

Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science



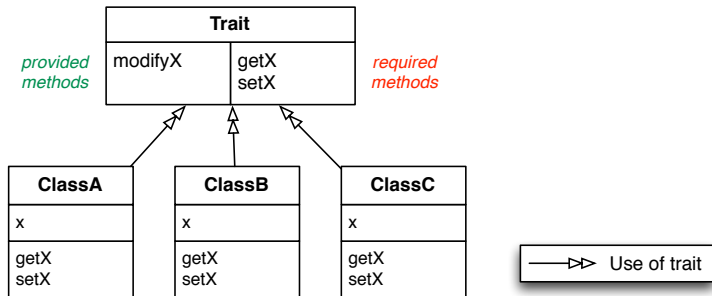
TECHNISCHE
UNIVERSITÄT
DARMSTADT

2015-05-12

Stateful Traits

Problem with Stateless Traits

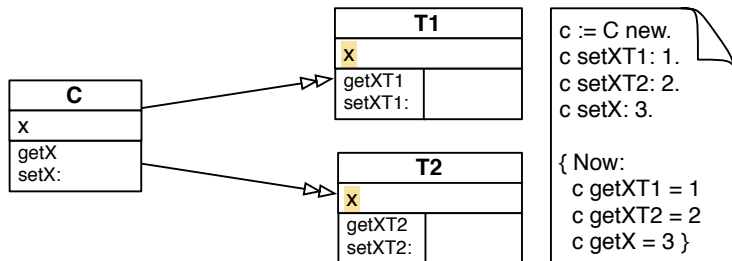
Stateless traits often require accessor methods.



- ▶ Reusability impacted because require interface is cluttered with uninteresting accessor methods.
- ▶ All client classes need to implement accessors (code duplication).
- ▶ Introduce new state in a trait \Rightarrow client classes need to change (code fragility).
- ▶ Public accessors might break encapsulation of client class.

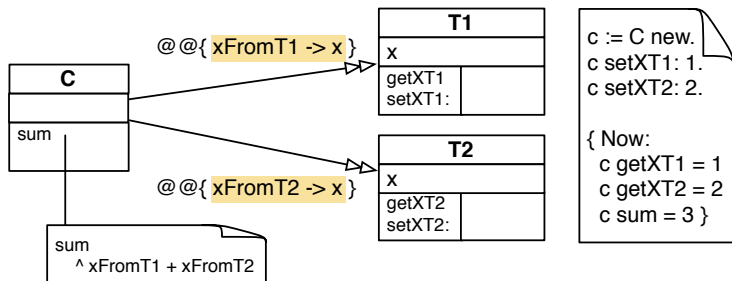
Stateful Traits

- ▶ Traits may have fields (instance variables).
- ▶ Fields are private to the scope of the trait that defines them.



Each **x** (in **C**, **T1** and **T2**) is distinct.

Granting Access to Trait Fields

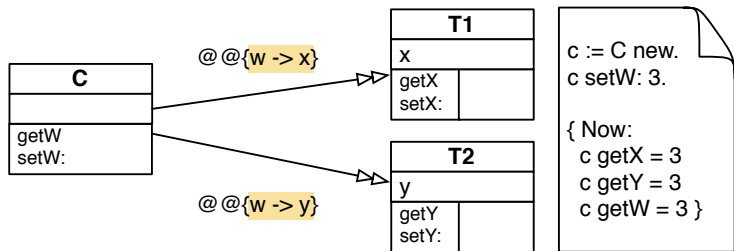


Granting access to x of T1 and x of T2 in C

- ▶ Fields are *private to the trait* where they are defined.
- ▶ Selected fields can be made accessible to the client (i.e. a class or composite trait), possibly mapping them to new names.

Merging Fields

- ▶ Client may merge variables of different traits by mapping them to a common name.
- ▶ Can never have two different variables of the same name in the same scope.



Merging variables x and y in C under name w:
w is shared between C, T1 and T2.

Beyond Inheritance: Aspect-Oriented Programming (AOP)

Motivation

Software Engineering Goals

- ▶ Reduce software complexity & improve understandability
- ▶ Promote reuse
- ▶ Facilitate evolution

Concepts & Methods

- ▶ Effective decomposition & composition mechanisms
- ▶ Reusable components, low coupling, non-invasive adaptation
- ▶ Low coupling, traceability across the software lifecycle

Our ability to achieve the goals of software engineering depends fundamentally on our ability to *keep separate all concerns* of importance in software systems.

[Tarr et al. 1999]

Separation of Concerns

The principle of dividing a program into distinct features with as little overlap in functionality as possible

Separation of concerns is achieved through mechanism of software decomposition and composition (modularisation).

Problems:

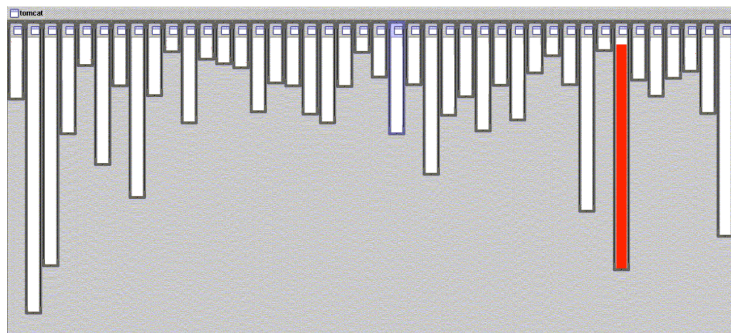
- ▶ Typically, concern separation along a single dimension (e.g: classification into an inheritance hierarchy)
- ▶ Some concerns cannot be easily separated and encapsulated using the available modularisation mechanism. They are *cross-cutting*.

Cross-cutting Concern

Behavior that cuts across the typical divisions of responsibility.

Examples: logging, security, persistence

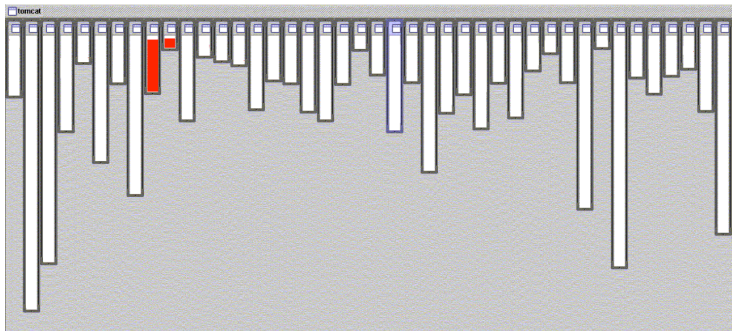
Example: Good Separation of Concerns



XML parsing in `org.apache.tomcat`

- ▶ Nicely fits in one class

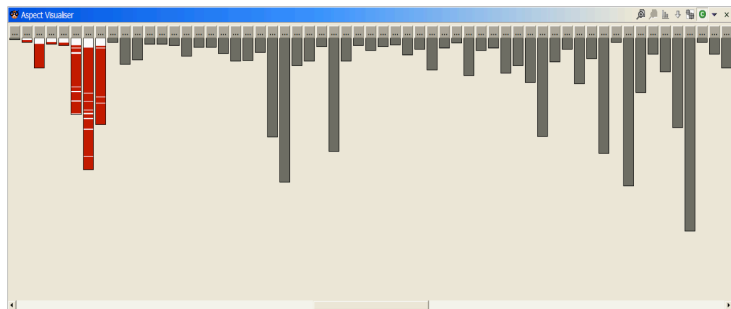
Example: Good Separation of Concerns



URL pattern matching in `org.apache.tomcat`

- ▶ Nicely fits in two classes (using inheritance)

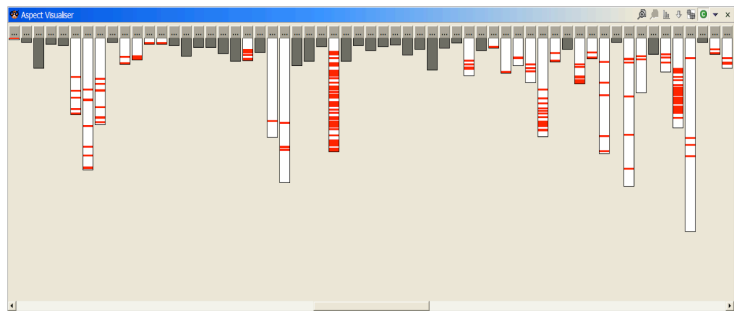
Example: Pretty Good Separation of Concerns



Class loading in `org.apache.tomcat`

- ▶ Mostly in one package (9 classes)

Example: Not So Good Separation of Concerns



Logging in `org.apache.tomcat`

- ▶ Code scattered across all packages and classes
- ▶ Logging is a cross-cutting concern
- ▶ Cannot be modularised using the *dominant decomposition* mechanism (i.e. classes)

The Tyranny of the Dominant Decomposition

The Tyranny

- ▶ Most languages provide a single (“dominant”) decomposition mechanism for separating concerns (e.g. procedures, functions, classes).
- ▶ Some concerns cannot be effectively separated using the provided decomposition mechanism – they end up scattered across many modules and tangled with one another.

Breaking the Tyranny

- ▶ Provide multiple decomposition mechanisms simultaneously, i.e. support for *multi-dimensional* separation of concerns.
- ▶ Modularise each concern in the system using a suitable decomposition mechanism.

[Tarr et al. 1999]

Aspect Oriented Programming: Idea

The hierarchical modularity mechanisms of object-oriented languages are inherently unable to modularize *all* concerns of interest in complex systems.

[Kiczales et al. 2001]

Cross-cutting concerns

- ▶ Have a clear purpose.
- ▶ Have a natural structure.

So let's capture the structure of cross-cutting concerns explicitly

- ▶ In a modular way,
- ▶ With linguistic and tool support.

Aspects are *well-modularised cross-cutting concerns*.

Logging Example: Without AOP

ApplicationSession

```
1 public class ApplicationSession {
2     private static final Logger logger =
3         LoggerFactory.getLogger(ApplicationSession.class);
4
5     private final Session session;
6
7     public ApplicationSession(Session session) {
8         this.session = session;
9     }
10
11     public void login(String username, String password) {
12         logger.info("User {} is logging in", username);
13         // ... login logic ...
14     }
15
16     public void logout() {
17         logger.info("User {} is logging out", session.getId());
18         // ... logout logic ...
19     }
20
21     // ... other methods ...
22 }
```

StandardSession

```
1 public class StandardSession {
2     private static final Logger logger =
3         LoggerFactory.getLogger(StandardSession.class);
4
5     private final Session session;
6
7     public StandardSession(Session session) {
8         this.session = session;
9     }
10
11     public void login(String username, String password) {
12         logger.info("User {} is logging in", username);
13         // ... login logic ...
14     }
15
16     public void logout() {
17         logger.info("User {} is logging out", session.getId());
18         // ... logout logic ...
19     }
20
21     // ... other methods ...
22 }
```

ServerSession

```
1 public class ServerSession {
2     private static final Logger logger =
3         LoggerFactory.getLogger(ServerSession.class);
4
5     private final Session session;
6
7     public ServerSession(Session session) {
8         this.session = session;
9     }
10
11     public void login(String username, String password) {
12         logger.info("User {} is logging in", username);
13         // ... login logic ...
14     }
15
16     public void logout() {
17         logger.info("User {} is logging out", session.getId());
18         // ... logout logic ...
19     }
20
21     // ... other methods ...
22 }
```

SessionInterceptor

```
1 public class SessionInterceptor {
2     private static final Logger logger =
3         LoggerFactory.getLogger(SessionInterceptor.class);
4
5     public void login(String username, String password) {
6         logger.info("User {} is logging in", username);
7         // ... login logic ...
8     }
9
10     public void logout() {
11         logger.info("User {} is logging out", session.getId());
12         // ... logout logic ...
13     }
14
15     // ... other methods ...
16 }
```

ServerSessionManager

```
1 public class ServerSessionManager {
2     private static final Logger logger =
3         LoggerFactory.getLogger(ServerSessionManager.class);
4
5     // ... session management logic ...
6
7     // ... other methods ...
8 }
```

StandardManager

```
1 public class StandardManager {
2     private static final Logger logger =
3         LoggerFactory.getLogger(StandardManager.class);
4
5     // ... session management logic ...
6
7     // ... other methods ...
8 }
```

StandardSessionManager

```
1 public class StandardSessionManager {
2     private static final Logger logger =
3         LoggerFactory.getLogger(StandardSessionManager.class);
4
5     // ... session management logic ...
6
7     // ... other methods ...
8 }
```


Logging Example: With AOP

1. **QUESTION**
 2. **ANSWER**
 3. **QUESTION**
 4. **ANSWER**
 5. **QUESTION**
 6. **ANSWER**
 7. **QUESTION**
 8. **ANSWER**
 9. **QUESTION**
 10. **ANSWER**
 11. **QUESTION**
 12. **ANSWER**
 13. **QUESTION**
 14. **ANSWER**
 15. **QUESTION**
 16. **ANSWER**
 17. **QUESTION**
 18. **ANSWER**
 19. **QUESTION**
 20. **ANSWER**
 21. **QUESTION**
 22. **ANSWER**
 23. **QUESTION**
 24. **ANSWER**
 25. **QUESTION**
 26. **ANSWER**
 27. **QUESTION**
 28. **ANSWER**
 29. **QUESTION**
 30. **ANSWER**
 31. **QUESTION**
 32. **ANSWER**
 33. **QUESTION**
 34. **ANSWER**
 35. **QUESTION**
 36. **ANSWER**
 37. **QUESTION**
 38. **ANSWER**
 39. **QUESTION**
 40. **ANSWER**
 41. **QUESTION**
 42. **ANSWER**
 43. **QUESTION**
 44. **ANSWER**
 45. **QUESTION**
 46. **ANSWER**
 47. **QUESTION**
 48. **ANSWER**
 49. **QUESTION**
 50. **ANSWER**
 51. **QUESTION**
 52. **ANSWER**
 53. **QUESTION**
 54. **ANSWER**
 55. **QUESTION**
 56. **ANSWER**
 57. **QUESTION**
 58. **ANSWER**
 59. **QUESTION**
 60. **ANSWER**
 61. **QUESTION**
 62. **ANSWER**
 63. **QUESTION**
 64. **ANSWER**
 65. **QUESTION**
 66. **ANSWER**
 67. **QUESTION**
 68. **ANSWER**
 69. **QUESTION**
 70. **ANSWER**
 71. **QUESTION**
 72. **ANSWER**
 73. **QUESTION**
 74. **ANSWER**
 75. **QUESTION**
 76. **ANSWER**
 77. **QUESTION**
 78. **ANSWER**
 79. **QUESTION**
 80. **ANSWER**
 81. **QUESTION**
 82. **ANSWER**
 83. **QUESTION**
 84. **ANSWER**
 85. **QUESTION**
 86. **ANSWER**
 87. **QUESTION**
 88. **ANSWER**
 89. **QUESTION**
 90. **ANSWER**
 91. **QUESTION**
 92. **ANSWER**
 93. **QUESTION**
 94. **ANSWER**
 95. **QUESTION**
 96. **ANSWER**
 97. **QUESTION**
 98. **ANSWER**
 99. **QUESTION**
 100. **ANSWER**

[illegible][illegible]

1. 在 2.1 节中, 我们定义了
 2. 在 2.2 节中, 我们定义了
 3. 在 2.3 节中, 我们定义了
 4. 在 2.4 节中, 我们定义了
 5. 在 2.5 节中, 我们定义了
 6. 在 2.6 节中, 我们定义了
 7. 在 2.7 节中, 我们定义了
 8. 在 2.8 节中, 我们定义了
 9. 在 2.9 节中, 我们定义了
 10. 在 2.10 节中, 我们定义了
 11. 在 2.11 节中, 我们定义了
 12. 在 2.12 节中, 我们定义了
 13. 在 2.13 节中, 我们定义了
 14. 在 2.14 节中, 我们定义了
 15. 在 2.15 节中, 我们定义了
 16. 在 2.16 节中, 我们定义了
 17. 在 2.17 节中, 我们定义了
 18. 在 2.18 节中, 我们定义了
 19. 在 2.19 节中, 我们定义了
 20. 在 2.20 节中, 我们定义了
 21. 在 2.21 节中, 我们定义了
 22. 在 2.22 节中, 我们定义了
 23. 在 2.23 节中, 我们定义了
 24. 在 2.24 节中, 我们定义了
 25. 在 2.25 节中, 我们定义了
 26. 在 2.26 节中, 我们定义了
 27. 在 2.27 节中, 我们定义了
 28. 在 2.28 节中, 我们定义了
 29. 在 2.29 节中, 我们定义了
 30. 在 2.30 节中, 我们定义了
 31. 在 2.31 节中, 我们定义了
 32. 在 2.32 节中, 我们定义了
 33. 在 2.33 节中, 我们定义了
 34. 在 2.34 节中, 我们定义了
 35. 在 2.35 节中, 我们定义了
 36. 在 2.36 节中, 我们定义了
 37. 在 2.37 节中, 我们定义了
 38. 在 2.38 节中, 我们定义了
 39. 在 2.39 节中, 我们定义了
 40. 在 2.40 节中, 我们定义了
 41. 在 2.41 节中, 我们定义了
 42. 在 2.42 节中, 我们定义了
 43. 在 2.43 节中, 我们定义了
 44. 在 2.44 节中, 我们定义了
 45. 在 2.45 节中, 我们定义了
 46. 在 2.46 节中, 我们定义了
 47. 在 2.47 节中, 我们定义了
 48. 在 2.48 节中, 我们定义了
 49. 在 2.49 节中, 我们定义了
 50. 在 2.50 节中, 我们定义了
 51. 在 2.51 节中, 我们定义了
 52. 在 2.52 节中, 我们定义了
 53. 在 2.53 节中, 我们定义了
 54. 在 2.54 节中, 我们定义了
 55. 在 2.55 节中, 我们定义了
 56. 在 2.56 节中, 我们定义了
 57. 在 2.57 节中, 我们定义了
 58. 在 2.58 节中, 我们定义了
 59. 在 2.59 节中, 我们定义了
 60. 在 2.60 节中, 我们定义了
 61. 在 2.61 节中, 我们定义了
 62. 在 2.62 节中, 我们定义了
 63. 在 2.63 节中, 我们定义了
 64. 在 2.64 节中, 我们定义了
 65. 在 2.65 节中, 我们定义了
 66. 在 2.66 节中, 我们定义了
 67. 在 2.67 节中, 我们定义了
 68. 在 2.68 节中, 我们定义了
 69. 在 2.69 节中, 我们定义了
 70. 在 2.70 节中, 我们定义了
 71. 在 2.71 节中, 我们定义了
 72. 在 2.72 节中, 我们定义了
 73. 在 2.73 节中, 我们定义了
 74. 在 2.74 节中, 我们定义了
 75. 在 2.75 节中, 我们定义了
 76. 在 2.76 节中, 我们定义了
 77. 在 2.77 节中, 我们定义了
 78. 在 2.78 节中, 我们定义了
 79. 在 2.79 节中, 我们定义了
 80. 在 2.80 节中, 我们定义了
 81. 在 2.81 节中, 我们定义了
 82. 在 2.82 节中, 我们定义了
 83. 在 2.83 节中, 我们定义了
 84. 在 2.84 节中, 我们定义了
 85. 在 2.85 节中, 我们定义了
 86. 在 2.86 节中, 我们定义了
 87. 在 2.87 节中, 我们定义了
 88. 在 2.88 节中, 我们定义了
 89. 在 2.89 节中, 我们定义了
 90. 在 2.90 节中, 我们定义了
 91. 在 2.91 节中, 我们定义了
 92. 在 2.92 节中, 我们定义了
 93. 在 2.93 节中, 我们定义了
 94. 在 2.94 节中, 我们定义了
 95. 在 2.95 节中, 我们定义了
 96. 在 2.96 节中, 我们定义了
 97. 在 2.97 节中, 我们定义了
 98. 在 2.98 节中, 我们定义了
 99. 在 2.99 节中, 我们定义了
 100. 在 2.100 节中, 我们定义了

1. **What is the main purpose of the document?**
The document is a **technical specification** for a **software system**.

2. **What are the key components of the system?**
The system consists of the following components:
- **Module A**: The main processing unit.
- **Module B**: The data management unit.
- **Module C**: The user interface.
- **Module D**: The system administration unit.

3. **What are the requirements for the system?**
The system must meet the following requirements:
- **Performance**: The system must be able to process data at a rate of **1000 transactions per second**.
- **Reliability**: The system must have a **99.99% uptime**.
- **Security**: The system must be able to protect data from unauthorized access.

4. **What are the installation and configuration instructions?**
The system can be installed on a **Windows** or **Linux** operating system. The configuration instructions are as follows:
- **Step 1**: Install the system files.
- **Step 2**: Configure the database connection.
- **Step 3**: Configure the user interface.

5. **What are the maintenance and support instructions?**
The system requires regular maintenance and support. The instructions are as follows:
- **Step 1**: Check the system logs for errors.
- **Step 2**: Update the system software.
- **Step 3**: Contact the support team for assistance.

[illegible]

1. 在 2.1 节中, 我们学习了如何从文本文件中读取数据, 并存储在列表中。现在, 我们将学习如何将数据写入文件。

2. 在 2.2 节中, 我们将学习如何将数据写入文件。

3. 在 2.3 节中, 我们将学习如何将数据写入文件。

4. 在 2.4 节中, 我们将学习如何将数据写入文件。

5. 在 2.5 节中, 我们将学习如何将数据写入文件。

6. 在 2.6 节中, 我们将学习如何将数据写入文件。

7. 在 2.7 节中, 我们将学习如何将数据写入文件。

8. 在 2.8 节中, 我们将学习如何将数据写入文件。

9. 在 2.9 节中, 我们将学习如何将数据写入文件。

10. 在 2.10 节中, 我们将学习如何将数据写入文件。

Two AOP Characteristics

Quantification

Aspects make **quantified** statements about the behavior of programs:

*In programs P , **whenever condition C arises**, perform action A .*

- ▶ Static: C is a condition over the program's structure
e.g. “when calling method m ”
- ▶ Dynamic: C is a condition that happens at runtime
e.g. “when value of variable x is negative”

Obliviousness

Quantifications hold over programs that are **oblivious** to these quantified statements: P is **not aware** of A .

In other words, aspects refer to core classes, but classes do not refer to aspects.

[Filman and Friedman 2000]

Two Kinds of Crosscutting Implementation

Static Crosscutting

- ▶ Changes the static structure of program
- ▶ By defining new operations on existing types
- ▶ Like *Open Classes* in Ruby, MultiJava, ...

Dynamic Crosscutting

- ▶ Modifies the runtime behaviour of a system
- ▶ By defining additional implementation to run at certain well-defined points in the program

Static Crosscutting: JastAdd

- ▶ JastAdd¹ is an aspect-oriented compiler compiler system.
- ▶ Abstract syntax of language is transformed to OO class hierarchy: *AST classes* model AST nodes.
- ▶ Aspects allow to add features to AST classes without having to syntactically edit those classes.

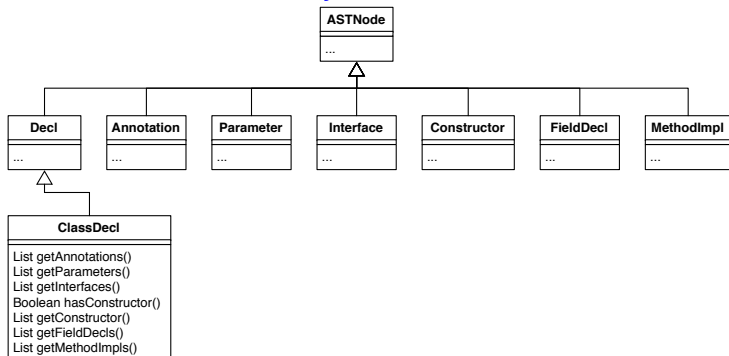
¹<http://jastadd.org/>

JastAdd Example (1)

Abstract syntax:

```
ClassDecl:Decl ::= Annotation* Parameter* Interface*  
                [Constructor] FieldDecl* MethodImpl* ;  
Annotation ::= ...  
Parameter  ::= ...
```

Translates to OO class hierarchy of AST classes:



JastAdd Example (2)

Aspect adds operations to AST classes:

```
aspect GenerateJavaCode {  
    ClassDecl.generateJava(PrintStream stream) {  
        for (MethodImpl m : getMethodImpls()) {  
            m.generateJava(stream);  
        }  
        ...  
    }  
    Annotation.generateJava(PrintStream stream) {...}  
    Parameter.generateJava(PrintStream stream) {...}  
    Interface.generateJava(PrintStream stream) {...}  
    Constructor.generateJava(PrintStream stream) {...}  
    FieldDecl.generateJava(PrintStream stream) {...}  
    MethodImpl.generateJava(PrintStream stream) {...}  
}
```

Advantage: Task-specific behavior (type checking, optimisations, code generation, etc.) can be grouped together in one place.

Dynamic Crosscutting: AspectJ

- ▶ AspectJ² is an aspect-oriented extension to Java.
- ▶ Supports both static and dynamic crosscutting.
- ▶ AspectJ code is compiled into standard Java bytecode.

²<https://eclipse.org/aspectj/>

AspectJ Example (1)

Example:

```
class Account {  
    int balance;  
  
    void deposit(int amount) {  
        balance = balance + amount;  
    }  
  
    boolean withdraw(int amount) {  
        if (amount <= balance) {  
            balance = balance - amount;  
            return true;  
        } else  
            return false;  
    }  
}
```

We want to keep track of (log) every deposit and withdraw operation.

AspectJ Example (2)

Logging, the traditional way:

```
class Account {  
    int balance;  
    Logger logger = new Logger();  
  
    void deposit(int amount) {  
        logger.log("deposit amount: " + amount);  
        balance = balance + amount;  
    }  
    boolean withdraw(int amount) {  
        logger.log("withdraw amount: " + amount);  
        if (amount <= balance) {  
            balance = balance - amount;  
            return true;  
        } else   
            return false;  
    }  
}
```

AspectJ Example (3)

The AspectJ way: define an aspect that modifies the behaviour of the deposit and withdraw methods by adding log method calls:

```
aspect Logging {
    Logger logger = new Logger();

    before(int amount) :
        call(void Account.deposit(int)) && args(amount) {
        logger.log("deposit amount: " + amount);
    }

    before(int amount) :
        call(boolean Account.withdraw(int)) && args(amount) {
        logger.log("withdraw amount: " + amount);
    }
}
```

AspectJ: Basic Mechanisms

Advice The code to be inserted

Join Point A well-defined point in the program where advice can be inserted

Pointcut A set of join points

Aspect A code unit where advice and pointcuts are combined

Weaving Integrating Java application and aspects

Join Point and Pointcut

A pointcut is a set of join points where advice can be inserted.

```
call(void Account.deposit(int)) && args(amount) {...}
```

“Insert advice wherever method `Account.deposit(int)` is called; make `amount` variable available to the advice code.”

More pointcut examples:

```
call(* *.*(..)); //Note: ".." is also a wildcard  
call(public *.*.set*(..));  
call(void *.*.set*(..));  
call(* *.*.set*(int));  
call(String com.foo.Customer.set*(..));  
call(* com.foo.Customer+.set*(..)); // "+" cross-cuts subclasses  
call(public void com..*.set*(int));  
call(* *.*.set*(int, ..));  
call(* *.*.set*(int, .., String));
```

Kinds of Join Points

Join points can be defined to capture various kinds of program events:

- ▶ Method calls, method execution
- ▶ Getting and setting fields
- ▶ Object construction (constructor calls and execution)
- ▶ Exception handler execution
- ▶ Program state
- ▶ many more...³

Recall **quantification**:

*In programs P, **whenever condition C arises**, perform action A.*

⇒ Join points quantify over program P by specifying the condition **C**.

³See the [AspectJ Programming Guide](#) for a comprehensive list

Advice

```
before(int amount) :  
    call(void Account.deposit(int)) && args(amount) { /* code */ }
```

Advice declarations define:

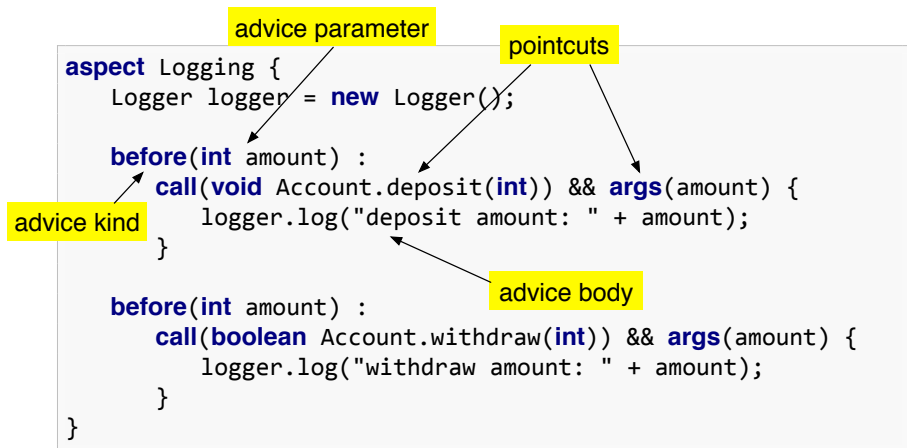
- ▶ the code body to insert
- ▶ where to insert the code (a set of join points, i.e. a pointcut)
- ▶ when to insert the code: **before**, **after** or **around** each join point

Around advice runs **in place of** the join point it operates over. It must be declared with a return type (like a method):

```
void around(int amount) :  
    call(void Account.deposit(int)) {  
        println("Depositing temporarily unavailable");  
    }
```

Aspect

An **aspect** is a code unit that encapsulates advice and pointcuts.



Weaving

General principle

Merge base program and aspects into an application that only contains base language constructs.

Example: AspectJ

Merge Java program and AspectJ aspects into standard Java bytecode.

AOP Benefits

- ▶ Prevents scattering of code that implements cross-cutting concerns.
- ▶ Lower coupling increases potential for reuse for both base language components and aspects
- ▶ Better understanding of software through higher level of abstraction and reduced complexity

AOP and OOP

- ▶ Object-Oriented Programming (OOP) decomposes a system into objects with specific behaviour.
- ▶ Certain behaviour cannot be isolated in a single object; instead it cross-cuts multiple objects.
- ▶ AOP encapsulates cross-cutting behaviour into aspects.
- ▶ AOP and OOP are orthogonal decomposition mechanisms.
- ▶ AOP does not depend on the OOP decomposition mechanism (AOP can extend e.g. procedural or functional languages).

AOP Criticism

- ▶ Aspects *localise* code belonging to one concern, but they sacrifice *locality*, i.e. the property that a statement is usually proximate to the statements executing around it.
- ▶ Understanding aspect-oriented code is difficult: need to examine advice code *and* code at program join points.
- ▶ IDE-support is essential for visualising cross-cutting concerns.
- ▶ Obliviousness of aspect application means that control flow is obscured (program cannot see aspects that apply to itself).
- ▶ AOP methodology claims to modularise crosscutting concerns, but it breaks modularity by not respecting interfaces or encapsulation.
- ▶ Monotony of motivating examples: logging, tracing, debugging – Do these require a new programming paradigm?

[Steimann 2006]

The Fragile Pointcut Problem

The quantification of pointcuts (defining where advice is executed) is very sensitive to changes in the program: renaming, moving, changing the signature of methods etc. can easily alter the set of join points.

Example. The pointcut:

```
before(int amount) :  
    call(void Account.deposit(int)) && args(amount) { /* code */ }
```

applies to:

```
class Account {  
    void deposit(int amount) {...}  
}
```

Consider what happens if we change the method's signature:

```
class Account {  
    void deposit(float amount) {...}  
}
```

This Week's Reading Assignment

- ▶ Steimann, F. *The paradoxical success of Aspect-oriented programming*. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (New York, NY, USA, 2006), OOPSLA '06, ACM Press
- ▶ Download link:
<http://dl.acm.org/citation.cfm?id=1167514>
- ▶ Freely accessible from within the TUD campus network

References I

- Filman, Robert E. and Daniel P. Friedman (2000). “Aspect-Oriented Programming is Quantification and Obliviousness”. In: *Aspect-Oriented Software Development*. Addison-Wesley.
- Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold (2001). “An Overview of AspectJ”. In: *European Conference on Object-Oriented Programming*. Vol. 2072. LNCS. Springer, pp. 327–354.
- Steimann, Friedrich (2006). “The Paradoxical Success of Aspect-oriented Programming”. In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. ACM Press, pp. 481–497.
- Tarr, Peri, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. (1999). “N Degrees of Separation: Multi-dimensional Separation of Concerns”. In: *International Conference on Software Engineering*. ICSE '99. IEEE Press, pp. 107–119.