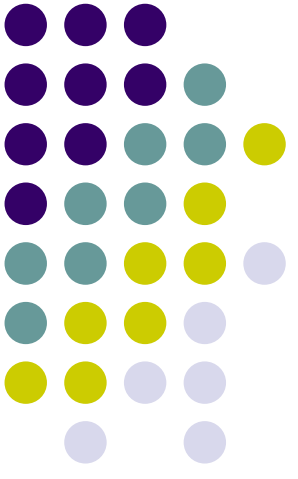
M1 – Middleware Adaptable

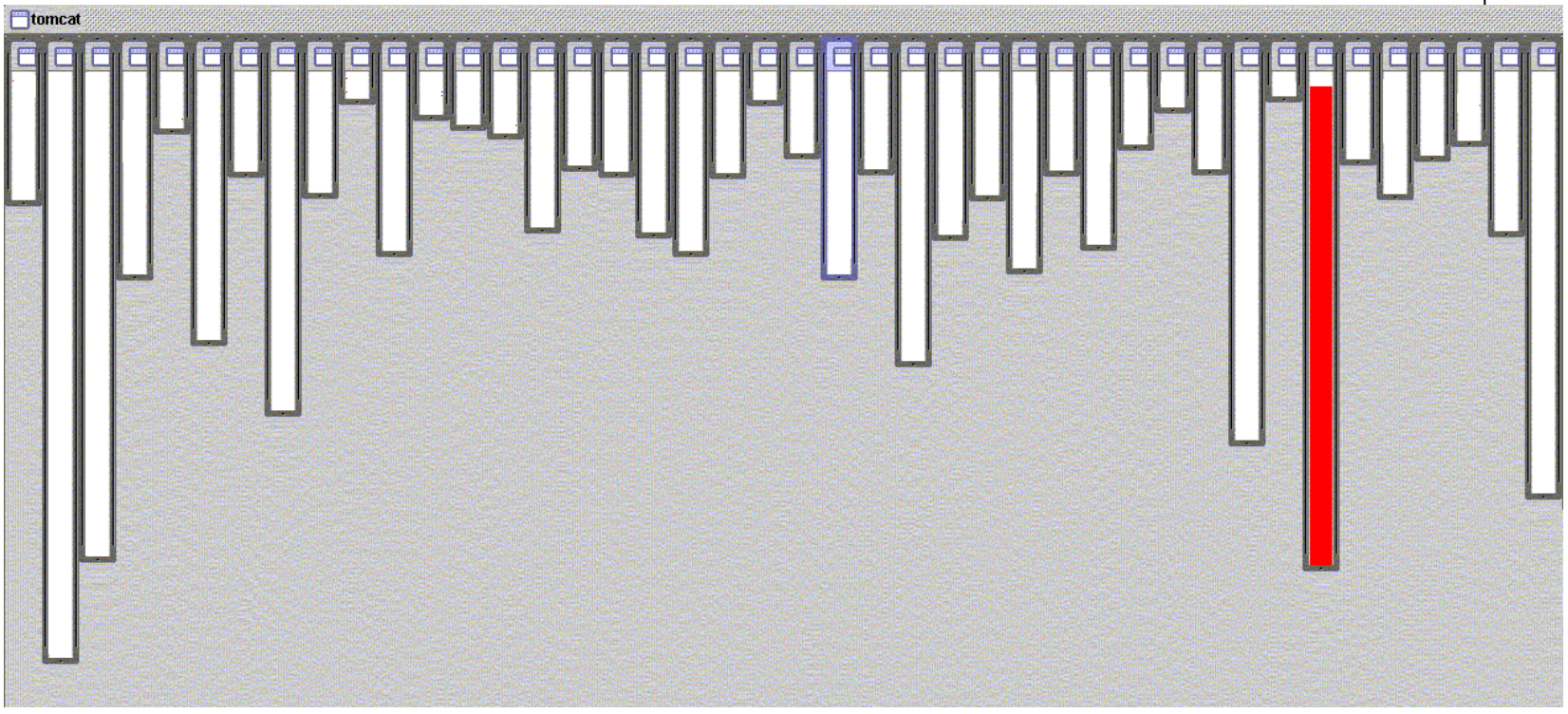
Aspect-Oriented Programming AspectJ

Sara Bouchenak

Sara.Bouchenak@inria.fr
http://sardes.inrialpes.fr/~bouchena/teaching

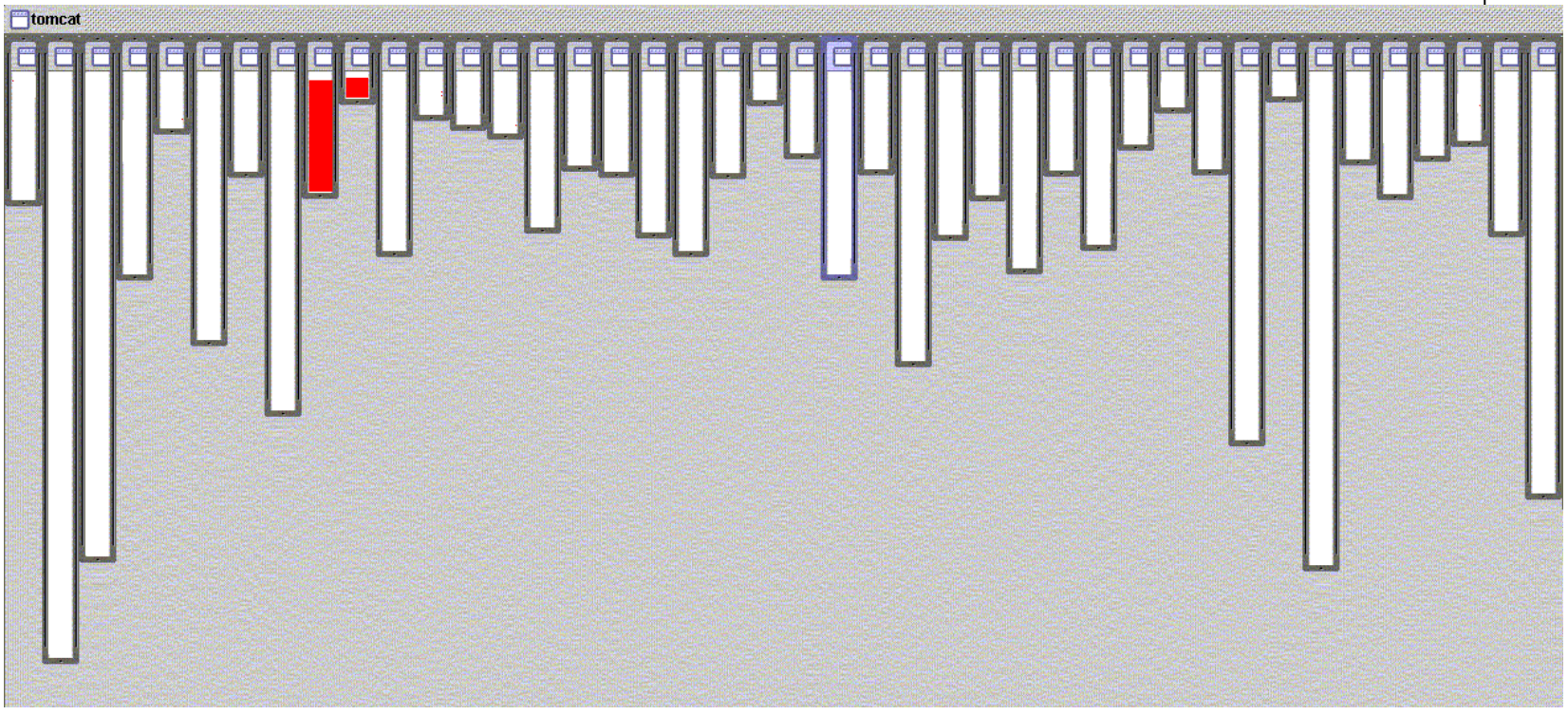


Bonne modularité



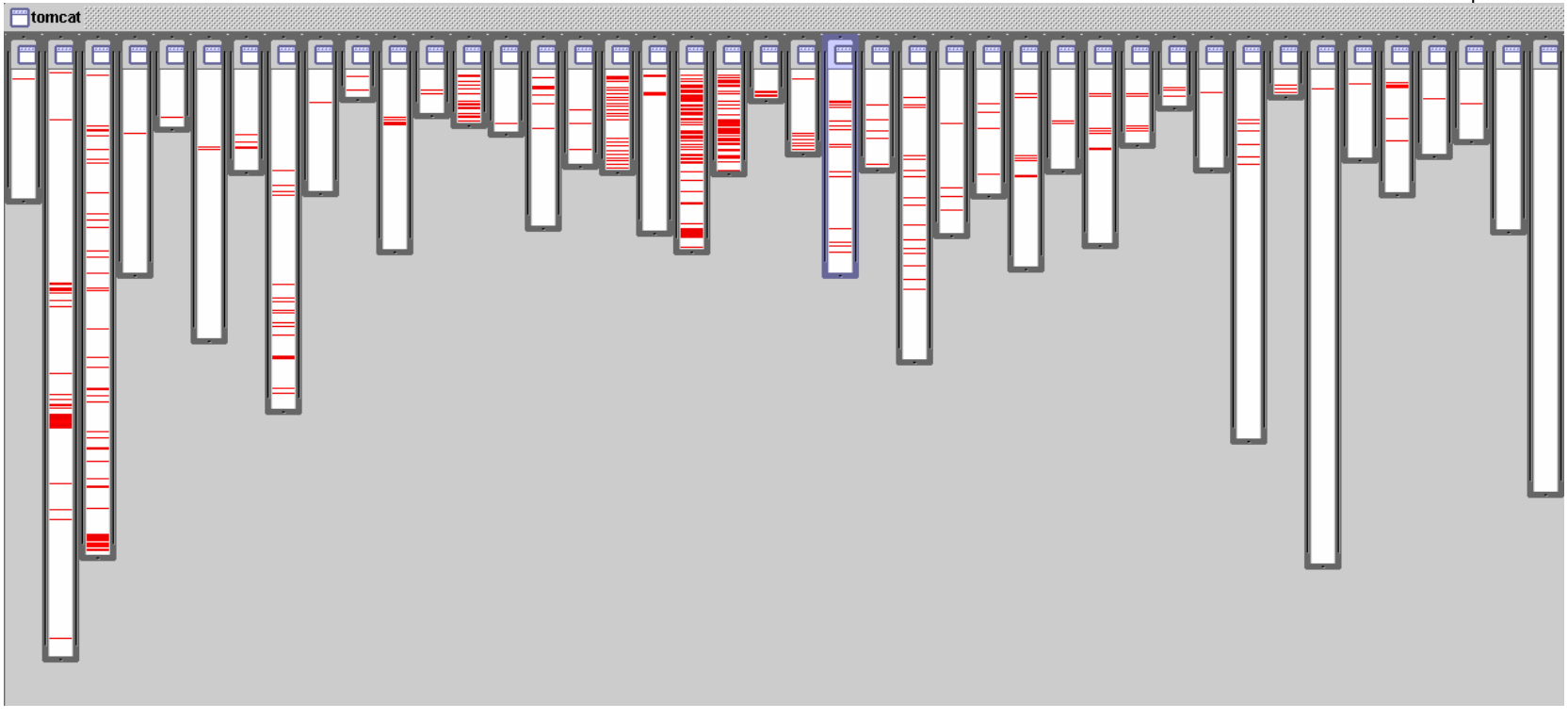
- Parsing XML dans org.apache.tomcat
  - lignes rouges montrent le code concerné
  - tout est regroupé dans un seul module (classe) ☺

Bonne modularité



- Reconnaissance de pattern d'URL dans org.apache.tomcat
  - lignes rouges montrent le code concerné
  - tout est regroupé dans deux modules (héritage) ☺

Problèmes



- Traçage dans org.apache.tomcat
  - lignes rouges montrent le code concerné
  - pas regroupé dans un seul module
  - pas dans un petit nombre de modules ☹

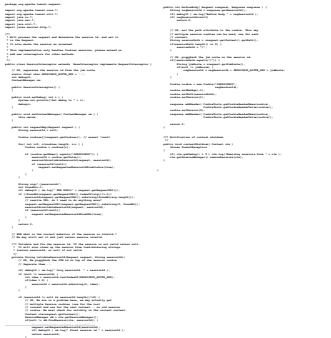
Expiration de session – non modulaire

Problèmes

ApplicationSession

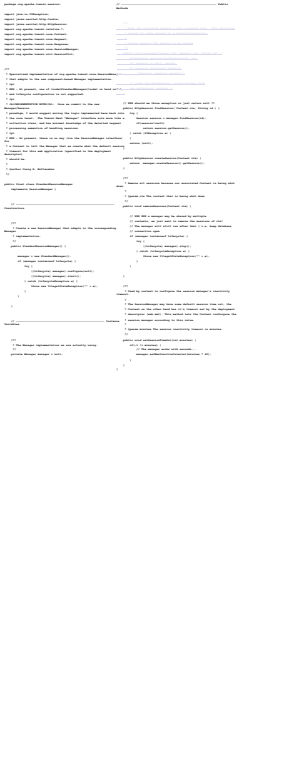
StandardSession

SessionInterceptor



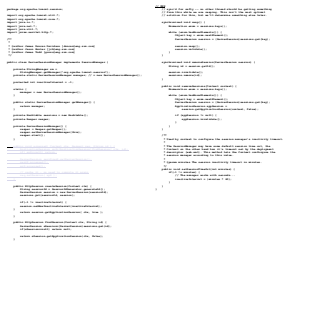
StandardManager

StandardSessionManager



ServerSession

ServerSessionManager



Synthèse

- Traitements très localisés
  - lecture de fichiers de config., parsing XML
- Traitements répartis sur peu de classes
  - traitement de requêtes HTTP, expressions régulières
- Traitements se retrouvant dans quasiment tout le code
  - traçage, diagnostic, rendre compte des requêtes traitées et des problèmes

Inconvénients d'un code non modulaire

- Code redondant
  - Même fragment de code dans plusieurs classes
- Code non clair
  - Structure non explicite
  - Image globale non claire
- Code difficile à modifier/maintenir
  - Trouver toutes les classes concernées
  - Etre sûr d'effectuer les modifications de façon cohérente
  - Très peu d'aide des outils de programmation OO

Principe de l'AOP

- Séparer les concepts (traitements) entrelacés
- Minimiser les dépendances entre eux
- Chaque traitement doit avoir
  - un objectif clair
  - une structure bien définie, modulaire

↳ Aspects
  - Traitements modulaires qui peuvent par la suite être entrelacés

Plan

1. Introduction à l'AOP
2. Introduction à AspectJ
3. Syntaxe AspectJ
4. Développement logiciel avec AspectJ
5. Synthèse et Références

Définition de l'AOP

- AOP = *Aspect-Oriented Programming*
  (programmation orientée aspect
  ou programmation par aspects)

- "La programmation par aspects est une technique permettant de factoriser certains traitements dont la réalisation est a priori dispersée à travers un système, fût-il orienté-objet."

Aspects : fonctionnels et orthogonaux

- Un système logiciel est constitué de :
  - **Aspects fonctionnels (*core concerns*)**. Modules logiciels qui mettent en œuvre la logique applicative du système logiciel.
  - **Aspects orthogonaux (*crosscutting concerns*)**. Modules qui mettent en œuvre des fonctionnalités système utilisées par plusieurs autres modules.

Méthodologie de l'AOP

- Mise en œuvre d'un système avec l'AOP :

1. Décomposition du système en aspects
2. Mise en œuvre des aspects
3. Recomposition des aspects

Méthodologie de l'AOP : 1. Décomposition en aspects

- Décomposition des besoins du système logiciel en :
  - Aspects fonctionnels.
    Exemple de système bancaire : clients, comptes, transactions entre banques, etc.
  - Aspects orthogonaux.
    Exemples : persistance des données, authentification, traçage, partage de ressources, performances, etc.

Méthodologie de l'AOP : 2. Mise en œuvre des aspects

- Mise en œuvre de chaque aspect indépendamment des autres
  - Aspects fonctionnels : clients, comptes, transactions bancaires, etc.
  - Aspects orthogonaux : persistance, authentification, performances, etc.
- Techniques de mise en œuvre
  - Langages procéduraux
  - Techniques orientées-objet, etc.

Méthodologie de l'AOP : 3. Composition des aspects

- Spécification des règles de composition des aspects
- Règles utilisées pour la construction du système final
- Processus de composition : intégration ou *weaving*

règle de composition — *aspect fonctionnel*

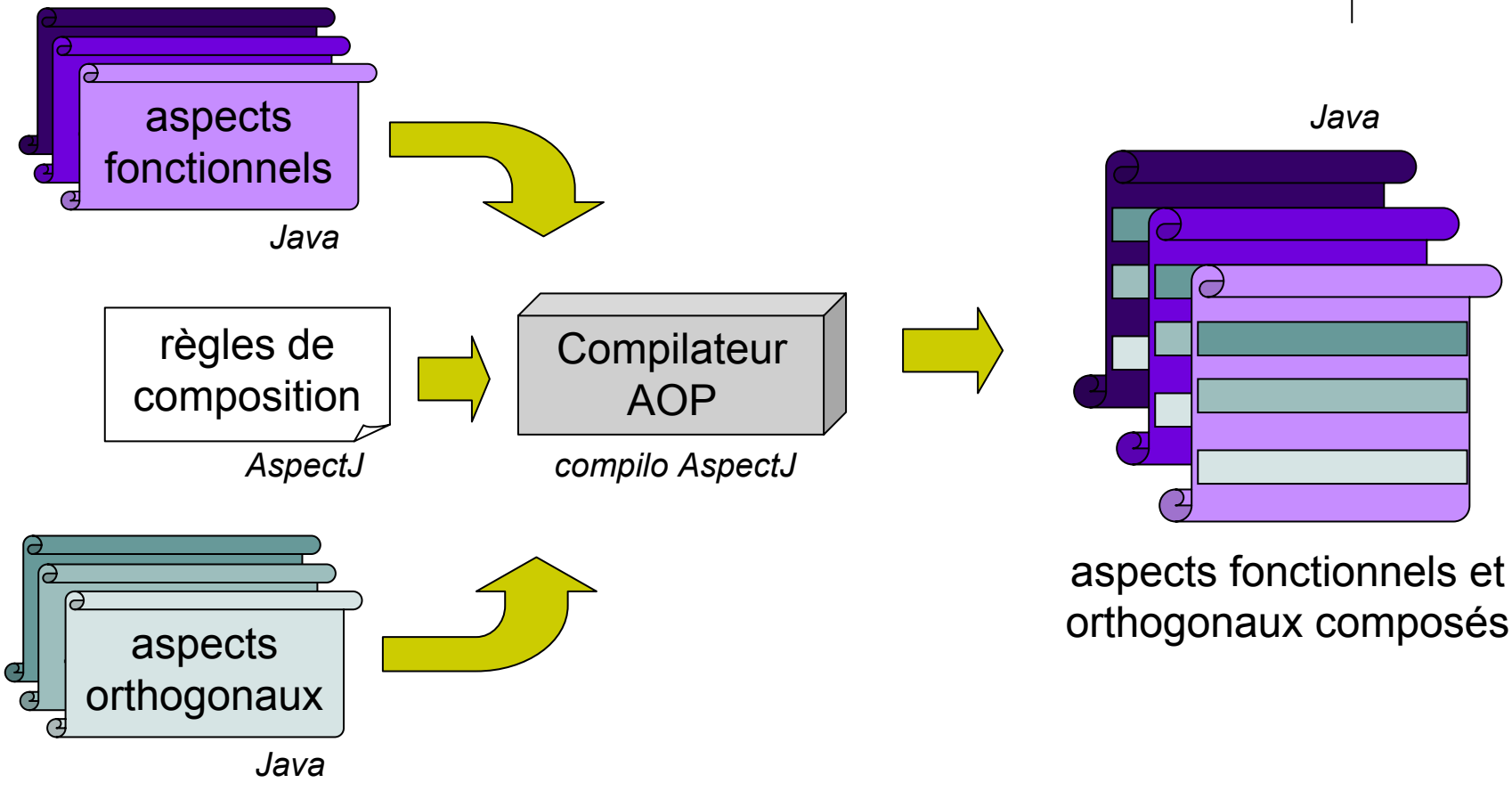
- Exemple : avant chaque opération sur le système bancaire, le client doit être authentifié.

aspect orthogonal

Réalisation de l'AOP

- AOP : une méthodologie de programmation
- Réalisation de la méthodologie AOP :
  - Langage de mise en œuvre des aspects (fonctionnels et orthogonaux) : C, C++, Java, etc.
  - Langage de spécification des règles de composition des aspects : AspectC, AspectJ, etc.
  - Outil d'intégration des aspects (*weaving*) : *aspect weaver*, compilateur AOP (AspectC, AspectJ, etc.)

Réalisation de l'AOP dans Java



Plan

1. Introduction à l'AOP
2. *Introduction à AspectJ*
   - *Composition dans AspectJ*
   - *join point, pointcut, advice, introduction, declaration, aspect*
   - *Méthodologie de programmation*
   - *Exemples*
3. Syntaxe AspectJ
4. Développement logiciel avec AspectJ
5. Synthèse et Références

Composition dans AspectJ

- Composition statique
  - Modification de la structure statique du système : classes et interfaces
  - Ajout d'attributs et méthodes
  - Déclaration de messages d'avertissement (*warnings*) ou d'erreurs affichés lors de la compilation
- Composition dynamique
  - Ajout d'un nouveau comportement à l'exécution "normale" du programme
  - Etendre ou remplacer une opération
  - Action effectuée avant/après l'exécution de certaines méthodes ou de certains traitants d'exceptions dans des classes

Eléments de composition dans AspectJ

- AspectJ
  - une extension du langage Java
  - pour définir les règles de composition statique et dynamique
- Eléments de composition dans AspectJ
  - *Join point*
  - *Pointcut*
  - *Advice*
  - *Introduction*
  - *Declaration* (at compile-time)
  - *Aspect*

Join point

- Définition
  - Un point identifié dans l'exécution d'un programme
  - Un appel de méthode ou l'accès à l'attribut d'un objet
- Exemple

```
public class Account {
    float balance;

    void credit(float amount) {
        this.balance += amount;
    }
}
```

  - *Join points* de la classe *Account* incluent
    - l'exécution de la méthode *credit*
    - l'accès à l'attribut *balance*

Pointcut

- Définition
  - Sélection d'un *join point*
  - Récupération du contexte au niveau de ce *join point*
- Exemple

```
execution (void Account.credit(float))
```

  - *Pointcut*
    - sélectionne le *join point* correspondant à l'appel de la méthode *credit* de la classe *Account*
    - récupère le contexte de cet appel : l'objet sur lequel la méthode est appelée, les paramètres de la méthode

Advice

- Définition
  - Code exécuté à un *pointcut*
  - Exécution avant, après ou en remplacement d'un *pointcut*
- Exemple

```
before() : execution (void Account.credit(float)) {
   System.out.println("About to perform credit operation");
}
```

  - *Advice*
    - affichage d'un message
    - avant l'exécution de la méthode *credit* de la classe *Account*

Introduction

- Définition
  - Introduction d'une modification statique à une classe ou interface
  - Ajout d'une méthode ou d'un attribut à une classe, extension de la hiérarchie des classes et interfaces
- Exemple

```
declare parents : Account implements BankingEntity;
```

```
private float Account._minimalBalance;
```

  - *Introduction*
    - extension de la hiérarchie
    - classe *Account* implémente l'interface *BankingAccount*
    - ajout du nouvel attribut *_minimalBalance* à la classe *Account*

Compile-time declaration

- Définition
  - Instruction de composition statique
  - Ajout de messages d'avertissement ou d'erreurs affichés lors de la compilation
- Exemple

```
declare warning : call (void Persistence.save(Object)) :
    "Deprecated method Persistence.save(),
     Consider using Persistence.saveOptimized()";
```

  - *Declaration*
    - déclaration d'un message d'avertissement
    - affichage si appel de la méthode *save* de la classe *Persistence*

Aspect

- Définition
  - Module de code qui contient la spécification des règles de composition statiques et dynamiques utilisées
  - *Introductions* + *declarations* + *pointcuts* + *advices* = *aspect*
  - Aspect dans AspectJ équivalent à une classe dans Java
- Exemple de fichier *ExampleAspect.java*

```
public aspect ExampleAspect {
    declare parents : Account implements BankingEntity;

    before() : execution (void Account.credit(float)) {
        System.out.println("About to perform credit operation");
    }

    declare warning : call (void Persistence.save(Object)) :
        "Deprecated method Persistence.save(),
         Consider using Persistence.saveOptimized()";
}
```

Méthodologie de programmation

- Conception
  1. Identifier les *join points* où il faut étendre/modifier un comportement
  2. Concevoir le nouveau comportement à introduire à ces points-là

- Mise en œuvre
  1. Ecriture d'un module *aspect* qui contient la mise en œuvre de cette conception
  2. Ecriture, dans l'aspect, des *pointcuts* qui sélectionnent les *join points* désirés
  3. Ecriture d'un *advice* associé à chaque *pointcut* pour définir le nouveau comportement à introduire à chacun de ces points
  4. Ajout de règles de composition statique (*introductions*, *declarations*) si nécessaire

Exemple Hello (1 / 4)

- Application

```
public class Communicator {
    public static void print(String message) {
        System.out.println(message);
    }
    public static void print(String person, String message) {
        System.out.println(person + ", " + message);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Communicator .print("Want to learn AspectJ?");
        Communicator .print("Tom", "how are you?");
    }
}
```

Exemple Hello (2 / 4)

- Compilation Java

  ➢ javac Communicator.java Test.java

- Exécution

  ➢ java Test

  Want to learn AspectJ?

  Tom, how are you?

Exemple Hello (3 / 4)

- Aspect

```
public aspect HelloAspect {                          ❶ Déclaration d'aspect

    pointcut printCall() : call (void Communicator.print(..));   ❷ Déclaration de pointcut

    before() : printCall() {                         ❸ Déclaration d'advice
        System.out.print("Hello! ");
    }

}
```

Exemple Hello (4 / 4)

- Compilation AspectJ

  - **ajc** Communicator.java Test.java HelloAspect.java

- Exécution

  - java Test

  Hello! Want to learn AspectJ?

  Hello! Tom, how are you?

Exemple de traçage (1 / 4)

- Application originale

```
package banking;

public class Account {
    float balance;

    void credit(float amount) {

        this.balance += amount;

    }

    void debit(float amount) {

        this.balance -= amount;

    }
}
```

Exemple de traçage (2 / 4)

- Application modifiée "à la main" pour traçage

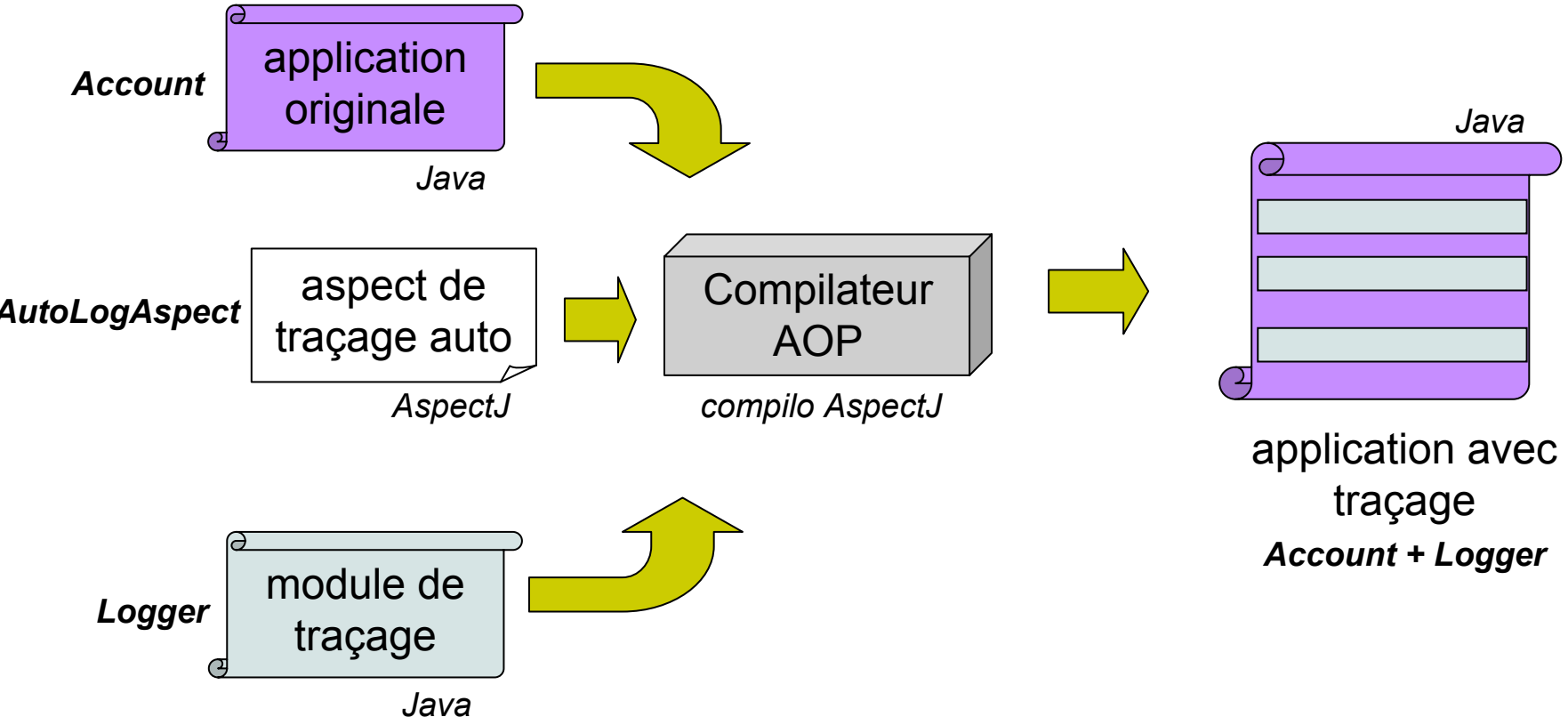
```
package banking;

public class Account {
    float balance;

    void credit(float amount) {
        Logger.entry("credit(float)");
        this.balance += amount;
        Logger.exit("credit(float)");
    }

    void debit(float amount) {
        Logger.entry("debit(float)");
        this.balance -= amount;
        Logger.exit("debit(float)");
    }
}
```

Exemple de traçage (3 / 4)

- Autre solution : Aspect de traçage automatique

```
public aspect AutoLogAspect {

    pointcut methodExecution() : execution (* banking.Account.*(..));


    before() : methodExecution() {
        Logger.entry(thisJoinPoint.getSignature().toString());
    }


    after() : methodExecution() {
        Logger.exit(thisJoinPoint.getSignature().toString());
    }

}
```

Exemple de traçage (4 / 4)



Plan

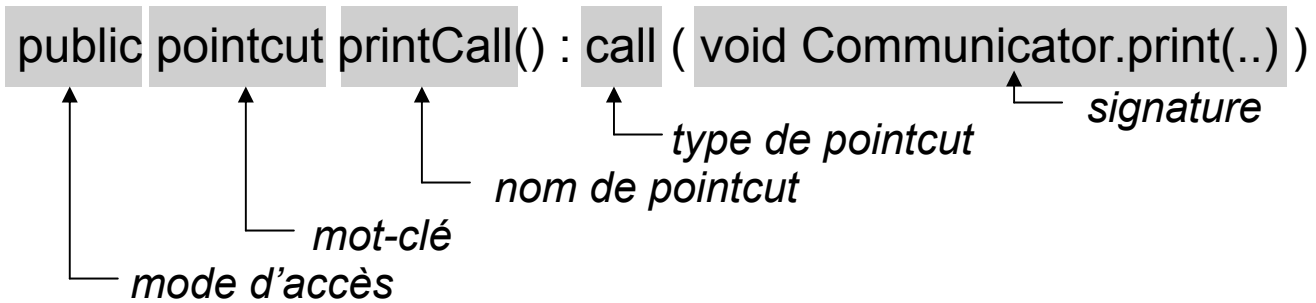
1. Introduction à l'AOP
2. Introduction à AspectJ
3. *Syntaxe AspectJ*
   - *Pointcut*
   - *Call vs. Execution*
   - *Pointcuts de flot de contrôle, de structure lexicale*
   - *Advice*
   - *Passage de paramètres entre pointcut et advice*
4. Développement logiciel avec AspectJ
5. Synthèse et Références

Pointcut

- Syntaxe d'un pointcut

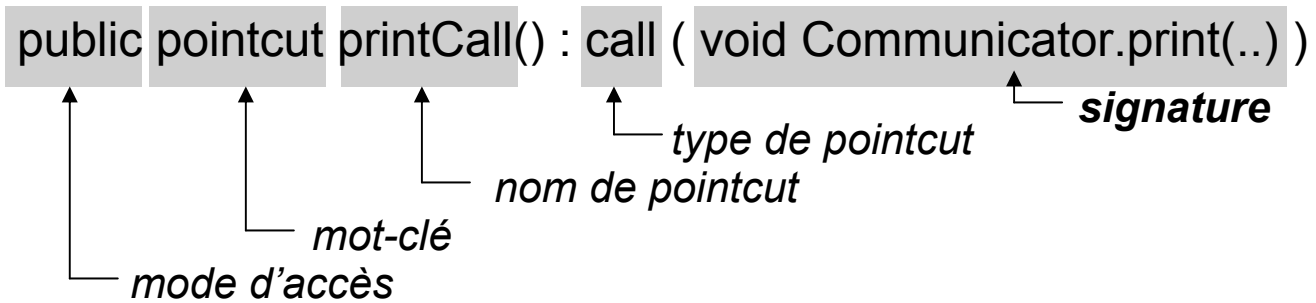
  [mode_accès] pointcut nom_pointcut([args]) : *definition_pointcut*

- Exemple



Signature de pointcut

- Exemple



- Signature d'un pointcut
  - Signature de type
  - Signature de méthodes/constructeurs
  - Signature d'attribut

Pointcut : Signature de type

Signature	Types associés
Account	Type (classe / interface) de nom Account
*Account	Type dont le nom se termine avec Account, tel que SavingsAccount, CheckingAccount
java.*.Date	Type Date dans tout sous-package direct du package java, tel que java.util.Date, java.sql.Date
java..Date	Tout type Date dans le package java ou un sous-package direct ou indirect du package java
java.util.List+	Tout type héritant de l'interface java.util.List (l'implémentant)

Pointcut : Signature de méthode

Signature	Méthodes associées
public void Collection.clear()	La méthode clear() de la classe Collection qui ne prend aucun paramètre et retourne void
public void Account.set*(*)	Toute méthode publique de la classe Account, dont le nom commence par set, qui prend un paramètre unique de type quelconque et qui retourne void
* Account.*()	Toute méthode de la classe Account, ne prenant pas de paramètre, quels que soient son type de retour et mode d'accès
public void Account.*(..)	Toute méthode publique de la classe Account, qui prend des paramètres de nombre et types quelconques et qui retourne void
* *.*(..) throws RemoteException	Toute méthode qui peut lever une exception de type RemoteException

Pointcut : Signature de constructeur

Signature	Constructeurs associés
public Account.new()	Le constructeur public de la classe Account, ne prenant aucun paramètre
public Account.new(int)	Le constructeur public de la classe Account, qui prend un paramètre unique de type int
public Account.new(..)	Tout constructeur public de la classe Account, qui prend un nombre et un type quelconques de paramètres
public Account+.new(..)	Toute constructeur public de la classe Account ou d'une de ses sous-classes

Pointcut : Signature d'attribut

Signature	Attributs associés
private float Account.balance	L'attribut privé balance de la classe Account
* Account.*	Tout attribut de la classe Account, quels que soient son nom, son type et son mode d'accès
public static * banking..*.*	Tout attribut public et statique du package banking ou d'un de ses sous-packages directs ou indirects
* *.*	Tout attribut de n'importe quelle classe

Opérateurs de pointcuts

- Opérateur unaire : !

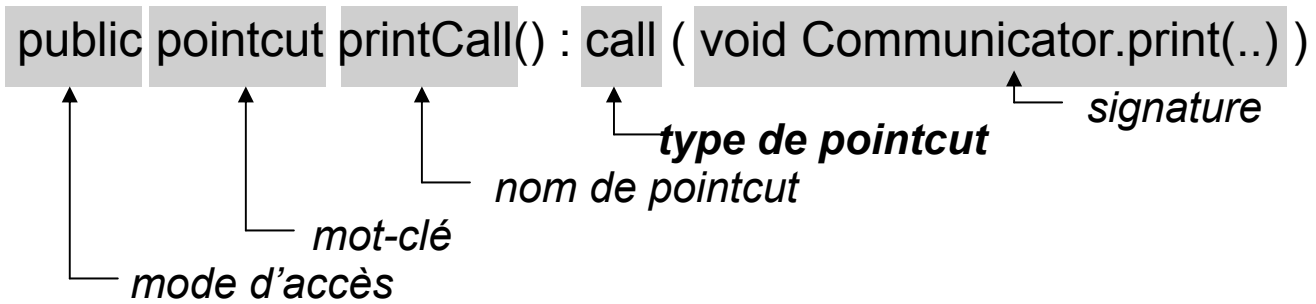
public !final *.*	Tout attribut public et non final de n'importe quelle classe
!Vector	Tout type autre que Vector

- Opérateurs binaire : && (et) || (ou)

Vector || Hashtable	Type Vector ou Hashtable
Java.util.RandomAccess+ && java.util.List	Tout type implémentant les deux interfaces, tel que java.util.ArrayList

Type de pointcut

- Exemple



- Type d'un pointcut
  - Appel de méthode, constructeur
  - Exécution de méthode, constructeur
  - Accès à un attribut en lecture, écriture, etc.

Types de pointcuts

Type de pointcut	Syntaxe
Appel de méthode	call(signature_méthode)
Exécution de méthode	execution(signature_méthode)
Appel de constructeur	call(signature_constructeur)
Exécution de constructeur	execution(signature_constructeur)
Initialisation de classe	staticinitialization(signature_type)
Accès à un attribut en lecture	get(signature_attribut)
Accès à un attribut en écriture	set(signature_attribut)
Traitant d'exception	handler(signature_type)

Call vs. Execution (1 / 5)

- Exemple

```
public class Account {
    float balance;

    void credit(float amount) {

        this.balance += amount;
    }
}
```

***execution* de la méthode credit**

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();

        accout.credit(100);
    }
}
```

***call* de la méthode credit**

Call vs. Execution (2 / 5)

- Aspect utilisant le *call*

```
public aspect AutoLogAspect_Call {

    pointcut creditMethodCall() : call (* Account.credit(..));


    before() : creditMethodCall() {
        Logger.entry(thisJoinPoint.getSignature().toString());
    }

}
```

Call vs. Execution (3 / 5)

- Application de l'AutoLogAspect_Call

```
public class Account {
    float balance;

    void credit(float amount) {

        this.balance += amount;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        Logger.entry("credit(float)");
        accout.credit(100);
    }
}
```

Call vs. Execution (4 / 5)

- Aspect utilisant l'*execution*

```
public aspect AutoLogAspect_Execution {

    pointcut creditMethodExecution() : execution (* Account.credit(..));


    before() : creditMethodExecution() {
        Logger.entry(thisJoinPoint.getSignature().toString());
    }

}
```

Call vs. Execution (5 / 5)

- Application de l'AutoLogAspect_Execution

```
public class Account {
    float balance;

    void credit(float amount) {
        Logger.entry("credit(float)");
        this.balance += amount;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();

        accout.credit(100);
    }
}
```

Pointcuts relatifs au flot de contrôle

Pointcut	Description
cflow(call (* Account.credit(..)))	Tout join point dans le flot de contrôle d'une méthode credit de la classe Account, y compris l'appel à la méthode credit elle-même
cflowbelow(call (* Account.credit(..)))	Tout join point dans le flot de contrôle d'une méthode credit de la classe Account, sauf l'appel à la méthode credit elle-même
cflow(creditMethodCall())	Tout join point dans le flot de contrôle du pointcut creditMethodCall
cflowbelow(execution (* Account.new(..)))	Tout join point dans le flot de contrôle d'un constructeur de la classe Account, sauf l'exécution du constructeur lui-même

cflow / cflowbelow : mais encore…

- Exemple

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();

        accout.credit(100);
    }
}
```

cflow / cflowbelow : mais encore…

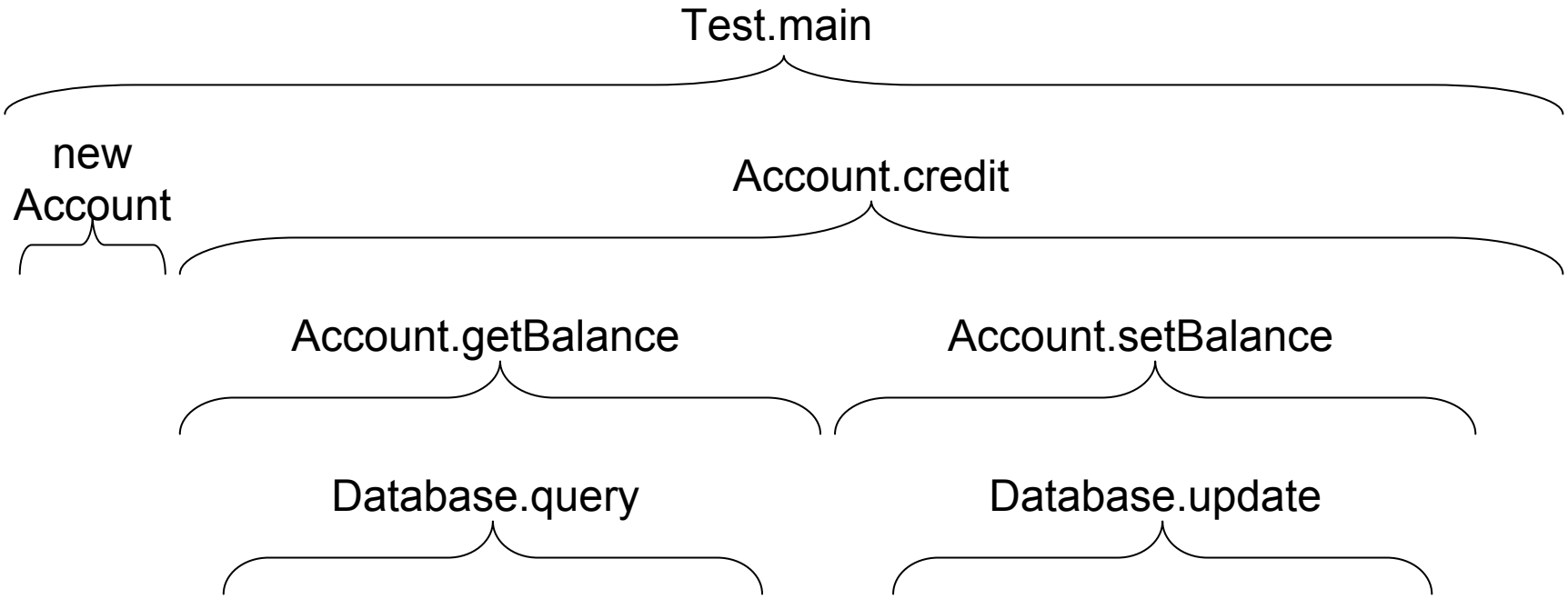
```
public class Account {
    int          id;          // account id
    Database database;  // associated database

    float getBalance() {

        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {

        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {

        setBalance(getBalance() + amount);
    }
}
```

cflow / cflowbelow : mais encore…

```
public class Database {
    float query(String sqlQuery) {
        … lecture de la base de données
    }
    void update(String sqlQuery) {
        … écriture dans la base de données
    }
}
```

cflow / cflowbelow : mais encore…

- Flot de contrôle à l'appel de la méthode credit



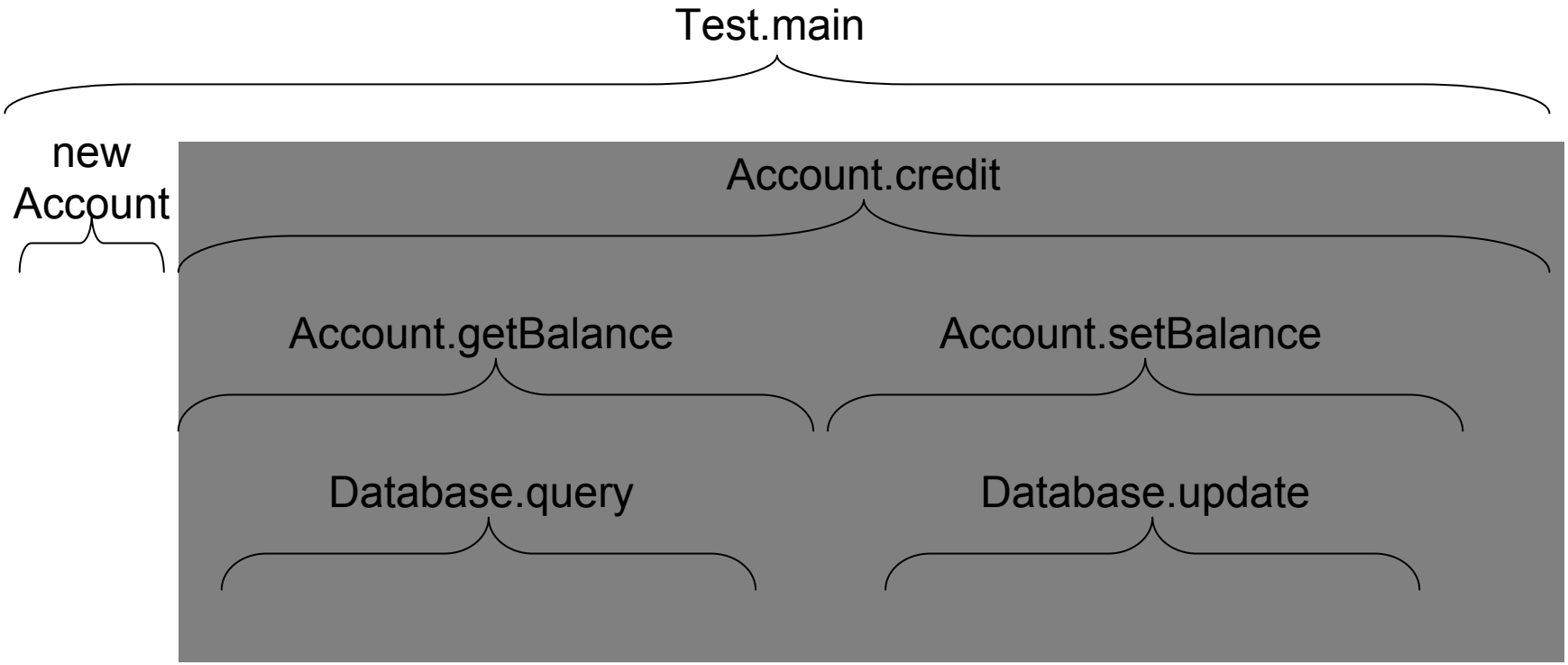
cflow : mais encore…

- Aspect utilisant le *cflow*

```
public aspect AutoLogAspect_CallF {

    pointcut creditMethodCallF() : cflow( call (* Account.credit(..)) );


    before() : creditMethodCallF() {
        System.out.println("Hello!");
    }

}
```

cflow : mais encore…

- cflow(call(* Account.credit(..)))



cflow : mais encore…

- Exemple

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        System.out.println("Hello!");
        accout.credit(100);
    }
}
```

cflow : mais encore…

```
public class Account {
    int          id;          // account id
    Database database;  // associated database

    float getBalance() {
        System.out.println("Hello!");
        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {
        System.out.println("Hello!");
        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {
        System.out.println("Hello!");
        setBalance(getBalance() + amount);
    }
}
```

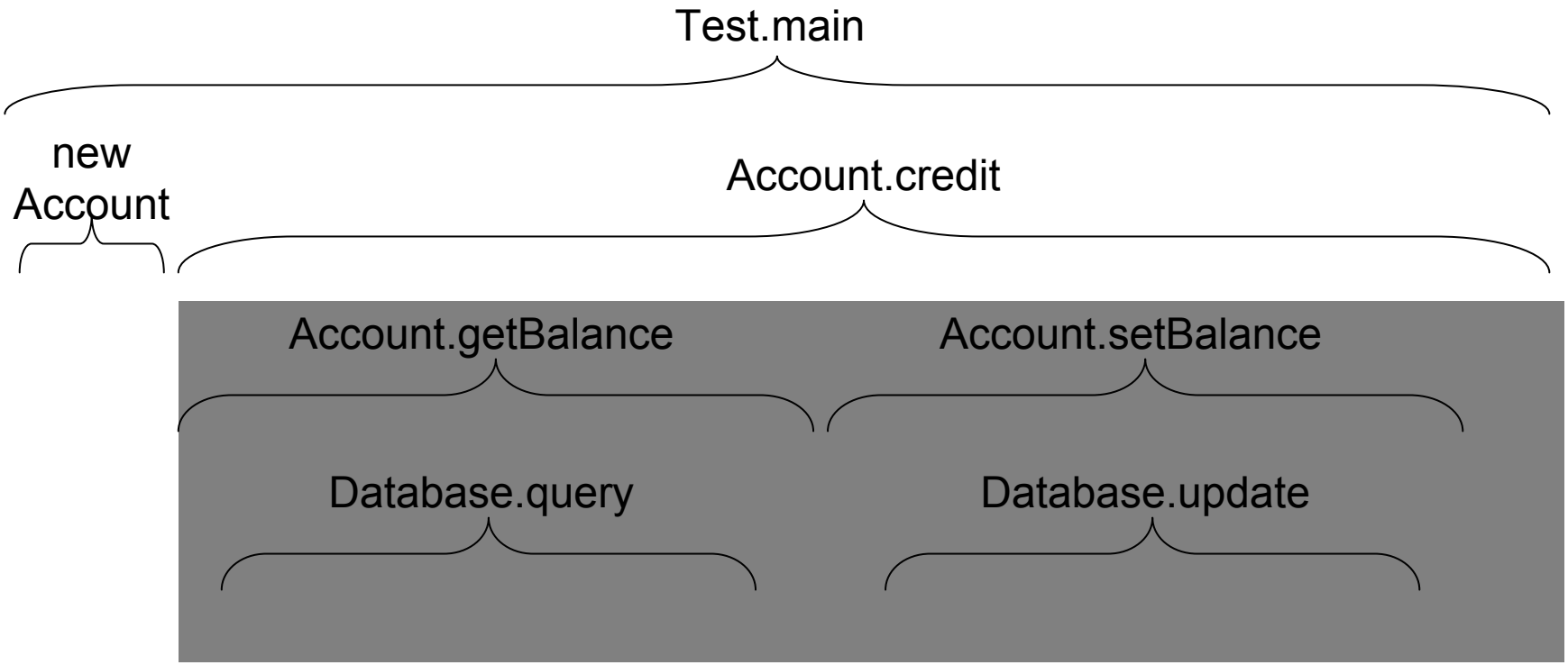
cflowbelow : mais encore…

- Aspect utilisant le *cflowbelow*

```
public aspect AutoLogAspect_CallFB {

    pointcut creditMethodCallFB() : cflowbelow( call (* Account.credit(..)) );


    before() : creditMethodCallFB() {
        System.out.println("Hello!");
    }

}
```

cflowbelow : mais encore…

- cflowbelow(call(* Account.credit(..)))



cflowbelow : mais encore…

- Exemple

```
public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        System.out.println("Hello!");
        accout.credit(100);
    }
}
```

cflowbelow : mais encore…

```
public class Account {
    int          id;          // account id
    Database database;  // associated database

    float getBalance() {
        System.out.println("Hello!");
        return database.query("SELECT balance FROM accounts WHERE id=" + id);

    }

    void setBalance(float b) {
        System.out.println("Hello!");
        database.update("UPDATE accounts SET balance="+b+" WHERE id=" + id);

    }

    void credit(float amount) {
        System.out.println("Hello!");
        setBalance(getBalance() + amount);
    }
}
```

Pointcuts relatifs à la structure lexicale

Pointcut	Description
within(Account)	Tout join point dans la classe Account (compris dans la portée lexicale de la classe Account)
within(Account+)	Tout join point dans la classe Account et ses sous-classes (compris dans la portée lexicale de ces classes)
withincode(* Account.credit(..))	Tout join point dans toute méthode credit de classe Account (compris dans la portée lexicale de cette méthode)

Exemple avec *within*

- Aspect utilisant l'*execution*

```
public aspect AutoLogAspect_Execution {

    pointcut methodExecution() : execution (* Account.*(..))
                                 && within(banking..*);


    before() : methodExecution() {
        Logger.entry(thisJoinPoint.getSignature().toString());
    }
}
```

Pointcuts relatifs à l'objet d'exécution

Pointcut	Description
this(Account)	Tout join point où le *this* est une instance de la classe Account ou d'une de ses sous-classes (appel de méthode ou accès à l'attribut d'un objet Account). Pointcut utilisé en combinaison avec *execution*
target(Account)	Tout join point où l'objet sur lequel une méthode est appelée est une instance de la classe Account ou d'une de ses sous-classes. Pointcut utilisé en combinaison avec *call*

- this(*Type*) ou this(*ObjectIdentifier*)
- target(*Type*) ou target(*ObjectIdentifier*)

Pointcuts arguments

Pointcut	Description
args(String,..,int)	Tous les join points de toutes les méthodes où le premier paramètre est de type String et le dernier de type int

- args(*Type* ou *ObjectIdentifier*,..)

Advice

- Advice :
  - Traitement à effectuer lors d'un pointcut

- Syntaxe d'un advice

```
specification_advice([args]) : nom_pointcut([args}) {
   … corps de l'advice
}
```

- Exemple

```
before () : printCall () {
   … corps de l'advice
}
```

nom de pointcut

specification d'advice

Advice : Syntaxe

- Syntaxe d'un advice

```
specification_advice([args]) : definition_pointcut {
   … corps de l'advice
}
```

- Exemple

```
before () : call ( void Communicator.print(..) ) {
   … corps de l'advice
}
```

specification d'advice

type de pointcut

signature

Types d'advices

- Advice before
  - s'exécute avant le join point
- Advice after
  - s'exécute après le join point
- Advice around
  - englobe l'exécution du join point

Advice before

- Exemple

```
before () : call ( * Account.*(..)) {
    … authentifier l'utilisateur
}
```

- Cas particulier
  - Si l'advice before lève une exception, la méthode concernée n'est alors pas exécutée
- Exemples d'utilisation
  - Traçage, authentification, etc.

Advice after

- Toute terminaison (normale ou exception)

```
after () : call ( * Account.*(..)) {

    … tracer toute terminaison
}
```

- Terminaison normale

```
after returning () : call ( * Account.*(..)) {

    … tracer la terminaison normale
}
```

- Terminaison en cas de levée d'exception

```
after throwing () : call ( * Account.*(..)) {

    … tracer la levée d'exception
}
```

Advice around

- Remplacer un traitement

```
around () : call ( void Account.credit(float)) {

    … nouvelle mise en œuvre de la méthode
}
```

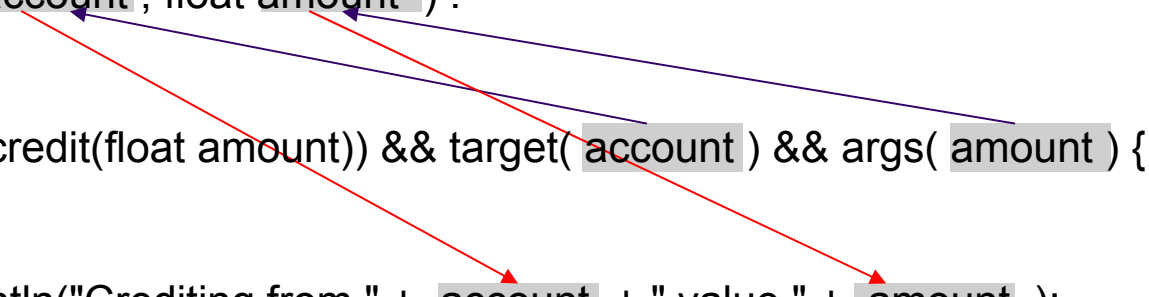
- Entourer un traitement

```
around (Account acount, float amount) :
  call ( * Account.credit(float amount)) && target(account) && args(amount) {

    … tracer le début de la méthode
    proceed(account, amount);
    … tracer la fin de la méthode
}
```

Passage de contexte entre le join point et l'advice (1 / 3)

- Passage de paramètres

```
before (Account account , float amount  ) :

  call ( * Account.credit(float amount)) && target( account ) && args( amount ) {

    System.out.println("Crediting from " +  account  + " value " +  amount  );
}
```



Passage de contexte entre le join point et l'advice (2 / 3)

- Récupération du résultat

```
after returning (Object object ) :

  call ( Object Account.*(..)) {

    System.out.println("Result " +  object );
}
```

Passage de contexte entre le join point et l'advice (3 / 3)

- Récupération de l'exception

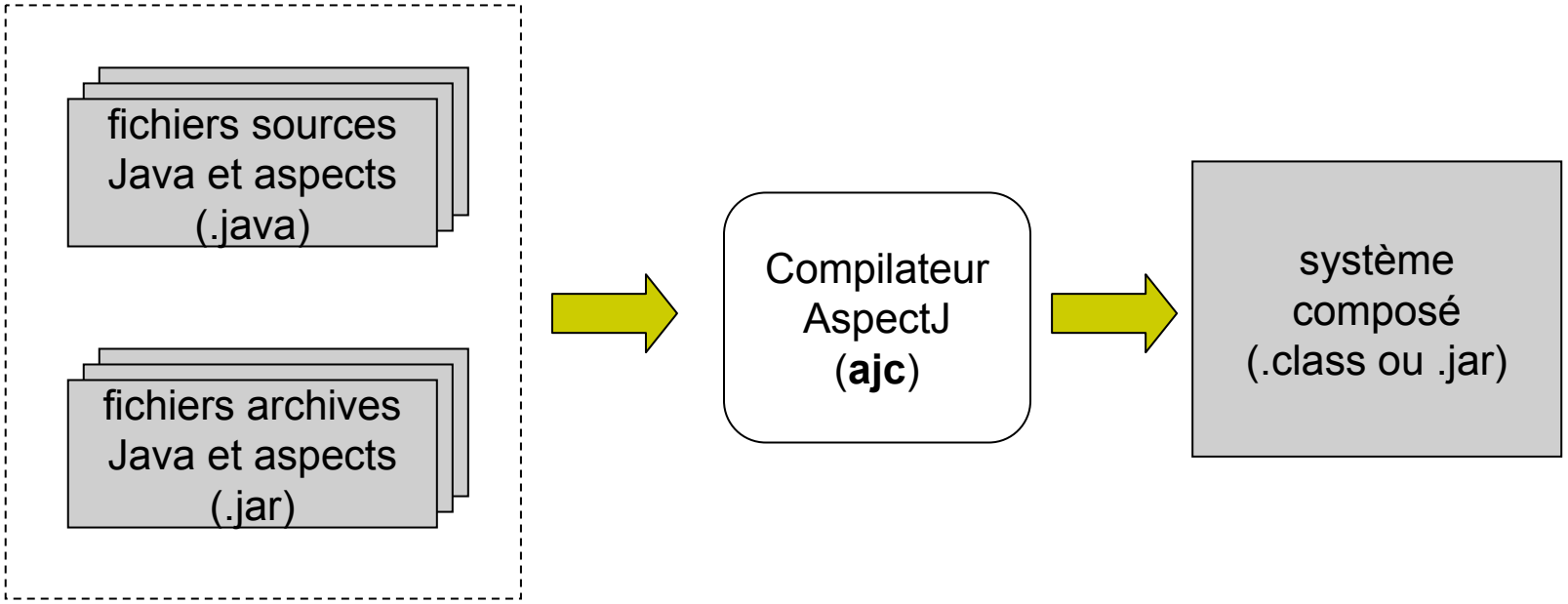
```
after throwing (RemoteException except ) :

  call ( * Account.*(..) throws RemoteException ) {

    System.out.println("Exception " + except );
}
```

Plan

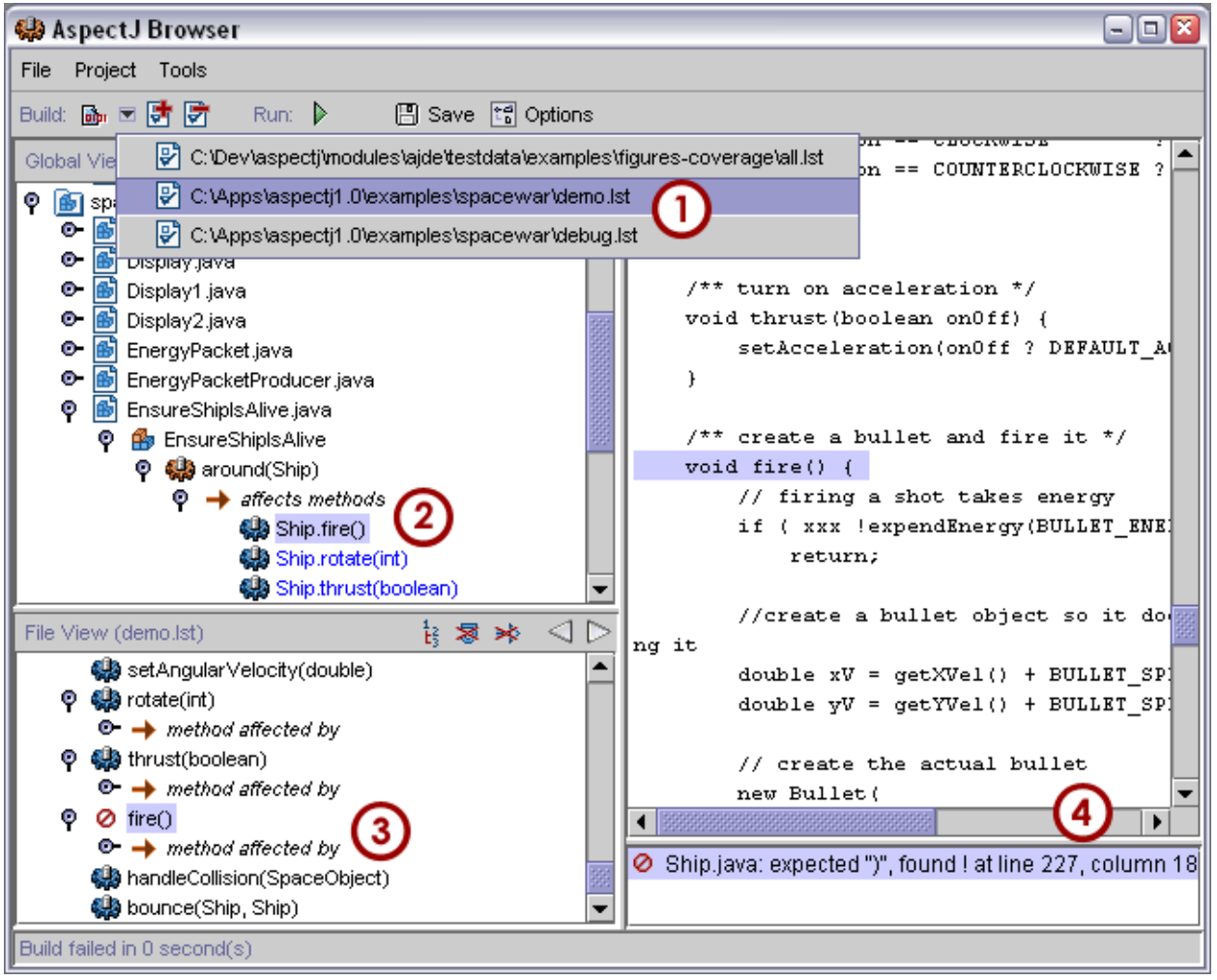
1. Introduction à l'AOP
2. Introduction à AspectJ
3. Syntaxe AspectJ
4. *Développement logiciel avec AspectJ*
5. Synthèse et Références

Compilateur AspectJ

- Commande *ajc*



Browser AspectJ (ajbrowser)



Synthèse : AOP en 2 phrases

- Intérêt de l'AOP
  - Extension d'un système existant pour prise en compte de nouvelles propriétés (aspects) orthogonaux
  - Conception modulaire d'un système impliquant plusieurs aspect (fonctionnels et orthogonaux)

Références Web

- Outils, articles et autres ressources relatives à la programmation par aspect : http://aosd.net
- AspectJ de Xerox, un logiciel libre de l'AOP pour Java : http://aspectj.org
- Techniques AOP utilisées dans d'autres langages que Java. Voir AspectC++, une extension de C++ pour l'AOP : http://www.aspectc.org/
- Techniques AOP techniques utilisées dans les langages procéduraux. Voir AspectC, une extension de C pour l'AOP : http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

Références bibliographiques

- **Aspect-Oriented Programming**, Kiczales et. al., ECOOP'1997
- **An Overview of AspectJ**, Kiczales et. al., ECOOP'2001
- **Getting Started with AspectJ**, Kiczales et. al., CACM 44(10), Oct. 2001
- **Aspect-Oriented Programming with AspectJ**, Ivan Kiselev, SAMS 2002
- **Mastering AspectJ: Aspect-Oriented Programming in Java**, Joseph D. Gradecki, Nicholas Lesiecki, John Wiley & Sons 2003