Realities of
(Scientific Software) Engineering

Jurgen Vinju

October 29, 2002
Methodology in General

- Meta + Hodos + Logia
- A regular and systematic way of accomplishing something
- Algorithms for humans
- One brings theory into practise using a methodology
- If theory and practise fail: methodology
- The human equation: intelligence, discipline, sociology, psychology
Software Engineering

Issues in this talk:

- What’s (different in Scientific) Software Engineering?
- What mistakes did we make and what solutions do we use?
- First mistake: Software Engineering $\equiv$ Programming
Management: Goals (1/2)

• 2nd mistake: Normal Software Engineering in a Research group

• The General Mission Statement:
  “Make Software That Makes Profit”
  → Efficient Engineering Process
  → Marketability
  → Timing
  → Competition
  → Money and investments
  → Continuity
Management: Goals (2/2)

- Our Mission Statement
  
  “Invent New Stuff, Proof that it works, Teach it”
  
  → Almost the same implications, plus...
  
  → No innovations, real inventions
  
  → General relevance, genericity
  
  → Explainability and simplicity
  
  → No profit ≡ No motivation for investments

- Conclusions:
  
  More work with less money and less time
  
  → Do less and more efficient!
  
  → Invest in more efficiency
Management: People (1/2)

- Roles/Actors in the general Software Engineering process:
  Managers: General, Sales, Technical, Floor
  Sales persons
  Designers: Architectural, Functional
  Programmers
  Testers
  Users

Number of People
Management: People (2/2)

- Reality: We do not have many people

- But we do have the same roles to act!

- Highly educated thinkers and speakers

- Conclusion: be aware of your role in the engineering process at any time, and be aware of the role of others
Management: conflicting interests

- Time: Papers versus Software
  Long term continuity versus short term results
  Usability versus new functionality

- Judgement
  Individual versus group
  Short term versus long term
  Internal versus external

- Shared conflicts
  Shared software $\equiv$ shared papers
  $\rightarrow$ Software supports papers
  $\rightarrow$ Papers support software
Management: Conclusions

- We need to be aware of our methodology
- Everybody is always involved in everything
  → Software and papers
- We have to deliver high(er) quality of software
- We have to be satisfied with less quantity
- We need to invest
Source Code
Source Code and the Laws of Murphy

- There’s always something stupid wrong: bugs are never interesting
- It’s always somebody else’s fault or it was a long time ago for you
- Everything is related to everything
- Everything is always similar, but not quite equal
- Everything always changes
  Requirements, Functionality, Context, People
- The documentation is always out of date
  And the source code comments too
- Nothing works when you actually need it
Source code: How to ask for trouble (and we did!)

- One programming language: everything in Lisp/C/C++
- A simple architecture: one program does everything
  
  A simple architecture does not mean a simple design
- A simple programming interface: everything is an ATerm
- A simple source tree: everything in the same source tree
- Simple code reuse: copy & paste
- Efficiency first: obfuscated code
- No dependencies: no reuse
- Everybody specializes: nobody knows anything
- Release the software only when its finished
- Change the formalism and the architecture simultaneously
Source code: solutions

1. Standardization
2. Architecture
3. Abstraction
4. Automation
5. Testing
6. Knowledge spreading

Each of the above is a **costly investment**, with **high rewards**
Source code: solution 1 - standardization

- We use CVS: there is one repository
- All tools have versions
- LGPL license
- Everything is represented as AsFix, or an ATerm
- Programming style: e.g. layout, nomenclature, length of procedures
- Interfaces: commandline, ToolBus, configure scripts
  For example; every tool has ’-h’ and ’-V’ and ’-v’ options.
- Keep most of the system stable, while improving other parts
Source code: solution 2 - architecture

• “A style and method of design and construction”
• ToolBus: separating computation from communication
• Separate source code packages:
  logical separation of functionality
  hierarchical layers of dependency
  units of reuse
• Example: ’asfsdf-meta’ uses ’sglr’ which uses ’pt-support’
  ’sglr’ can be used without ’asfsdf-meta’ (e.g. in ’elan-meta’)
  ’pt-support’ can be used without ’sglr’ (e.g. in ’asf-compiler’)

Source code: solution 3 - abstraction

- Create packages for every component
- Create a commandline/ToolBus tool for every basic functionality
- Create API’s for every data structure
- Create procedures for every computation
- Abstraction implies:
  - documentation
  - reusability
  - replacability
- Choice of abstractions
  - Arbitrary, logical or enforced by an interface
  - Stratification
Source code: solution 4 - automation

- gmake, automake, autoconf: generate makefiles and configure scripts
- getopt: command line parsing
- ApiGen: generates abstract data types
- SDF: parser generation
- ASF: transformation tools
- autobuild: repetitive commandline work and reporting
- dbs (daily build system):
  - cvs checkout, configure, build, check, install, distcheck
- autobundle:
  - automated source tree composition and downloading
documents dependencies between packages
Source code: solution 5 - tests

- Write automated test procedures and programs
- Functionality/Unit testing
  Higher confidence in correctness
  Documents functionality or fixed bugs
- Regression/Component testing
  Higher confidence in overall functionality
- Integration testing
  Testing the communication between components
Source code: solution 6 - knowledge spreading

- Pair programming
- ChangeLog/CVS messages: all small changes explained by short descriptions
- Presentations/Mailing lists/Papers
- **Refactoring**: by changing something a little bit, you get to know it
  - Everything that is wrong or ugly
  - Delete it, change it or reimplement it
  - It will bother you later anyway
Measurable Results

- Much, much less code is left
- Less boring work
- Less bugs
- Self-documenting (for the programmer, not the user)
- Changes/Replacements/bugfixes are made very quickly
- Papers on the new tools
Conclusion

- I have told you a lot
- Awareness of the process and the technique
- Made many errors, fixed them one step at a time
- Set of solutions that require investments:
  - Teamwork
  - Standards
  - Tools
  - Refactorings
  - Tests