Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

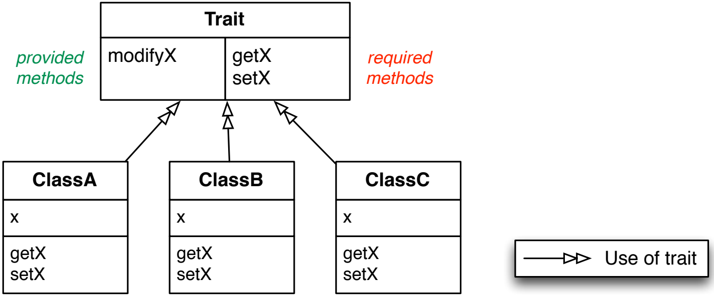
TECHNISCHE UNIVERSITÄT DARMSTADT

2015-05-12

Stateful Traits

Problem with Stateless Traits

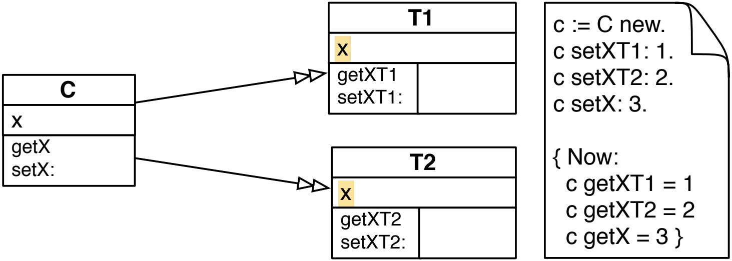
Stateless traits often require accessor methods.



- 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.

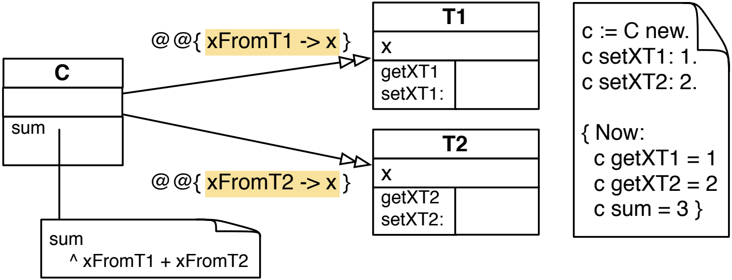
Stateful Traits

- 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

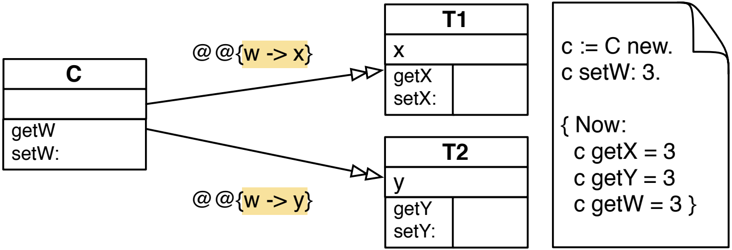


Granting access to x of T1 and x of T2 in C

- 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).

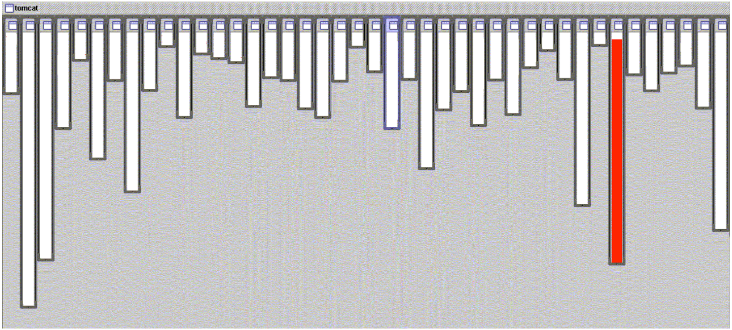
Problems:

- 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

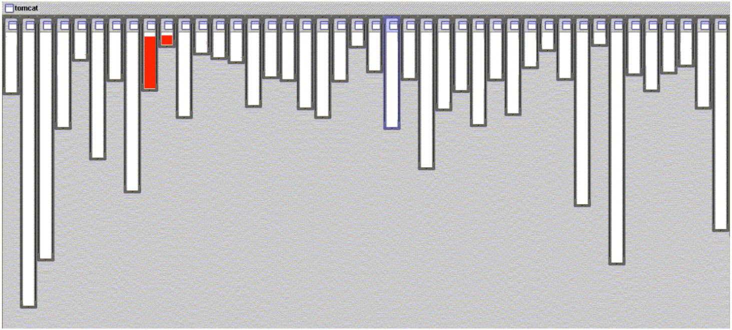
Example: Good Separation of Concerns



XML parsing in `org.apache.tomcat`

- Nicely fits in one class

Example: Good Separation of Concerns



URL pattern matching in `org.apache.tomcat`

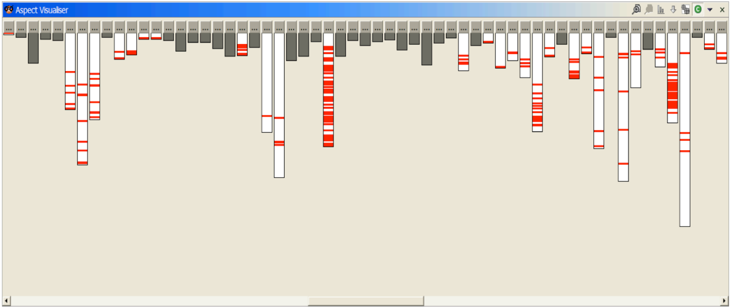
- Nicely fits in two classes (using inheritance)

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

Logging Example: With AOP

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

- JastAdd[1] 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.

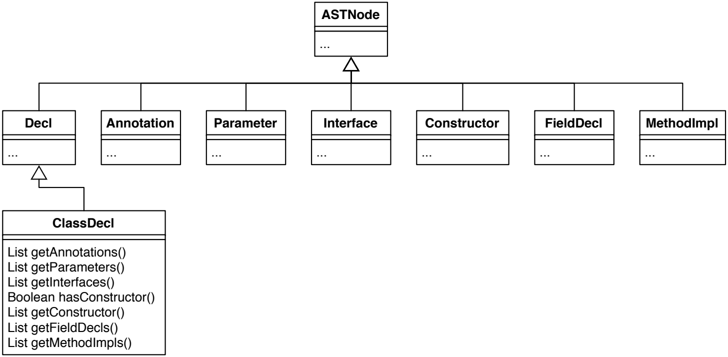
[1] `http://jastadd.org/`

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

- AspectJ[2] 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 tradtional 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.

[3] See 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.

```
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);
        }
}
```

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.