veritas
iustitia
libertas
Freie Universität Berlin

Seminar „Selected Topics in in Software Engineering“

Reuse

Christopher Oezbek
Freie Universität Berlin, Department for CS
http://www.inf.fu-berlin.de/inst/ag-se/

- Introduction
- Terminology
- Ideas for Research
- Brainstorming

Introduction

- What is Reuse?
  - *"...the use of existing software artifacts or knowledge to create new software..."* [FraTer96]
  - This includes all types of artifacts created.
  - Internal Reuse Vs. External Reuse
- Why Reuse?
  - Because we (the CS people) are inventing the wheel over and over again and wasting enormous resources doing so.
  - We hope that there is a way to make integration and design of the reusable component cheaper than redevelopment.
  - There is no other way to build large applications.

Historical Developments

- From the very beginning of computing in the fifties subroutine-libraries have been used.
- 1968 McIlroy Paper at NATO Conference on SE
- 1970s Development of substantial libraries for graphics and numerical calculations, Ada (1979)
- 1980s Software Crisis => Can reuse solve these problems?
- 1990s Libraries are so large by now that the size of them is hindering their use.

Terminology

- Library
  - Set of individual functions or classes that can be reused mostly independently (functional reuse).
  - "a discrete, stand-alone, context independent part of a solution"
- Framework
  - A unit of design reuse; coupling several library classes.
  - "an abstract design for a particular kind of application"
- Component
  - Independent unit of reuse.
  - Technical definition by given set of import and export mechanism.
  - Interface is usually restricted to an in/out mechanism.
  - Automated parts (deployment J2EE, interface query COM, dependency resolution OSGi)
- API: Usually a framework plus library parts (for instance JDK).

Freie Universität Berlin

Terminology (II)

- Source
  - The element of design (at any level) that is chosen to be reused.
  - Sums up all the terms like component, library...
- Target
  - The problem that needs to be solved

Reusable Aspects of Software Development

1. Architectures
2. Source Code
3. Data
4. Designs
5. Documentation
6. Templates
7. Human Interfaces
8. Plans
9. Requirements
10. Test Cases

=> Functionality?

How reuse is supposed to work

Avg. development time:	Normal	Reuse
Convex Hull	12,4	2,2
Readers / Writers	4,7	1,8
Producer / Consumer	3,9	1,9
Shortest Path	33,3	1,4
Parallel Prefix	20,0	1,4
Divide Region	20,0	3,5
Sort / Merge	8,5	1,5

So where is the problem?

- Wrong type of problems!
  - All of them can be ranked highly on the algorithmic complexity and low on the coupling dimension.
  - => We are going to investigate this with our first experiment.
- The subjects were given an extensive library of domain relevant functions (for instance a function to determine the hull of a set of points from one side).
  - The subjects did not have to search for the source.

Where does reuse work?

- Program families
- Between successive versions
- If the same developer who developed the code continues to do so in a different project.

- As soon as you move code too far away from people who have knowledge about it, reusability decreases dramatically
- These aspects point toward the cognitive dimension of the problem.

Reuse Research

- What areas of research are there in the domain of reuse?
  - A large portion of reuse research deals with quantifying reuse (Metrics). For instance

$$C = (b + (E/n) - 1)R + 1 .$$

  - C = cost, E = relative developing cost for a reusable component, b=relative integration cost for the component, n= number of reuses, R=proportion of reused code in the product
  - This is not so interesting for me, since we lack the industry relations to have access to projects that could be used for this kind of research.
  - => Move in the direction of individual programmers and their usage of APIs

Freie Universität Berlin

Failures to reuse [FraFox96]

- The following failures modes for component reuse have been identified:
  - No Attempt to Reuse
  - Part Does Not Exist
  - Part Is Not Available
  - Part Is Not Found
  - Part Is Not Understood
  - Part Is Not Valid
  - Part Can Not Be Integrated

DfR > Automatic Refactoring

- Problem: How to balance complexity of libraries and reusability?
- *"Finding new abstractions is difficult. In general, it seems that an abstraction is usually discovered by generalizing from a number of concrete examples."* [JohFoo88]
- Idea: Automatic Refactoring
- Extract the code fragment from a number of large projects that lead to highest reduction in code-size.
- The idea is similar to Huffman codes or dictionary based compression.

- Found on a Monday night: Clone Doctor http://www.semdesigns.com/Products/Clone/index.html

DfR > Code Harvesting

- *"In my experience, the best way to do CodeHarvesting is to tag any copy-pasted block with a predefined FixmeComment, which I periodically grep for."* [c2/GlyphLefkowitz]
- This would require Micro Process Encoding as Sebastian is going to investigate.
- The problem is that it would be difficult to detect architectural changes that go beyond 10 lines of code.

DfR > Performance and Trust

- If you use a component it is highly likely that you rely on somebody's messy code:
  - You need a lot of trust.
  - You need to get a feeling for the quality of the solution.
    - This means the extent of documentation, available examples, how broad the user base is, support,...
- Performance on the other hand is something that becomes especially interesting with components.
  - You will make calls into a unknown black-box component.
  - When will it return?
  - Does it comply with your performance requirements?
- Research in that direction would try to counter the "Not invented here"-syndrome

DfR > Reuse as a documentation problem

- *"In a component-based-programming paradigm, the information overload of the API reference documentation can have a serious effect on the programming task and therefore ultimately on the resulting software."* [BerglundE99]
- An idea would be to integrate examples into the API documentation by searching existing projects.
- Another to add a prioritized list of classes, functions and identifier.

JavaDoc and Patterns?

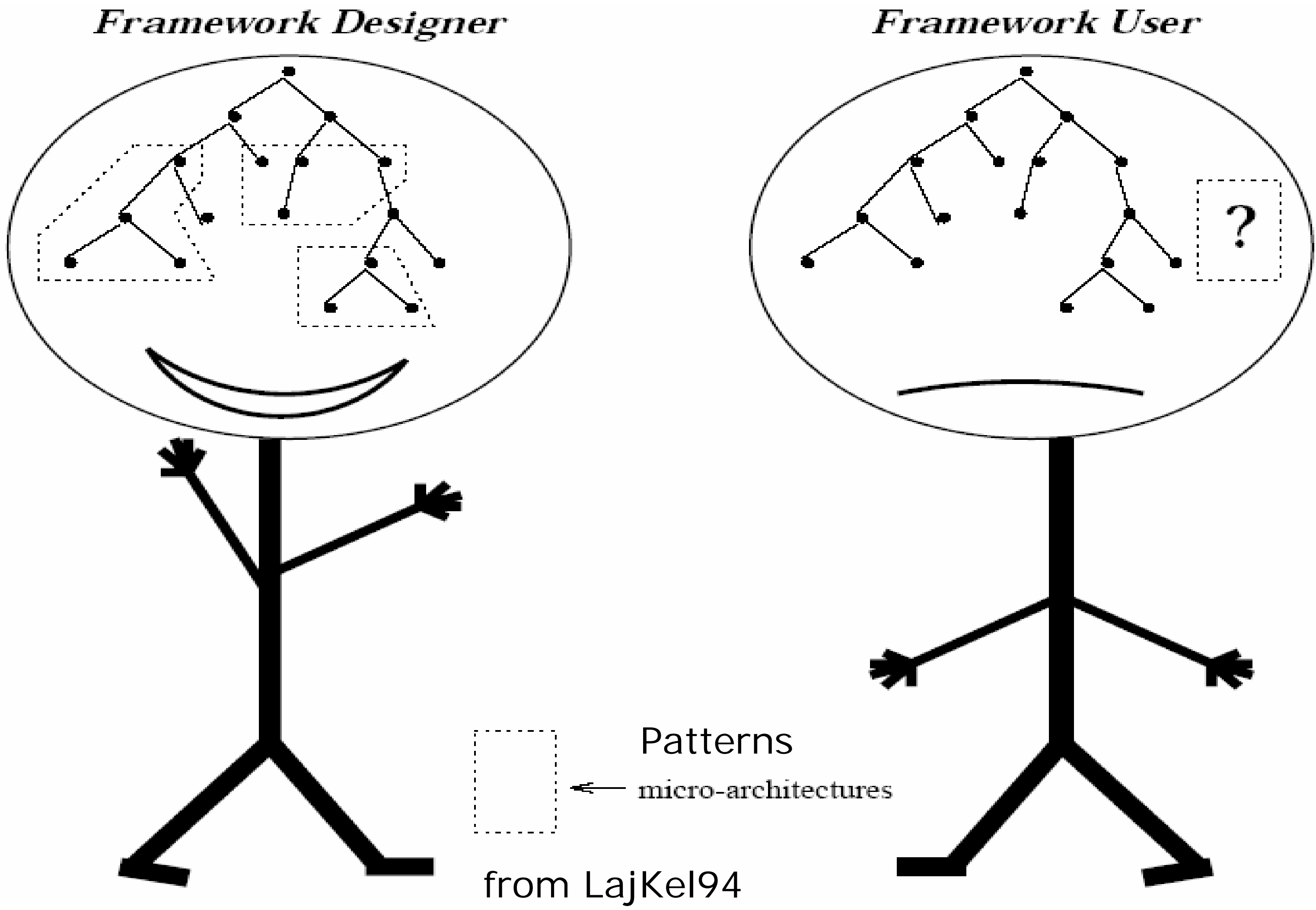


Figure 2: Conceptual views of frameworks: designer view (left) versus user view (right).

DfR > Reuse as a cognitive problem

- Discovering an API seems rather to be an cognitive problem than a technical issue.
- The sheer size of the name space of current APIs (Java 1.5: 3279 classes in 166 packages) makes it impossible for a single human to have a detailed insight.
- Ways to solve the problem:
  - Turn SE into linguists
  - Constructivist learning theory => Learn the API by rewriting it. This is what is happening in the real-world.
  - Reduce the complexity of the APIs.
  - Use cognitive dimensions (Steven Clarke) like for instance Role Expressiveness to tailor APIs to user's preexisting notions of use

DfR > What do people change in a language

- Another idea based on cognitive load:
- Unlike Java, the programming language Ruby allows for modification of the base library
  - For instance one can alter the way the String.concat function behaves (with all its consequences).
- By looking into existing projects it should be easy to generate a list of modifications applied to the language.
- Sharing these extension can be useful for future developers.
- To prevent from creating an ever increasing library it would be mandatory to create a hierarchical scheme:
  - Core functions < Helpers < Convenience Functions

DfR > Understanding the Theoretical Framework

- I could also try to understand the existing cognitive aspects people have identified when reusing software.
- For instance:
  - To what extent do encapsulation mechanism like modules and classes promote reuse? How do they accomplish this? Is it just because programmers can chunk a large number of lines?
- Problem:
  - I feel that I am getting more and more impatient and that my probing inside the problem space is not getting me towards the frontier of research.
  - Just the seminar and theoretical understanding is not enough hands on stuff.

Another Reuse Dimension: Pipes and CLI

- This is often cited as being Reuse, too.
- The amount of work that one has to do to glue these tools together can be high.
- Difference of `Ping.exe` (Windows XP) and `ping` (Linux)
  - `Reply from 192.168.239.132: bytes=32 time=101ms TTL=124`
  - `64 bytes from 160.45.111.116: icmp_seq=2 ttl=127 time=0.5 ms`
- => Paradise for every Regular-Expression-Fan (but did s/he consider IPv6?)
- Even worse: Linux-Ping changed its output-format between versions
- => Reuse requires stable interfaces (COM)

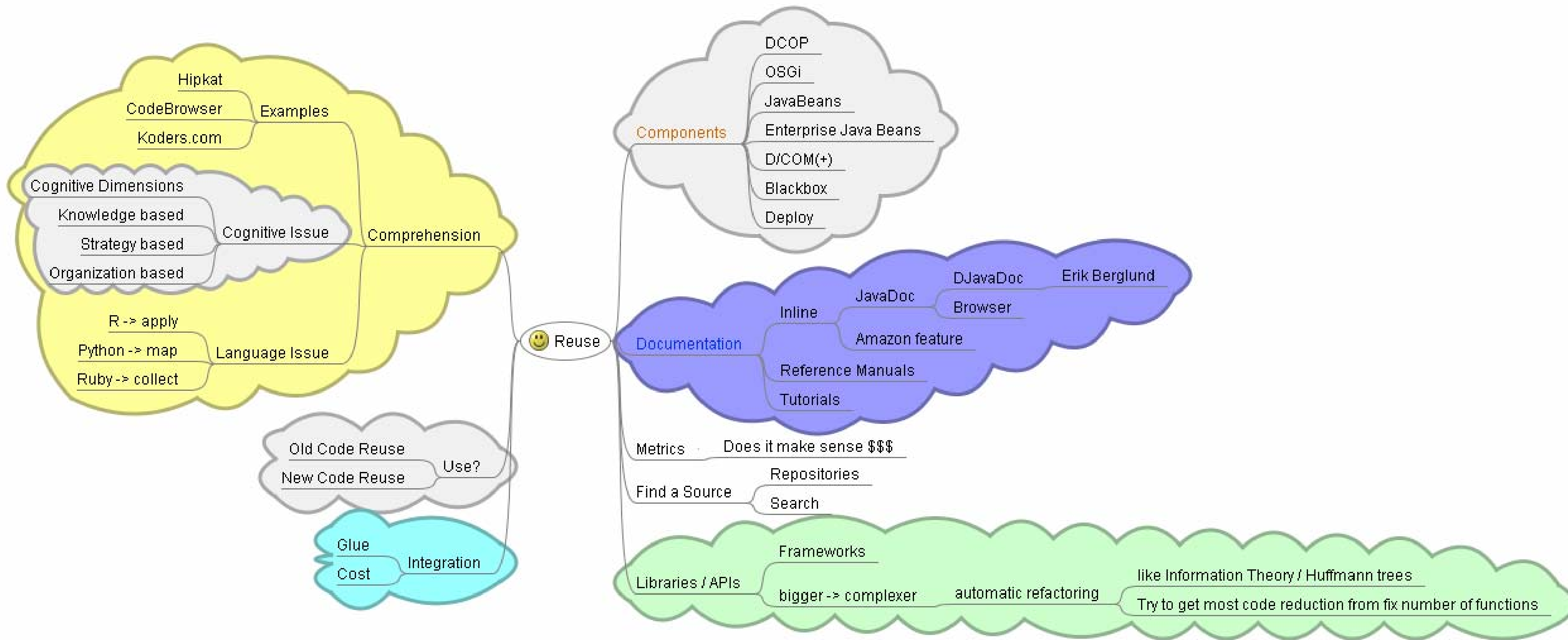
The Tool side of Things

- Why tools?
  - Because of course they would be something to work on.
  - They chew up all the time a PhD can take.
  - One can publish a number of papers with them.
  - Because that's just what CS-people do.
- A lot of interesting tool ideas lurking in my mind as you have seen on the previous slides. In addition:
  - Java Design Recovery and Modification Tool (called Java Toolbox on my homepage)
- How is the feeling about this?
- If I should rather go into the direction of more empirical work then I need some help in figuring out what to do.

Software Product Lines

- Predictive vs. Opportunistic Reuse
  - *"Rather than put general software components into a library in hopes that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well defined product line."* [http://www.softwareproductlines.com]
- The idea is then to identify in advance these points of reuse that will span across the family or line of product.
- Special languages, generators or customization tools can then be used to distinguish the individual products from the reusable core.

The whole thing as a mind map…



Thank you!