

Méthodes de Développement Industriel (MDI)

Mathieu Acher

<http://www.mathieuacher.com>

Associate Professor

University of Rennes 1

Objectifs de MDI

- Méthodes de développement industriel (MDI)
 - En fait: génie logiciel / software engineering
 - Comment développer des systèmes logiciels de plus en plus complexe?
- #1 Prendre conscience de la complexité des systèmes logiciels actuels et à venir
 - Les enjeux et l'impact sur le métier
- #2 Modélisation
 - UML, SysML
- **#3 Design patterns, refactoring, test**
 - **OO avancé**
- **#4 Méthodes**

Tête la première Design Patterns



O'REILLY*

Eric Freeman & Elisabeth Freeman
avec Kathy Sierra & Bert Bates
Traduction de Marie-Océane Baland



Un café?

Decorator

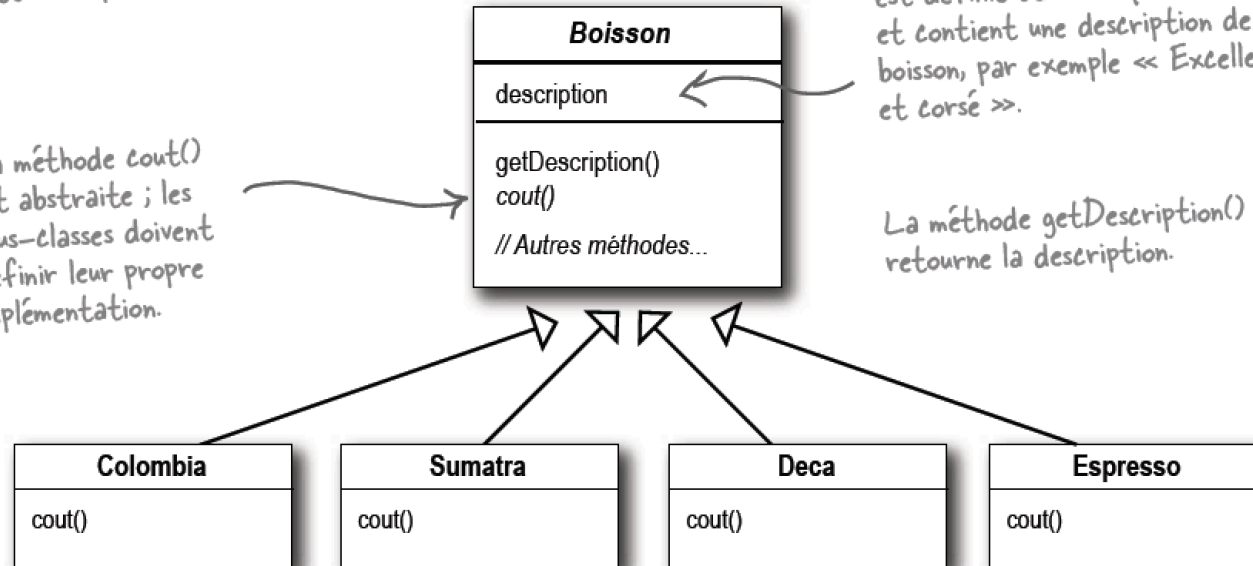


Boisson est une classe abstraite sous-classée par toutes les boissons proposées dans le café.

La méthode `cout()` est abstraite ; les sous-classes doivent définir leur propre implémentation.

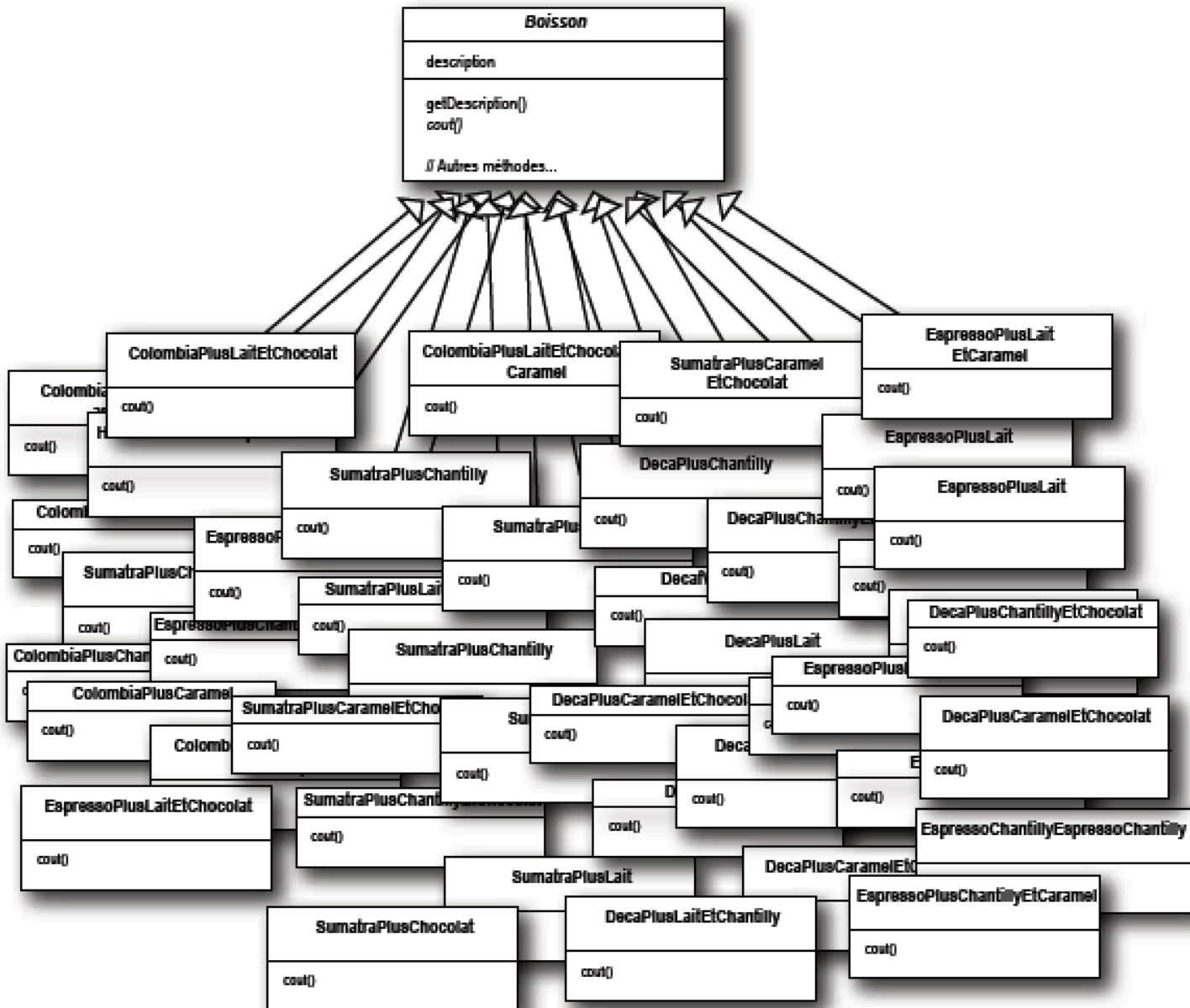
La variable d'instance `description` est définie dans chaque sous-classe et contient une description de la boisson, par exemple « Excellent et corsé ».

La méthode `getDescription()` retourne la description.



Chaque sous-classe implémente `cout()` pour retourner le coût de la boisson.

#1 Héritage

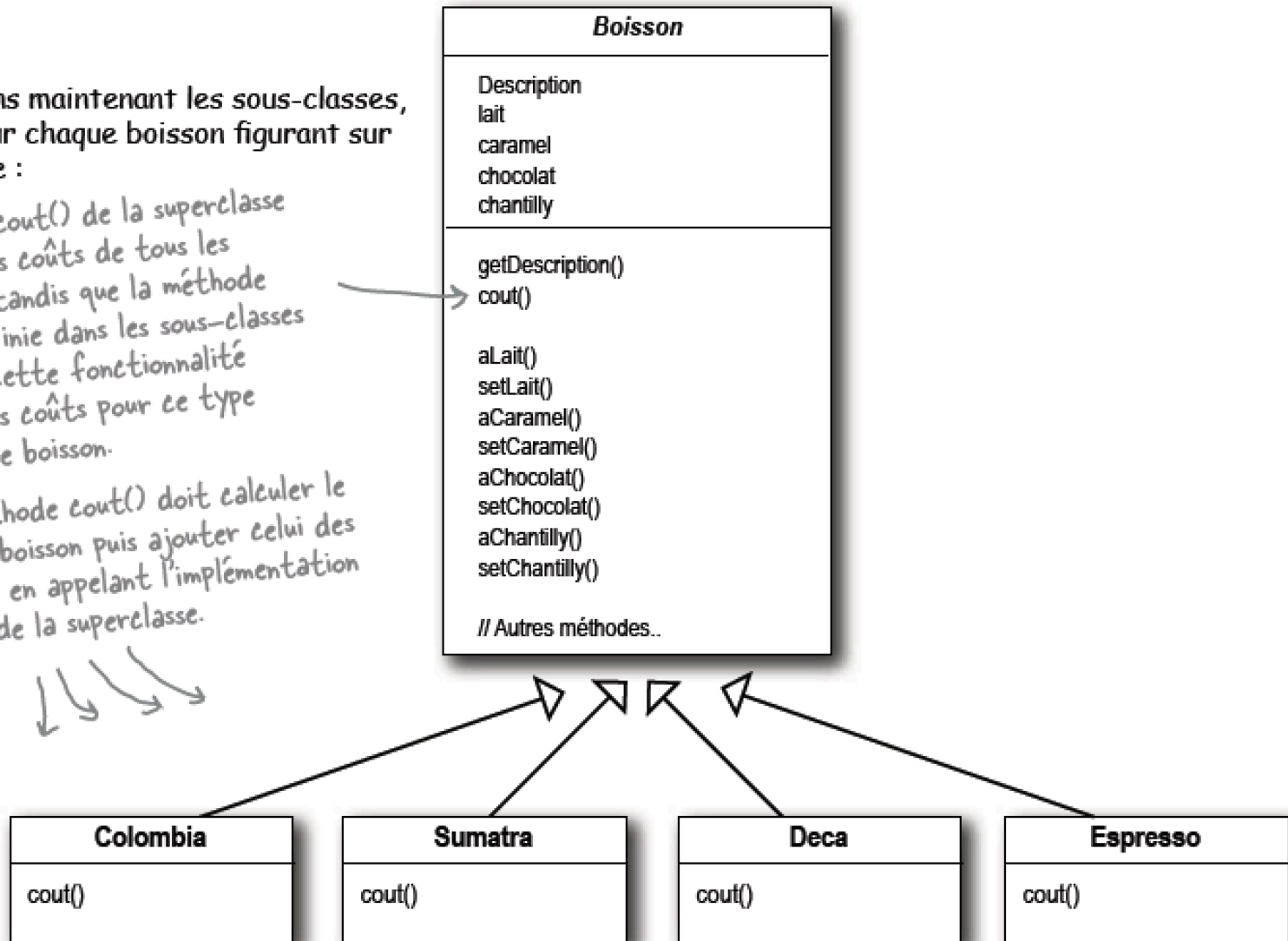


#2 Une autre solution

Ajoutons maintenant les sous-classes, une pour chaque boisson figurant sur la carte :

La méthode `cout()` de la superclasse va calculer les coûts de tous les ingrédients, tandis que la méthode `cout()` redéfinie dans les sous-classes va étendre cette fonctionnalité et inclure les coûts pour ce type spécifique de boisson.

Chaque méthode `cout()` doit calculer le coût de la boisson puis ajouter celui des ingrédients en appelant l'implémentation de `cout()` de la superclasse.

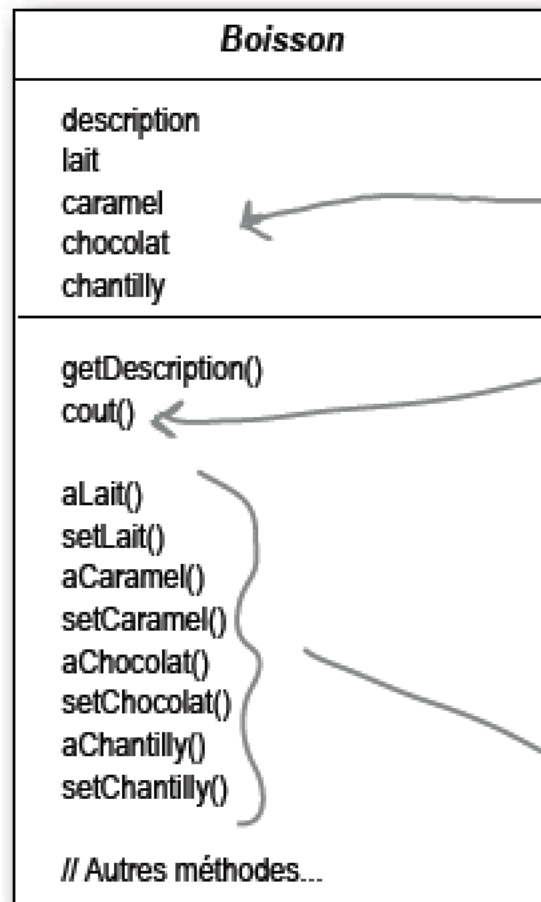


Augmentation du prix des ingrédients oblige à **modifier le code existant**.

Quid des nouveaux ingrédients?

Quid des nouvelles boissons?

Les classes doivent être ouvertes à l'extension, mais fermées à la modification.

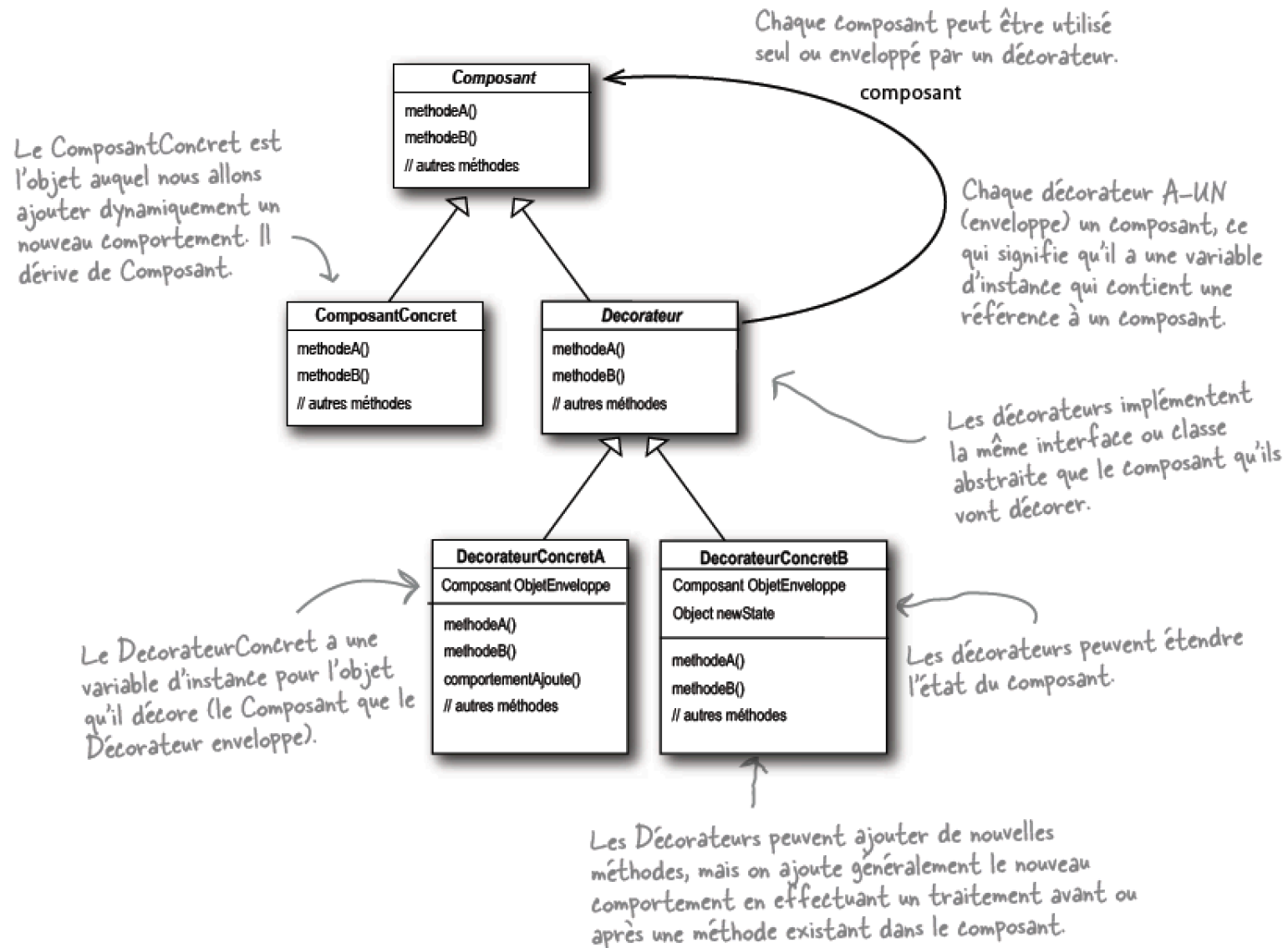


Nouvelles valeurs booléennes pour chaque ingrédient.

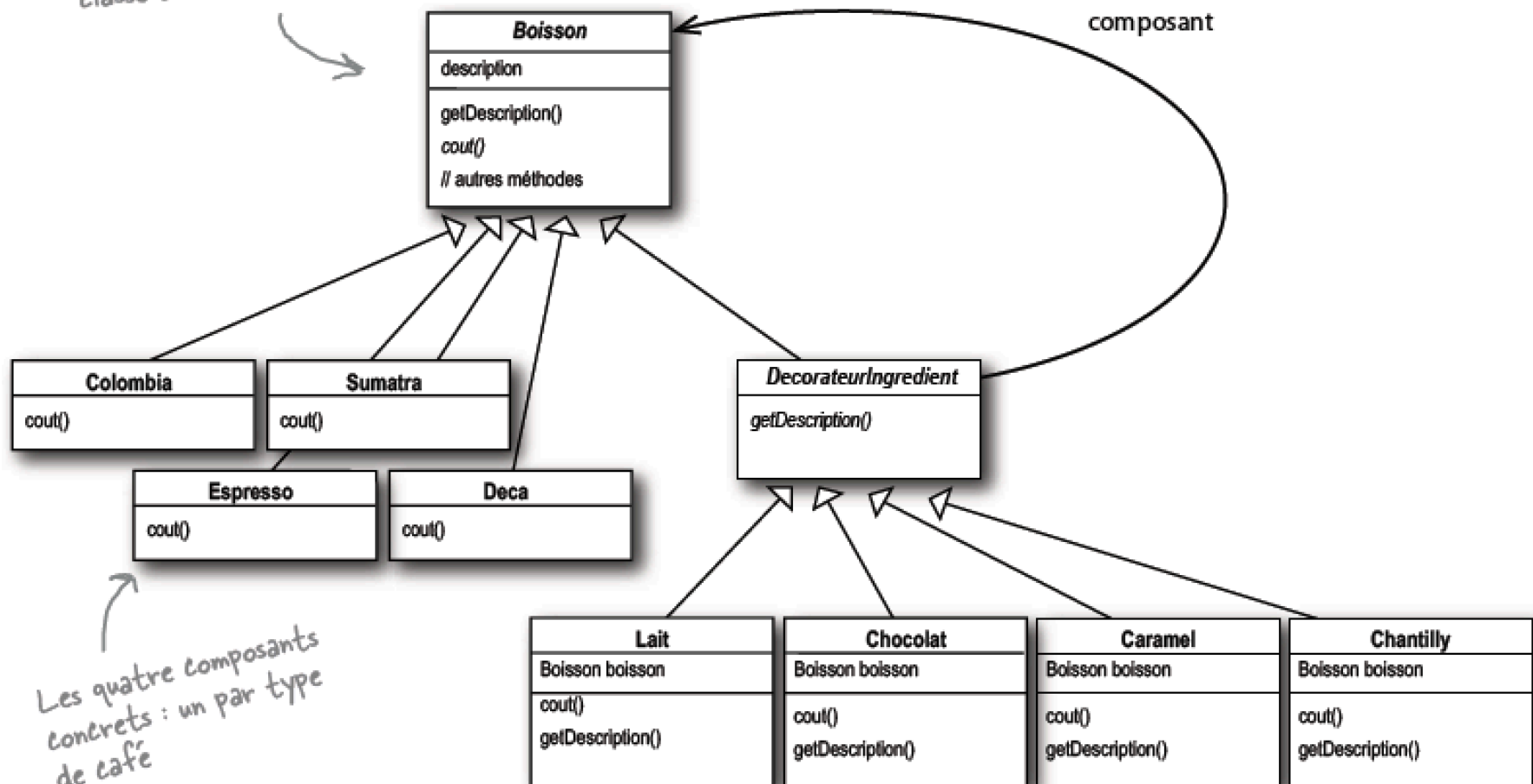
Maintenant, nous implémentons `cout()` dans `Boisson` (au lieu qu'elle demeure abstraite), pour qu'elle puisse calculer les coûts associés aux ingrédients pour une instance de boisson donnée. Les sous-classes redéfiniront toujours `cout()`, mais elles appelleront également la super-version pour pouvoir calculer le coût total de la boisson de base plus celui des suppléments.

Ces méthodes lisent et modifient les valeurs booléennes des ingrédients.

Le pattern **Décorateur** attache dynamiquement des responsabilités/fonctionnalités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour étendre les fonctionnalités.



Boisson joue le rôle de notre classe abstraite, Composant.



Les quatre composants concrets : un par type de café

Et voici nos décorateurs pour les ingrédients. Remarquez qu'ils ne doivent pas seulement implémenter `cout()` mais aussi `getDescription()`. Nous verrons pourquoi dans un moment...

```
public abstract class Boisson {  
    String description = "Boisson inconnue";  
  
    public String getDescription() {  
        return description;  
    }  
  
    public abstract double cout();  
}
```


Boisson est une classe abstraite
qui possède deux méthodes :
getDescription() et cout().

getDescription a déjà été
implémentée pour nous, mais
nous devons implémenter
cout() dans les sous-classes.

```
public class Colombia extends Boisson {  
    public Colombia() {  
        description = "Pur Colombia";  
    }  
  
    public double cout() {  
        return .89;  
    }  
}
```




D'abord, comme elle doit être interchangeable avec une Boisson, nous étendons la classe Boisson.



```
public abstract class DecorateurIngredient extends Boisson {  
    public abstract String getDescription();  
}
```

Nous allons aussi faire en sorte que les ingrédients (décorateurs) réimplémentent tous la méthode getDescription(). Nous allons aussi voir cela dans une seconde...



Chocolat est un décorateur : nous étendons DecorateurIngredient.

Souvenez-vous que DecorateurIngredient étend Boisson.

Nous allons instancier Chocolat avec une référence à une Boisson en utilisant :

```
public class Chocolat extends DecorateurIngredient {  
    Boisson boisson;  
  
    public Chocolat(Boisson boisson) {  
        this.boisson = boisson;  
    }  
  
    public String getDescription() {  
        return boisson.getDescription() + ", Chocolat";  
    }  
  
    public double cout() {  
        return .20 + boisson.cout();  
    }  
}
```

(1) Une variable d'instance pour contenir la boisson que nous enveloppons.

(2) Un moyen pour affecter à cette variable d'instance l'objet que nous enveloppons. Ici, nous allons transmettre la boisson que nous enveloppons au constructeur du décorateur.

Nous devons maintenant calculer le coût de notre boisson avec Chocolat. Nous déléguons d'abord l'appel à l'objet que nous décorons pour qu'il calcule son coût. Puis nous ajoutons le coût de Chocolat au résultat.

La description ne doit pas comprendre seulement la boisson – disons « Sumatra » – mais aussi chaque ingrédient qui décore la boisson, par exemple, « Sumatra, Chocolat ». Nous allons donc déléguer à l'objet que nous décorons pour l'obtenir, puis ajouter « Chocolat » à la fin de cette description.


```
public class StarbuzzCoffee {
```

```
    public static void main(String args[]) {  
        Boisson boisson = new Espresso();  
        System.out.println(boisson.getDescription()  
            + " €" + boisson.cout());
```

```
        Boisson boisson2 = new Sumatra();
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chantilly(boisson2);
```

```
        System.out.println(boisson2.getDescription()  
            + " €" + boisson2.cout());
```

```
        Boisson boisson3 = new Colombia();
```

```
        boisson3 = new Caramel(boisson3);
```

```
        boisson3 = new Chocolat(boisson3);
```

```
        boisson3 = new Chantilly(boisson3);
```

```
        System.out.println(boisson3.getDescription()  
            + " €" + boisson3.cout());
```

```
    }
```

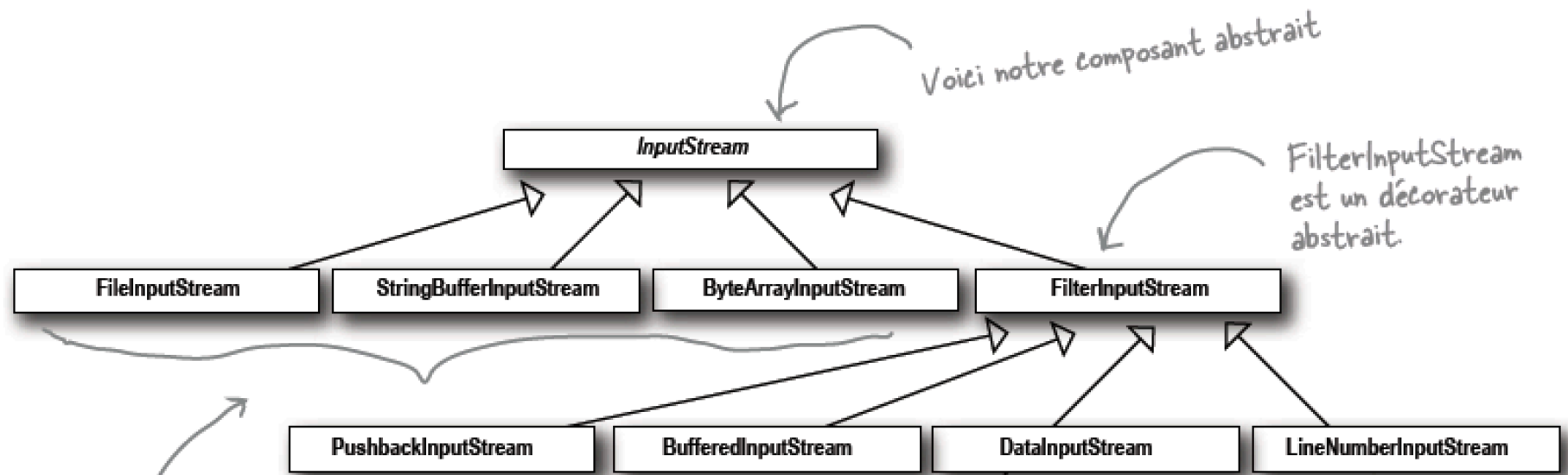
```
}
```

Commander un espresso, pas d'ingrédients
et afficher sa description et son coût.

Créer un objet Sumatra.
L'envelopper dans un Chocolat.

L'envelopper dans un second Chocolat.
L'envelopper de Chantilly.

Enfin nous servir un Colombia avec
Caramel, Chocolat et Chantilly.



Outils et Méthodes

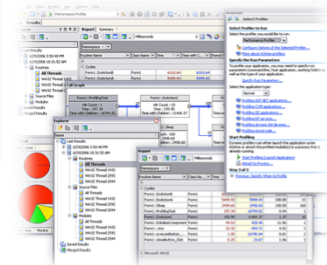
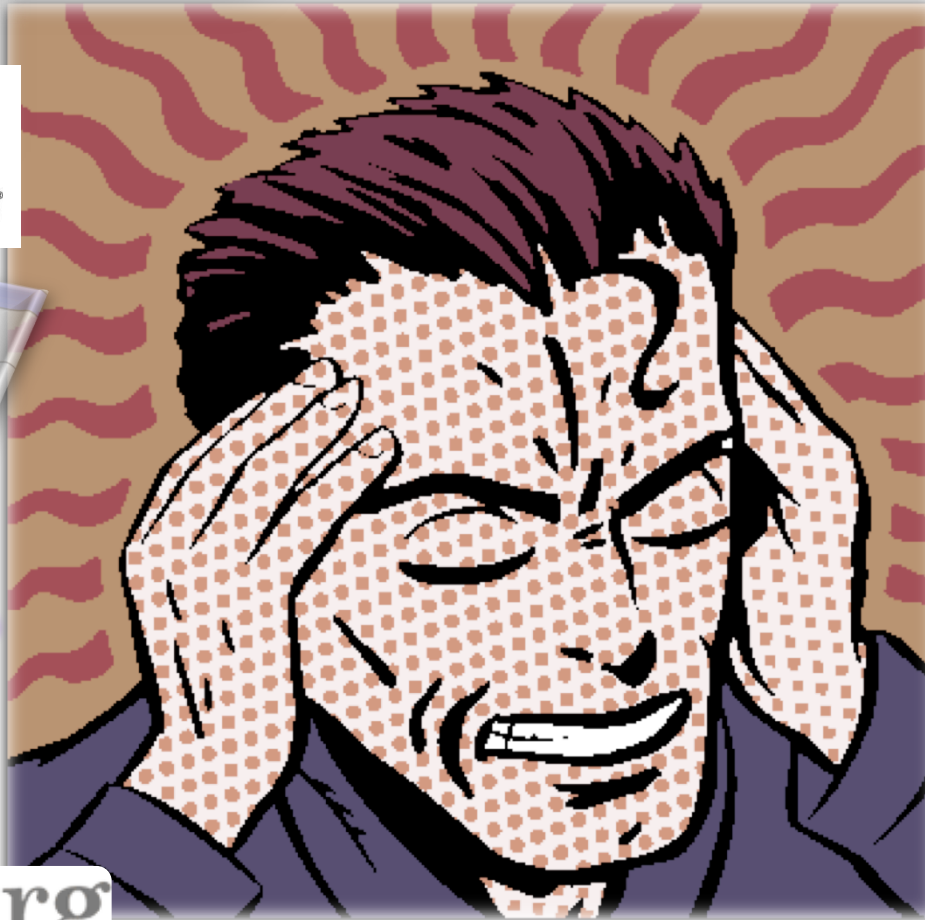
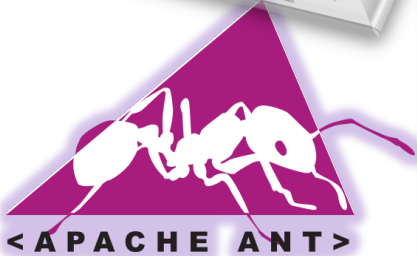
Software Engineering



Visual Basic



Code::Blocks
Studio



Documentation and Source Code (for C1)

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 <code>Action</code> interface provides a useful extension to the <code>ActionListener</code> interface in cases where the same functionality may be accessed by several controls.
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 <code>KeySelectionManager</code> .
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.
MenuItem	Any component that can be placed into a menu should implement this interface.
MutableComboBoxModel	A mutable version of <code>ComboBoxModel</code> .
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 <code>getXXX(key)</code> 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 <code>getXXX(key)</code> 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 `JRootPane` instead of 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 not throw an 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
 - Design pattern!
- Minimizes technical debt
- Keep development at speed
- To make the software easier to understand
- To help find bugs
- To “Fix broken windows”

Non exhaustive (code smell)

(and not necessarily smells in all situations)

- Duplicated code
- Feature Envy
- Inappropriate Intimacy
- Comments
- Long Method
- Long Parameter List
- Switch Statements
- Improper Naming

Code Smell examples (1)

```
public void display(String[] names) {  
    System.out.println("-----");  
    for(int i=0; i<names.length; i++){  
        System.out.println(" + " + names[i]);  
    }  
    System.out.println("-----");  
}
```

```
public void listMember(String[] names) {  
    System.out.println("List all member: ");  
    System.out.println("-----");  
    for(int i=0; i<names.length; i++){  
        System.out.println(" + " + names[i]);  
    }  
    System.out.println("-----");  
}
```

Duplicated code

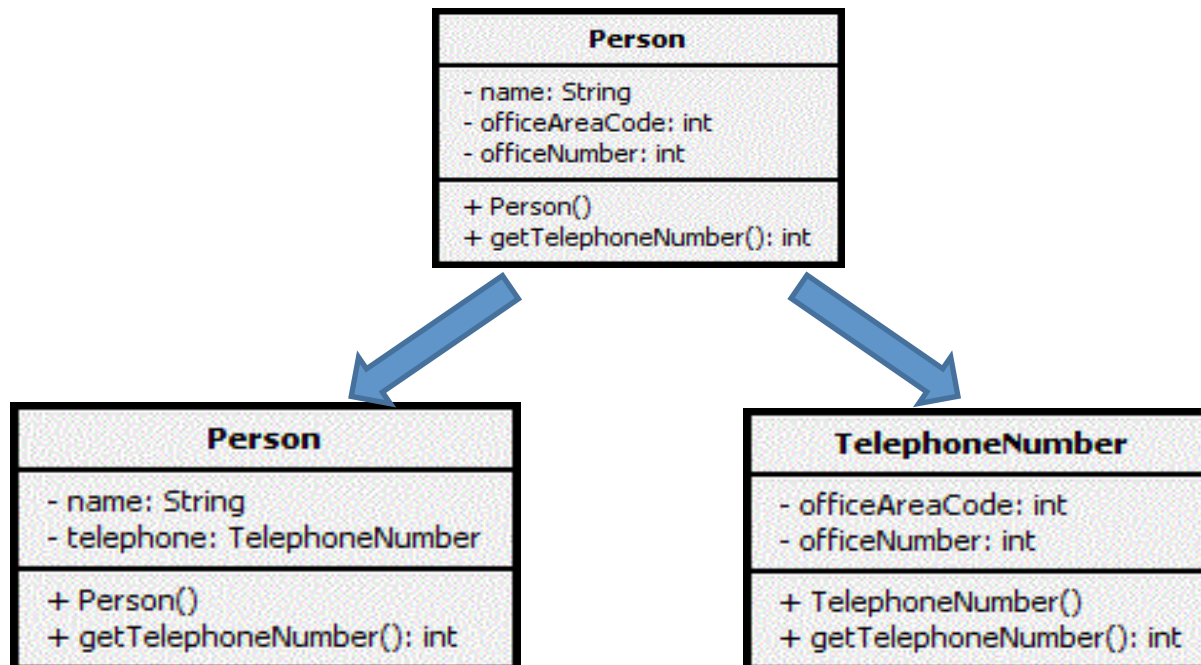
Code Smell examples (2)

```
public String formatStudent( int id,  
                             String name,  
                             Date dob,  
                             String province,  
                             String address,  
                             String phone ){  
  
    //TODO:  
    return null;  
}
```

Long list of parameters

Improving design

- Move Method or Move Field – move to a more appropriate Class or source file
- Rename Method or Rename Field – changing the name into a new one that better reveals its purpose
 - Pull Up – in OOP, move to a superclass
 - Push Down – in OOP, move to a subclass



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;
        for (int i = 0; i <= 100; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

Problems Javadoc Declaration Console Error Log

<terminated> MyFirstClass [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

Hello Eclipse!

5050

Extract Method

Method name:

Access modifier: ☐ public ☐ protected ☐ default ☒ private

Parameters:

Type	Name
int	sum

☐ Declare thrown runtime exceptions

☐ Generate method comment

☐ Replace additional occurrences of statements with method

Method signature preview:

```
private static int calculateSum(int sum)
```

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

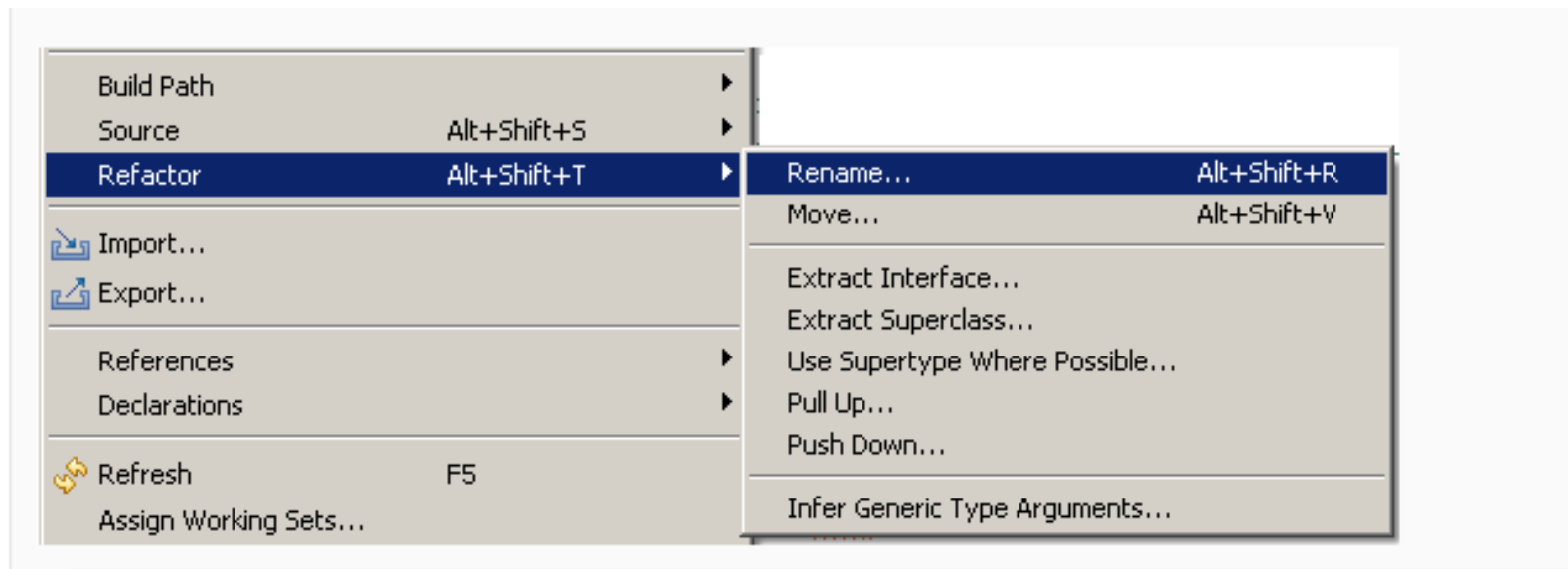
public class MyFirstClass {

    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



You have constructors on subclasses with mostly identical bodies.

Create a superclass constructor; call this from the subclass methods.

Pull Up Constructor Body

You

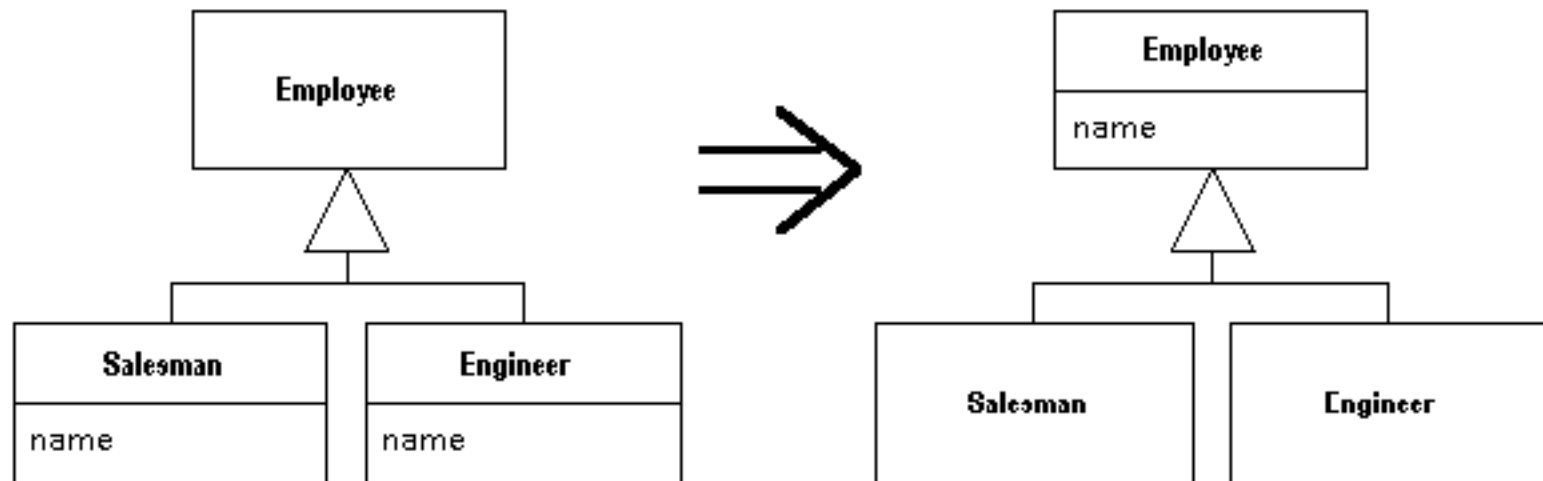
Create

```
class Manager extends Employee...  
    public Manager (String name, String id, int grade) {  
        _name = name;  
        _id = id;  
        _grade = grade;  
    }
```

```
    public Manager (String name, String id, int grade) {  
        super (name, id);  
        _grade = grade;  
    }
```

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

- User 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 (see javadoc)
- 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 »

This is the text to dictate...

Copy

Insert

Editeur (1) [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (4 avr. 2014 08:04:49)

Editeur

Selection courante: 0 0

Deplacer le (d)ebut de la selection

Deplacer la (f)in de la selection

(C)opier

Co(l)ler

(I)nsrer du texte

(Q)uitter

Votre choix:



Copier

Coller

Inserer

Rpter

Annuler



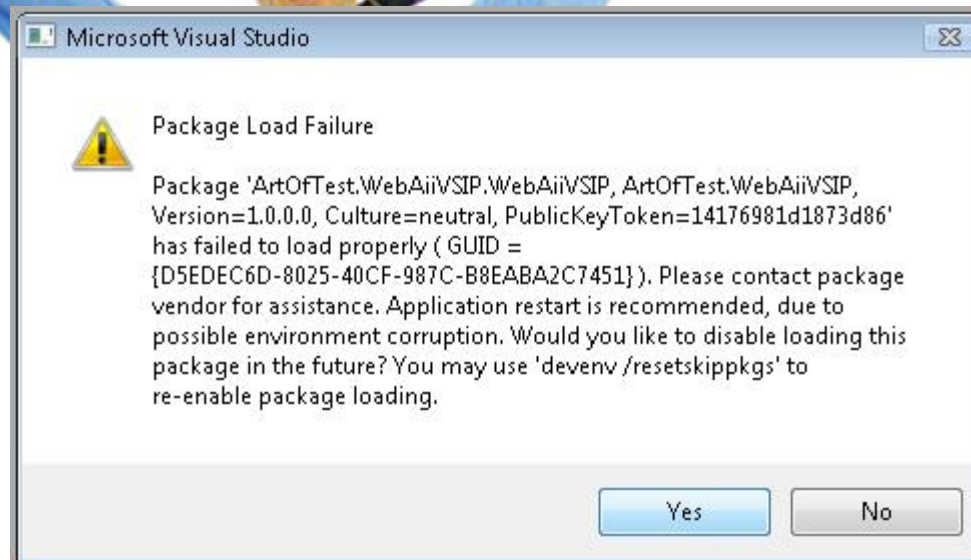
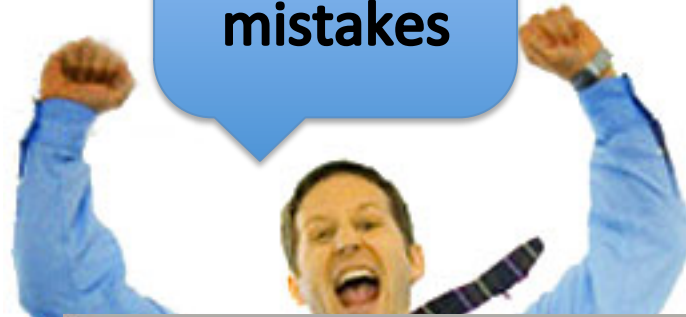
This part is largely
inspired by Thomas
Zimmermann slides

Dijkstra



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

**I don't
make
mistakes**



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 = maintenance »

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



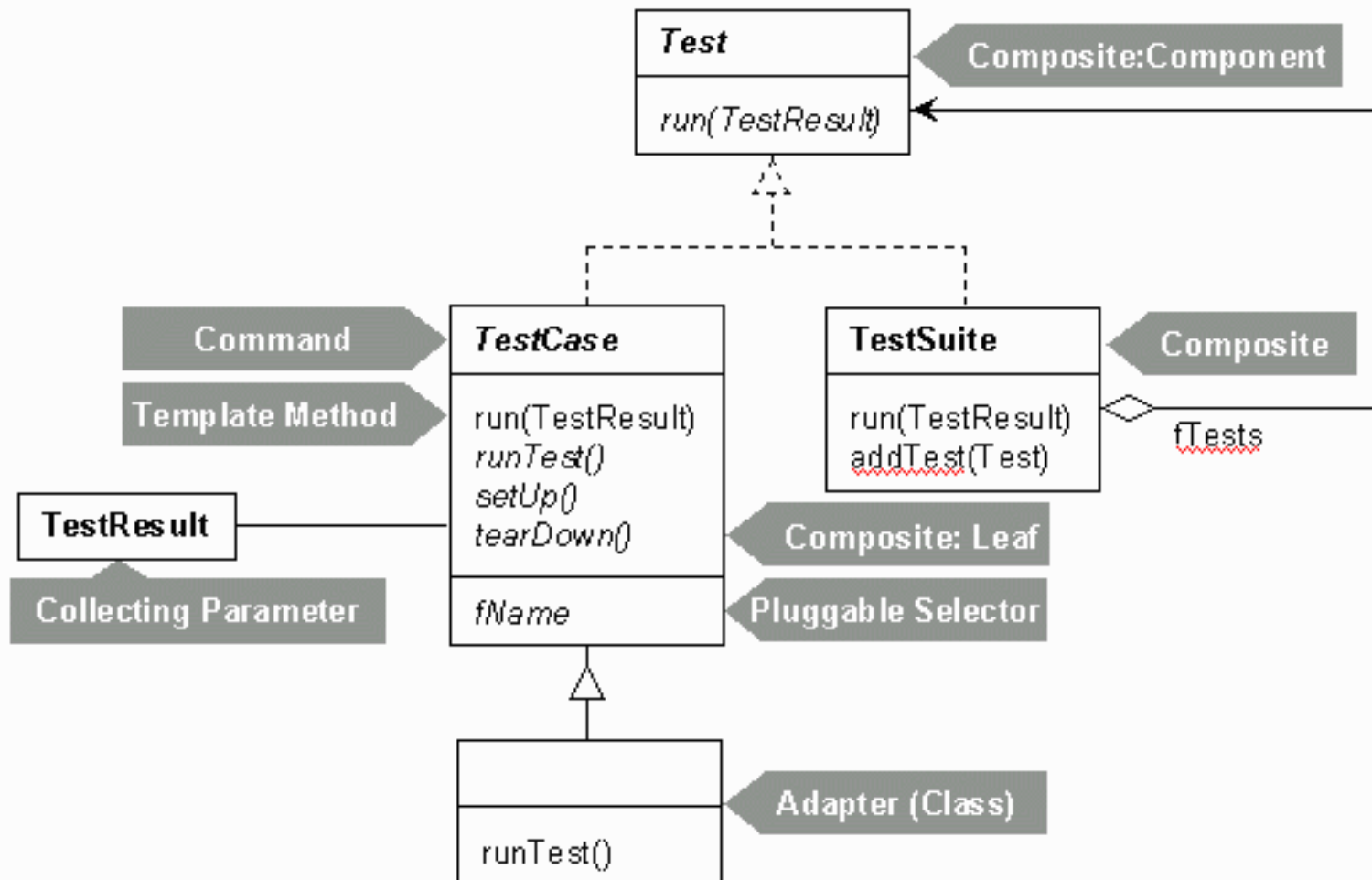
Unit testing on individual units of source code (=smallest testable part).

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

Price: **CDN\$ 10.94**
In Stock
Ships from and sold by Amazon.ca

Quantity:

 Add to Shopping Cart


or
[Sign in](#) to turn on 1-Click ordering.

1 Add product

 **Shopping Cart** Already a customer?
[Sign in](#)

 See more items like those in your cart

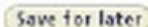
Subtotal: CDN\$ 10.94

Make any changes below? 

Shopping Cart Items--To Buy Now

Item added on
April 26 2007

Harry Potter and the Half-Blood Prince (Book 6) [Adult Edition] - J. K. Rowling; **Mass Market Paperback**
In Stock





Price: **CDN\$ 10.94** **Qty:**

CDN\$ 10.94
You Save:
CDN\$ 4.05
(27%)

2 Remove 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(String name) {
        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*

`assertNotNull(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

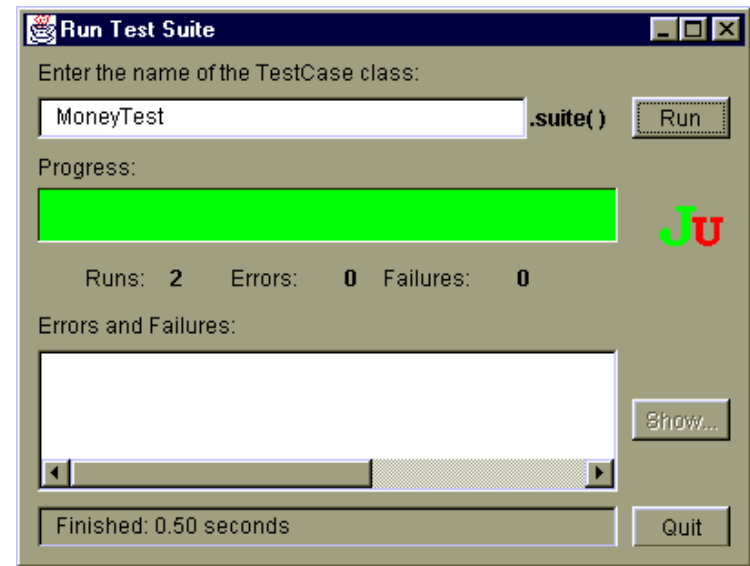
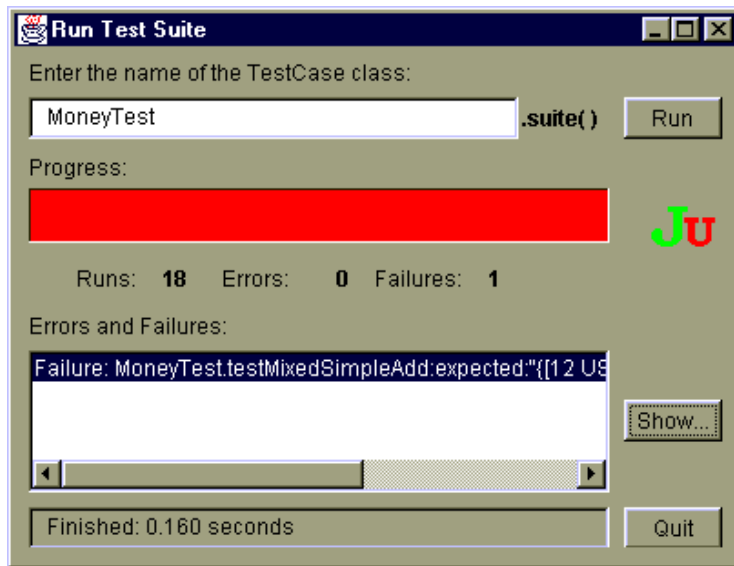
```
// Tests removing a product from the cart.  
public void testProductRemove() throws NotFoundException {  
    Product book = new Product("Harry Potter", 23.95);  
    _bookCart.removeItem(book);  
  
    assertTrue(!_bookCart.contains(book));  
  
    double expected = 23.95 - book.getPrice();  
    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 and 4 <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://membres-liglab.imag.fr/donsez/cours/junit.pdf>

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.**



Documenting,
Testing,
Design Patterns,
Refactoring,
Debugging

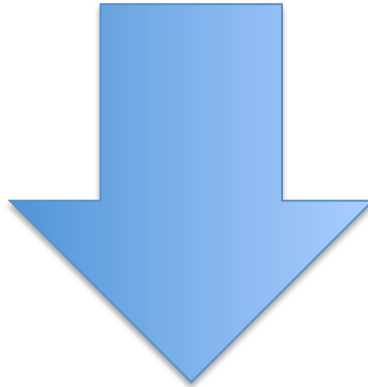
#1 What is the link?

- Documenting
 - Understanding (readability, maintainability)
- Refactoring
 - Improving the design (readability, maintainability, extensibility)
- The activity of documenting can somehow be replaced 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) && // browser 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;

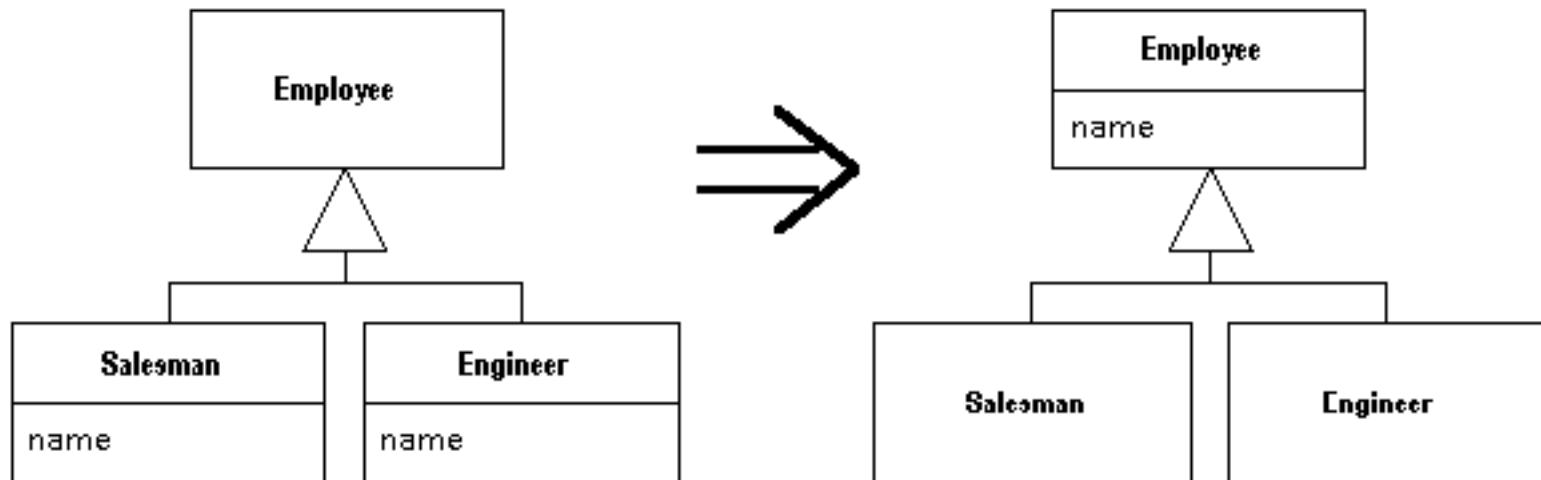
if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
    // do something
}
```

#2 What is the link?

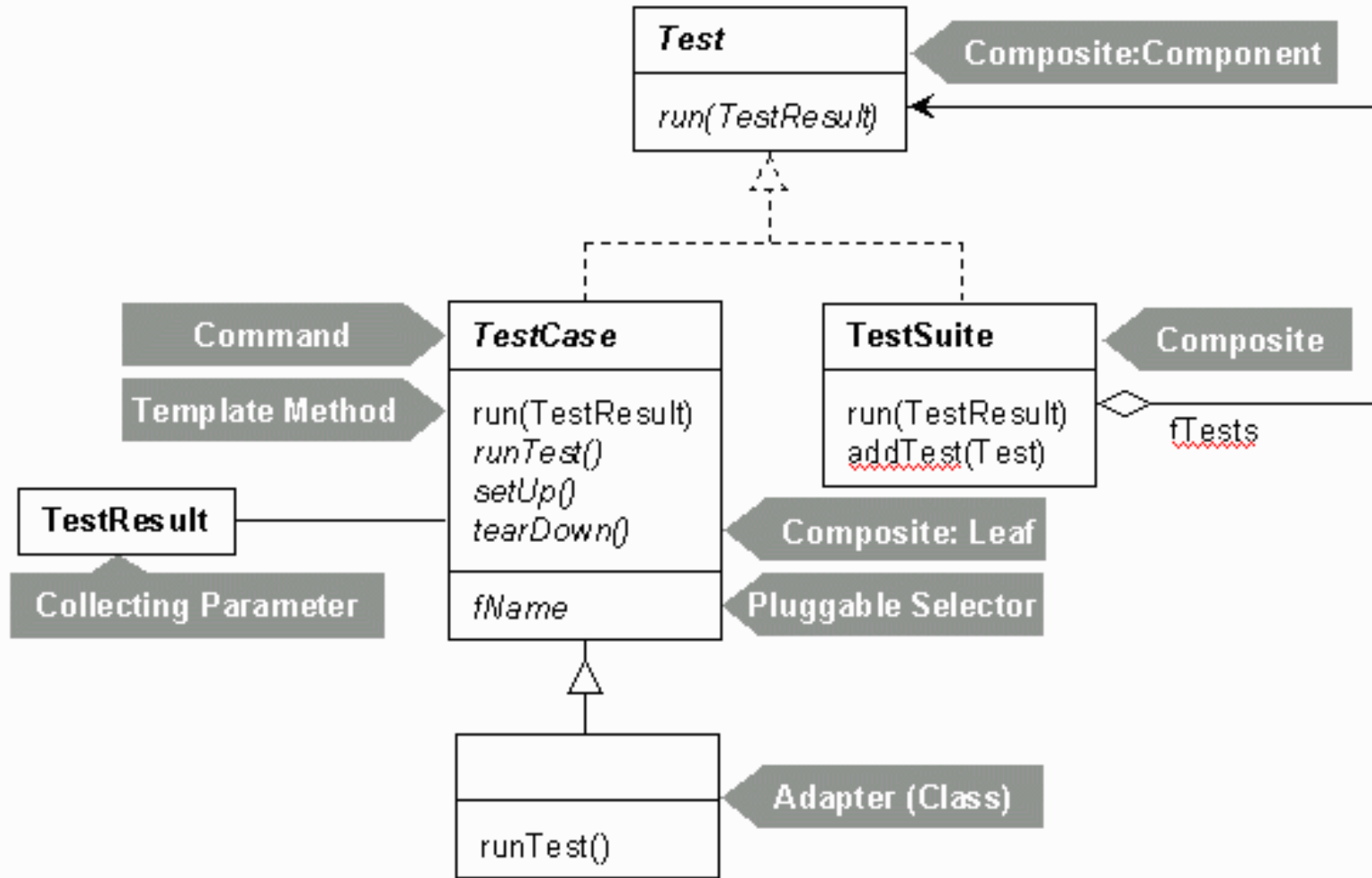
Design patterns: there are refactorings

Two subclasses have the same field.

Move the field to the superclass.



JUnit and... Design patterns



Worth reading!

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

#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.**

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
Understandability
Maintainability

Design

Document, refactor... Execute your tests... Debug.. Write test..

Documenting

And so on!

Debugging

Refactoring

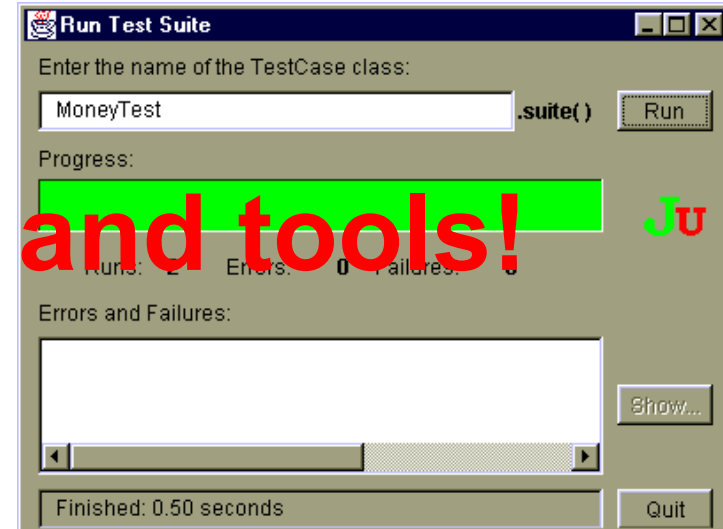
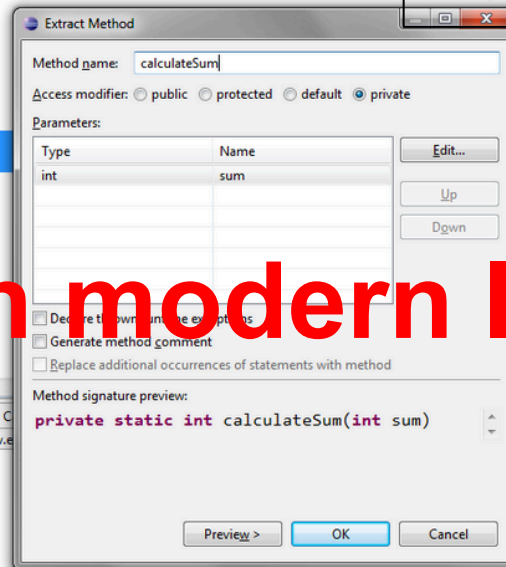
Testing

With modern IDE and tools!

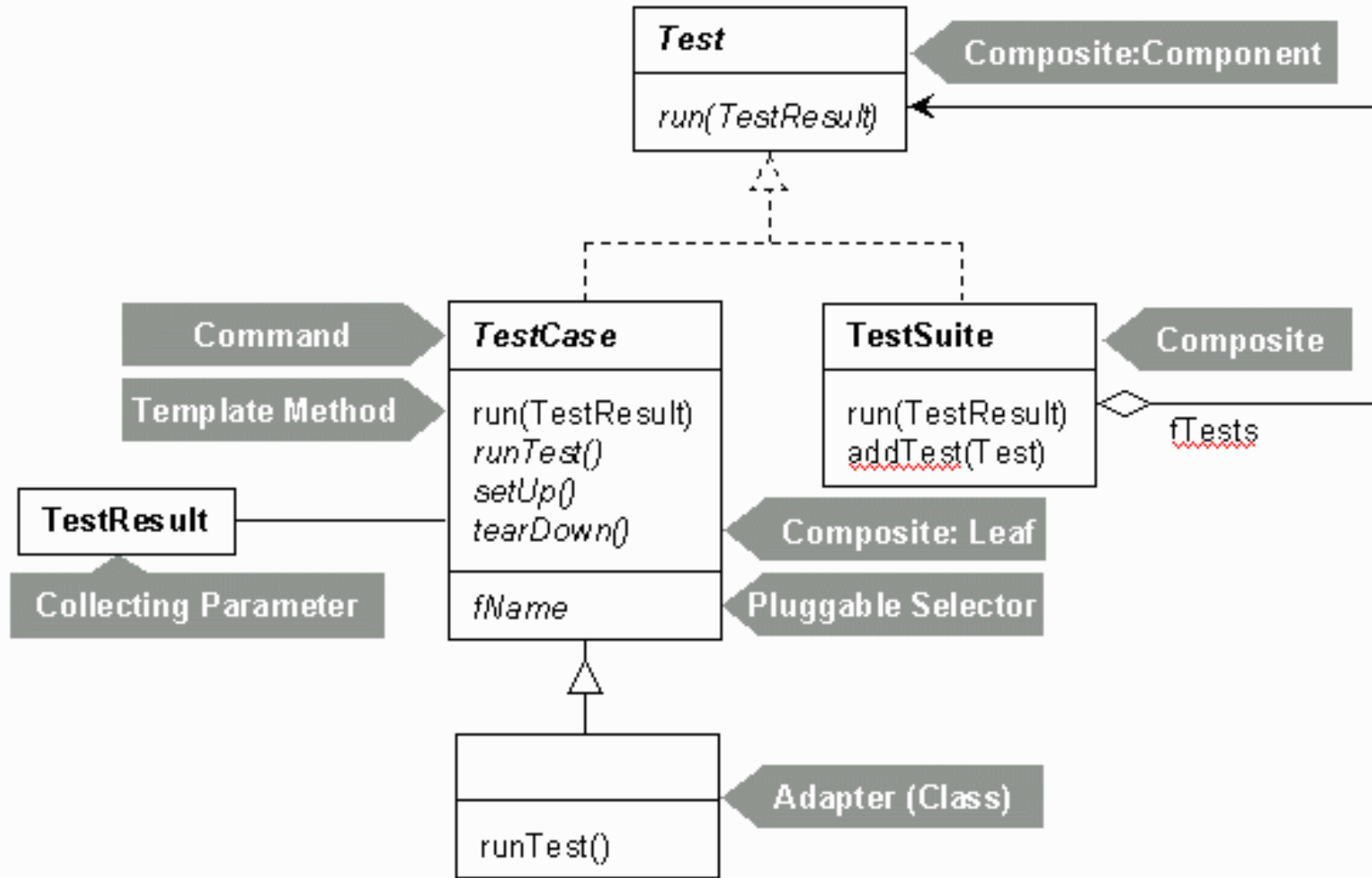
```
package de.vogella.eclipse.ide.first;

public class MyFirstClass {

    public static void main(String[] args) {
        System.out.println("Hello Eclipse!");
        int sum = 0;
        for (int i = 0; i <= 100; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```



JUnit and... Design patterns



Worth reading!

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

Et dans le futur ? *(from E. Gamma)*

a new categorization

▪ Core

- Composite
- Strategy
- State
- Command
- Iterator
- Proxy
- Template Method
- Facade
- *Null Object*



▪ Creational

- Factory method
- Prototype
- Builder
- *Dependency Injection*

▪ Peripheral

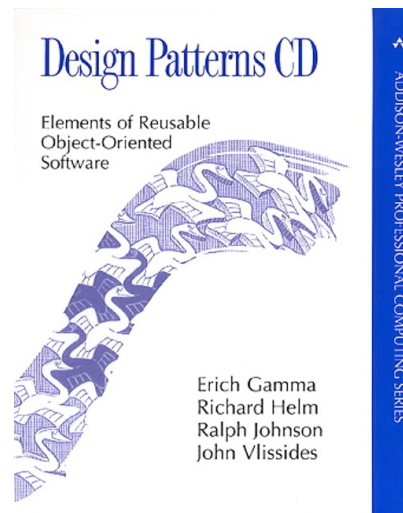
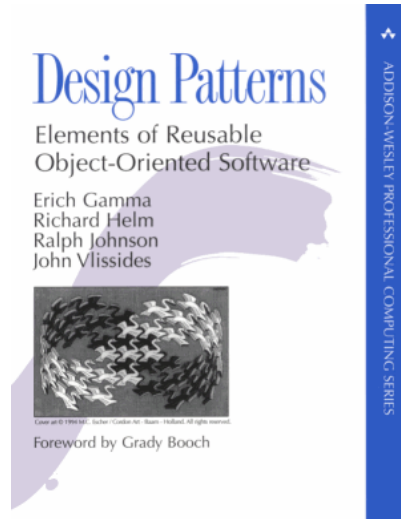
- Abstract Factory (peripheral)
- Memento
- Chain of responsibility
- Bridge
- Visitor
- *Type Object*
- Decorator
- Mediator
- Singleton
- *Extension Objects*

▪ Other (Compound)

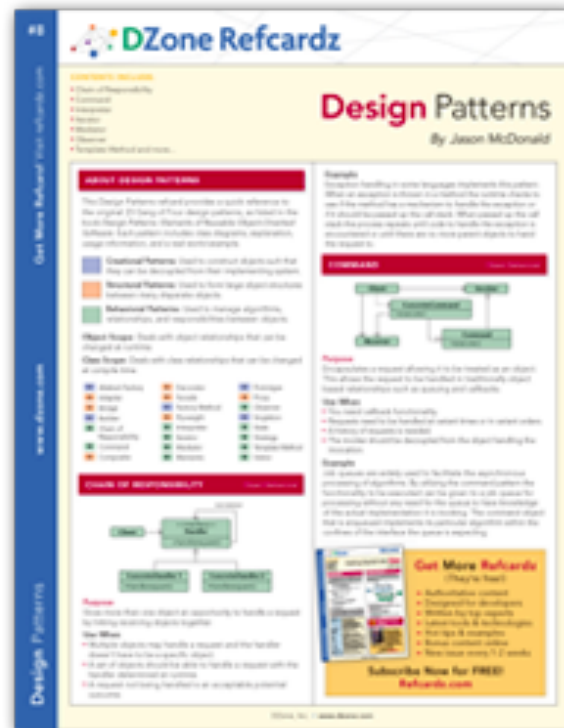
- Interpreter
- Flyweight



References



References



<http://refcardz.dzone.com/refcardz/design-patterns>