatic numbers, …
    - JavaNCCS, Eclipse Metrics …

Refactoring

What's Code Refactoring?

"A series of *small* steps, each of which changes the program's *internal structure* without changing its *external behavior*"

Martin Fowler

Example

Which code segment is easier to read?

Sample 1:

```
if (markT>=0 && markT<=25 && markL>=0 && markL<=25){
        float markAvg = (markT + markL)/2;
        System.out.println("Your mark: " + markAvg);
}
```

Sample 2:

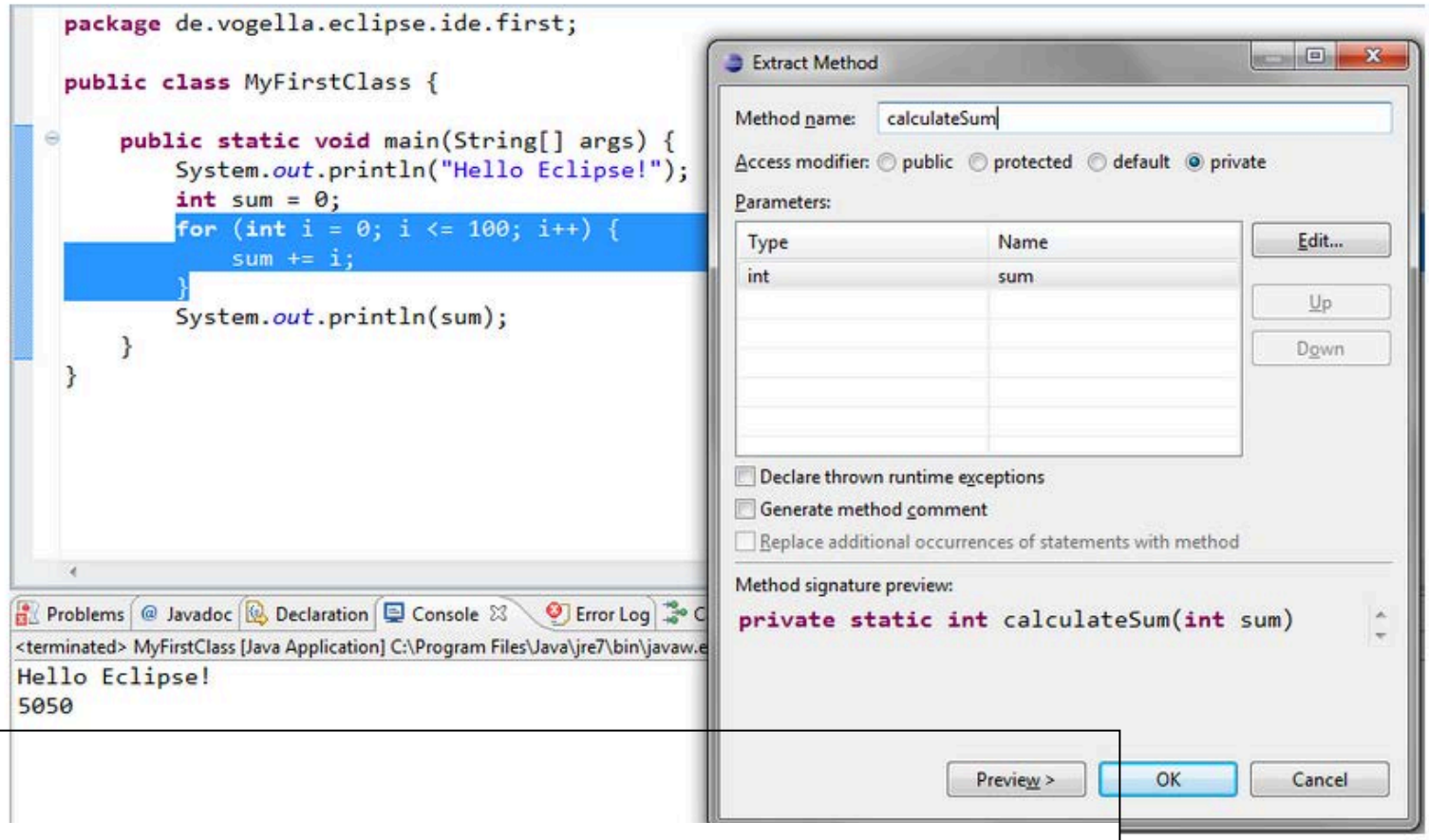
```
if (isValid(markT) && isValid(markL)){
        float markAvg = (markT + markL)/2;
        System.out.println("Your mark: " + mark);
}
```

Why do we Refactor?

- Improves the design of our software
  - Apply design pattern / remove anti pattern
- Minimizes technical debt
- Keep development at speed
- To make the software easier to understand
- To help find bugs
- To “Fix broken windows”

How do we Refactor?

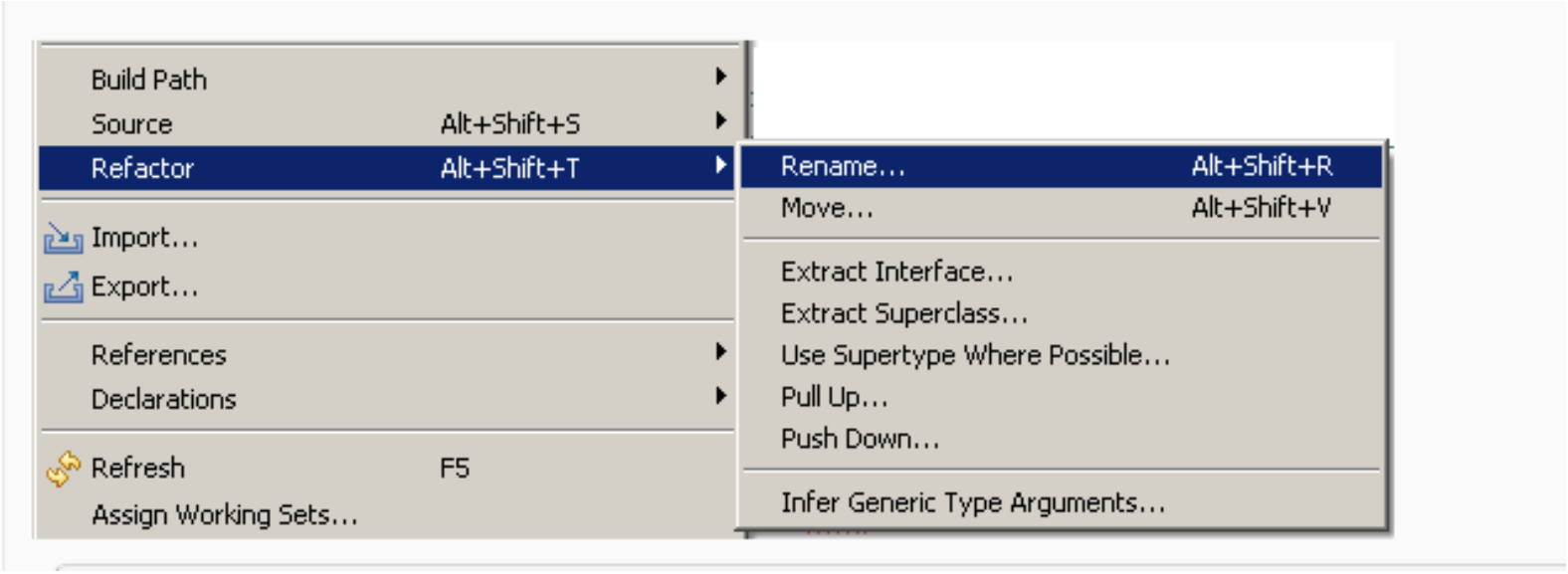
- Manual Refactoring
  - Code Smells
- Automated/Assisted Refactoring
  - Refactoring by hand is time consuming and prone to error
  - Tools (IDE)
- In either case, **test your changes**




```
package de.vogella.eclipse.ide.first;

public class MyFirstClass {

  public static void main(String[] args) {
    System.out.println("Hello Eclipse!");
    int sum = 0;
    sum = calculateSum(sum);
    System.out.println(sum);
  }

  private static int calculateSum(int sum) {
    for (int i = 0; i <= 100; i++) {
      sum += i;
    }
    return sum;
  }
}
```

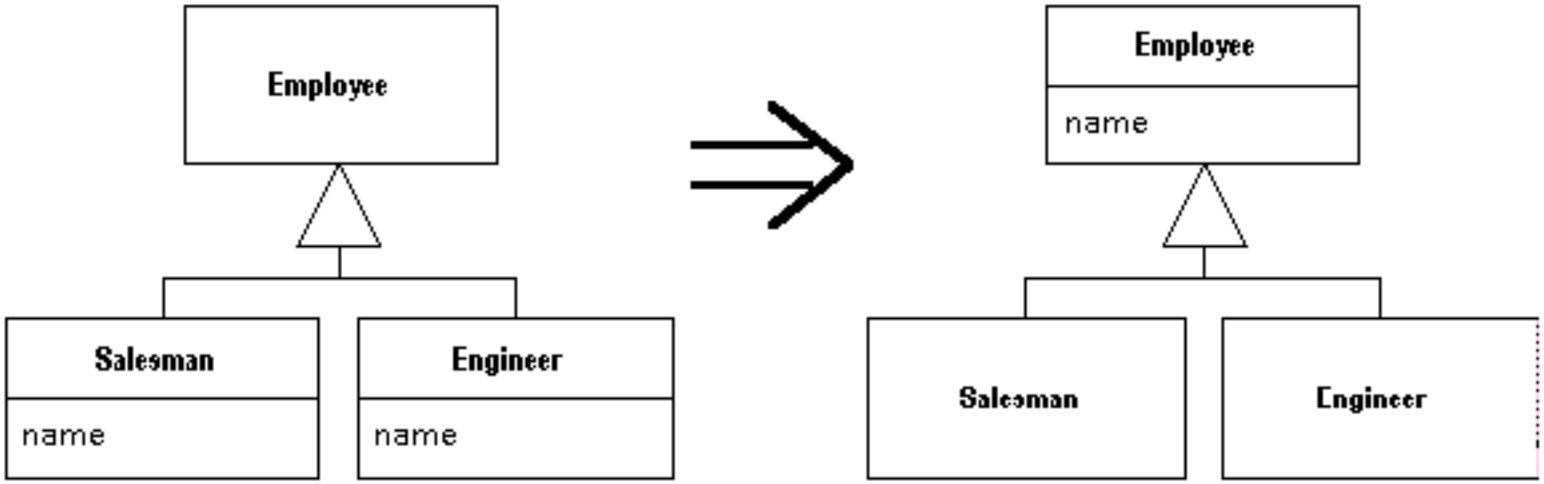
Typical refactoring patterns

- Rename variable / class / method / member
- Extract method
- Extract constant
- Extract interface
- Encapsulate field



Two subclasses have the same field.

Move the field to the superclass.



You have a complicated expression.

Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
       wasInitialized() && resize > 0 )
 {
   // do something
 }
```

```
        final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
        final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
        final boolean wasResized  = resize > 0;

        if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
        {
                // do something
        }
```

Logging

Debugging

- Symbolic debugging
  - javac options: -g, -g:source,vars,lines
  - command-line debugger : jdb (JDK)
    - commands look like those of dbx
  - graphical « front-ends » for jdb (AGL)
  - Misc
    - Multi-threads, Cross-Debugging (-Xdebug) on remote VM , ...

Monitoring

- Tracer
  - TRACE options of the program
  - can slow-down .class with TRACE/←TRACE tests
    - solution : use a pre-compiler (excluding trace calls)
  - Kernel tools, like OpenSolaris DTrace (coupled with the JVM)

Logging

- Logging is chronological and systematic record of data processing events in a program
  - e.g. the Windows Event Log
- Logs can be saved to a persistent medium to be studied at a later time
- Use logging in the development phase:
  - Logging can help you **debug** the code
- Use logging in the production environment:
  - Helps you **troubleshoot problems**

Logging, why? (claims)

- Logging is easier than debugging
- Logging is faster than debugging
- Logging can work in environments where debugging is not supported
- Can work in production environments
- Logs can be referenced anytime in future as the data is stored

Logging Methods, How?

- The evil System.out.println()

- Custom Solution to Log to various datastores, eg text files, db, etc…

- Use Standard APIs
  - Don’t reinvent the wheel

Log4J

- Popular logging frameworks for Java
- Designed to be reliable, fast and extensible
- Simple to understand and to use API
- Allows the developer to control which log statements are output with arbitrary granularity
- Fully configurable at runtime using external configuration files

Log4J Architecture

- Log4J has three main components: loggers, appenders and layouts
  - Loggers
    - Channels for printing logging information
  - Appenders
    - Output destinations (console, File, Database, Email/SMS Notifications, Log to a socket, and many others…)
  - Layouts
    - Formats that appenders use to write their output
- Priorities

Logger

- Responsible for Logging
- Accessed through java code
- Configured Externally
- Every Logger has a name
- Prioritize messages based on level
  - TRACE < DEBUG < INFO < WARN < ERROR < FATAL
- Usually named following dot convention like java classes do.
  - Eg com.foo.bar.ClassName
- Follows inheritance based on name

Logger API

- Factory methods to get Logger
  - `Logger.getLogger(Class c)`
  - `Logger.getLogger(String s)`
- Method used to log message
  - `trace(), debug(), info(), warn(), error(), fatal()`
  - Details
    - `void debug(java.lang.Object message)`
    - `void debug(java.lang.Object message, java.lang.Throwable t)`
  - Generic Log method
    - `void log(Priority priority, java.lang.Object message)`
    - `void log(Priority priority, java.lang.Object message, java.lang.Throwable t)`

Root Logger

- The root logger resides at the top of the logger hierarchy. It is exceptional in two ways:
  1. it always exists,
  2. it cannot be retrieved by name.

- Logger.getRootLogger()

Appender

- Appenders put the log messages to their actual destinations.
- No programatic change is require to configure appenders
- Can add multiple appenders to a Logger.
- Each appender has its Layout.
- ConsoleAppender, DailyRollingFileAppender, FileAppender, JDBCAppender, JMSAppender, NTEventLogAppender, RollingFileAppender, SMTPAppender, SocketAppender, SyslogAppender, TelnetAppender

Layout

- Used to customize the format of log output.
- Eg. HTMLLayout, PatternLayout, SimpleLayout, XMLLayout
- Most commonly used is PatternLayout
  - Uses C-like syntax to format.
    - `Eg. "%-5p [%t]: %m%n`
    - `DEBUG [main]: Message 1 WARN [main]: Message 2`

Log4j Basics

- Who will log the messages?
  - The *Loggers*
- What decides the priority of a message?
  - *Level*
- Where will it be logged?
  - Decided by *Appender*
- In what format will it be logged?
  - Decided by *Layout*

Log4j in Action

```
// get a logger instance named "com.foo"
Logger  logger = Logger.getLogger("com.foo");

// Now set its level. Normally you do not need to set the
// level of a logger programmatically. This is usually done
// in configuration files.
logger.setLevel(Level.INFO);

Logger barlogger = Logger.getLogger("com.foo.Bar");

// This request is enabled, because WARN >= INFO.
logger.warn("Low fuel level.");

// This request is disabled, because DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// The logger instance barlogger, named "com.foo.Bar",
// will inherit its level from the logger named
// "com.foo" Thus, the following request is enabled
// because INFO >= INFO.
barlogger.info("Located nearest gas station.");

// This request is disabled, because DEBUG < INFO.
barlogger.debug("Exiting gas station search");
```

Log4j Optimization & Best Practises

- Use logger as private static variable
- Only one instance per class
- Name logger after class name
- Don’t use too many appenders
- Don’t use time-consuming conversion patterns
- Use Logger.isDebugEnabled() if need be
- Prioritize messages with proper levels

You can’t test everything
(so one advice by Martin Fowler)

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.

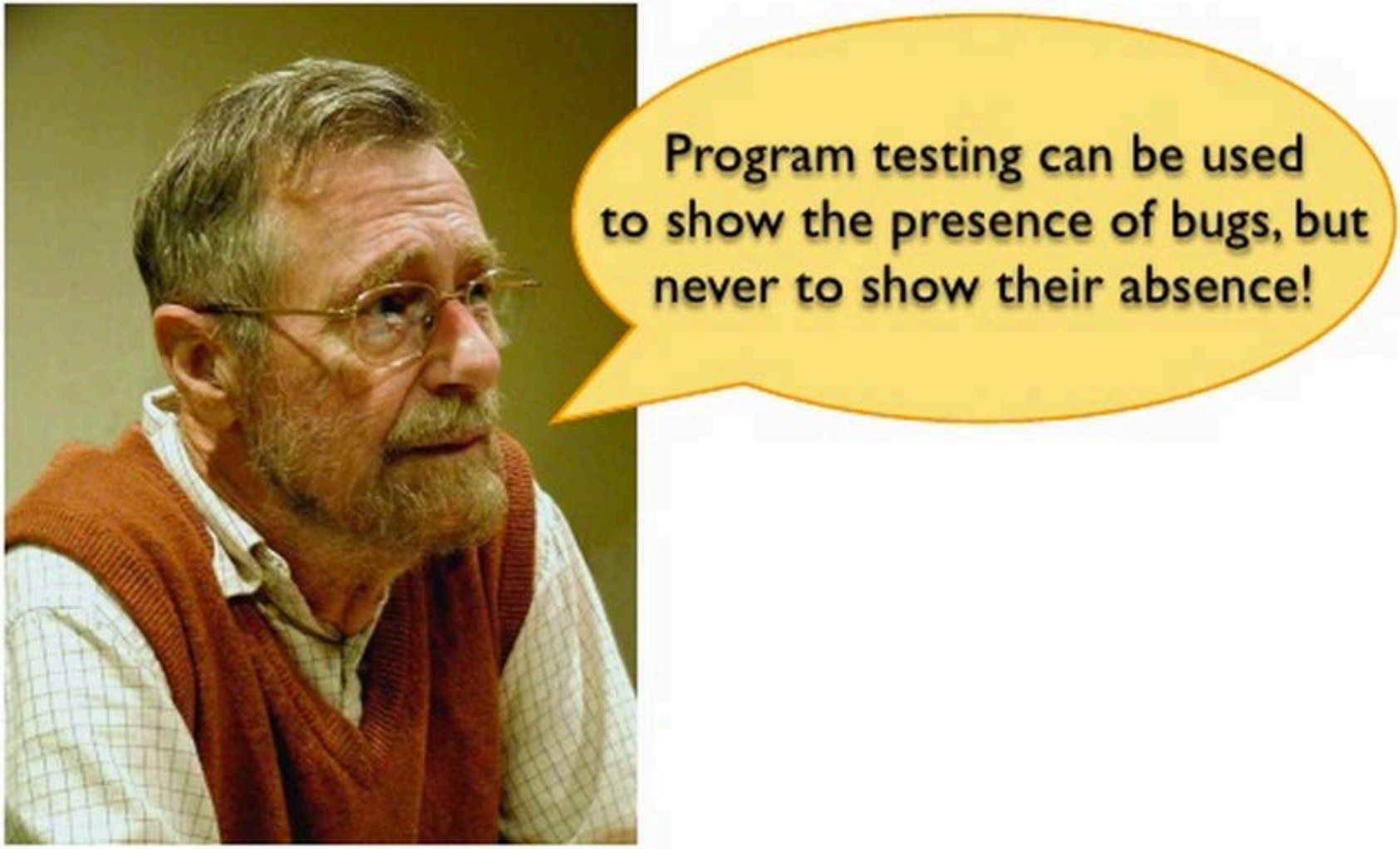
Testing

…the activity of finding out whether a piece of code (a method, class or program) produces the intended behavior

Your **hope** as a programmer

« A program does exactly what you expected to do »

Djikstra


Program testing can be used
to show the presence of bugs, but
never to show their absence!


I don’t make mistakes
Microsoft Visual Studio
Package Load Failure
Package 'ArtOfTest.WebAiiVSIP.WebAiiVSIP, ArtOfTest.WebAiiVSIP, Version=1.0.0.0, Culture=neutral, PublicKeyToken=14176981d1873d86' has failed to load properly (GUID = {D5EDEC6D-8025-40CF-987C-B8EABA2C7451}). Please contact package vendor for assistance. Application restart is recommended, due to possible environment corruption. Would you like to disable loading this package in the future? You may use 'devenv /resetskippkgs' to re-enable package loading.
Yes
No

Master 2 (Apprentis)

15 « jobs », 15 aim at
Testing (critical or non critical) applications
Correcting anomalies and ensuring that they won't appear in the future
Maintaining

« 1 day of producing code
= 3 days of testing code »

« 70% of a software project = maintainance »

10. HealthCare.gov didn't have enough testing before going live.

This became clear in a series of Congressional hearings, where federal contractors testified that end-to-end testing only began in the final weeks of September, right before the Oct. 1 launch. When pressed on how much time would have been ideal for testing, one contractor told lawmakers that "months would have been nice."

http://www.washingtonpost.com/blogs/wonkblog/wp/2013/11/01/thirty-one-things-we-learned-in-healthcare-govs-first-31-days/

Test phases

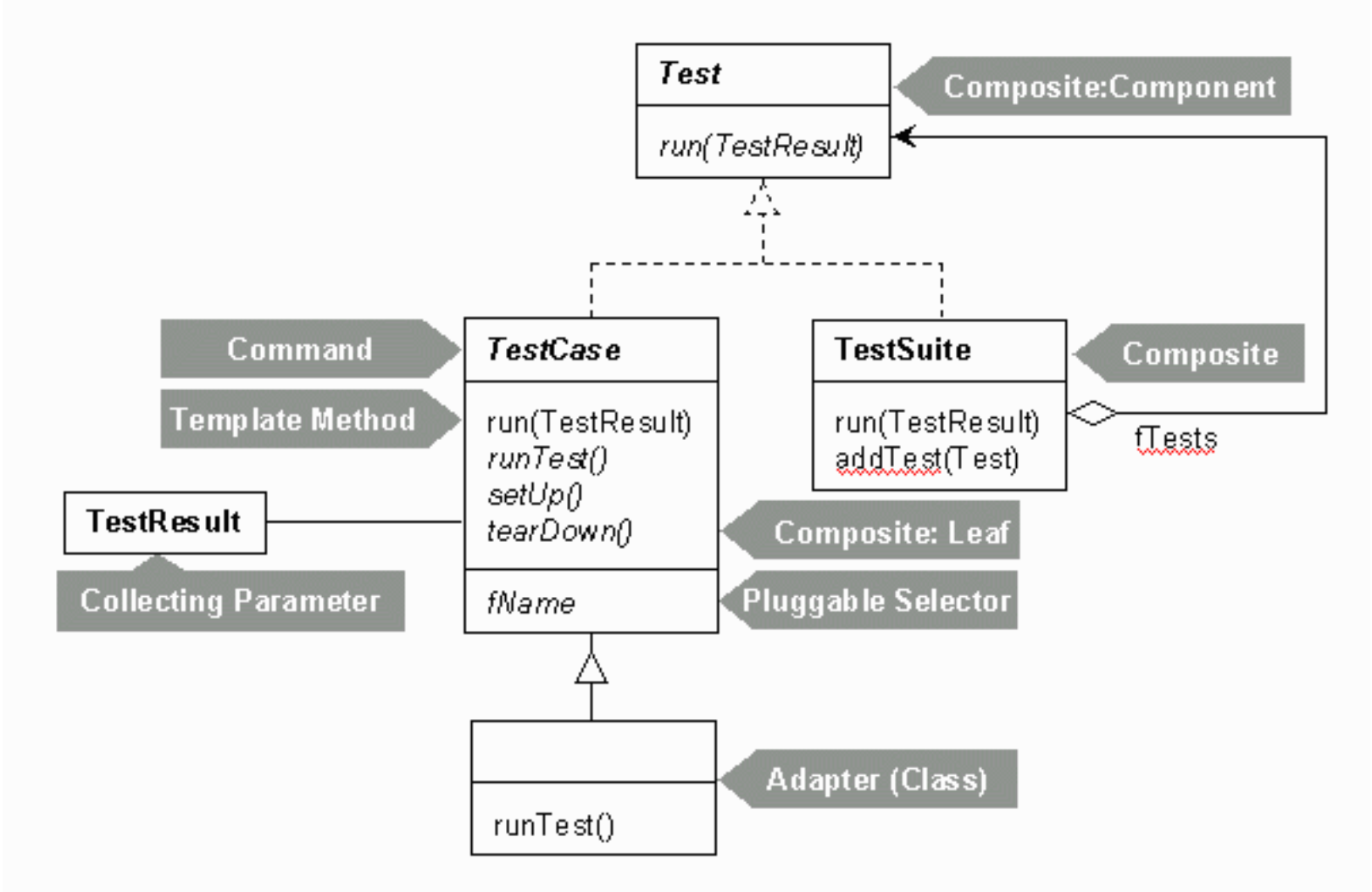
Unit testing on individual units of source code (=smallest testable part).

Integration testing on groups of individual software modules.

System testing on a complete, integrated system (evaluate compliance with requirements)

Junit and… Design Patterns

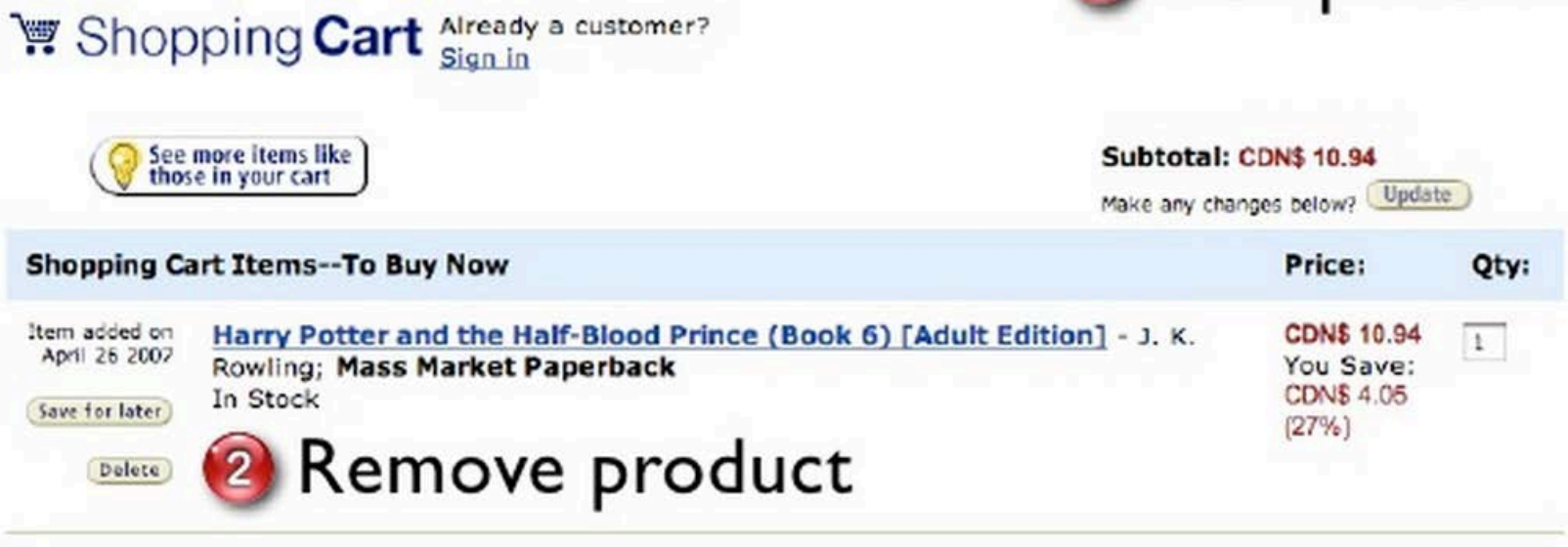
http://junit.sourceforge.net/doc/cookstour/cookstour.htm



Running example

1. Set of products
2. Number of products
3. Balance

1. Add product



Init

Constructor + Set up and tear down of fixture.

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;

    // Creates a new test case
    public ShoppingCartTest(St
        super(name);
    }
```

```
    // Creates test environment (fixture).
    // Called before every testX() method.
    protected void setUp() {
        _bookCart = new ShoppingCart();

        Product book = new Product("Harry Potter", 23.95);
        _bookCart.addItem(book);
    }

    // Releases test environment (fixture).
    // Called after every testX() method.
    protected void tearDown() {
        _bookCart = null;
    }
```

Assertions

`fail(msg)` – triggers a failure named *msg*

`assertTrue(msg, b)` – triggers a failure, when condition *b* is false

`assertEquals(msg, v1, v2)` – triggers a failure, when $v1 \neq v2$

`assertEquals(msg, v1, v2, ϵ)` – triggers a failure, when $|v1 - v2| > \epsilon$

`assertNull(msg, object)` – triggers a failure, when *object* is not *null*

`assertNonNull(msg, object)` – triggers a failure, when *object* is *null*

Example #1

```
// Tests adding a product to the cart.
public void testProductAdd() {
      Product book = new Product("Refactoring", 53.95);
      _bookCart.addItem(book);

      assertTrue(_bookCart.contains(book));

      double expected = 23.95 + book.getPrice();
      double current = _bookCart.getBalance();

      assertEquals(expected, current, 0.0);

      int expectedCount = 2;
      int currentCount = _bookCart.getItemCount();

      assertEquals(expectedCount, currentCount);
}
```

Example #2

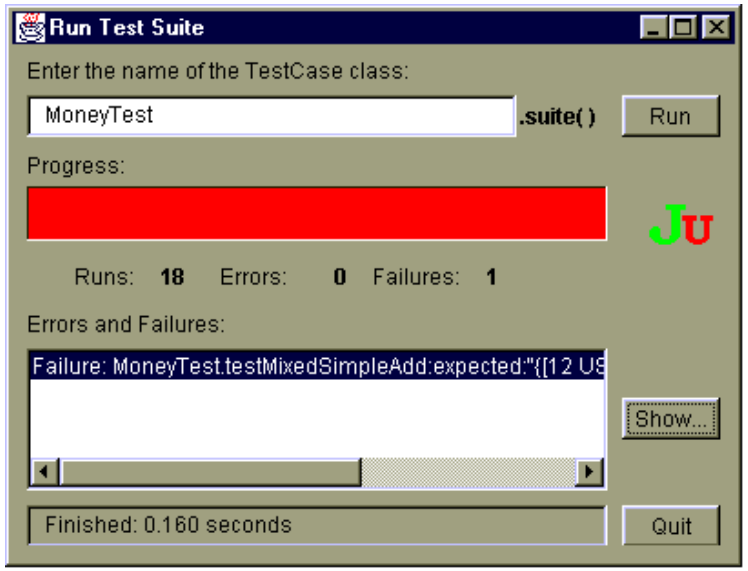
```
// Tests removing a product from the cart.
public void testProductRemove() throws NotFoundException {
      Product book = new Product("Harry Potter", 23.95);
      _bookCart.removeItem(book);

      assertTrue(!_bookCart.contains(book));

      double expected = 23.95 - book.getPrice();
      double current = _bookCart.getBalance();

      assertEquals(expected, current, 0.0);

      int expectedCount = 0;
      int currentCount = _bookCart.getItemCount();

      assertEquals(expectedCount, currentCount);
}
```

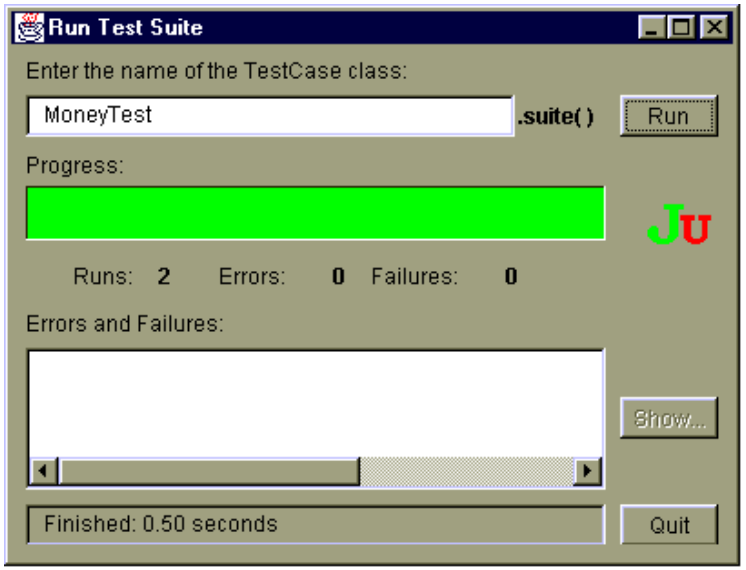
```
public static Test suite() {

    // Here: add all testX() methods to the suite (reflection).
    TestSuite suite = new TestSuite(ShoppingCartTest.class);

    // Alternative: add methods manually (prone to error)
    // TestSuite suite = new TestSuite();
    // suite.addTest(new ShoppingCartTest("testEmpty"));
    // suite.addTest(new ShoppingCartTest("testProductAdd"));
    // suite.addTest(...);

    return suite;
}
```

Unit Test

JUnit 3, 4 & 5 http://www.junit.org

- Test pattern
  - Test, TestSuite, TestCase
  - Assertions (assertXX) that must be verified
- TestRunner
  - Chain tests and output a report.





- See JUnit course:
  - http://people.irisa.fr/Benoit.Combemale/teaching/vv

Documenting,
Testing,
Design Patterns, bad smells
Refactoring,
ebugging, monitoring, loggi

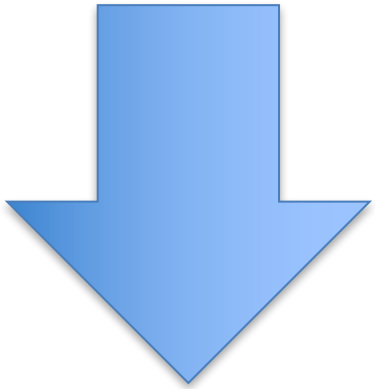
#1 What is the link?

- Documenting
  - Understanding (readability, maintainability)
- Refactoring
  - Improving the design (readability, maintainability, extensibility)
- The activity of documenting can somehow be replaced/automated by the activity of refactoring
  - if the code and architecture is comprehensible by itself

refactoring.com

Documentation and Refactoring

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&  // platform is MacOS
     (browser.toUpperCase().indexOf("IE") > -1) &&  // brower is IE
      wasInitialized() && resize > 0 )
 {
   // do something
 }
```



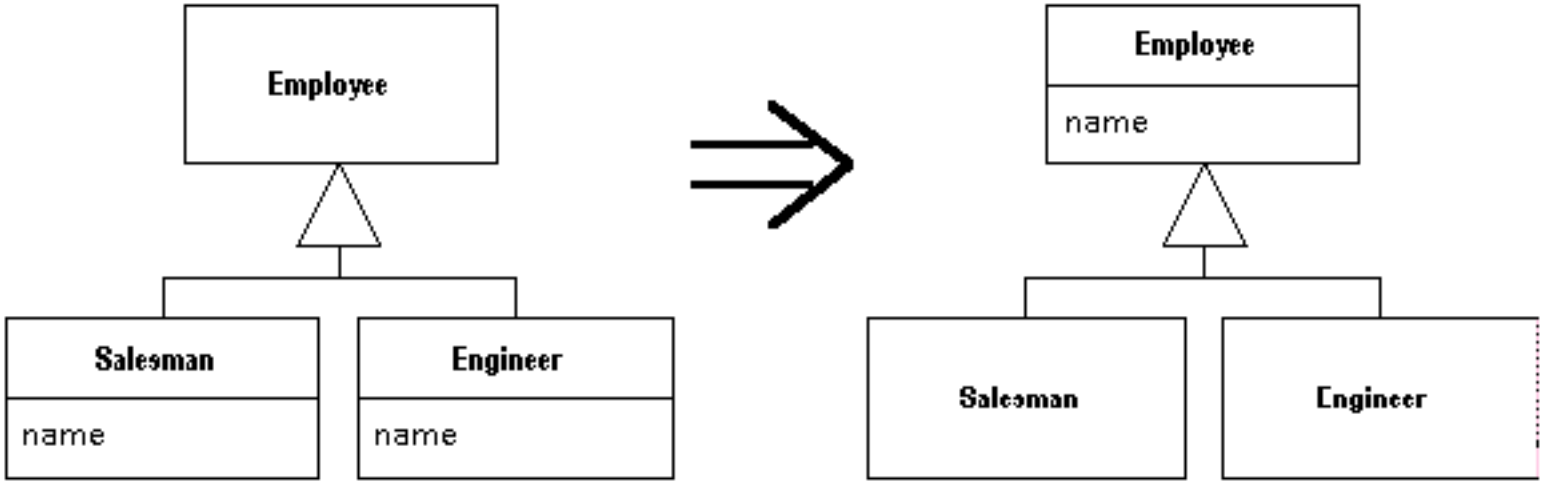
```
final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized  = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
        // do something
}
```

#2 What is the link?

Design patterns: there are refactorings

Two subclasses have the same field.

Move the field to the superclass.



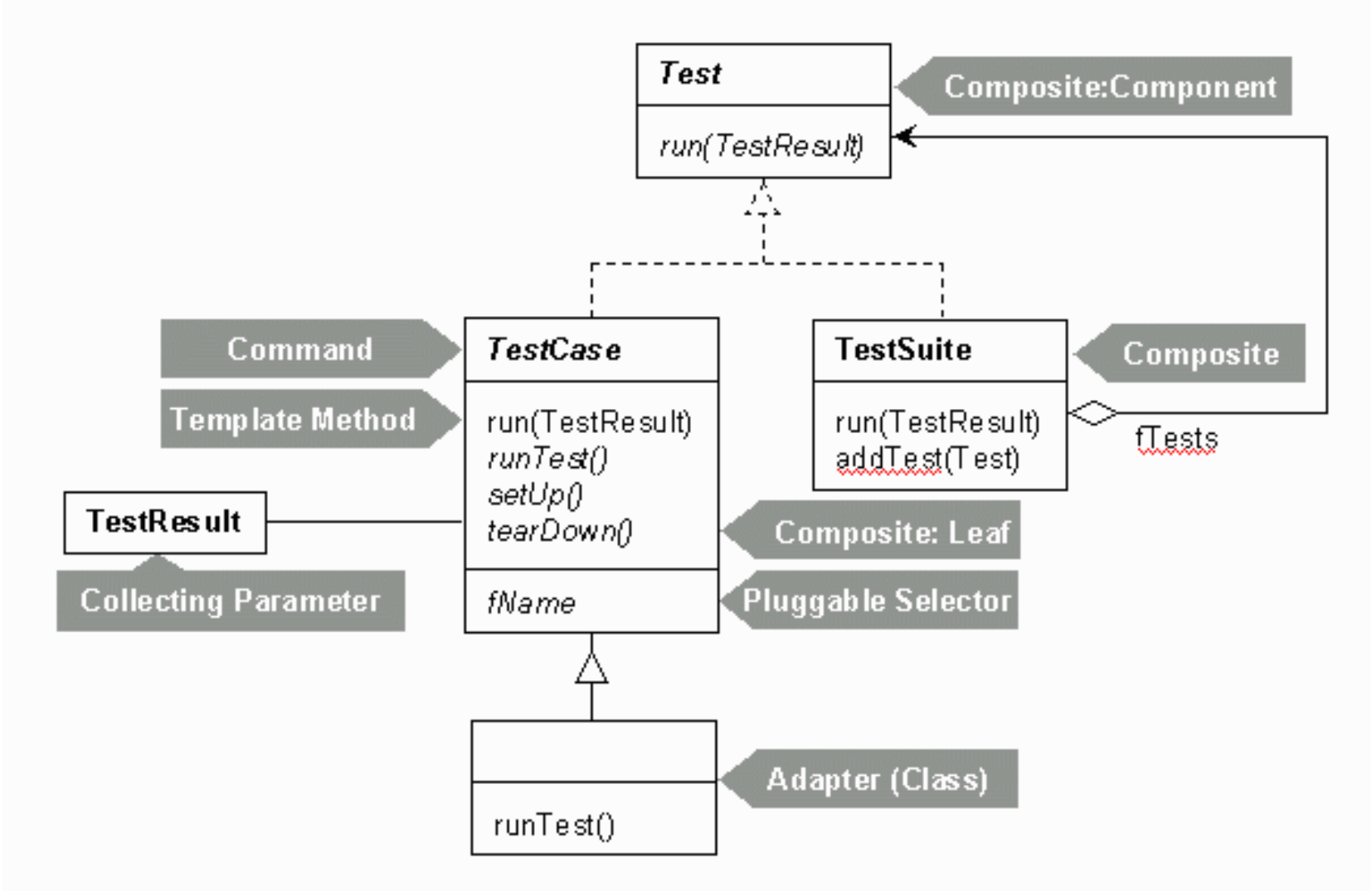
#3 What is the link?

- Testing: “the activity of finding out whether a piece of code produces the intended behavior”
  - Debugging can help
  - Testing is better

than debugging

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.

JUnit and… Design patterns



Worth reading!

http://junit.sourceforge.net/doc/cookstour/cookstour.htm

What is the link?

- Testability
  - degree to which a system or component **facilitates the establishment** of test criteria and the performance of tests to determine whether those criteria have been met."
  - Controllability + Observability
- **Controllability** ability to manipulate the software's input as well as to place this software into a particular state
- **Observability** deals with the possibility to observe the outputs and state changes that
- How to improve Testability?
  - Refactoring, Design patterns

What is the link?

Testing/Refactoring/Design Patterns

- How to improve testability?
- Test-driven Development
  - Write tests first ~ Test-driven design

Let say your first piece of code is… a test

```
// Tests removing a product from the cart.
public void testProductRemove() throws NotFoundException {
    Product book = new Product("Harry Potter", 23.95);
    _bookCart.removeItem(book);

    assertTrue(!_bookCart.contains(book));

    double expected = 23.95 - book.getPrice();
    double current = _bookCart.getBalance();

    assertEquals(expected, current, 0.0);

    int expectedCount = 0;
    int currentCount = _bookCart.getItemCount();

    assertEquals(expectedCount, currentCount);
}
```

What is the link?

- Testing
- Documenting

- Unit tests are one of the best source of documentation
  - One of the entry point to understand a framework
  - It documents the properties of methods, how objects collaborate, etc.

What is the link?

Documenting

Refactoring

Debugging

Testing

Readability
Understandibility
Maintainability

Design

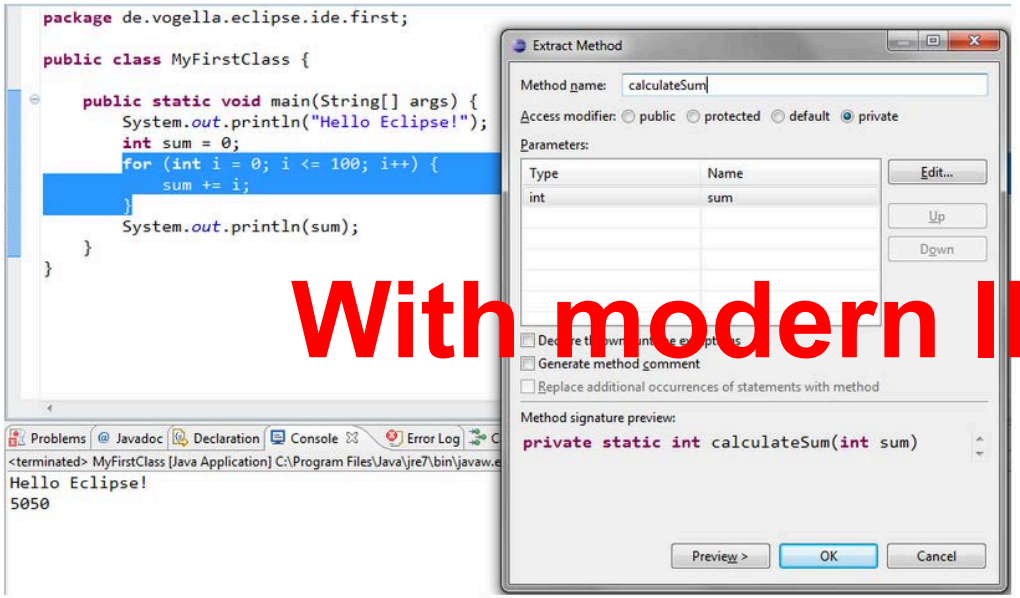
Document, refactor… Execute your tests… Debug.. Write test.. And so on!

Documenting

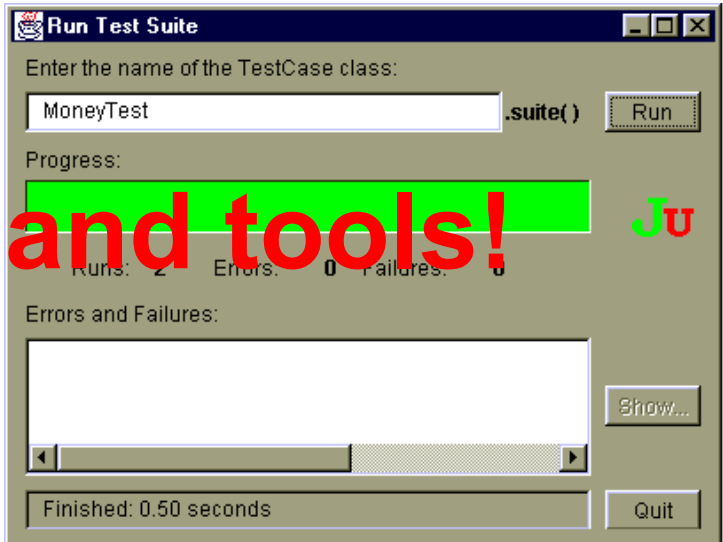
Debugging

Refactoring

Testing



With modern IDE and tools!



INTEGRATION CONTINUE

Jenkins

INTRODUCTION

- Qu’est ce que l’intégration continue ?
- Pourquoi automatiser ?
- Par où commencer ?
- Le cycle vertueux de l’intégration continue

Définitions

"L'intégration continue est un ensemble de pratiques utilisées en génie logiciel. Elles consistent à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression de l'application en cours de développement."

Wikipedia

"Une pratique considérant différemment l'intégration, habituellement connue comme pénible et peu fréquente, pour en faire une tâche simple faisant partie intégrante de l'activité quotidienne d'un développeur."

Documentation
CruiseControl.NET

Qu’est ce que l’intégration continue ?

- Technique puissante permettant dans le cadre du développement d’un logiciel en équipes de:
  - Garder en phase les équipes de dév
  - Limiter risques de dérive
  - Limiter la complexité

- A intervalles réguliers, vous allez construire (build) et tester la dernière version de votre logiciel.

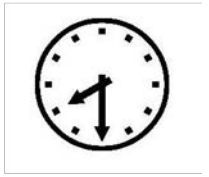
- Parrallèlement, chaque développeur teste et valide (commit) son travail en ajoutant son code dans un lieu de stockage unique.

Pourquoi automatiser ?

- Gagner du temps
  - Vous ne faites pas de taches répétitives
- Gagner en confiance
  - Indépendant de votre efficacité du moment
  - Procédures répétables
- Diminue le besoin de documentation
  - Pour nouveaux entrants projet, utiliser scripts !
  - … et + en analysant le script.

Par où commencer ?

- 1) Outil gestion versions code sources
  - Lieu unique de partage
  - Retour arrières, snapshots, branches…
- 2) Tests automatisés
  - Chaque développeur



- 3) Scripts
  - Coté serveur pour automatiser (Ex : crontab)
- 4)Outils de communication
  - Mail, Tél, Rss…

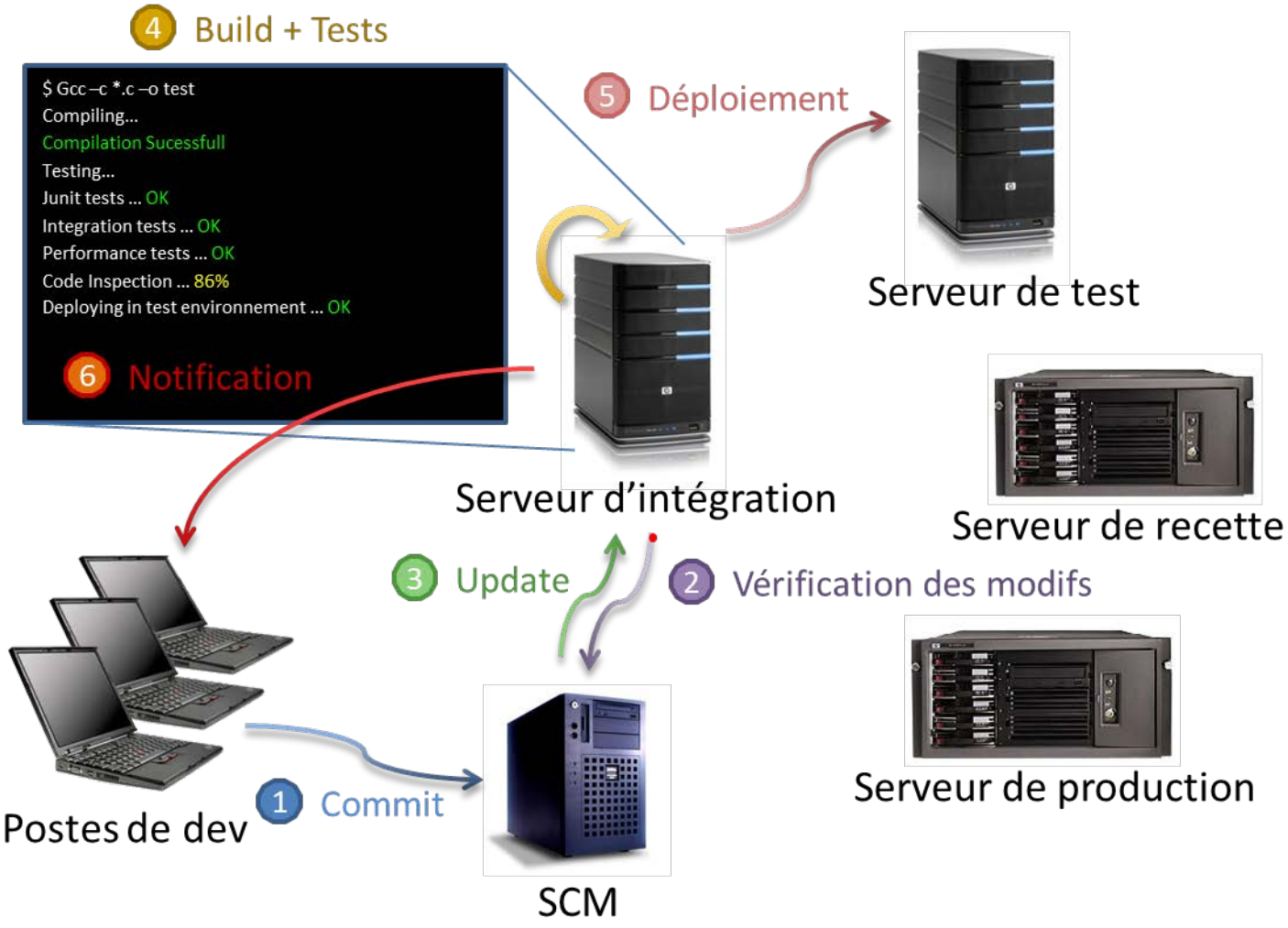
Architecture d’un logiciel d’intégration

Un fonctionnement actif

- Les développeurs « committent »
- Le serveur d'intégration surveille le serveur SCM (Cron)

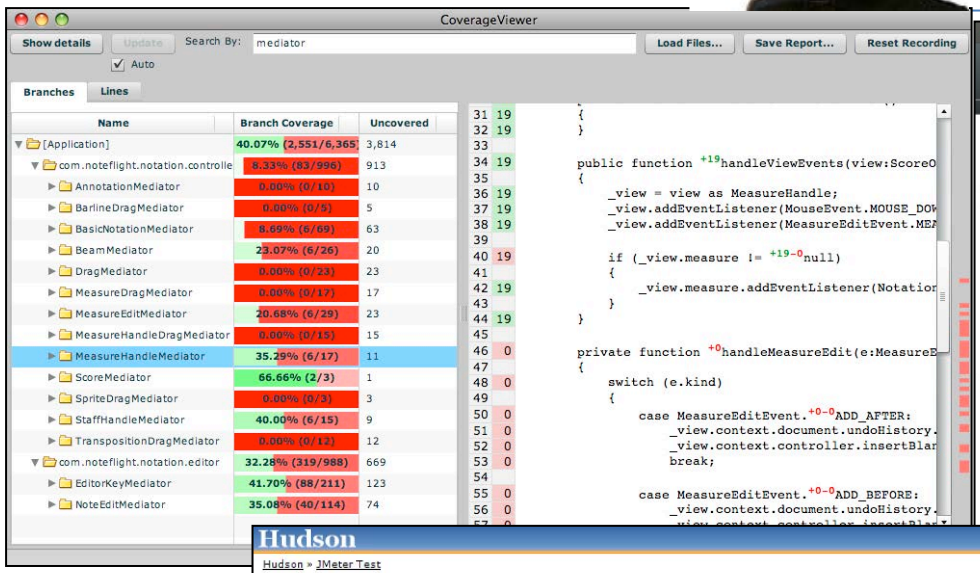
Cas d'utilisation

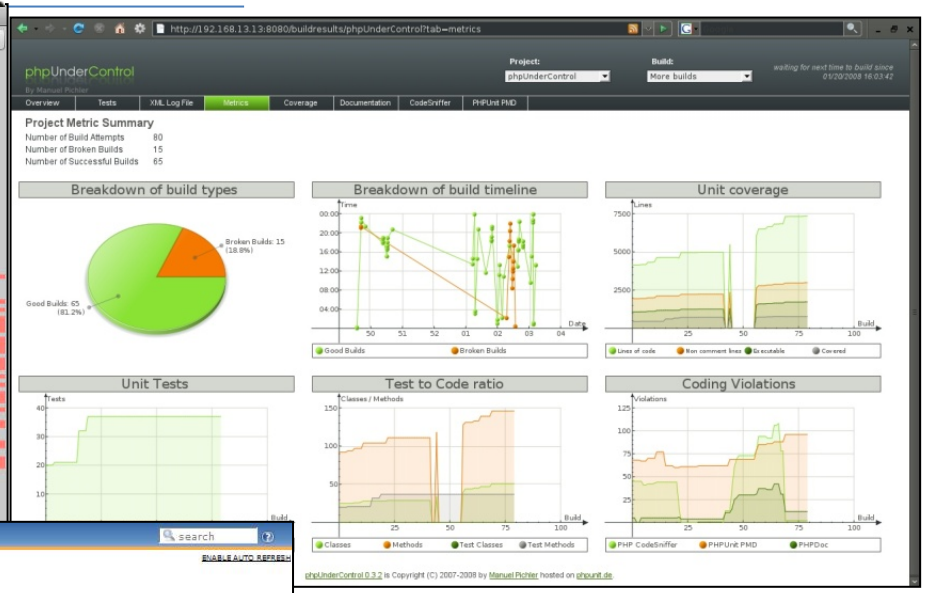
Le développeur soumet une modification

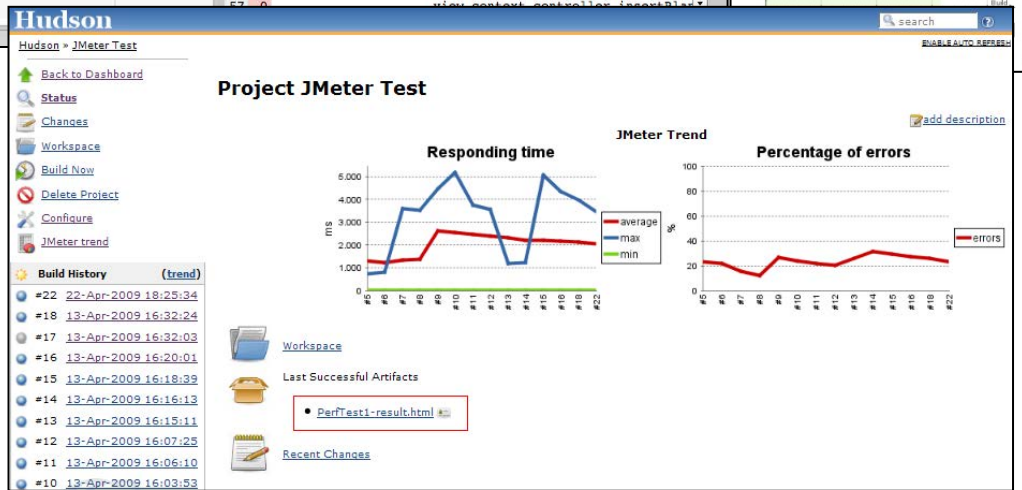


Cas d'utilisation

Le chef de projet analyse le reporting







Les technologies existantes

- Hudson, jenkins
- CruiseControl / CruiseControl.NET
- Apache Continuum
- QuickBuild (open-source: LuntBuild)
- Et beaucoup d’autres …