universitätbonn

Aspektorientierte Softwareentwicklung

Vorlesung im Sommersemester 2008

Dr. Günter Kniesel
gk@cs.uni-bonn.de

Daniel Speicher
dsp@cs.uni-bonn.de

universitätbonn ROOTS

Organisatorisches

– Wann, was, wie, warum? –

Leistungspunkte, Übungen

- Vorlesung mit Übungen
  - 2+1 SWS / 4 Leistungspunkte im Bereich B
  - Übungen verpflichtend für DPO'03 (Auch sonst sehr empfohlen)
- Erstes Übungsblatt nächsten Mittwoch
- Erste Tutorien
  - Voraussichtlich am Freitag in der darauffolgenden Woche
- Bitte tragen Sie sich in die Teilnehmerliste ein!
- Bitte abonnieren Sie die Mailingliste der Vorlesung!
  - https://sewiki.iai.uni-bonn.de/service/knowhow/mailingliste
  - https://mailbox.iai.uni-bonn.de/mailman/listinfo.cgi/swt-vorlesung

Übersicht über die Tutorientermine

Gruppe	Zeit	Raum
01	Freitag 10:00	A-106
02	Freitag 10:30	A-106
03	Freitag 11:00	A-106
04	Freitag 11:30	A-106
05	Freitag 12:00	A-106
06	Freitag 12:30	A-106
07	Freitag 13:00	A-106
08	Freitag 13:30	A-106

Bitte bilden Sie Gruppen mit Kommilitonen,
die nach derselben DPO studieren!

Prüfungen, Nachfolgeveranstaltungen

- Mündliche Modulprüfung nach neuer DPO
  - am 22.07.2008 und 23.07.2008.
  - Falls Sie zu diesen Terminen nicht können, bitte frühzeitig melden.

- Mündliche C-Prüfung nach alter DPO
  - durch Prof. Cremers oder anderen habilitierten Prüfer. Themenkombination mit passenden Spezialvorlesungen in Absprache mit dem Prüfer.
  - in Verbindung mit Vorlesungen
    - ⇨ “Design Patterns SS2002”
    - ⇨ “Refactoring SS2002”
  - und Seminar (oder äquivalenten Stoff aus Buch)
    - ⇨ “Component Engineering SS2003”

- Nachfolgeveranstaltungen (soweit schon bekannt):
  - Praktikum „Agile Software Development“: 4 ½ Wochen x 8 Std./Tag (27.08.- 26.09.2008)

Skript und Literatur

- Skript
  - Gemischt deutsch / englisch
  - Zeitnah zur Vorlesung
  - https://sewiki.iai.uni-bonn.de/teaching/lectures/aosd/2008/folien
- Literaturverweise, Werkzeuge
  - https://sewiki.iai.uni-bonn.de/teaching/lectures/aosd/2008/literaturlinks
  - Wird im Laufe des Semesters ergänzt

Literatur: Empfohlene Bücher / Zeitschriften

- AOSD
  - Gregor Kiczales, et al.: “Aspect-Oriented Programming” Proceedings European Conference on Object-Oriented Programming,1997 [AOP97]
  - Communications of the ACM, Volume 44, Issue 10 (October 2001) [AOSD01]
  - Robert E. Filman, Tzilla Elrad, Siobhán Clarke, Mehmet Aksit: “Aspect-Oriented Software Development”, Addison-Wesley, 2005 [AOSD05]
  - http://aosd.net/

- AspectJ
  - Ramnivas Laddad: AspectJ in Action, Manning Publications, 2003 [AJIA03]
  - http://www.eclipse.org/aspectj/

universitätbonn ROOTS

The Goal: Modularity + 1-1 Concepts & Modules

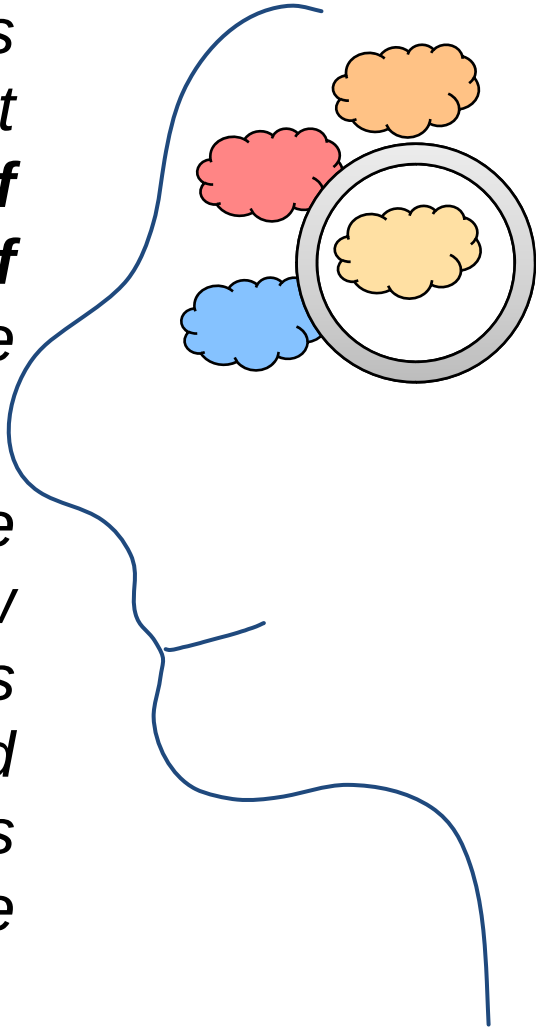
– OOP Recapitulation –

Separation of Concerns

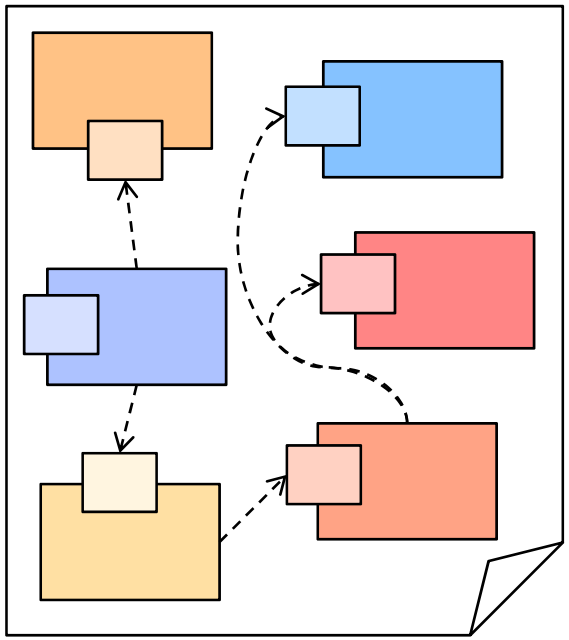
Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing ***to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency****, all the time knowing that one is occupying oneself only with one of the aspects.*

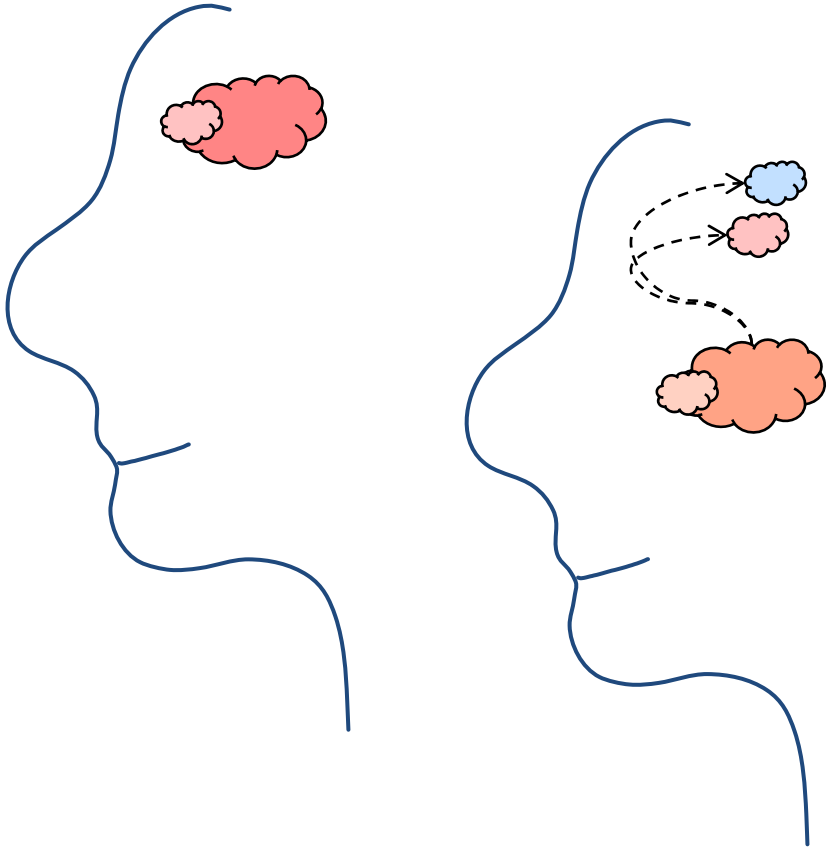
*We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "****the separation of concerns****" […]*

[Edsger W. Dijkstra, "On the role of scientific thought", 1974]



Sichtbare Schnittstellen, verborgene Details: Module





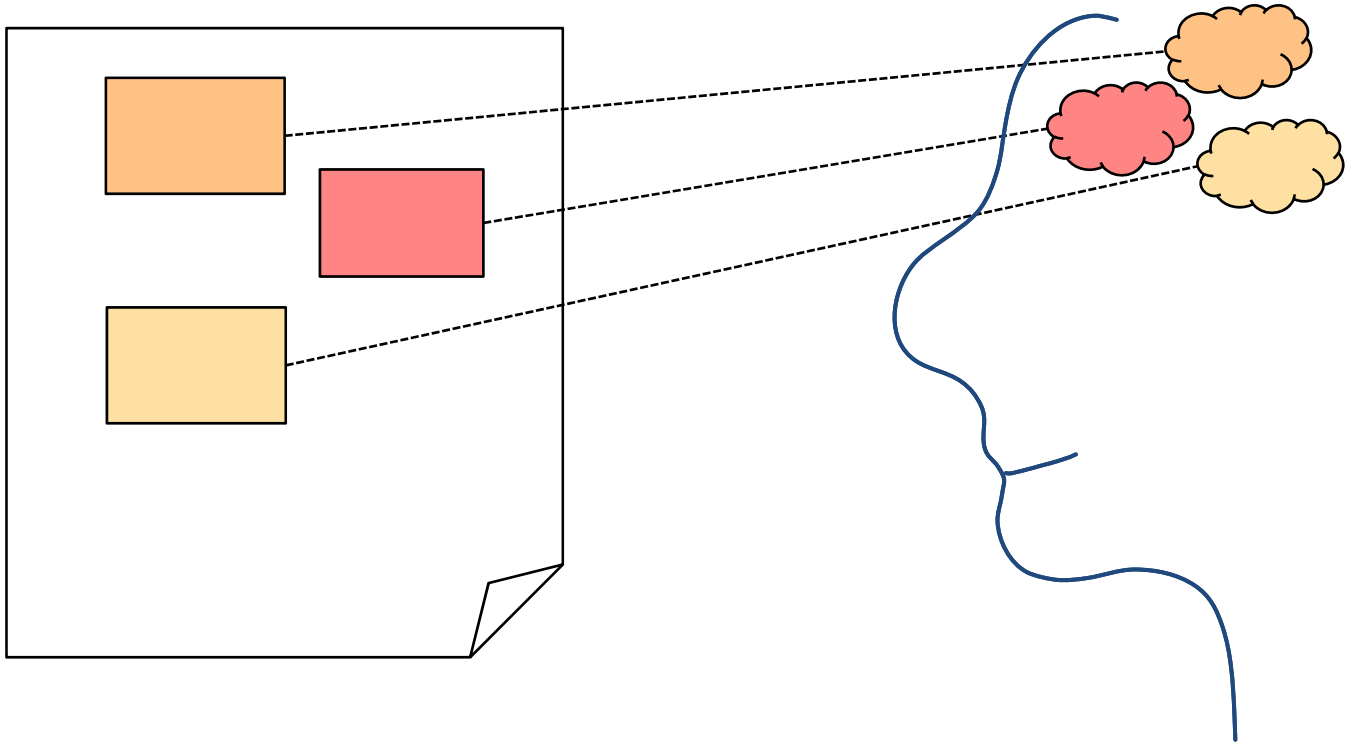
Nutzen der Modularisierung

- Parallele Entwicklung + Änderbarkeit + Verständlichkeit
  [David L. Parnas: *On the Criteria to Be Used in Decomposing Systems into Modules*, 1972]
- Wiederverwendbarkeit allgemeiner/abstrakter Funktionalität
  [Dependency Inversion Principle: „Depend on abstractions not on concretions.”]

Sichtbare Schnittstellen, verborgene Details

- Prozeduren, Funktionen, Methoden
  - Schnittstelle ist die Signatur
  - Beispiel: sort(Comparable[] items)
- Statische Module
  - C: *.h & *.c --- C++: *.hpp & *.cpp --- Modula: *.def & *.mod
- Objekte
  - Interfaces und Abstrakte Klassen als Schnittstellen
  - Kapselung: Zustandsänderung nur über Methoden
- Komponenten
- (Web) Services

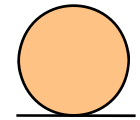
Ideal: Konzepte und Module in 1-1 Relation



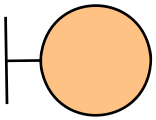
- → Verständlichkeit
- ← Natürliche Implementierung
- ← Leichte Identifikation zu ändernder Teile

OO basierter Modularität ist natürlich

- Objekt = Identität + Zustand + Verhalten
- Objekte ⇔ Dinge
- Ganzes besteht aus Teilen
- Abstraktion
  - Ersetzbarkeit ⇔ Kann betrachtet werden als …
  - Vererbung ⇔ ist ein … und hat/kann daher …



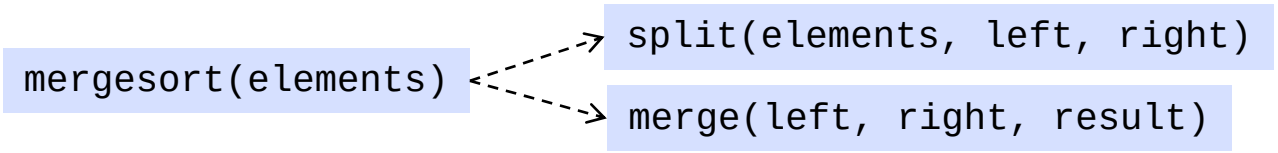
1-1 Modellierung realer Entitäten



Programmieren durch Zusammensetzen virtueller Entitäten. Insbesondere graphische Benutzer-schnittstellen.

Wiederverwendbare Abstraktionen

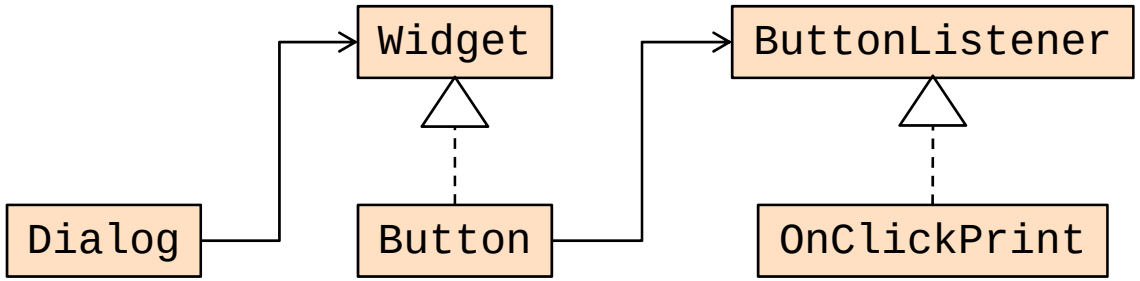
- Algorithmen: Kleine abstrakte Prozeduren.



- Große Anwendungen: Umfassender Kontrollfluss.

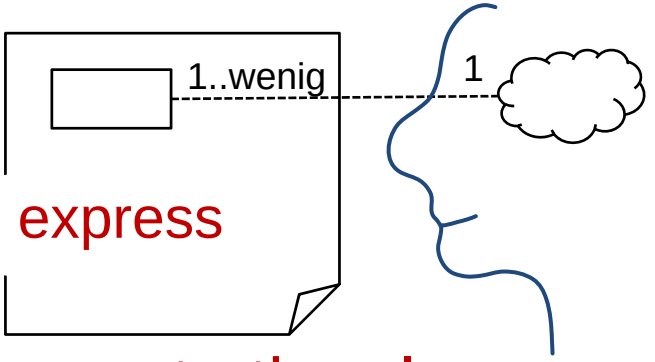
  [Open Close Principle: „Modules should be open for extension but close for modification“]

  - Beispiel: Dialog mit Eingabefeldern und Schaltern.
    - ⇨ Wiederzuverwenden ist das Zusammenspiel dieser Elemente.
    - ⇨ Erreicht z.B. durch Einsetzen eines Objekt mit geeigneter Beobachter-Schnittstelle. [Einsetzbarkeit: Liskov Substitution Principle]
  - Pattern: Observer, Template-Method, Composite…



Rules of Simple Design

- Some time ago, Kent Beck offered the following "rules" for simple design. In priority order, the code must:
  1. Run all the tests
  2. Contain no duplicate code
  3. Express all the ideas the author wants to express
  4. Minimize classes and methods



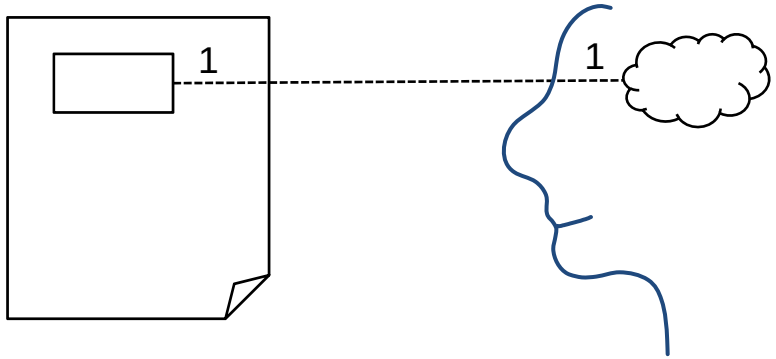
- Expressing ideas gives meaningful names to the classes and methods we have and to those that are created by the duplication rule. As well, we are often moved to create new classes and methods just for improved expression. For example, it is common to replace a switch statement with a few classes with a polymorphic method. When we see the essential idea that underlies the switch, the classes, and the name of the method, tend to pop right out.

(Ron Jeffries: http://www.xprogramming.com/xpmag/expEmergentDesign.htm)

XP: Refactor mercilessly

- Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. Make sure everything is expressed once and only once. In the end it takes less time to produce a system that is well groomed.

(Don Wells, http://www.extremeprogramming.org/rules/refactor.html)



universitätbonn ROOTS

Motivation: There is a problem to solve

– We can’t reach our goal with OO –

Limitations of OOP

- Object-oriented programming has matured to the point we are now beginning to see its limitations.
- Many requirements do not neatly decompose into behaviors centered on a single locus.
- The tyranny of the dominant decomposition: Every OO decomposition results in *cross-cutting concerns*.

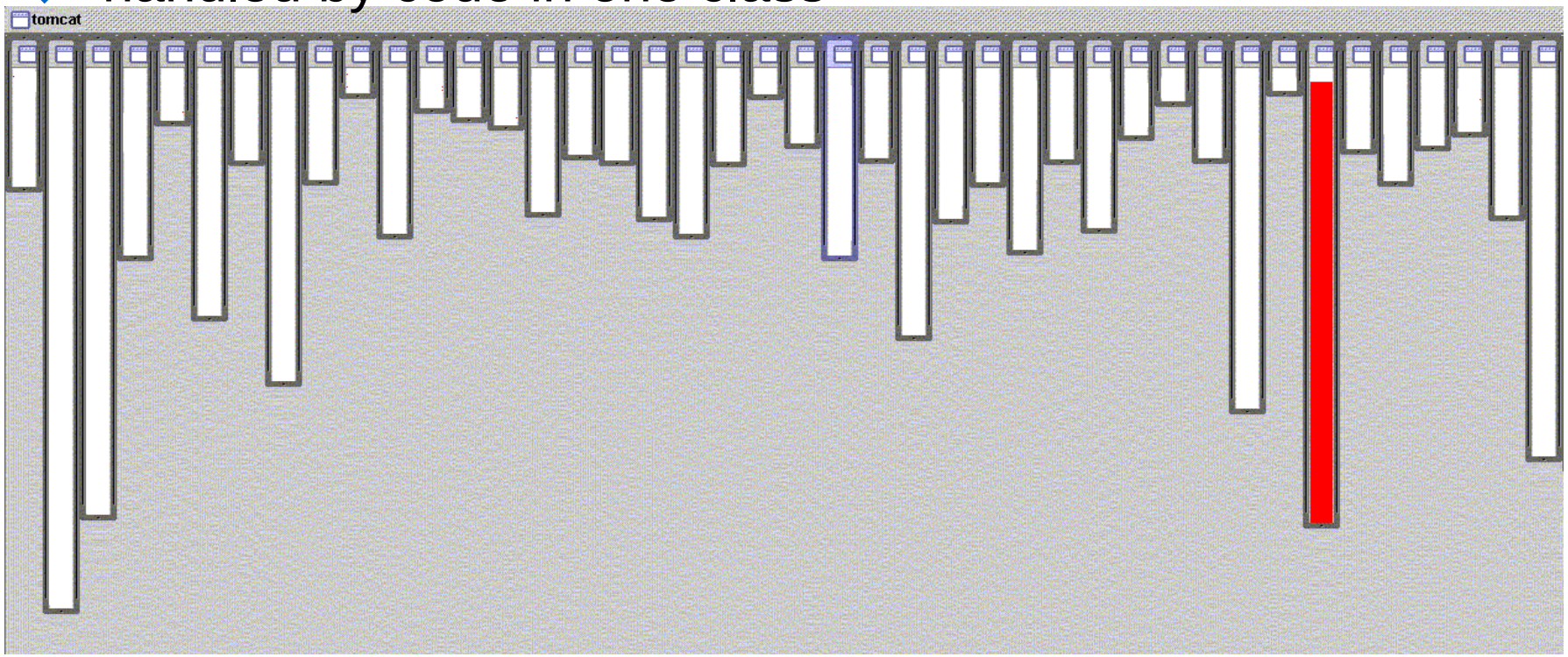
Peri Tarr et al. in “N degrees of separation: Multi-dimensional separation of concerns” in Proceedings of the 21st international conference on software engineering (ICSE’99)

An Example for the dominant decomposition

- Ivar Jacobsen
  - Use Cases and Aspects Working Seamlessly Together
  - http://aosd.net/archive/2003/jacobson-aosd-2003.ppt
  - Slides 36 – 38
  - [Slide 39: The symmetric approach to a solution: HyperJ]

XML parsing in org.apache.tomcat

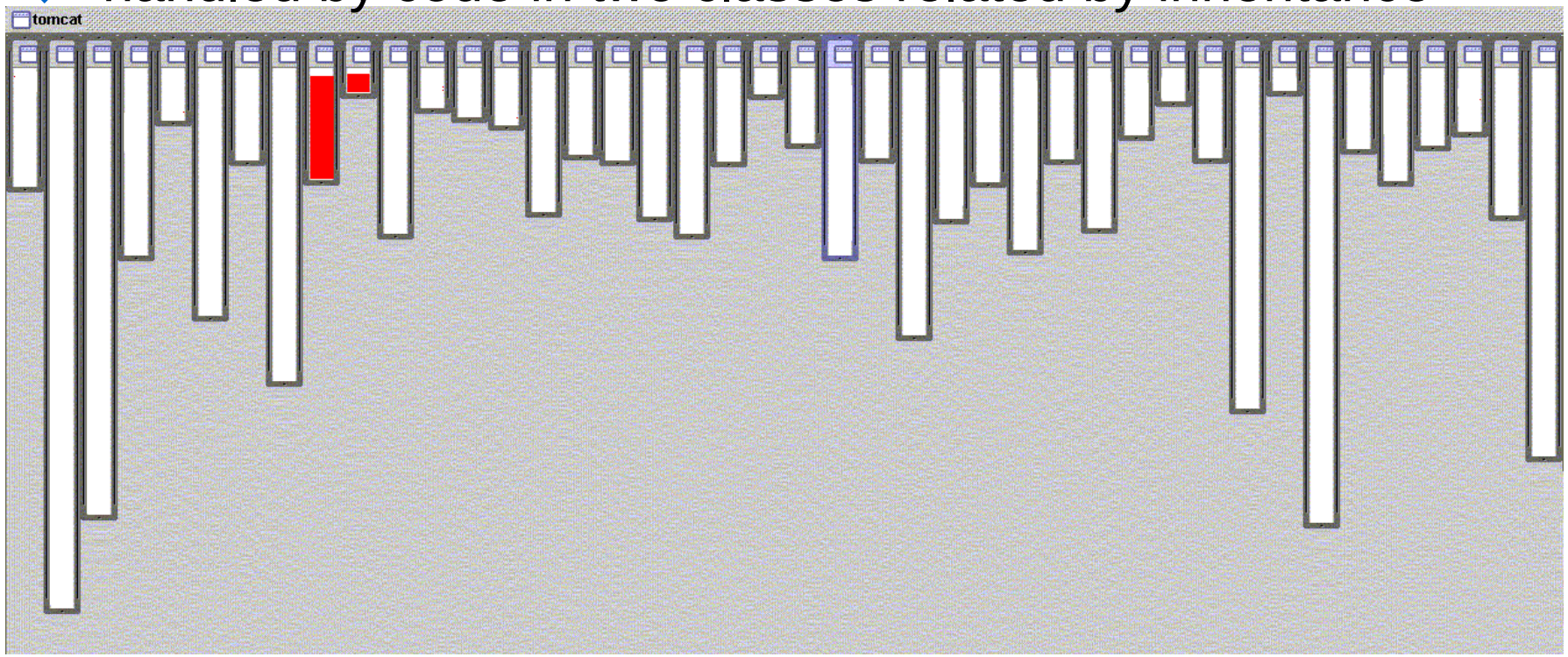
- Good modularity:
  - handled by code in one class



[Picture taken from the aspectj.org website]

URL pattern matching in org.apache.tomcat

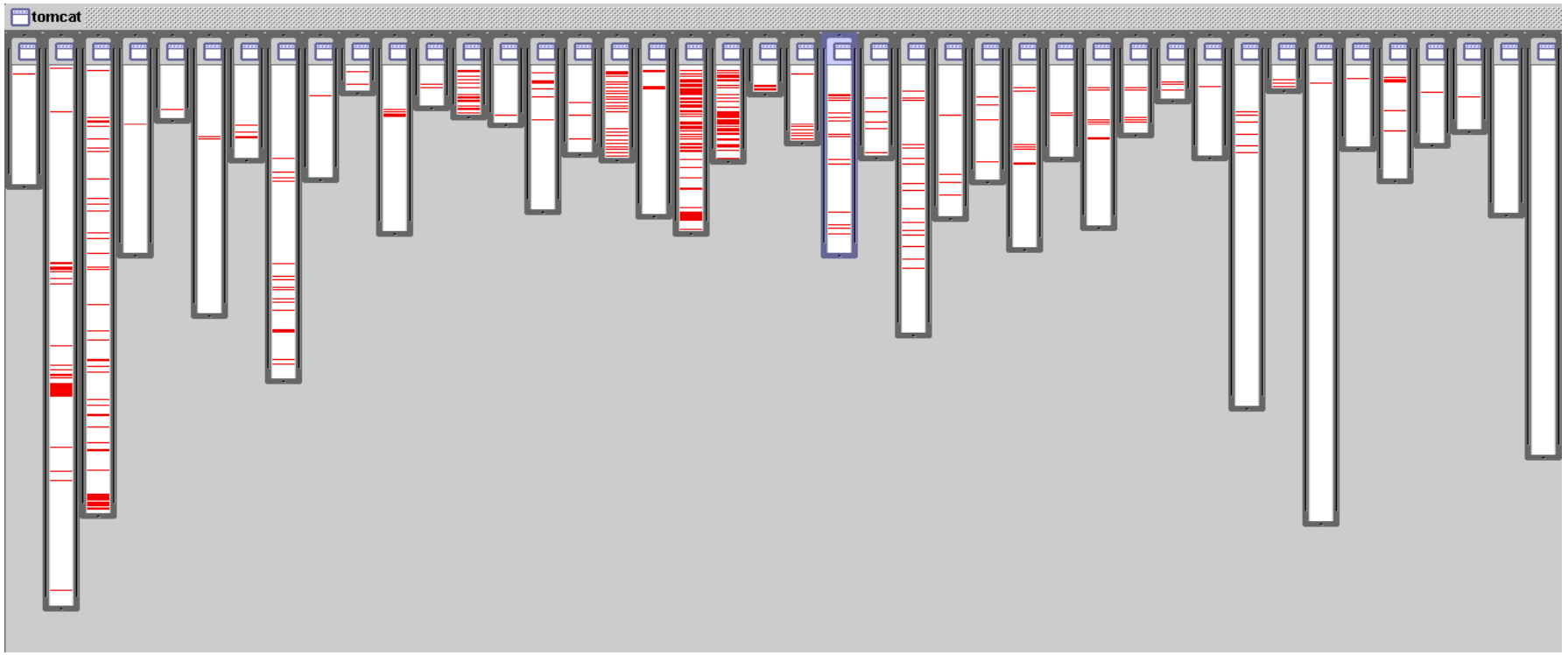
- Good modularity
  - handled by code in two classes related by inheritance



[Picture taken from the aspectj.org website]

Logging in org.apache.tomcat

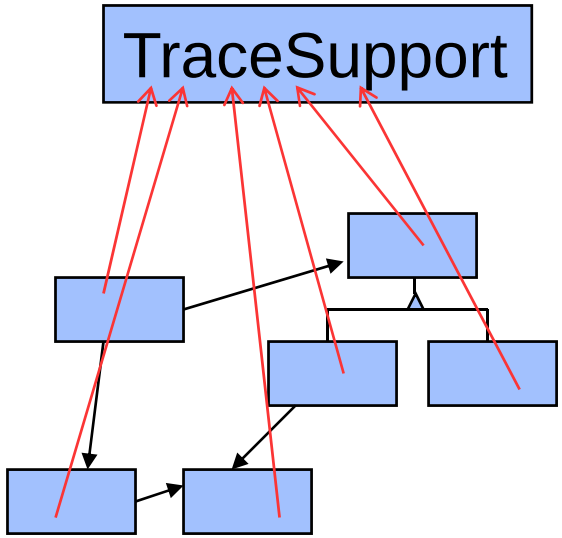
- Bad modularity
  - handled by code scattered over almost all classes



[Picture taken from the aspectj.org website]

➔ Logging is a Crosscutting Concern

Example: Tracing




```
class TraceSupport {
  static int TRACELEVEL = 0;
  static protected PrintStream stream = null;
  static protected int callDepth = -1;

  static void init(PrintStream _s) {stream=_s;}

  static void traceEntry(String str) {
    if (TRACELEVEL == 0) return;
    callDepth++;
    printEntering(str);
  }
  static void traceExit(String str) {
    if (TRACELEVEL == 0) return;
    callDepth--;
    printExiting(str);
  }
}
```

```
class Point {
  void set(int x, int y) {
    TraceSupport.traceEntry("Point.set");
    _x = x; _y = y;
    TraceSupport.traceExit("Point.set");
  }
}
```

Consistent trace form but *using* it is cross-cutting...

Other Cross-Cutting Concerns

- Systemic, “Non-Functional”
  - security (authorization, auditing, encryption)
  - logging, debugging, error handling
  - synchronization and transactions
  - usability (GUI features)
  - distribution, fault tolerance, monitoring
  - persistence and many more non-functional requirements

- Functional
  - business rules and constraints
  - traversal of complex object graphs
  - accounting mechanisms (timing and billing)
  - view update (observer)
  - filter
  - aggregation mechanisms (component gluing)

Aspect-Oriented Software Development …

- Recognises crosscutting concerns:
  - Have a clear purpose
  - Have a natural structure
  - Are inherent in any complex system
- Captures crosscutting concerns explicitly:
  - In a modular way
  - With linguistic and tool support
- In most approaches the system is divided into
  - core concerns (classes)
  - crosscutting concerns (aspects)

  ➔ Asymmetric approach

Cross Cutting leads to Bad modularity

- Scattering
  - Code addressing one concern is spread across different regions of a program
- Tangling
  - Code in one region addresses different concerns
- Scattering and tangling tend to appear together.
  - They describe different facets of the same problem.

Cost of scattered and tangled code

- Redundancy
  - Same or similar fragments of code in many places
- Difficult to understand and reason about
  - Non-explicit structure
  - The big picture isn't clear
- Difficult to change
  - Find all the code involved
  - ... and be sure to change it consistently
  - ... and be sure not to break it by accident
  - No help from OO tools

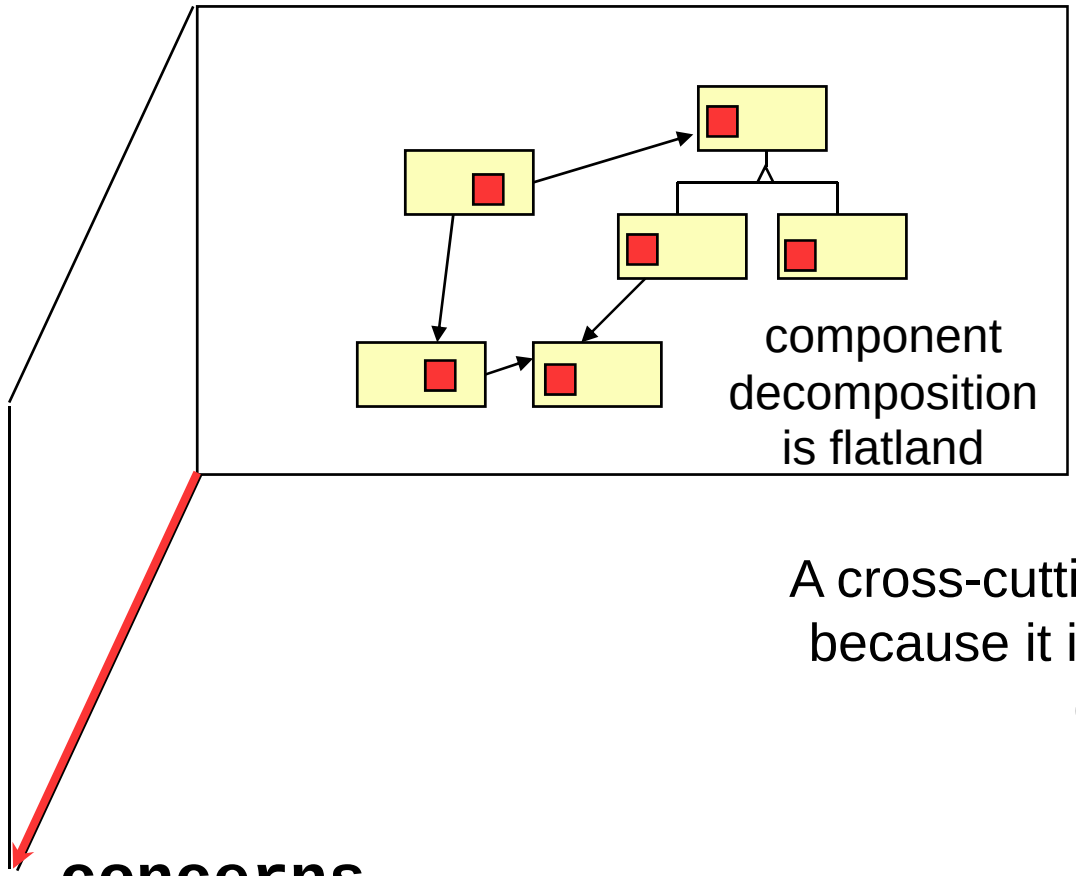
Good modularity

- Separated
  - Implementation of a concern can be treated as relatively separate entity
- Localized
  - Implementation of a concern appears in one part of program.
- Modular
  - Localized + well defined interface to the rest of the system.

Aspect-Oriented Programming

- AOP attempts to realize cross-cutting concerns as first-class elements and eject them from the object structure into a new, previously inaccessible, dimension of software development.
- AOP encapsulates behavior that OOP would scatter throughout the code by extracting cross-cutting concerns into a single textual structure called an aspect.

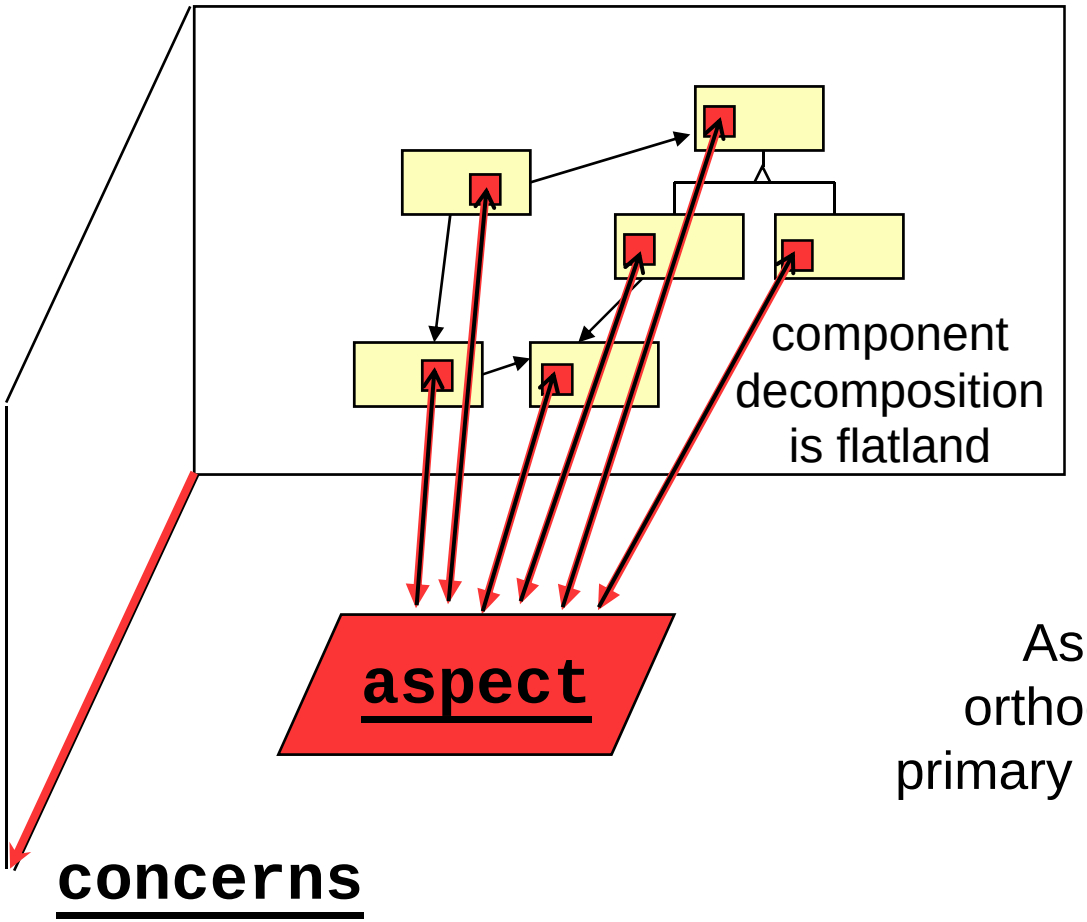
AOP: Opening a New Dimension in Software Development



A cross-cutting concern is scattered because it is realized in the wrong dimension!

Problem: Scattering & Tangling

AOP: Opening a New Dimension in Software Development



Aspects are
orthogonal to the
primary decomposition

Solution: Quantification & Obliviousness

Quantification and Obliviousness

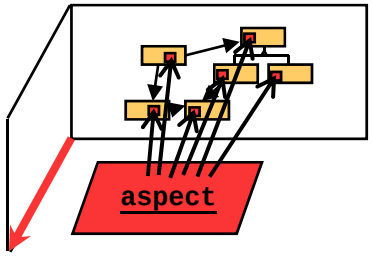
- Quantification
  - „At every occurrence of join point X do Y.“
  - ➔ Gives us the ability to enforce system-wide invariants.
- Obliviousness
  - The modified entity does not need to know about the aspect and does not need to provide any specific hooks for enabling the aspect.
  - ➔ Gives us the ability to perform unanticipated software evolution in a modular, non-invasive way!

➔ Debate: Is Obliviousness a core property of AOP?

- ➔ Without obliviousness, we might need scattered markers that say „please apply your aspect to me“.
- ➔ That might be a less severe form of scattering but it still is scattering.
- ➔ AOP without obliviousness is just half AOP.

The Four Technical Ingredients of AOP

1. Joinpoint model: common frame of reference to define the scope of cross-cutting concerns



2. Means to identify joinpoints

3. Means to influence structure and behavior at joinpoints

4. Means to weave everything together into a functional system

Expected Benefits of AOP

- Good modularity, even for crosscutting concerns
  - less tangled code
  - more natural code
  - shorter code
  - easier maintenance and evolution
    - ⇨ easier to reason about, debug, change
  - more reusable
    - ⇨ library aspects
    - ⇨ plug and play aspects when appropriate
      - – even at runtime
- Additional benefits
  - reduced dependencies, dependency inversion
  - composition transparency
  - (un) pluggability
- Challenges
  - interactions, interferences

Impact of AOP

- AOP done right will not only clean up code, but will impact the entire software development process as well. In fact, MIT Technology Review lists AOP as one of the top 10 emerging technologies that will change the world and names it in one breath with brain-machine interfaces, flexible transistors, and microphotonics.

–(MIT Technology Review, January 2001)

universitätbonn ROOTS

Course Overview

– What will you learn? –

Themen-Übersicht (1)

Teil A. AOP: Abstraktionen und Sprachen

1. Einführung: Motivation, Beispiele, allgemeine Prinizipien, Vorlesungs-Übersicht
2. Sprachkonzepte von *AspectJ* und Demo des *AJDT*
3. *Java/AspectJ 5.0* (Insbesondere Annotationen, Generics)
4. Generische Aspekte: *LogicAJ* und *LogicAJ2*
5. Dynamische Aspekte: Z.B. *Ditrios* & *CSLogicAJ*, *AspectL*, *AspectS*, *AspectJVMs*, *JBOSS AOP*

Teil B. Anwendungen

1. Z.B. Persistenz als Aspekte, Design Patterns als Aspekte

Themen-Übersicht (2)

Teil C. AOM + AOSD:

1. Aspect Mining und Refactoring to Aspects
2. Aspect Oriented Software Development
3. Design Prinzipien, OO und AO Design Patterns

Teil D. Semantik, Implementierung und Modellierung

1. Semantik von Aspekten: Ausführung, Weaving, Compilierung
   →Aspekte als Bedingte Transformationen (CTs)
2. Analyse von Aspekt-Interferenzen
3. Aspect-Aware Refactorings

Rückblick: Prüfungsstoff, Ausblick: DA-Themen und Folgeveranstaltungen

universitätbonn ROOTS

Reference Articles

– Instead of a summary –

Weiterführende Literatur zur heutigen Vorlesung

- Robert E. Filman, Daniel P. Friedman:
  Aspect-Oriented Programming is Quantification and Obliviousness
  [AOSD05], Chapter 2. Vorversion auch online.

  Beantwortet die Frage nach dem Kern von AOP. Gilt als Referenzartikel. Diskutiert für einige Technologien inwieweit sie als AOP anzusehen sind.

- Adrian Colyer:
  The Ted Neward Challenge (AOP without the buzzwords)*
  http://www.aspectprogrammer.org/blogs/adrian/2004/05/the_ted_neward.html

  Arbeitet die “1 zu 1”-Vision aus. Im Netz als eine der besten Einführungen in die Grundidee von AOSD gehandelt.

*: Banned buzzwords: scattering, tangling, crosscutting, modularity, encapsulation, abstraction, dominant decomposition, concern

- Christina Videira Lopes:
  AOP: A Historical Perspective (What‘s in a Name?)
  [AOSD05], Chapter 5

  Historische und philosophische Reflexion aus erster Hand über die Entwicklung, die Grundidee und die Zukunft von AOP.

Was ist ein Aspekt?

- Christina Videira Lopes:
  So, what makes an Aspect an Aspect, before we even think of programming it with AspectJ? Given the name, we choose for it, which clearly influences our perception, Aspects are software concerns that affect what happens in the Objects but that are more concise, intelligible, and manageable when written as separate chapters of the imaginary book that describes the application.

- Adrian Colyer:
  So what we've really got in any non-trivial software application is not the ideal 1-to-1 mapping between concept and implementation, but an n:m mapping. … AOP addresses the problem by introducing a new construct known as an aspect that is able to capture in one place the implementation of design requirements, such as the view-notification requirement in MVC, which OOP cannot.

- Robert E. Filman, Daniel P. Friedman:
  AOP can be understood as the desire to make quantified statements about the behavior of programs and to have these quantifications hold over programs that have no explicit reference to the possibility of additional behavior.

universitätbonn ROOTS

Soviel für heute.

Nächste Woche lernen Sie AspectJ!