FZI Forschungszentrum Informatik
an der Universität Karlsruhe

Tool Supported Software Quality Assessments

Markus Bauer

Olaf Seng

VTT Electronics, Oulu, 25 June 2003

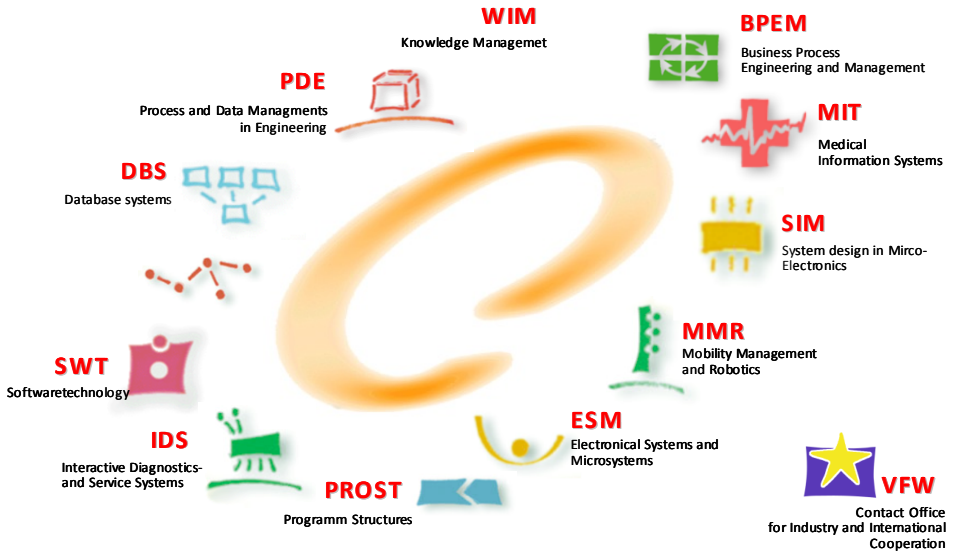
Programmstrukturen

Overview

- About us
- Introduction
  - Software quality
  - Techniques for tool supported quality assessment
- Case studies
  - SolidTech's database server
  - VTT's eXpert web application
  - Telecommunication System
- Discussion

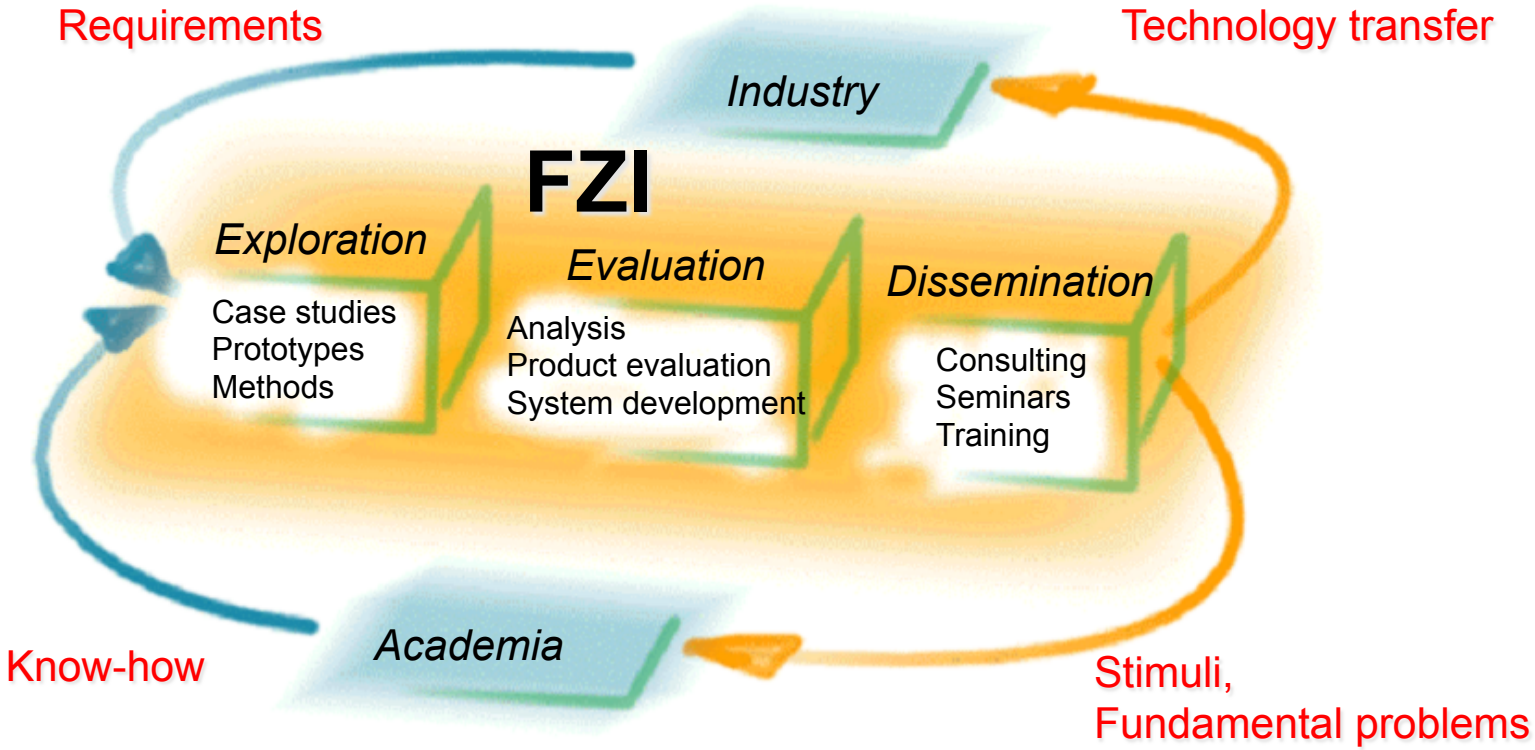
About Us

- 13 research departments, covering a variety of different computer science fields
- Turnover: 8 Mio. EUR (2000)



- Over 200 projects per year
- About 120 employees, about 100 researchers
- Publicly funded foundation, founded in 1985

Bridging the gap...



Program Structures Department

- **Our vision:**
  Software construction is an engineering discipline!

- **Services we provide:**
  - Design and development of innovative software solutions
  - Design support, consulting and training
  - Software assessments, quality management
  - Support for the evolution of software systems
  - Software modelling
  - Compiler construction
  - Evaluation of development tools and processes

PROST – People

Prof. Dr. G. Goos
Director

T. Genssler
Department Head
Software Evolution
Compiler Construction

M. Bauer
Vice Department Head
Component Mining
Software Evolution

H. Bär
Program Analysis
Web Applications

C. Andriessens
Program Analysis
Software Quality

M. Winter
Components
Generators

A. Trifu
Software Quality
Components

O. Seng
Software Quality
Software Evolution

V. Kuttruff
Software Transformation
Generators

Topics and Projects

Web-based Computing

App2Web, Vivian, JiniLAB

Software Evolution

FAMOOS, TROOP,
TOCODA, App2Web

**SPI &
Software Quality**

proSoft, IMPROVE,
QBench

Compiler Construction

DD2DTM, aXMLerate, **Inject/J**
Lazy XML evalutation

**Software Engineering
Methodology**

PECOS, GeCoRi, **CompoBench**
MODALE

Our Partners and Customers

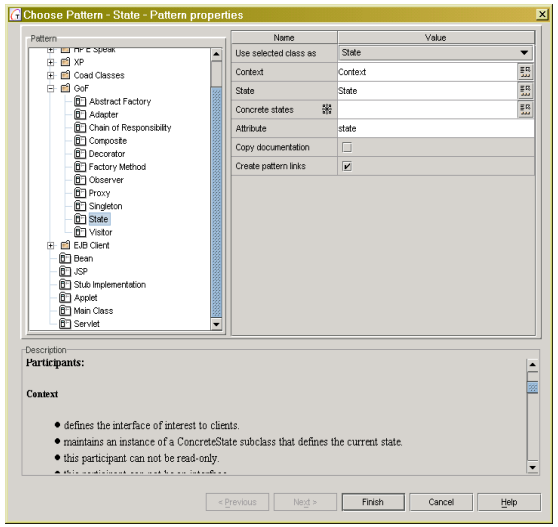
Object Technology Intl.

Our Know-How in Commercial Products

- TogetherSoft Together/J
  - Design pattern technology, metrics support
- Telelogic AuditSuite (aka. SEMA Audit)
  - Fact extraction, metrics for OO-systems
- CAS genesisWorld (Groupware System)
  - LDAP server, Windows CE synchronization, SyncML module
- Bundesanstalt für Wasserwirtschaft
  - Rheingold (Analysis and visualization of hydraulic data)

Project Reference TROOP

- Tool supported software restructuring using design patterns
- Integration the technique into the CASE tool Together
- Training on OO and design patterns
- Partners:
  - CAS (User),
  - Object International (now: TogetherSoft)



"In nine years of reviewing (European) Commission projects, this is the closest I have seen to a textbook example of how the process should work. In less than two years, an innovation has moved from being a doctoral dissertation to being embodied in a commercial CASE tool, having been rigorously evaluated in the development of a commercial software along the way."

Trykve Renskaug, 1999, TROOP Final Review

Our Focus Today: Software Quality

- Our goal:
  To construct high quality software!
- This sounds good, but:
  - 1968: NATO proclaims the *software crisis*: software systems have bad product quality and cause unreasonably high maintenance costs!
  - 1994: IBM: Survey on large software projects with 24 IT companies – 88% of the software systems require a major redesign!
- Software quality is (still) an issue!

Software Quality

- *“Quality is, if the customer returns, and not the product”.* (Hans-Helge Stechl, board member, BASF)

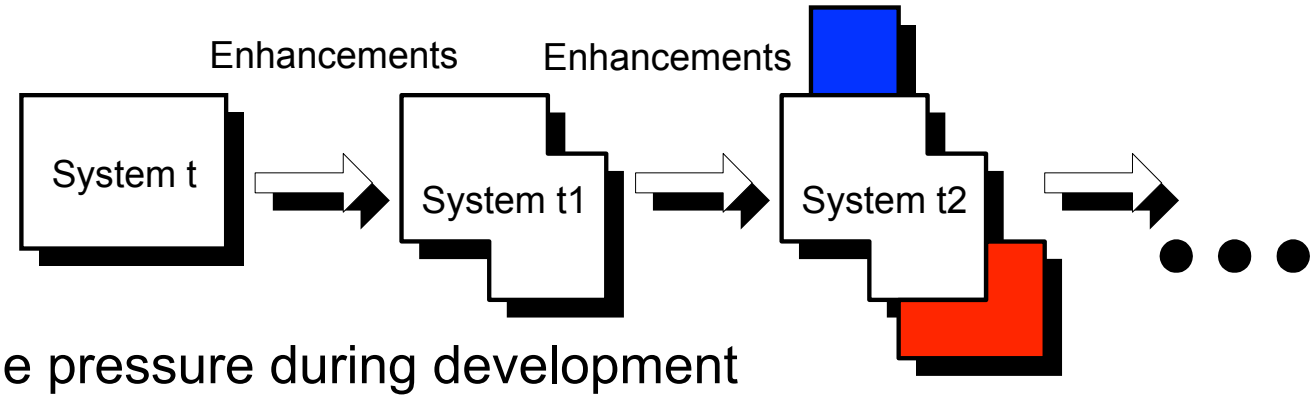
- External quality = customer‘s perspective:
  - Ease of use
  - Performance
  - Reliability
- Internal quality = developer‘s perspective:
  - Good design
  - Understandability, maintainability

- Good internal quality is a requirement for good external quality!

Why do we have quality problems?

- Requirements for software systems are difficult to analyse, requirements change („law of constant change“)
  - Gap between domain experts and software experts
  - Long software life cycles
  - Need to adapt towards new usage scenarios

⇒ Software structures erode...



- Time pressure during development
- Lack of developer know-how

Good Software Design?

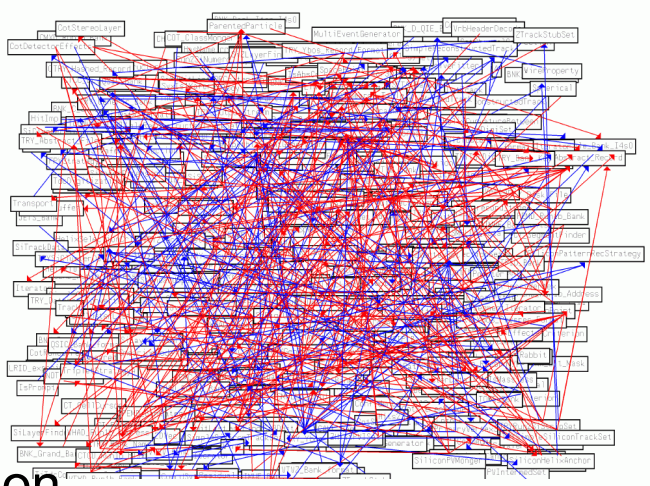
- A primary goal of good design is to contribute to low development and maintenance costs.
- Good design is a compromise. It should be simple and it should be flexible.
- Design is bad if it is unnecessarily complicated or inflexible.

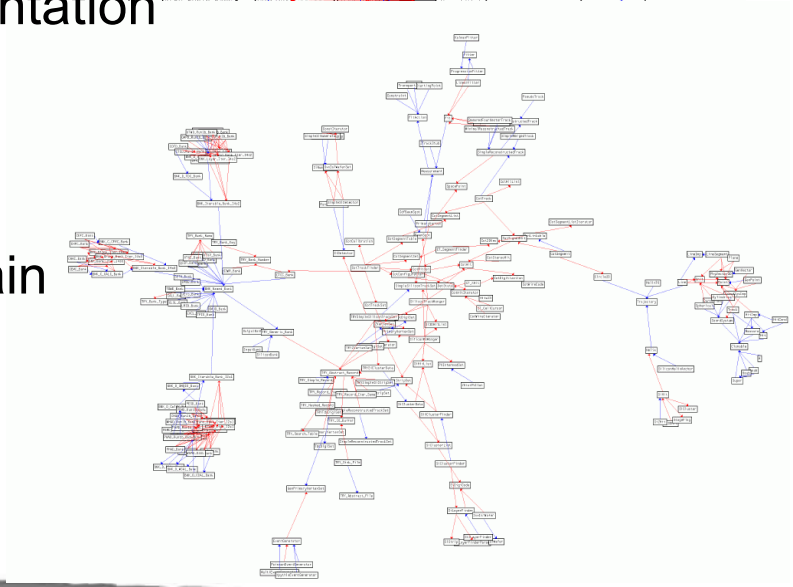
- Good design (internal quality) has often positive effects on other quality factors (external quality: e.g. performance, stability,...)!

What does that mean?

Principles of „good design“:

- Modularity:
  Reduce the complexity of the system; decompose it into manageable units (→subsystems)
- Encapsulation:
  Separate interfaces from implementation (→interfaces)
- Abstraction:
  Create simplified views on the concepts of your application domain (→data types, classes)

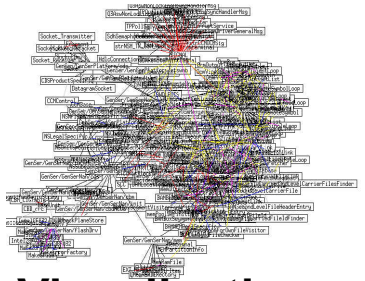




Measure design quality?

- Why do we want do do this?
  - Tom De Marco: *"You cannot control what you cannot measure."*
  - Clerk Maxwell: *"To measure is to know."*
  - Lord Kelvin: *"The degree to which you can express something in numbers is the degree to which you really understand it."*
- How?
  - Use software metrics and check design rules or guidelines!
  - Identify architecture violations, check dependencies, measure coupling, cohesion and complexity properties

Tool support for quality assessments



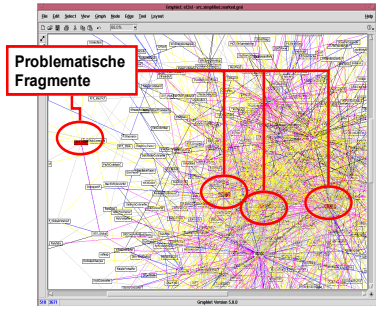
Visualisations

Abstraction

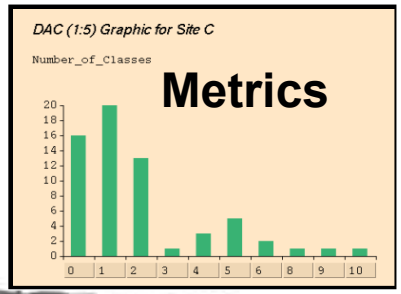
Source code

Design-DB

Fact extraction

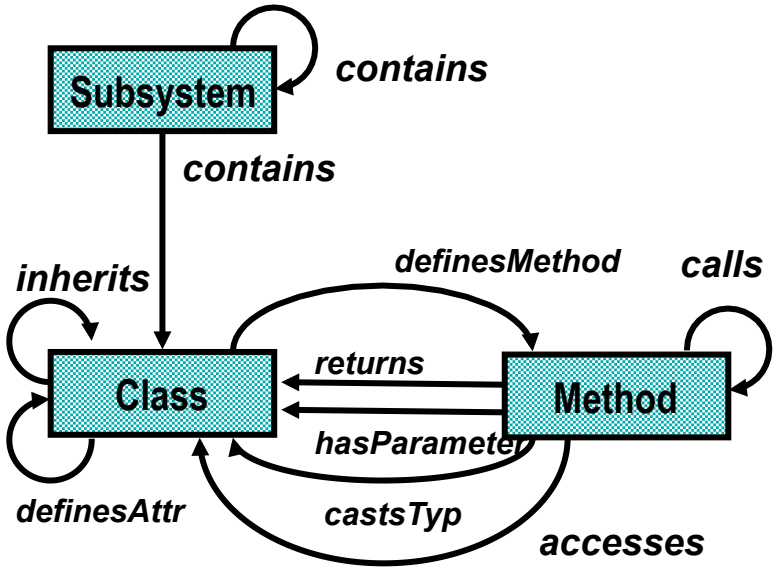


Problem detection using heuristics



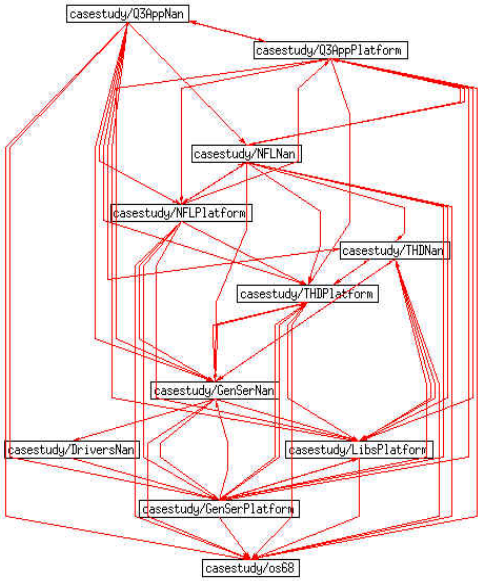
Fact Extraction

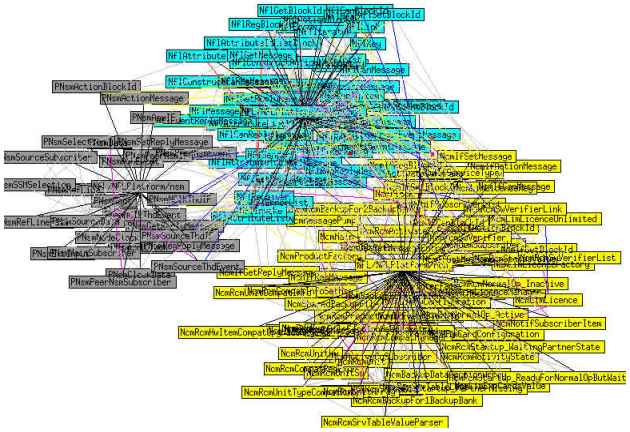
- Objective
  - An abstract, semi-formal model of the system → „design database“ as a foundation for further analyses
- Techniques:
  - Compiler techniques
  - Graph theory to abstract the design database
- Problems:
  - Programming language issues (C/C++: macros!)
  - Incomplete or defective source code



Visualisations

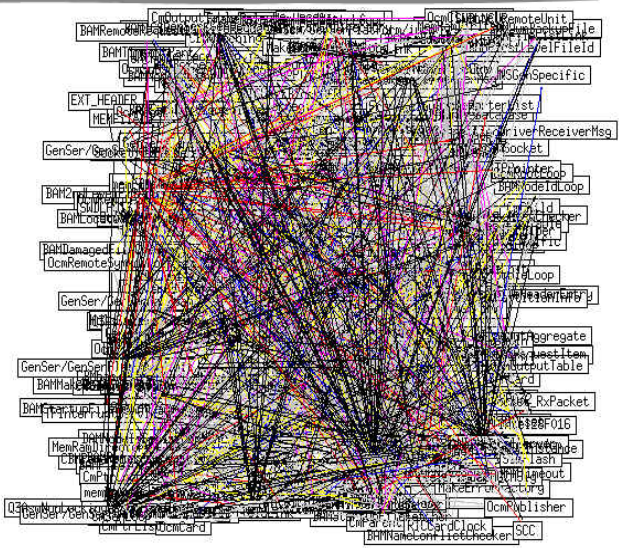
- Objective:
  Use the fact that a picture may say more than a 1000 words! Visualisations help to grasp complex structures.
- Usage Scenarios:
  - Understand architectures and structures
  - Check dependencies (layered architecture, framework vs. application code)



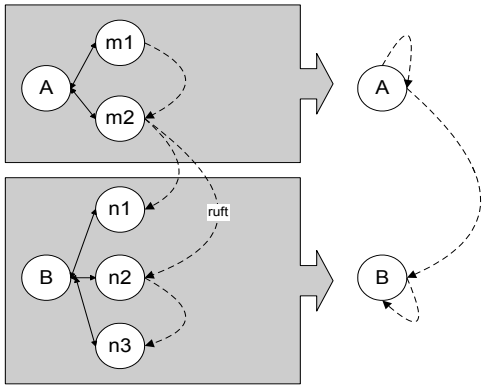


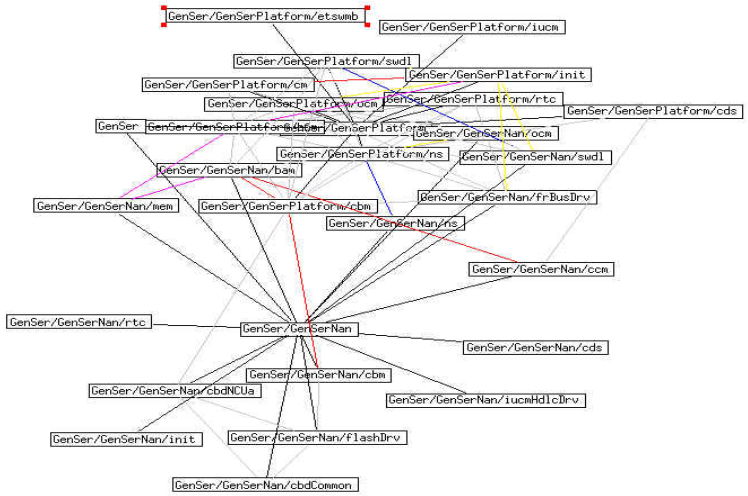
Abstraction

- Objective:
  Improve the design database, generate facts that provide additional information to developers
- Techniques:
  - Filtering
  - Grouping / Aggregation of low level elements to high level elements



Example: Group methods to classes or operations with abstract data types

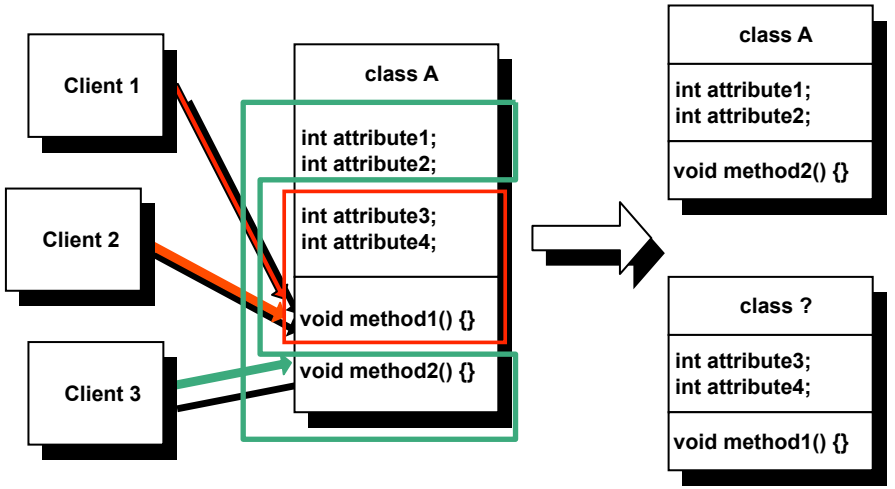




Software Metrics

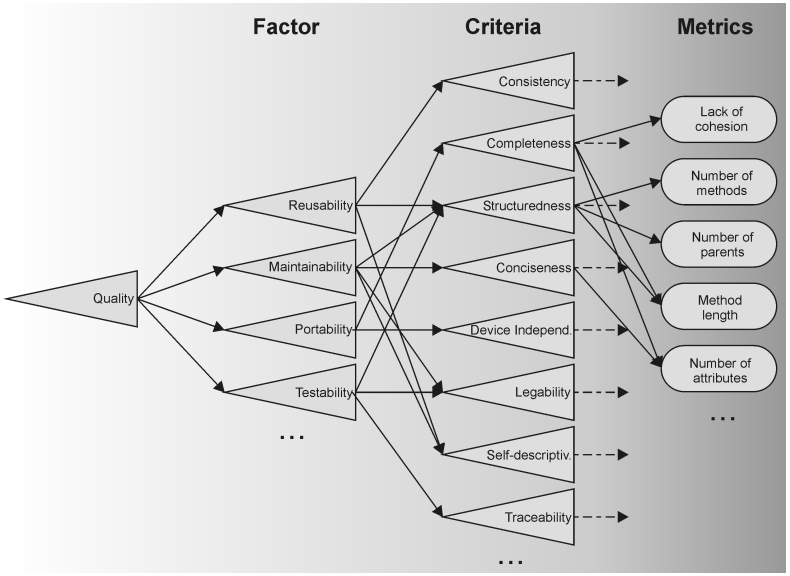
- Objective:
  - Discover weak spots
- Examples:
  - Complexity Metrics: May point to hard to maintain parts
  - Coupling Metrics: Identify fragile classes
  - Cohesion Metrics: Identify classes that do not represent suitable abstractions (one concept per class)

1. TCC: Class A has low cohesion
2. Analysis of client code: Class A is used with two different usage patterns => it implements two separate concepts
3. Class A can be split

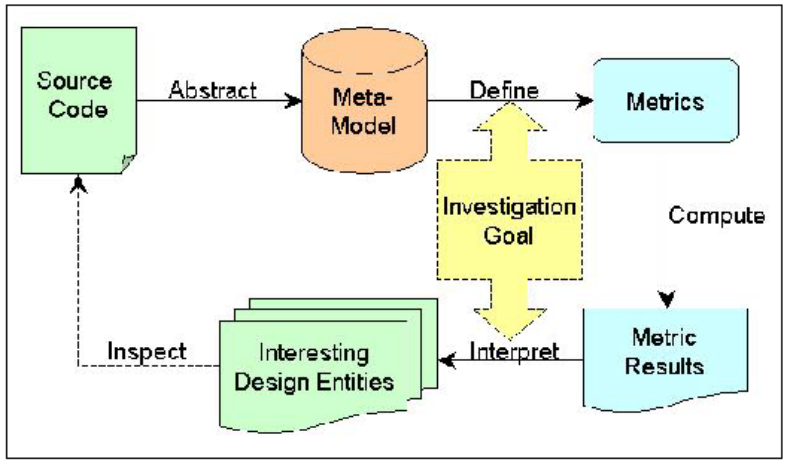


Quality Management Using Metrics

- Quality Models
  - Objective: Mapping between quality factors and metrics
  - Example: Factor-Criteria-Metrics

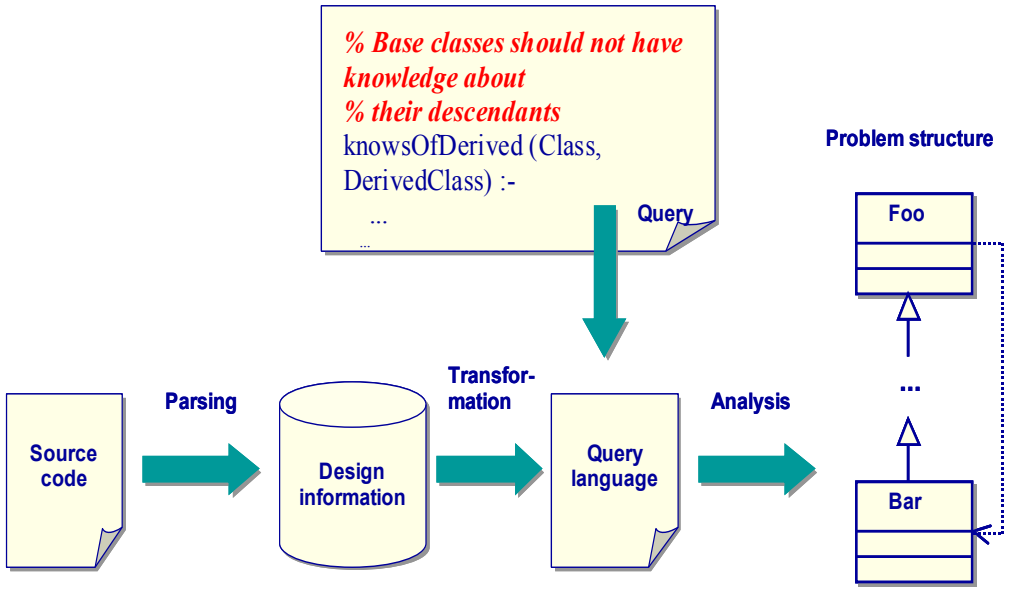


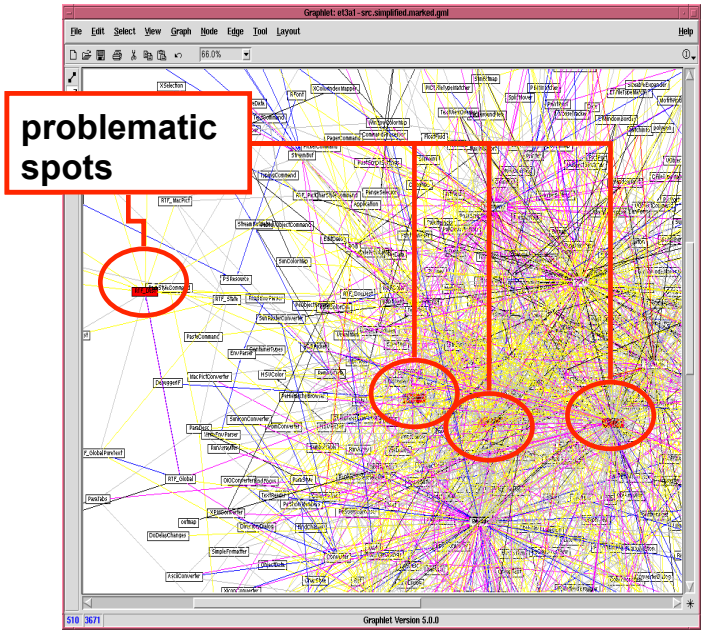
- Process Models
  - Objective: Goal-driven measurement und usage of data
  - Example: Goal-Question-Metric



Running Design Queries

- Objectives:
  - Find artefacts with certain structural properties
  - Problem detection using common design heuristics
  - Architecture checks, enforcing design guidelines
- Example: „Super classes should not know subclasses!“





Examples for Design Heuristics

Classes:

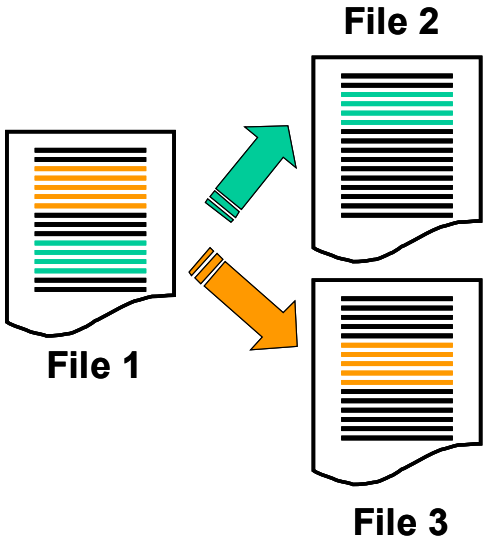
- Classes should not depend on subclasses
- Use inheritance only for polymorphism
- Avoid unused inheritance (Keep your inheritance hierarchy simple)
- No bottleneck classes
- No god classes

Subsystems:

- Lean, well defined subsystem interfaces
- No fragile classes in interfaces
- Classes in interfaces should not depend on (too many) classes in other subsystems
- Decoupling of subsystems
- No cyclic inheritance between subsystems

Identifying Duplicated Code

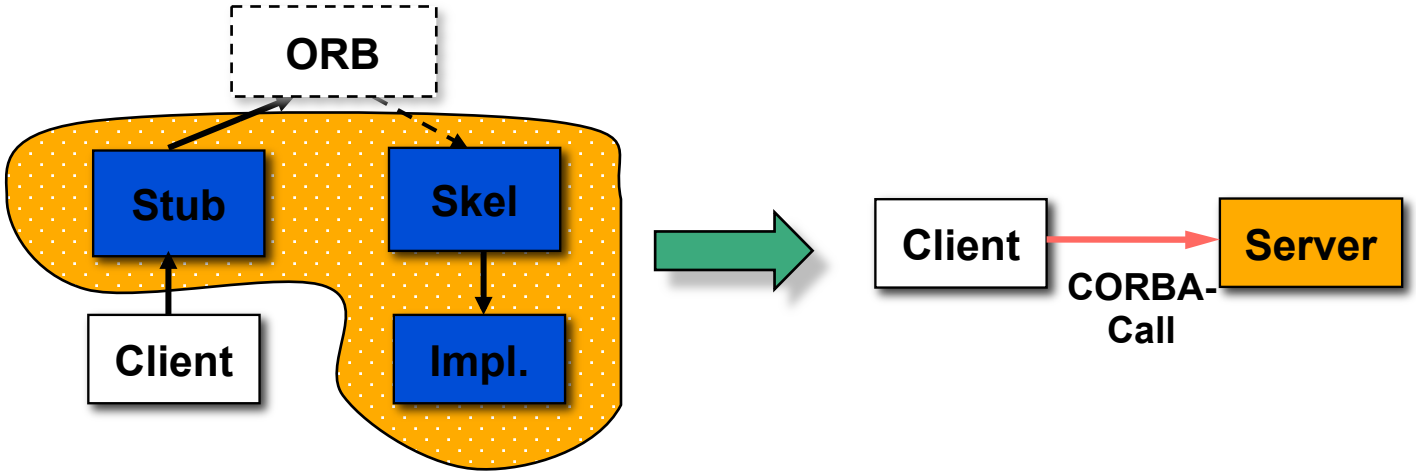
- Objective:
  - Detect code fragments which have been copied/cloned from other locations
  - Duplicated code is hard to maintain
    - Code size increases, much code to read when maintaining the system
    - Bug fixes will usually fix only one version
    - Often: Factoring out duplicated code into a reusable method improves understandability



- Techniques:
  - Simple line based pattern matching + „clustering“
  - Language independent
  - Possible extension: Identify „fuzzy“ clones

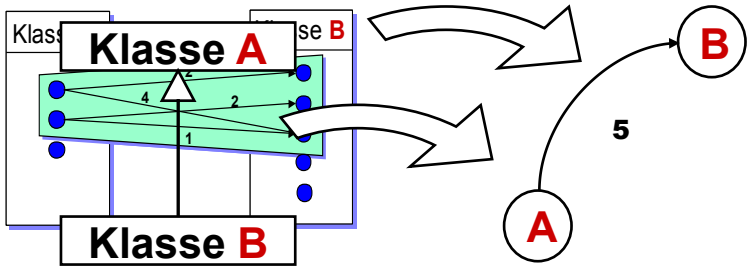
Working with Components

- Problem: How to deal with component infrastructures?
  - System dependencies are hidden by the run-time environment
  - Source code is „polluted“ by infrastructure related code
- Solution: Apply abstraction techniques
  (Filter and aggregation operations on the design database)



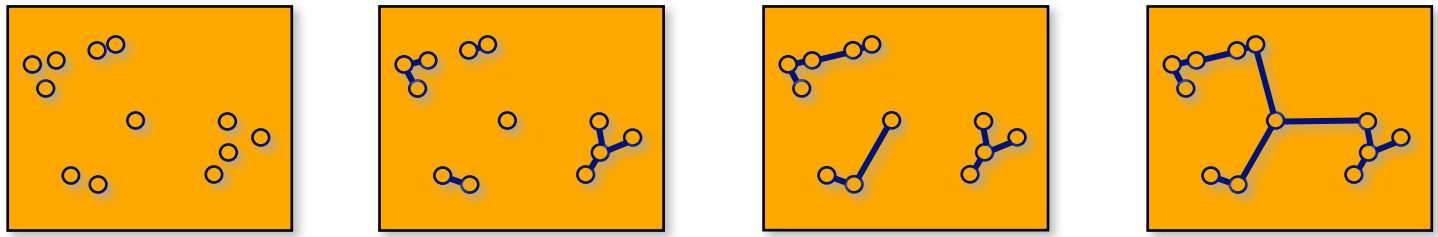
Analyse Component Structures (I)

- Idea:
  - Compute an ideal decomposition which minimizes coupling and maximizes internal cohesion of components/subsystems
  - Compare this ideal decomposition with the component/subsystem structure declared by the designers
- Technique: Represent the system‘s structure as a graph
  - Classes and modules = nodes
  - Dependencies = weighted edges
    - Inheritance
    - Calls
    - Variable accesses and other type dependencies



Analyse Component Structures (II)

- Compute clusters
  - Foundation: algorithms from data mining
  - Greedily group nodes according to their coupling



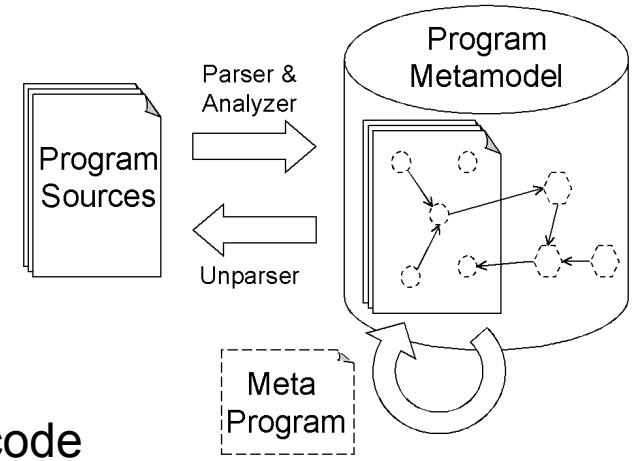
- Compare clusters with decomposition given by the designers
- In the future: combine this with checks for architectural styles/patterns/rules

Tool Supported Refactoring

- Objectives:
  - Improve previously identified weaknesses
  - Systematic and automated refactoring of a system‘s structure
  - Reduction of error-prone hand-made code changes

Supported transformations:

1. Basic refactorings:
   Create, move, rename
   classes, methods,...; insert new code fragments
2. Complex transformations:
   e.g. introduction of design patterns
3. Scripting environment for user specific code changes for Java available as plugin for Eclipse



Tool support

- FZI‘s tool prototypes :
  - GOOSE: Fact extraction, design heuristics, metrics
  - Echidna: Software visualization
  - PRODEUS: Metrics
  - JAMES: Analysis of components
  - Recoder, Inject/J: Software transformation
- FZI‘s Know-How is available in commercial products:
  - Object International Together Enterprise Edition – Software transformation, metrics
  - Telelogic Audit Suite, SEMA Audit –
    Fact extraction, metrics for oo-systems

Coffee Break

After the coffee break:

- Examples from three case studies in order to illustrate the concepts

Case Studies

- Numerous software assessments (1998 - today):
  ABB, DaimlerChrysler, IBM, Nokia, Debis, SOLID Technologies, Telekom Deutschland, VTT
- Systems from research labs
  ET++, 65 kLOC, C++, 770 classes
  eXpert web application, 5 kLOC, 16 classes, Java, JSP (❷)
- Engineering software
  150 kLOC, C++/DCOM
- Telecommunication software
  500 kLOC, C++, C, Assembler
  2 MLOC, C,C++
  1 MLOC C++, 120 kLOC Java, CORBA (❸)
  20 MLOC, 120 kLOC, Chill
- Contract management for an insurance company
  1MLOC, Java/EJB, 6000 classes
- Database engine for embedded systems
  1MLOC, C, 28 subsystems, ~300 complex data types, 14.000 functions (❶)

Case Study 1: SolidTech‘s DB server

- You will see typical steps and techniques used in a tool supported software assessment

- Techniques
  - Architecture and dependency analyses
  - Assessment of code complexity, coupling and encapsulation
  - Analysis of data objects and functions

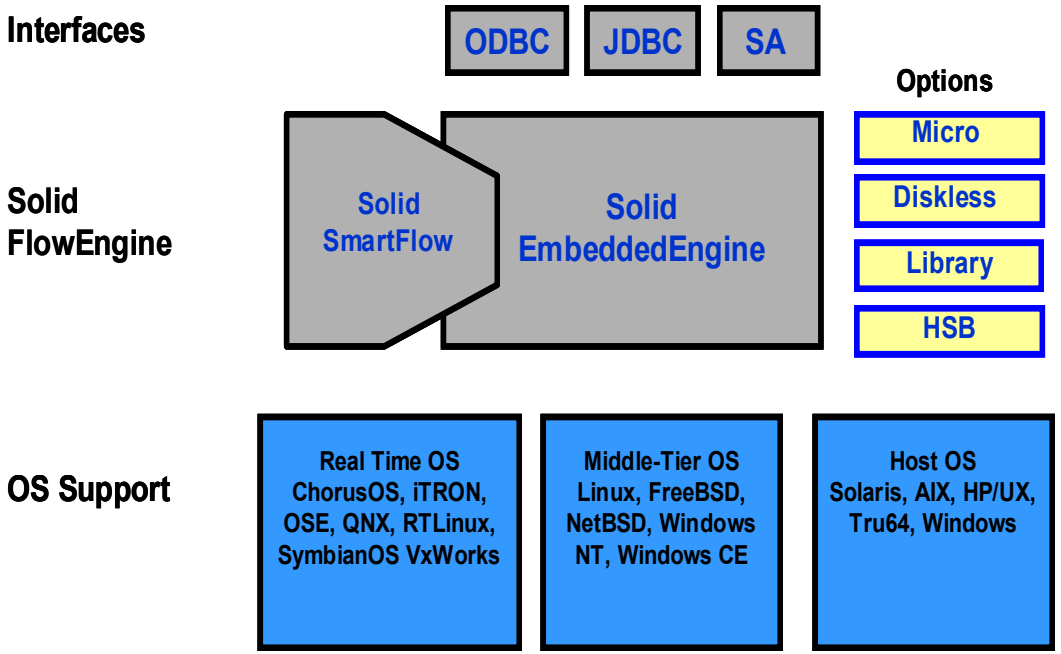
Scope of the Assessment at SolidTech

Objectives:

- Put the theory into practice for Solid's benefit!
- Assess the quality of one of Solid's database products

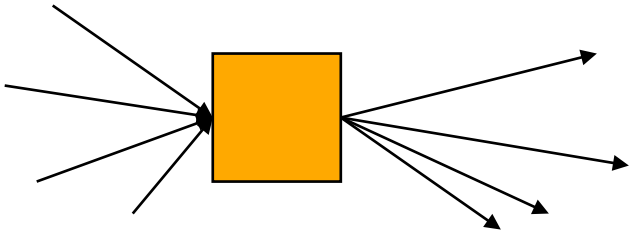
Case study:

- 28 subsystems, 1347 files, ~1.000.000 LOC C
- 14000 functions
- Core part: 274 data types

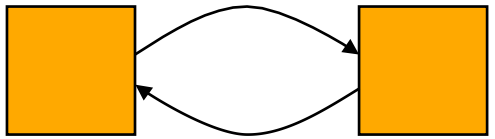


Assessment: Typical Steps (I)

- Check for architecture violations
  - Define and check rules for dependencies between subsystems
    Example: `shouldNotDependOn('sputsrv/dbe',X) :- not(isUtilityLayer(X)).`
- Check for dependency bottlenecks and cycles
  - Bottlenecks and cycles hinder understandability and maintainability
  - Heuristic applies to subsystems and to data types



Bottleneck



Cycle

Assessment: Typical Steps (II)

- Measurement of coupling, encapsulation and complexity of subsystems and complex data types
  - Coupling:
    High coupling between components
    ➔ System may be hard to understand and to maintain
  - Complexity:
    High internal complexity (complex control flow)
    ➔ Component is difficult to understand, error-prone
  - Complex components should be well encapsulated
    ➔ low coupling, lean interfaces

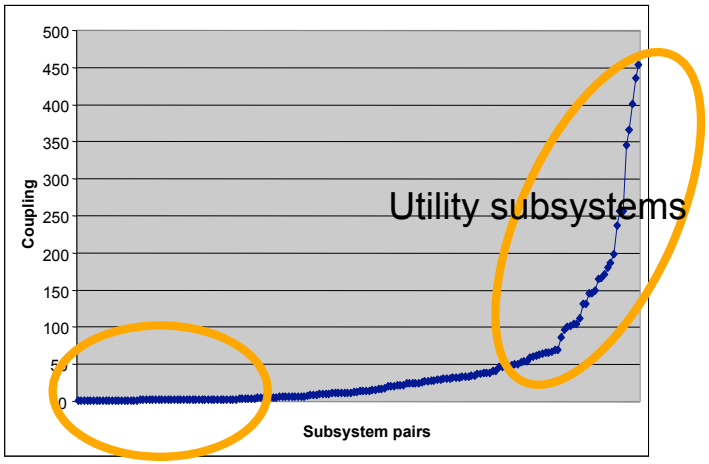
- Measurement of complexity and call dependencies of functions

Results from SolidTech's Casestudy

- Architectural violations:
  - Architectural style: layered architecture
  - Only a very few violations of the dependency rules (= forbidden dependencies between subsystems)
- Bottlenecks:
  - Some, most of them uncritical (and possibly unavoidable)
- Cycles:
  - Very few cycles, most of them uncritical

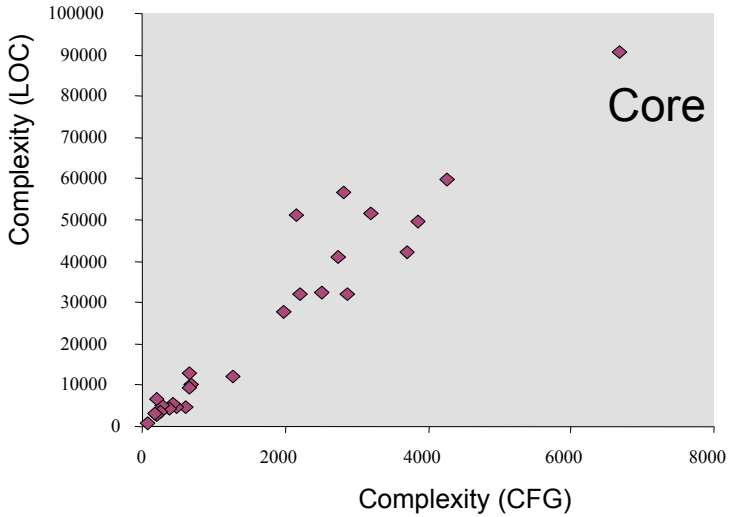
Subsystems: Coupling and Complexity Results

Coupling



Coupling values involving Core

Complexity



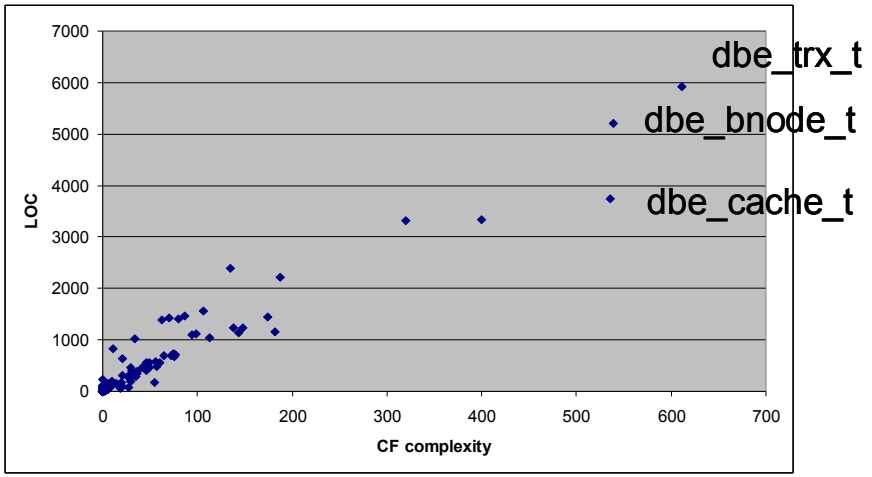
- Good: Mostly low coupling between subsystems, reasonable complexity for most subsystems
- Some parts in Solid‘s code (Core) are complex but well encapsulated

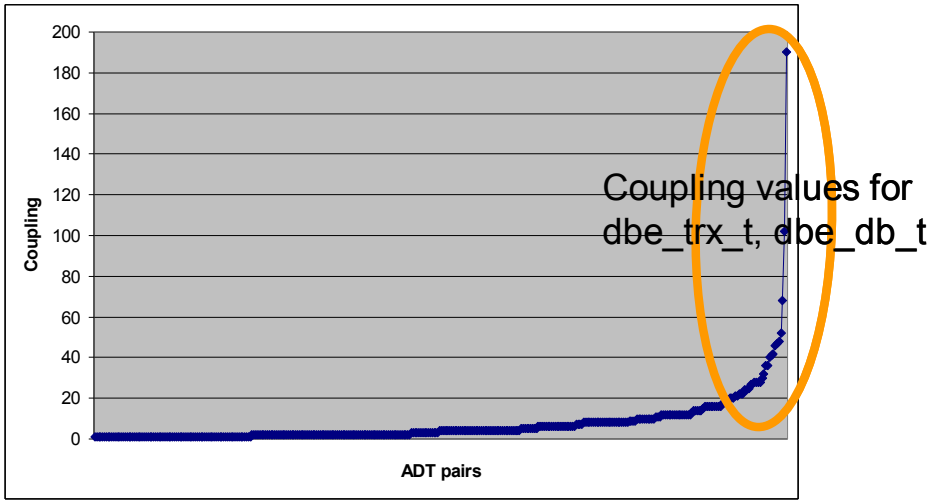
Complex Data Types

- Solid: Procedural programming in C, OO-style thinking: Data types (C structs)
  - model important concepts of the application domain
  - logically group data and corresponding operations/functions
- Use abstraction techniques to apply design queries to data types:
  - Infer complexity and coupling properties from individual operations/functions on data types
  - Heuristic: An operation is associated with a data type by naming conventions and first parameter type
  - Apply design queries on data type objects to compute complexity, coupling and check for guidelines (bottlenecks, cycles)

Data Types: Complexity and Coupling Results

- Complexity analysis:
  Good: Only a few data types have high complexity values
- Coupling:
  Good: Only a small number of data types are involved with high coupling
- Problems:
  - Some data types (dbe_trx_t) are complex and tightly coupled
  - They represent central concepts; probably this cannot be avoided





Analysis of functions

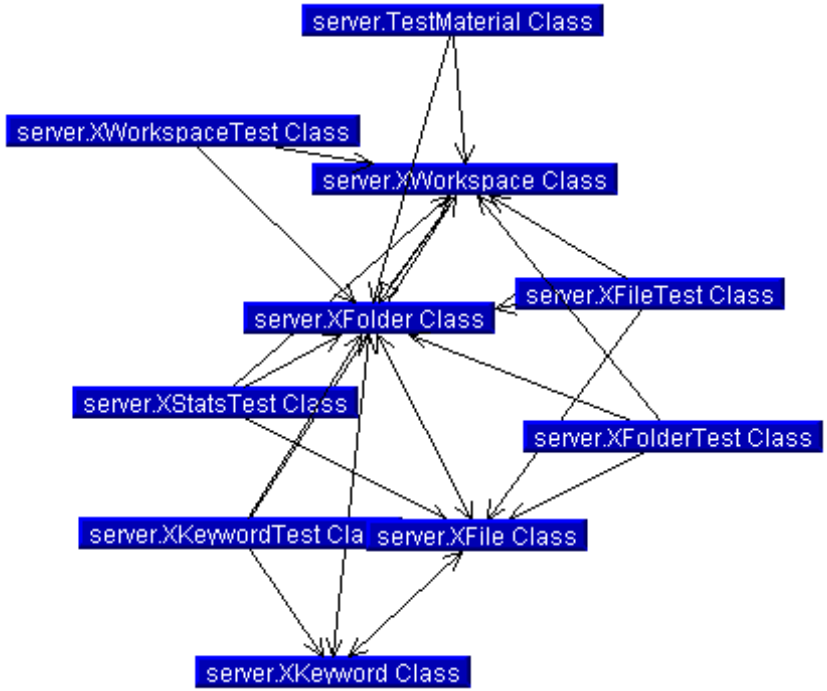
- Complexity analysis: similar results as with data types
- Other things to check:
  Recursion chains and number of function parameters :
  - Recursion chains: series of function calls that make a loop
  - Long chains: probably not created on purpose; might (in rare cases) lead to infinite loop of function calls, potentially causing memory exhaustion, process-abortion,...
  - Number of parameters: usage gets more complex as number of parameters increase
- Results :
  - Recursion chains:
    Solid: Good results: few chains, max. length: 7;
    Mozilla: No so good: more chains, max. length: 114
  - Number of parameters:
    Results OK, but some weaknesses in API-like parts

Verdict on SolidTech‘s Code

- Bottom line: SolidTech‘s database server code seems to be in a very good shape
- Good:
  - Consequent usage of complex data types improves encapsulation and abstraction properties
- Interesting findings for SolidTech:
  - Some parts in the server‘s core are highly complex
  - But: these parts have been well encapsulated
  - A few minor design flaws have only local effects and can therefore be easily removed

Case Study 2: Web Application

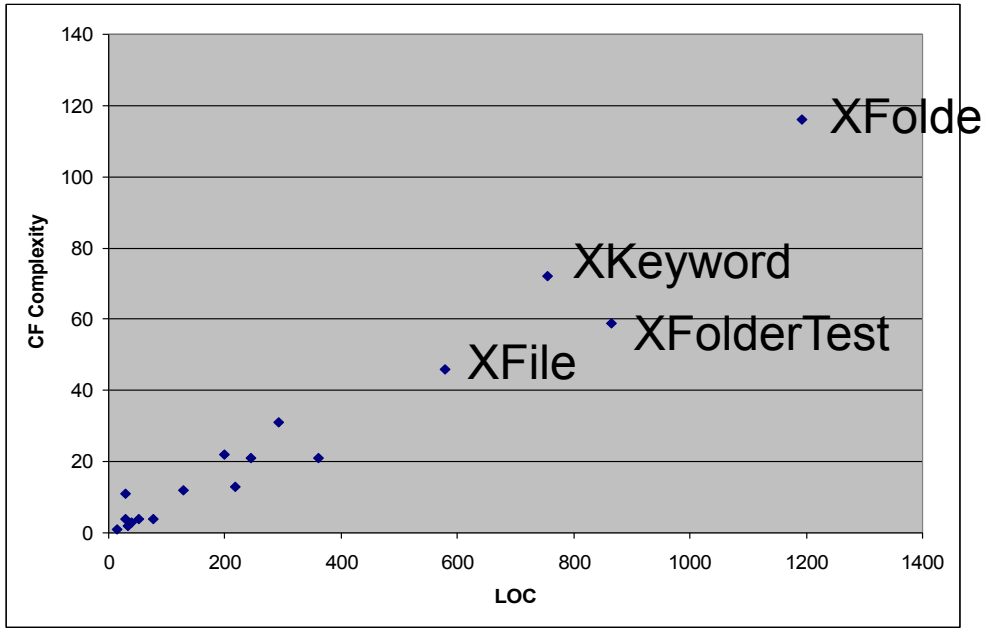
- VTT's eXpert system
  – a web-based knowledge store
- Server classes of the web application in Java:
  - 16 classes, 7 (8) test classes
  - ~ 5100 LOC, 173 methods
- Techniques:
  - Complexity measurements
  - Test coverage for complex parts
  - Comment density
  - Checking of design heuristics
  - Duplicated code analysis



Complexity Measurements

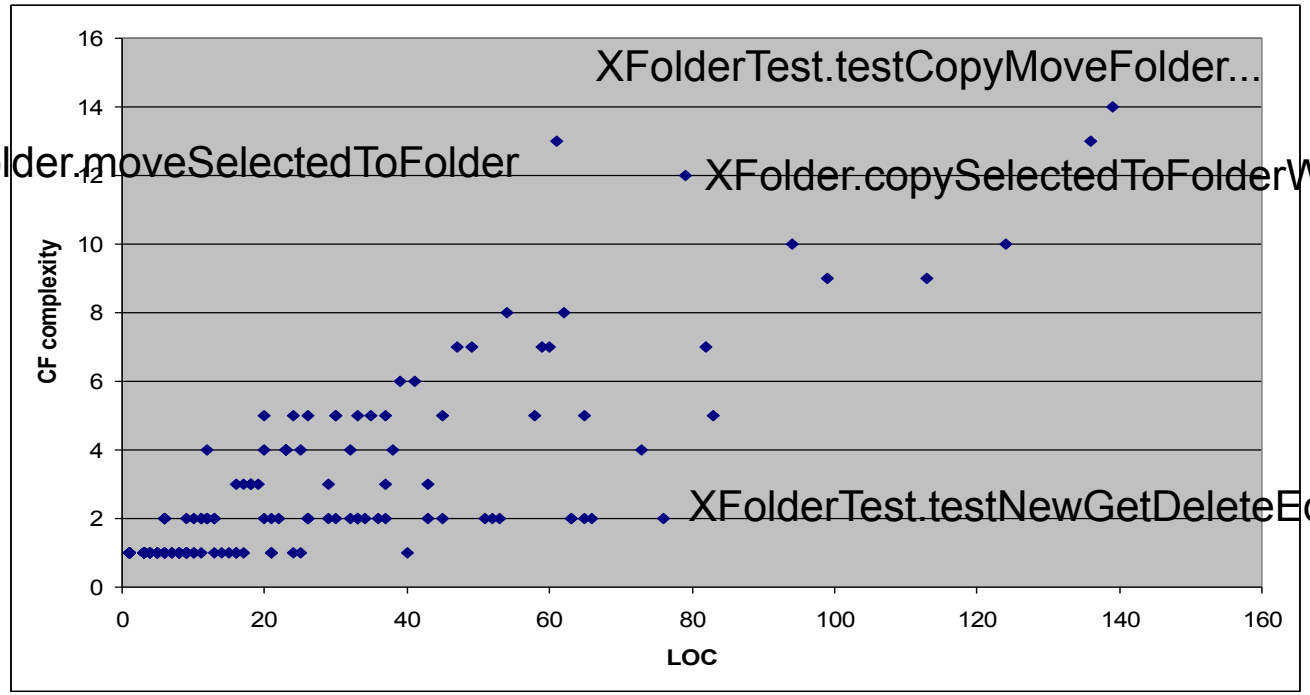
- Complex code
  - is hard to understand
  - is hard to maintain
  - should be carefully tested
  - should be hidden by encapsulation
- Types of measures used within assessment
  - LOC: not including comments
  - Control flow: similar to McCabe
- High values may point to problematic spots in the system

Complexity - Classes



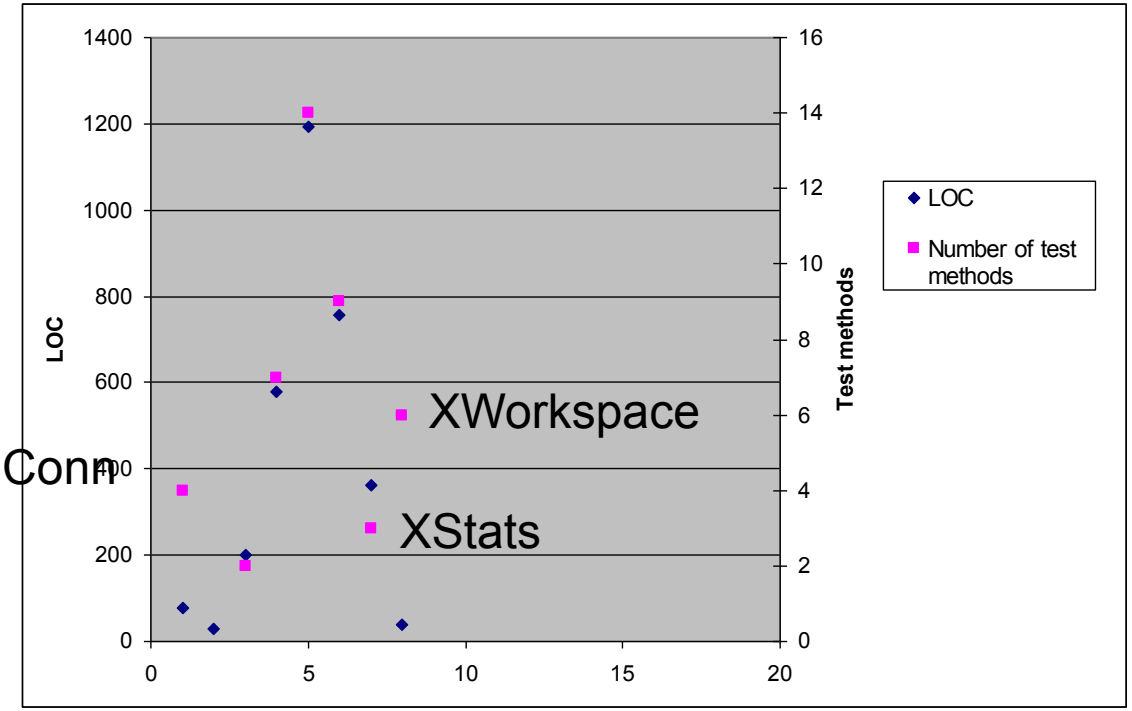
- XFile, XFolder, XKeyword are large, complex
- Good: XFolder seems to be tested well
- Complexity ratio is very homogenous
- Maintainability OK, since there are no classes with extremely complex and compact code

Complexity - Methods



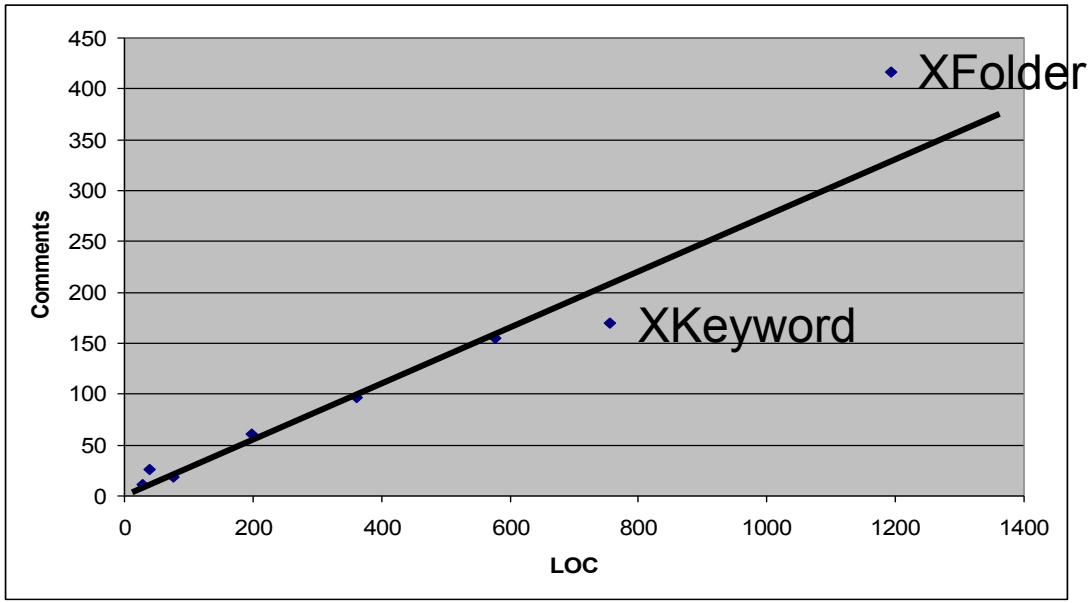
- XFolder/XFolderTest:
  - copy/moveSelectedToFolder have high relative complexity
  - Test methods (testCopyMoveFolder...) are complex and large
  - testNewGetDeleteEditFiles is relatively simple
  - Relative complexity: 0.10 (compared with JHotDraw: 0.23)

Test Coverage



- Number of test methods proportional to LOC (or CF complexity)
- XWorkSpace, DBConn are „well tested“
- XStats‘ test coverage is below average

Comment density



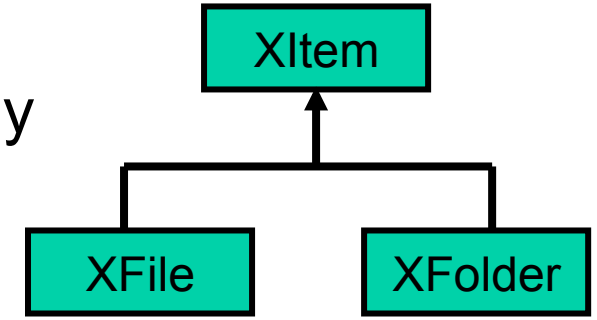
- Comment density homogenous: 20-30% comments
- In comparison with JHotDraw: 26% average density, but high variation

Checking Design Heuristics

Base classes should not know anything about their derived classes	+
Frequently used classes should be stable (in-degree > 5, out-degree> 3)	+
Do not turn an operation into a class	+
Divide large classes, containing more than 25 methods	XFolder 44 - 19 get()/set()
Unused Inheritance	Next Slide
Inheriting same class twice	N/A
Avoid multiple inheritance	N/A

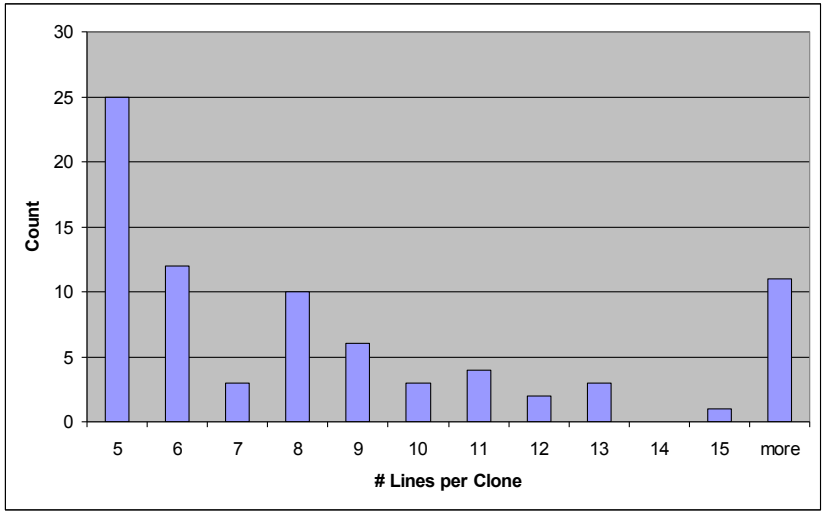
Unused Inheritance

- Detect inheritance, which is never used for the sake of polymorphism
- Interface XItem is not used explicitly
- Discussion:
  - Analysis used only server code
  - JSP code using the server code was not analysed
  - Consequence: probably a false alarm, if XItem is used in JSP or other client code.

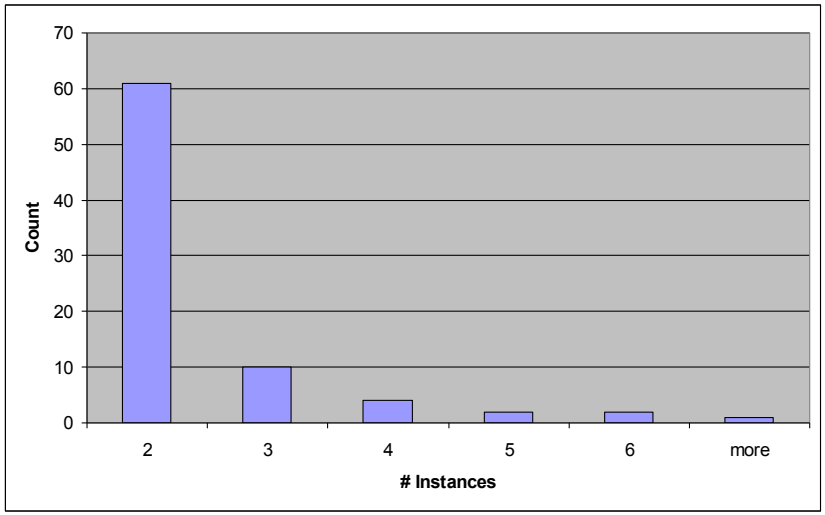


Code Duplication

- Observation:
  - More than 60 cloned blocks (with length >5)
  - Largest clone: 36 lines
  - Many clones involve XFile, XFolder.

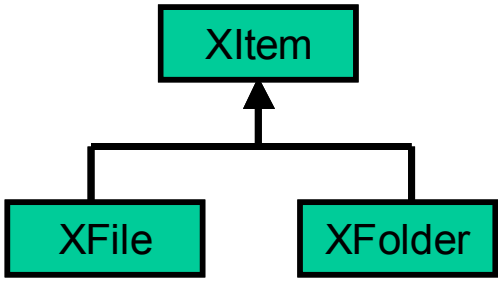


- Discussion:
  - For a system that small, this is quite a high amount of code duplication
  - Much code duplication in test cases



Detailed Analysis

- High duplication between XFolder and XFile
- Duplicated blocks (with length > 10):



36	XFile.java(37-78)	XFolder.java(48-90)	*getOrderByClause*
29	XFile.java(488-520)	XFolder.java(1096-1129)	*getName, getOwner, getRootID, getUpdated*
21	XFile.java(443-466	XFolder.java(1059-1082)	*toString, getDescr*
18	XFolder.java(625-644),	XFolder.java(827-846)	
18	XFolder.java(292-309),	XFolder.java(393-410)	
15	XFile.java(315-331),	XFolder.java(663-679)	*large chunks from checkFileExists*
13	XFolder.java(330-342),	XFolder.java(423-435)	
12	XFile.java(283-294),	XFile.java(342-353)	
11	XFile.java(554-564),	XFolder.java(1176-1187)	*parts from textSearch*
11	XStats.java(163-175),	XStats.java(219-231)	
11	XFile.java(531-541),	XFolder.java(1152-1163)	*parts from textSearch*

- Suggestion:
  - Move duplicated code into common super class
  - Maybe: Transform Interface XItem into a abstract class

Verdict on VTT‘s eXpert Code

- Good:
  - good design concepts
  - extensive set of test cases
  - complexity values OK
- Chances for improvement:
  - „Bad smells“ in XFolder, XFile (key concepts of the server!)
  - Code duplication
- Discussion:
  - Is code duplication a consequence rapid prototyping? (Get it to run first, then worry about refactoring!)

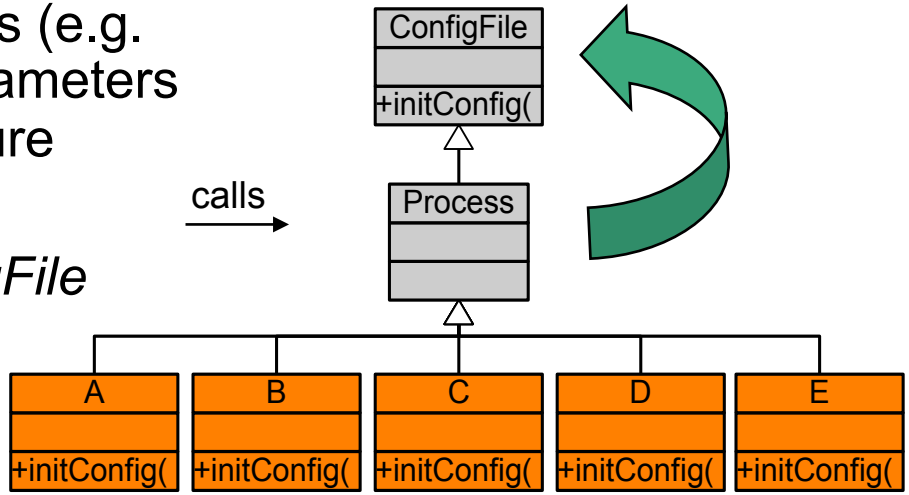
Case Study 3: Telecom System

- Network management software
  - Technologies: Java, C/C++, CORBA,
  - Size: approx. 1 Mio LOC

- Lessons Learned:
  - One fundamental design flaw often has numerous symptoms – many analysis techniques reveal weaknesses!
  - It is often difficult to find out the cause of the flaw!
  - After you‘ve done that, removing the flaw is rather simple!

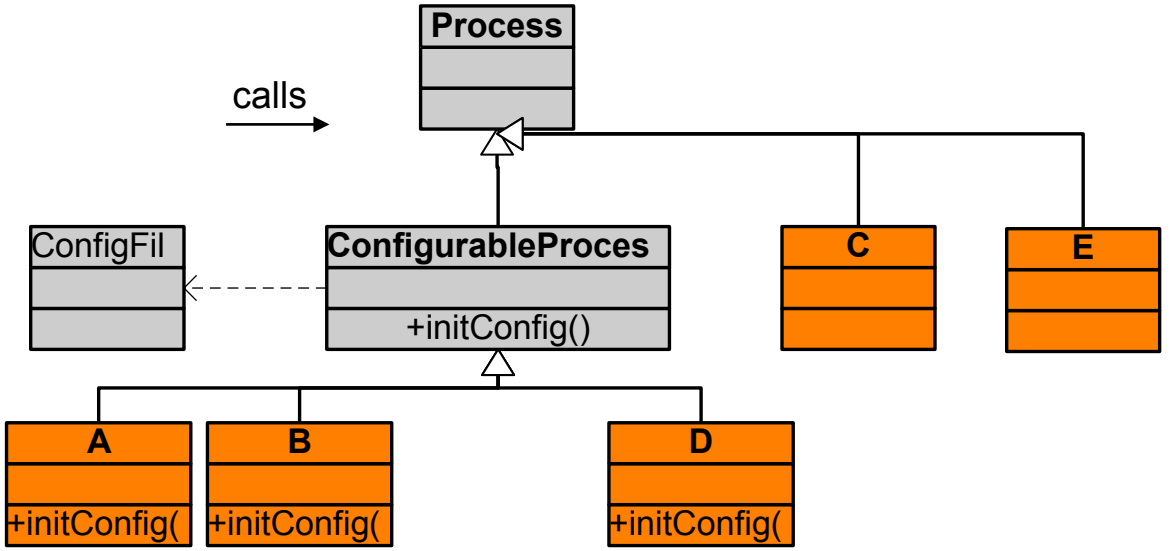
Weak Spot – Situation

- *ConfigFile* defines operations (e.g. *initConfig()),* which read parameters from a file in order to configure processes.
- *Process* inherits from *ConfigFile*



- Special types of processes *A* to *E* overwrite *initConfig():*
  - *C* and *E*'s implementation of *initConfig()* is empty
  - *A, B* and *D*'s implementations of *initConfig()* are very similar to the implementation in *ConfigFile* (dode duplication!); they are quite complex.
- Client code always uses the interface of Process; *ConfigFile* is never used!
- (*initConfig()* contains case statements on types *A, B* and *E*)

Weak Spot – Resolved



- We now have configurable and non-configurable processes
- Inheritance is now semantically OK: Specialization
- Common functionality of *initConfig()* in *A, B* and *D*:
  - Implementation outline (template) in *ConfigurableProcess*
  - Hook methods for specific variation of the behaviour (Template Method Pattern) in *A, B, D*
- Reading the configuration file can be delegated to *ConfigFile*

Lessons Learned

- Tool supported quality assessment often lead to surprising results
- Interpreting findings is like a puzzle – after a while, you get a pretty good picture about the quality of a system:
  - Critical spots affected by many metrics/heuristics often point to severe design problems
  - Bad results may not be that critical!
    Example: High subsystem complexity is not a problem, if the subsystem is well encapsulated
- Analysis techniques give only hints – an experienced developer still has to inspect and evaluate the findings
- Architectural rules + company specific style guides help to preserve a system‘s quality

Benefits of Software Assessments

- Regular assessments help to keep your system's quality under control
- External experts can provide new perspectives on a system and its quality
- Quality assessments may stimulate discussions among developers about quality and possible improvements
  => first steps to a quality aware company!
- Quality workshops help to build up developers' intuition about good design
- Demonstrate, that your company is quality aware!

Are you interested?

Our offer: Tool supported assessments of your software

Features:

- Two experts from FZI thoroughly inspect your Java, C or C++ code for 5 working days (peer work!)
- Confidentiality guaranteed, on-site work at your company
- Workshop to discuss and analyse the assessment results with your developers

Costs:

- 10 person days for the assessment
- Travel + accommodation costs
- Optional: 3 person days for a detailed quality report
- (Some developer resources at your side, e.g. for the workshop)

Our Personal Experience...

- The structure of your software system is a key factor to it‘s success!
- KISS: Keep it simple, stupid! (A. Tanenbaum)
  Whenever you have the feeling that something is complicated, simplify it!
- Use decomposition to reduce the complexity!
  Miller‘s Law: A good structure should allow you to keep only seven (+/-2) things in mind at one time.
- Name artefacts meaningfully!
  If you cannot think of a suitable name for a concept, you have not properly understood it (or it is not a valuable concept at all)! => Rethink it!

Contact

For more information contact:

Markus Bauer, <bauer@fzi.de>
Department PROST
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe
http://www.fzi.de
Phone: +49 721 9654 630
FAX: +49 721 9654 609