Separation of Concerns (SoC) and Aspect-Oriented Programming (AOP)

Department of Computer Science and Software Engineering
Concordia University

Summer 2013

"To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. […]

I usually refer to it as Separation of Concerns, because one tries to deal with the difficulties, the obligations, the desires, and the constraints one by one."

E. W. Dijkstra, A Discipline of Programming, 1976, last chapter, In Retrospect

Separation of Concerns (SoC)

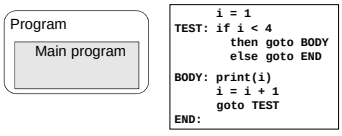
- "Separation of Concerns" (Dijkstra, Parnas): realization of problem domain concepts into separate units of software.
- There are many benefits associated with having a concern of a software system being expressed in a single modular unit.
  - Better analysis and understanding of systems, easy adaptability, maintainability and high degree of reusability.
- Crucial to software development.
- How to best achieve it is an open issue.

From machine language to assembly programming

- Machine language: **`10110000 01100001`**
- Equivalent assembly representation: **`mov al, 061h`**

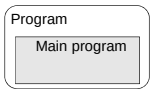
  which means to move the hexadecimal value 61 (97 decimal) into the processor register with the name "al".
- It resembles the way a machine is organized.
- Tedious and error prone.

Unstructured programming



- With GOTO, no control structures.
- Escapes from machine dominance, but it was still low level and tedious.
- Complex code: Difficult to read and write.
- Poor evolvability.
- Poor maintainability.
- Poor reusability.

Structured programming

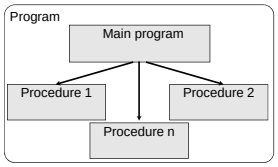



```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```

- Enabled developers to adopt high-level language constructs.
- Language features for common program patterns.
- Support for control flow: selection and loop.
- Allowed (some) abstraction from the underlying machine.
- Poor evolvability.
- Poor maintainability.
- Poor reusability.

7

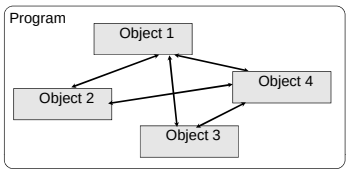
Procedural programming



- + Procedural abstractions + Parameter passing + Recursion; Program = data structures + functions.
- Allows functional decomposition of a problem.
- Top-town design and stepwise refinement allow the problem to be broken down into (manageable) subproblems.
- Modular composition is made possible as solutions to subproblems can be coded into units of functionality (procedures and functions) or components.

8

Object-oriented programming (OOP)



- + Encapsulation + Polymorphism + Inheritance
- Provides linguistic mechanisms to support functional decomposition along the notion of class.
- Allows us to view computation as a set of collaborating objects.
- Classes allow us to hide implementation details beneath interfaces.

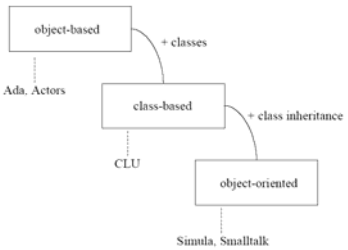
9

- Inheritance allows for reusing implementation and interface, to support an *is-a* relationship: specialization, specification, extension, and combination.
- Composition can describe a *has-a* relationship.
- Polymorphism provides a common behavior and interface for related concepts.
- Overall, OOP gets closer to the way we think.
- Has been a great success towards SoC.

10

The origins of OOP

- OOP is an evolutionary, not a revolutionary approach.



11

"The purpose of thinking is to reduce the detailed reasoning needed to a doable amount, and a separation of concerns is the way we hope to achieve this reduction. The crucial choice is, of course, what aspects to study in isolation, how to disentangle the original amorphous knot of obligations, constraints and goals into a set of concerns that admit a reasonably effective separation.
The knowledge of the goal of separation of concerns is a useful one: we are at least beginning to understand what we are aiming at."

E. W. Dijkstra, A Discipline of Programming, 1976, last chapter, In Retrospect.

12

Languages and paradigms

- Programming languages and paradigms define the way we communicate with machines.
- Each paradigm offers its own set of decomposition rules through linguistic mechanisms.
  - e.g. procedures, functions, objects, etc.
- Each paradigm has advanced our ability to achieve clear separation of concerns at the source code level.
  - It allows a more natural mapping of requirements to programming constructs.
  - Problem decomposition and programming languages have been mutually supportive.

13

Paradigms and complexity

- Evolution of programming paradigms lets us manage the increasing complexity of systems.
- The converse of this fact is equally true (and very interesting): We have allowed the existence of increasingly complex requirements because various paradigms provide mechanisms to support their implementation.
- From conventional to intelligent computing

14

Programming language paradigms and thought

- Does a natural language affect the way we think?
  - Discovery channel had a documentary about a tribe that has no notion of numbers.
  - So we cannot say things like *"how many?"*, *"in fifteen minutes"*, etc.
- Does a programming language paradigm affect the way we think about providing solutions to problems?
  - We tend to perform functional decomposition and stepwise refinement of problems in terms of programming constructs (procedures, objects, functions, etc.)

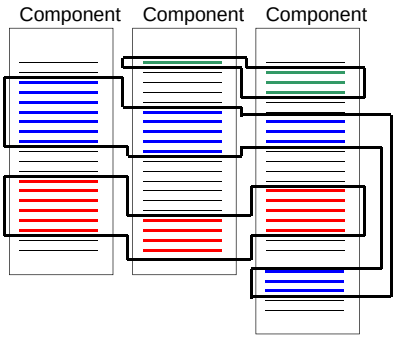
15

Crosscutting

- In OOP, decomposition (through mechanisms provided by current languages) is one-dimensional focusing on the notion of a class.
- In large systems, interaction of components is very complex.
- Current programming languages do not provide constructs to address certain properties in a modular way.
- OOP cannot address the design or implementation of behavior that spans over many modules (often unrelated).

16

An initial picture of crosscutting



17

A first example of crosscutting

- Authentication, contract checking and logging are tangled with the core operation (methods) of the component (class).

```
public class BusinessLogic {
    // data members for business logic,
    authentication, contract checking, logging.
    public void someOperation {
        // perform authentication
        // ensure preconditions
        // log the start of an operation
        … perform core operation
        // authentication
        // ensure postconditions
        // log successful termination of
        // operation
    }
    // more operations similar to the above}
```

18

Some observations

Two issues are of interest here:

1. Implementation for authentication, contract checking and logging is not localized.
   - Code spans over many methods of potentially many classes and packages.
2. Implementation of **`someOperation()`** does much more than performing some core functionality.
   - It contains code for more than one concerns.

19

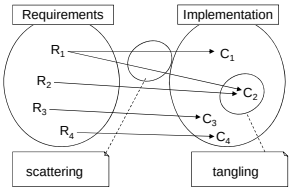
Symptoms of crosscutting

- Crosscutting imposes two symptoms on software development:
  - Code scattering: implementation of some concerns not well modularized but cuts across the decomposition hierarchy of the system.
  - Code tangling: a module may contain implementation elements (code) for various concerns.
- Scattering and tangling describe two different facets of the same problem.

20

Where does crosscutting originate?

- There might not always be a one-to-one mapping from problem domain to solution space.
- Requirements space is n-dimensional, but Implementation space is one-dimensional (In OOP everything must belong to a class).



21

Implications of crosscutting

- As a result of crosscutting, the benefits of OOP cannot be fully utilized, and developers are faced with a number of implications:
  - Poor traceability of requirements: Mapping from an n-dimensional space to a single dimensional implementation space.
  - Lower productivity: Simultaneous implementation of multiple concerns in one module breaks the focus of developers.
  - Strong coupling between modular units in classes that are difficult to understand and change.

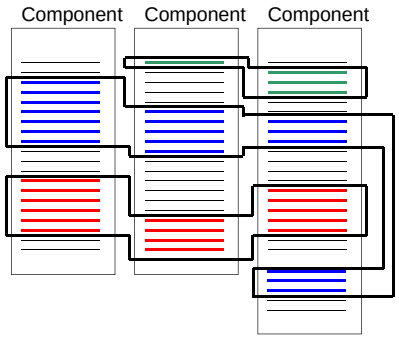
22

- Low degree of code reusability. Core functionality impossible to be reused without related semantics, already embedded in component.
- Low level of system adaptability.
- Changes in the semantics of one crosscutting concern are difficult to trace among various modules that it spans over.
- Programs are more error prone.
- Difficult evolution.

- Crosscutting affects the quality of software.

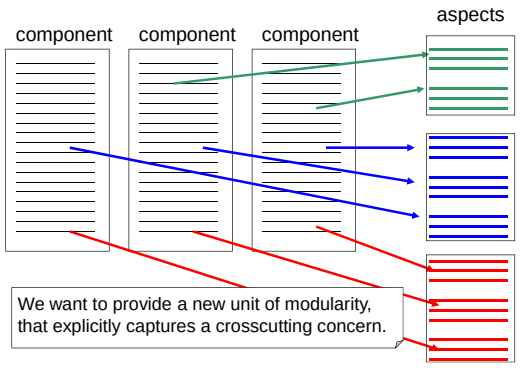
23

We would like to move from this picture…



24

…to this one!



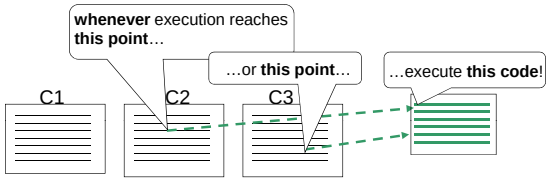
25

Aspect-oriented programming (AOP)

- Motivation: Better linguistic mechanisms are needed for SoC.
- Aspect-Oriented Programming (AOP) refers to approaches and technologies that can explicitly capture crosscutting concerns.
  - Concerns can be functional (e.g. business rules) or non-functional (e.g. synchronization).
- In OOP, the tendency is to find commonality among classes and push them up (vertically) in the inheritance hierarchy.
- In AOP, we identify scattered concerns and eject them horizontally from the object structure.
- Just as OOP did not discard the idea of block structure and structured programming, AOP does not reject existing technology.

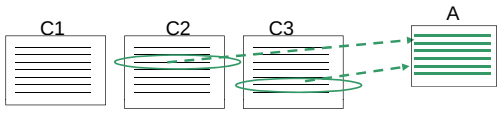
26

Principles of AOP: 1. Quantification



- *"In program P, when condition C occurs, execute action A."*

27

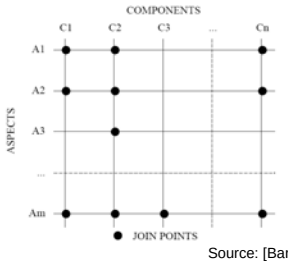


- In the above example, the points in C2 and C3 are not random; they must be well-defined.
- A join point is a well-defined point in the execution of a program.
  - Examples include calls to methods and execution of methods.

28

Components, aspects and joinpoints

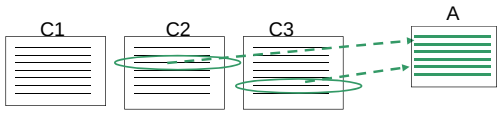
- Join points are those places where aspect code interacts with the rest of the system.



Source: [Bardou, '98]

29

Principles of AOP: 2. Obliviousness



- Note that neither **C2** nor **C3** make calls to **A** !
- Even though components **C2** and **C3** are enhanced by the aspectual behavior defined by **A**, they are not aware of this (nor have they been implemented to accept such enhancements).
- Contrast this with procedure calls or message passing.

30

Related technology: Common Lisp Object System

- CLOS methods can be augmented by *auxiliary methods* (as opposed to primary methods) including before-, after- and around-methods.
- Before- and after-methods allow us to wrap new behavior around the call to the primary method.
- before-methods allow us to say *"When a primary method is called, **before** running the code that should run, execute the code of this auxiliary method."*
- after-methods allow us to say *"When a primary method is called, **after** running its code, execute the code of this auxiliary method."*

31

- Around-methods are called instead of the primary methods. In its own discretion, an around-method may invoke its primary method via **`call-next-method`**.
- around-methods allow us to say *"When a primary method is called, **instead of** running the code that should run, execute the code of this auxiliary method."*

32

Implementing an aspect-oriented program

1. Aspectual decomposition: Identify core functionality and crosscutting concerns (aspects)
   - e.g. business logic, authentication, contract checking, logging.
2. Implement each concern (relatively) separately.
   - Core functionality: Implement core functionality in the existing unit of modularity, e.g. class.
   - Aspects: May provide linguistic support for the explicit definition of aspects.
3. Provide rules of composition between core functionality and aspects.

33

4. Composition: Can be achieved by a number of ways; the most dominant way is a linguistic approach:
   - 4.1 Provide linguistic mechanisms (constructs) to explicitly capture aspects.
   - 4.2 Provide special compilers (weavers) to combine components and aspects based on rules. The underlying AOP environment weaves or composes concerns together into a coherent program.

34

Composition rules

- Composition rules define two things:
  1. The points of communication between components and aspects (join points).
  2. The semantics of aspects to be performed on certain join points.
- In the linguistic model (implemented by languages such as AspectJ), composition rules are embedded within the aspect definitions.
- Weaver may directly produce executable, i.e. an intermediate source is not necessary.

35

Structure of an AOP program

- Program = {components}
  + {aspects}
  + {composition rules}
- Composition rules are not part of component definitions (though they may be placed inside aspect definitions).

```
class C1 {
 ...
}
```

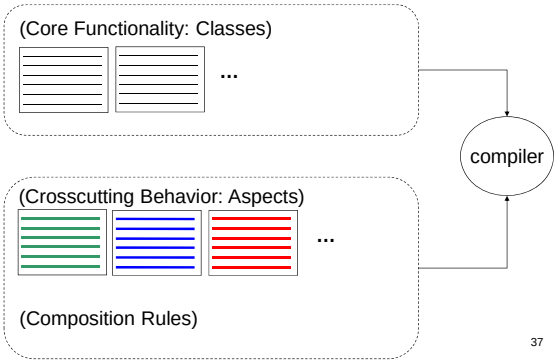
```
class C2 {
 ...
}
```

```
class C3 {
 ...
}
```

```
aspect A {
 when <condition>
 perform <action>
}
```

36

Putting everything together



37

```
public class BusinessLogic {
    … data members for business logic
    public void someOperation {
        … perform core operation}
   // more operations similar to the above}

public aspect Authenticator {
   … data members for authentication
   public void authenticate() {…}}

public aspect Logger {
   … data members for logging
   public void log() {…}}

public aspect ContractChecker {
 … data members for contract checking
    public boolean precondition() {…}
    public boolean postcondition(){..}}
```

compiler

38

Benefits of AOP

- Provisions of AOP should be compared with those of other paradigms, such as OOP.
- Clear (two-dimensional) separation of concerns.
  - Less tangled and less scattered code.
- Improves modularity: Makes concerns easier to manipulate (reason about, debug, change, and reuse).
- Higher level of abstraction.
- Good maintenance and higher level of adaptability.
- AOP is not bound to OOP.
  - In the literature you can find proposals for AOP with functional, logical and procedural programming.

39

"This is what I meant by "focusing one's attention upon a certain aspect"; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extend that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of."

E. W. Dijkstra, *A Discipline of Programming*, 1976
last chapter, In retrospect

40

References

- T. Elrad, R. E. Filman and A. Bader, Aspect-oriented programming, *Communications of the ACM*, Vol. 44, No. 10, October 2001, pp. 29-32.
- R. E. Filman and D. P. Friedman, Aspect-oriented programming is quantification and obliviousness, Proceedings of the OOPSLA Workshop on Advanced Separation of Concerns, 2000. Also RIACS Technical Report 01.12, May 2001.
- R. Laddad, I Want My AOP!; Part 1. Separate software concerns with aspect-oriented programming, *JavaWorld*, January 2002.
- F. Steimann, The paradoxical success of aspect-oriented programming, In Proceedings of OOPSLA, 2006.

41