

# Les livrables

La liste des livrables et leur calendrier de rendu constituent l'axe directeur du projet logiciel :

- documentation de conception (ou spécification fonctionnelle des classes),
- code source documenté avec documentation d'API,
- rapport de codage, rapport de test, rapport de mesures,
- exécutables et bibliothèques réutilisables (JAR),
- documentation d'installation,
- documentation de maintenance.

Pour chaque livrable, il faut identifier les objectifs (contenu) et la cible (public utilisateur) et adapter la rédaction en conséquence : intérêt de maintenir un site (Web) de projet.

# Documentation et approche documentaire

# Documentation embarquée

Les inconvénients de la séparation source/documentation :

- rédaction de la doc. négligée et reportée,
- lors de la maintenance le suivi de la doc. n'est pas assuré.

**La seule façon de maintenir la documentation à jour  
est d'imposer que sa manipulation se fasse  
simultanément à celle du code.**

Dans la situation actuelle, placer toute la documentation technique d'un module de programme dans le source de ce programme (embarquement)

# Cohérence de la documentation

Dès qu'une information est dupliquée, le risque est pris d'une incohérence future.

La recherche d'**unicité de définition** de l'information doit être constante.

Le système de documentation doit mettre en oeuvre divers mécanismes pour permettre cette unicité :

- limitation du nombre de sources,
- possibilité de capter l'information qui est dans le code,
- utilisation de liens de type hypertexte pour permettre le contrôle de la cohérence quand la duplication ne peut être évitée

# Organisation de la documentation

- Utiliser **JavaDoc**,
- Commentaire **header structuré** pour la méta-information et le copyright (en XML)
- Ne pas négliger le **commentaire de classe**, toujours y inclure un **exemple d'emploi**,
- Choix de **noms significatifs** pour les variables et les méthodes : évite de longs commentaires

# Organisation de la documentation

- Typer les commentaires :
  - commentaires **javadoc** (`/** ... */`) pour la description du protocole d'emploi,
  - **blocs de commentaires** (`/* ... */`) positionnés avant le code qu'ils commentent,
  - commentaires de **fin de ligne** (`// ...`) pour marquer la structure,
  - **masquage de code** (`// ...` en début de ligne) pour conserver du code détruit lors d'une modification
  - commentaires techniques après la fin de la classe

# Commentaires Javadoc

- Utiliser des commentaires Javadoc pour documenter les différentes parties du code :

- Classes
- Méthodes
- Attributs

```
/**  
 * description du code à suivre, la description  
 * peut être sûr plusieurs lignes  
 * @param val1 description du paramètre val1  
 * @param val2 description du paramètre val2  
 * @return description de la valeur de retour  
 */
```

- Une documentation de la seule partie visible peut être produite
- Génération de la documentation avec l'outil Javadoc :  
**javadoc -d dest [package] sourcefiles**
- La documentation peut être générée automatiquement !

# Balises Javadoc

- **@author** auteur du code
- **@deprecated** indique que ce code est déprécié et qu'il ne doit plus être utilisé (certains IDEs avertissent le développeur en cas d'appel à un code déprécié)
- **@exception** ou **@throws** documente l'exception lancée par une méthode
- **@param** documente un paramètre de méthode, requis pour chaque paramètre
- **@return** documente la valeur de retour (le cas échéant)
- **@see** permet d'indiquer un lien vers un autre code documenté
- **@since** indique depuis quelle version de l'API ce code est présent
- **@version** indique la version du code



# Pratique de documentation

- Faire des commentaires courts et utiles,
- Adopter des formes de phrases adaptées aux familles de méthodes,
- Eviter les répétitions et les formules lourdes, ne pas plagier les contrats ou le code,
- Toujours écrire les commentaires avant le code qu'ils concernent,
- Dès qu'une formulation plus formelle est disponible, l'utiliser (contrats, assertions)
- Faire relire et contrôler les commentaires par un tiers,
- Maintenir une documentation extractible en permanence.

# Automatisation des tâches de développement

# ANT (Another Neat Tool)

- Projet open source de la fondation Apache  
<http://ant.apache.org>
- Objectif : automatisation des tâches dans le processus de développement logiciel :
  - compilation,
  - documentation,
  - tests,
  - archivage sous forme distribuable
- Exécution en ligne de commande ou via un IDE  
**ant [options] [target [target2 [target3] ...]]**

# Fonctionnement de ANT

- Les traitements à effectuer sont indiqués dans un fichier XML (build.xml)
- Un projet compte des cibles (target) correspondant à des activités : compilation, installation, exécution, ...
- Chaque cible est composée de tâches (task) correspondant à des commandes usuelles (javac, jar, copy, ...)

# Tâches dans ANT

Pas seulement javac ou javadoc !

Ant	Defaultexcludes	Javadoc/Javadoc2	Pvcs	
AntCall	Delete	Javah	Record	
ANTLR	Deltree	JDepend	Rename	
AntStructure	Depend	JJDoc	RenameExtensions	
AntVersion	Dependset	JJTree	Replace	
Apply/ExecOn	Diagnostics	Jlink	ReplaceRegExp	
Apt	Dirname	JspC	ResourceCount	
Attrib	Ear	JUnit	Retry	Translate
Augment	Echo	JUnitReport	RExec	Truncate
Available	Echoproperties	Length	Rmic	TStamp
Baseline	EchoXML	LoadFile	Rpm	Typedef
Bindtargets	EJB Tasks	LoadProperties	SchemaValidate	Unjar
BuildNumber	Exec	LoadResource	Scp	Untar
BUnzip2	Fail	Local	Script	Unwar
BZip2	Filter	MacroDef	Scriptdef	Unzip
Cab	FixCRLF	Mail	Sequential	Uptodate
Continuus/Synergy	FTP	MakeURL	ServerDeploy	Microsoft Visual
Tasks	GenKey	Manifest	Setproxy	SourceSafe Tasks
CvsChangeLog	Get	ManifestClassPath	SignJar	Waitfor
Checksum	GUnzip	MimeMail	Sleep	War
Chgrp	GZip	Mkdir	SourceOffSite	WhichResource
Chmod	Hostinfo	Move	Sound	Weblogic JSP
Chown	Image	Native2Ascii	Splash	Compiler
Clearcase Tasks	Import	NetRexxC	Sql	XmlProperty
Componentdef	Include	Nice	Sshexec	XmlValidate
Concat	Input	Parallel	Sshsession	XSLT/Style
Condition	Jar	Patch	Subant	Zip
Copy	Jarlib-available	PathConvert	Symlink	
Copydir	Jarlib-display	Perforce Tasks	Sync	
Copyfile	Jarlib-manifest	PreSetDef	Tar	
Cvs	Jarlib-resolve	ProjectHelper	Taskdef	
CVSPass	Java	Property	Telnet	
CvsTagDiff	Javac	PropertyFile	Tempfile	
CvsVersion	JavaCC	PropertyHelper	Touch	

# Exemple de build.xml

```
<project name="MyProject" default="dist" basedir=". ">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
```

# Exemple de build.xml

```
<target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>

<target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
</target>
</project>
```

# Autre exemple

```
<project>

  <target name="clean">
    <delete dir="build"/>
  </target>

  <target name="compile">
    <mkdir dir="build/classes"/>
    <javac srcdir="src" destdir="build/classes"/>
  </target>

  <target name="jar">
    <mkdir dir="build/jar"/>
    <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
      <manifest>
        <attribute name="Main-Class" value="oata.HelloWorld"/>
      </manifest>
    </jar>
  </target>

  <target name="run">
    <java jar="build/jar/HelloWorld.jar" fork="true"/>
  </target>

</project>
```

Exécution par : ant compile jar run



Tests

# Contrôles et Tests Unitaires

Any program feature  
without an automated test  
simply doesn't exist.

Ken Beck, p. 57, Extreme Programming Explained

# La gestion des tests

Une approche voisine de l'eXtreme Programming (XP)

ou développement agile :

- définition **préalable** des protocoles et des unités de test (lors de la spécification d'interface)
- utilisation très fréquente des **tests unitaires** en mode **vérification** pendant le développement incrémental du code
- validation finale de la classe sur la séquence de test complète et fabrication du **rapport de test**

# Ce qu'il ne faut pas faire ...

... que l'on fait pourtant !

- truffer le code de traces temporaires,
  - lancer l'application, appuyer sur quelques boutons pour «remuer» la fonctionnalité en cours,
  - vérifier dans les traces le bon fonctionnement de l'ensemble.
- 
- l'environnement de test est mal défini, on ne peut savoir ce qui a été testé vraiment, pas de trace
  - le test ne porte que sur la fonctionnalité en cours, sans voir d'éventuelles régressions ailleurs

# Buts du test

- Dans une optique d'assurance qualité, la conception des tests et les tests doivent d'abord s'attacher à la **prévention des bugs**
- S'ils ne permettent pas d'éviter tous les bugs, ils doivent découvrir les symptômes causés par les bugs et fournir un diagnostic pour les **corriger** aisément.

***Savoir qu'un programme est incorrect  
n'implique pas de connaître les bugs***

# Conception des tests

- L'activité de test doit être spécifiée, conçue, documentée et testée au même titre que l'activité de codage.
- Ce n'est pas une simple concaténation de cas de tests ajoutés au gré du débogage.
- Les phases et techniques de test doivent être prévues à l'avance et positionnées explicitement dans le cycle de fabrication du logiciel :  
**pas de pavé «test and debug» en fin de projet !**

# Le projet

L'activité de test ne peut s'envisager que par rapport à un projet dont les éléments sont bien identifiés.

- les acteurs
- les buts (objectifs, spécifications, ...)
- les moyens (machines, langages, ...)

# Prévision et Oracles

- Le test «pour jouer» consiste à voir, après coup, que la sortie observée est bien la sortie escomptée. Dans le vrai test, la sortie doit être **prédite et documentée** avant l'exécution du test (voire l'écriture du code).
- Si un programmeur ne sait pas prédire sûrement le résultat d'un test, c'est qu'il n'a pas compris comment le programme fonctionne ou ce qu'il doit faire exactement.



# Prévision et Oracles

- La notion d'**oracle** a été définie en 1978 par W.E.Howden : c'est n'importe quel programme, processus ou ensemble de données qui spécifie la sortie attendue d'un jeu de test appliqué à un élément de programme. Il y a autant de types d'oracles qu'il y a de façon de tester.
- L'oracle le plus commun est l'**oracle d'entrée/sortie** («input/outcome oracle») qui spécifie la sortie attendue pour une entrée donnée.

# Tests unitaires

- Unité
  - portion élémentaire d'un programme
  - classe ou méthode
- Test unitaire
  - teste une unité
  - assertions à vérifier sur différents cas critiques
  - tests indépendants

# Régression

- Bug de régression
  - évolution des programmes au cours du temps
  - apparition de nouveaux bugs
  - nécessité de tester l'ensemble du programme après chaque modification

Le Framework JUnit

# Présentation

- Origine :
  - framework de test écrit en Java par Gamma et Beck
  - open source : [www.junit.org](http://www.junit.org)
  - en constante évolution, aujourd'hui en version 4.8.2
  - attention, versions  $< 3.8 \neq 4.x$
- Objectifs :
  - test des applications Java
  - faciliter la création de tests
  - tests de non régression

# JUnit Framework

Un framework est un ensemble de classes et de collaborations entre les instances de ces classes.

Le source d'un framework est disponible mais ne s'utilise pas directement, il se spécialise :

par ex., pour créer un cas de test, on hérite de la classe `TestCase` (3.8) ou on utilise l'annotation **@Test()** (4.x)

Un framework peut être vu comme un programme à trous qui offre la partie commune des traitements et chaque utilisateur le spécialise pour son cas particulier.

# Organisation du code des tests

- Cas de Test :
  - JUnit 3.8 : **public void testXXX()**
  - **@Test()** (**TestCase**) pour définir un test unitaire
  - **@Before()** (**setUp**) pour les phases d'initialisation des méthodes
  - **@After()** (**tearDown**) pour les phases de terminaison des méthodes
- Suite de Test :

une séquence d'unités de test, de cas de test et/ou de suites de test
- Lancement des tests :
  - Les tests sont exécutées de manière indépendante
  - Le TestRunner exécute par défaut tous les tests / une suite de test
  - L'automatisation des tests permet de s'assurer de la non-régression

# Assertions

- Différentes assertions pour différents tests :
  - égalité : `assertEquals (... , ...)`
  - condition : `assertTrue (...)` ou `assertFalse (...)`
  - nullité : `assertNull (...)` ou `assertNotNull (...)`
  - similitude : `assertSame(..., ...)` ou `assertNotSame(..., ...)`



# Exemple : une classe Money

- Représentation de sommes d'argent dans plusieurs devises
- Taux de change variables entre devises et selon le temps

```
public class Money implements IMoney{
    private int fAmount;          // le montant
    private String fCurrency;    // la devise

    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }
    public String currency() {
        return fCurrency;
    }
    ...
}
```

# Exemple : une classe Money

```
...  
// Vérifier que deux objets Money sont égaux  
public boolean equals(Money aMoney) {  
    return aMoney.currency().equals(currency())  
        && amount() == aMoney.amount();  
}  
  
// Ajout de deux sommes d'une même devise  
public Money addSimple(Money m) {  
    return new Money(amount()+m.amount(),  
currency());  
}  
...
```

# TestCase @Test()

```
import org.junit.*;
import static org.junit.Assert.*;
public class MoneyTest {
    //constructeur
    public MoneyTest(String name){
        super(name);
    }
    @Test()
    public void testEquals() {
        Money m12EUR = new Money(12, "EUR");    // création de données (fixture)
        Money m14EUR = new Money(14, "EUR");
        assertTrue(!m12EUR.equals(null));
        assertEquals(m12EUR, m12EUR);           // comparaison
        assertEquals(m12EUR, new Money(12, "EUR"));
        assertFalse(m12EUR.equals(m14EUR));
    }
    @Test()
    public void testAddSimple() {
        Money m12EUR = new Money(12, "EUR");    // création de données (fixture)
        Money m14EUR = new Money(14, "EUR");
        Money expected= new Money(26, "EUR");
        Money result= m12EUR.addSimple(m14EUR); // exécution de la méthode testée
        assertTrue(expected.equals(result));    // comparaison
    }
}
```

# setUp @Before et tearDown @After

```
import org.junit.*;

public class MoneyTest {
    ...
    private Money m12EUR;                // objet pour fixture
    private Money m14EUR;

    @Before()                            // appel avant chaque test, syntaxe 4
    protected void setUp() {             // appel avant chaque test, syntaxe 3.8
        m12EUR = new Money(12, "EUR");    // création des objets
        m14EUR = new Money(14, "EUR");
    }
    ...
    @Test()                              // définition d'un test, syntaxe 4
    public void testAddSimple() {         // définition d'un test, syntaxe 3.8
        Money expected= new Money(26, "EUR"); // création de données complémentaires
        Money result= m12EUR.addSimple(m14EUR); // exécution de la méthode testée
        Assert.assertTrue(expected.equals(result)); // comparaison
    }
    ...
    @After()                             // appel après chaque test, syntaxe 4
    protected void tearDown() {          // appel après chaque test, syntaxe 3.8
        m12EUR = null;                  // nettoyage des objets
        m14EUR = null;
    }
}
```

# Regrouper les méthodes de test

- utilisation d'un autre runner que celui par défaut, le **`org.junit.runners.Suite`**
- changer de runner : annotation **`@RunWith(Class)`**
- classe vide annotée **`@RunWith(Suite.class)`**
- pour indiquer comment former la suite de test : annotation **`@SuiteClasses(Class[])`**

# Regrouper les méthodes de test

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
```

```
@RunWith(Suite.class)
@SuiteClasses(value = {MoneyTest.class,
MoneyBagTest.class})
public class AllTests { }
```

```
// pour compatibilité avec JUnit 3.x
public class AllTests {
    public static Test suite() {
        return new JUnit4TestAdapter(AllTests.class);
    }
}
```

# Rapport de test

Les résultats de test de validation sont stockés dans des fichiers XML :

- archivage à long terme pour faire des statistiques,
- génération de rapports à l'aide de scripts XSLT

[Home](#)

**Packages**
  
[arachnid.spider.parser](#)
  
[arachnid.spider.robots](#)

**Classes**
  
[RobotExclusionTest](#)
  
[RobotsFacadeTest](#)
  
[SimpleHTMLTokenTest](#)

# Unit Test Results

Designed for use with [JUnit](#) and [Ant](#).

## Summary

Tests	Failures	Errors	Success rate	Time
<a href="#">41</a>	<a href="#">0</a>	<a href="#">0</a>	100.00%	0.344

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

## Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
<a href="#">arachnid.spider.parser</a>	1	0	0	0.128	2009-02-10T16:05:33	pc-mna-161.univ-ubs.fr
<a href="#">arachnid.spider.robots</a>	40	0	0	0.216	2009-02-10T16:05:33	pc-mna-161.univ-ubs.fr

# JUnit : Situation

- Avantages :
  - gratuit,
  - simple,
  - intégré à de nombreux outils ou IDE (Ant, JBuilder, Eclipse, Kawa, VisualAge, ...)
- Inconvénients :
  - documentation, mais il existe de bons tutoriels et livres sur le sujet,
  - exploitation des résultats, mais les environnements comme Ant compensent ce manque (action `<junitreport>`)



# Evolution

- Evolue régulièrement tous les 6-8 mois : amélioration des fonctionnalités et surtout des rapports de tests,
- Le concept se généralise : analyse plus conceptuelle de l'activité dans des ouvrages récents, extension à l'ensemble des langages.
- La philosophie Xprogramming s'étend :
  - l'importance des tests est reconnue,
  - les outils de tests sont indispensables.