Selected topics in software development

Today:

Habits and tools

Speaker:

Jyrki Katajainen

- habits — of individual developers
- processes — of larger development teams

Course home page:

http://www.diku.dk/forskning/performance-engineering/Software-development/

Development environment

The IDE used for the development was Emacs, selected for its power, extensibility, and excellent portability, including portability to hand-held and wearable devices that I often used for development on the move. I also appreciated the availability of Emacs's `cperl` mode, which manages to offer pretty good auto-formating for Perl code, even though "only `perl` can parse Perl".

[Ashish Gulhati, Beautiful code, p. 168]

Iterative design

I typically write out little snippets of pseudocode that describe how I want something to work (a "code story"), play a bit with input and output formats, do a bit of coding, and—if I'm not satisfied with how the system is fitting together—go back and rework the code story. For anything larger than a toy application, I implement little bits of the system and test them out in stand-alone programs before deciding whether to move forward with that part of the design. I keep my notes in a "stream of consciousness" text file, and commit code to my CVS repository often. I try to make all the code visually appealing and elegant. If it isn't elegant, something is wrong with the design, and I go back to the drawing board.

[Lincoln Stein, Beautiful code, p. 192]

Hands dirty early

I'm a firm believer in diving right into coding as soon as possible. You can only learn so much from a book, an article on a web site, or a set of API documents. By getting your hands dirty early on in the process, you'll uncover a lot of things you may not have thought about by simply studying the problem.

[Andrew Patzer, Beautiful code, p. 458]

Work processes vs. tools

While debugging with `ddd` is usually more fun than using a command-line tool, it does not necessarily make you a more efficient debugger. For this, the debugging **process** is far more important than the tool.

[Andreas Zeller, Beautiful code, p. 464]

Debugging TEX

Watching the program execute itself in this "dynamic order" has always been insightful for me, after I've desk-checked it in the "static order" of my original code.

the code for typesetting mathematics; this took another four days. (Well, the "days" were nights actually; I worked during the night to avoid delays due to time-sharing.)

How big was TEX at the time? I estimated...the total number of statements...4600. Thus the debugging strategy I used allowed me to verify about 35 statements per hour.

[Donald E. Knuth, The errors of TEX]

Figures from his notebook

I kept track of how long this process took, so that I'd be better able to estimate the duration of future programming projects.

Day	Time (hours)	Day	Time (hours)
10 Mar 1978	6	19 Mar 1978	7.5
11 Mar 1978	7	20 Mar 1978	10
12 Mar 1978	8	21 Mar 1978	8
13 Mar 1978	7	22 Mar 1978	6
14 Mar 1978	8	23 Mar 1978	7.5
15 Mar 1978	8	25 Mar 1978	7
16 Mar 1978	7	26 Mar 1978	6
17 Mar 1978	7	27 Mar 1978	8
18 Mar 1978	8	29 Mar 1978	6

Assembler programming → structured programming → literate programming

The total debugging time, 132 hours, was extremely encouraging to me, because it was much less than the 41 days it had taken me to write the program. Previously I had needed to devote about 70% of program development time to debugging, but now the figure had dropped to about 30%. I considered this to be a tremendous victory for structured programming, since my programming time had also decreased from what it had been with old **habits**. Later, with the WEB system, I noticed even further gains in productivity.

[Donald E. Knuth, The errors of TEX]

Step through your code

The best way to write bug-free code is to actively step through all new or modified code to watch it execute, and to verify that every instruction does exactly what you intended it to do.

And what's the best way programmers can test their code? It's by stepping through it and taking a microscopic look at the intermediate results. I don't know many programmers who consistently write bug-free code, but the few I do know habitually step through all of their code.

remember that that the goal is to catch bugs at the earliest possible moment. Stepping through code helps to achieve that goal.

[Steve Maguire, Writing solid code, §4]

Online notebook

- Open a window and keep an editor running while you work; cut & paste
- An online notebook can be used both for design and programming
- Experiences from other places confirm that this is one of the most valuable skills learnt

For an example of an online notebook, see

CPH STL repository:

`CPHSTL/Program/Bitset` Jeppe's `ChangeLog`

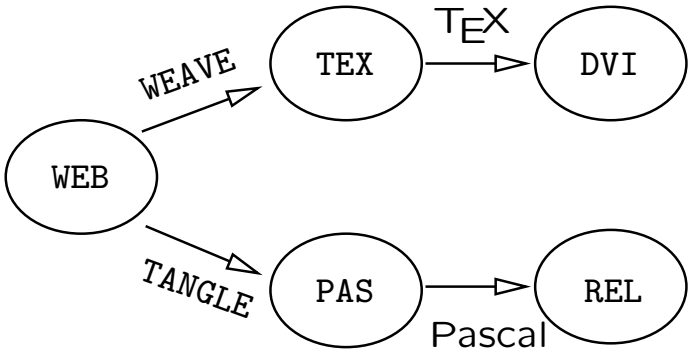
Literate programming

Consider programs to be works of **literature**.

The practitioner of literate programming can be regarded as essayist, whose main concern is with exposition and excellence of style.

WEB = TEX + Pascal

CWEB = TEX + C



[Donald E. Knuth, Literate programming]

Online exercise (5–10 min)

What is your first impression of Silvio Levy's and Donald E. D. Knuth's word-count program?

Explain how the word counting is actually done.

Defensive programming

In defensive programming, the main idea is that if a routine is passed bad data, it won't be hurt, even if the bad data is another routine's fault.

The best form of defensive coding is not inserting errors in the first place. Using **iterative design**, **writing pseudocode** before code, **writing test cases** before writing the code, having low-level **design inspections** are all activities that help prevent inserting defects.

The whole point of defensive programming is guarding against errors you don't expect.

[Steve McConnell, Code complete, §8]

Assertions

code will work in every case except the one in which `malloc` fails.

The compiler can't catch this bug or any like it. Nor can the compiler catch bugs in your algorithms, verify your assumptions, or in general check the validity of data being passed around.

Finding these kinds of bugs is hard. It takes a skilled programmer or tester to consistently root them out. But finding these kinds of bugs automatically is easy, if you know how.

[Steve Maguire, Writing solid code, §2]

Implementing ASSERT

```
#ifdef DEBUG
  void __assert(char*, unsigned int); /* prototype */

  #define ASSERT(condition)        \
    if (condition) {               \
    }                              \
    else {                         \
      __assert(__FILE__, __LINE__) \
    }
#else
  #define ASSERT(condition)
#endif

void __assert(char* strFile, unsigned int uLine) {
  fflush(NULL);
  fprintf(stderr, "\nAssertion failed: %s, line %u\n",
          strFile, uLine);
  fflush(stderr);
  abort();
}
```

[Steve Maguire, Writing solid code, §2]

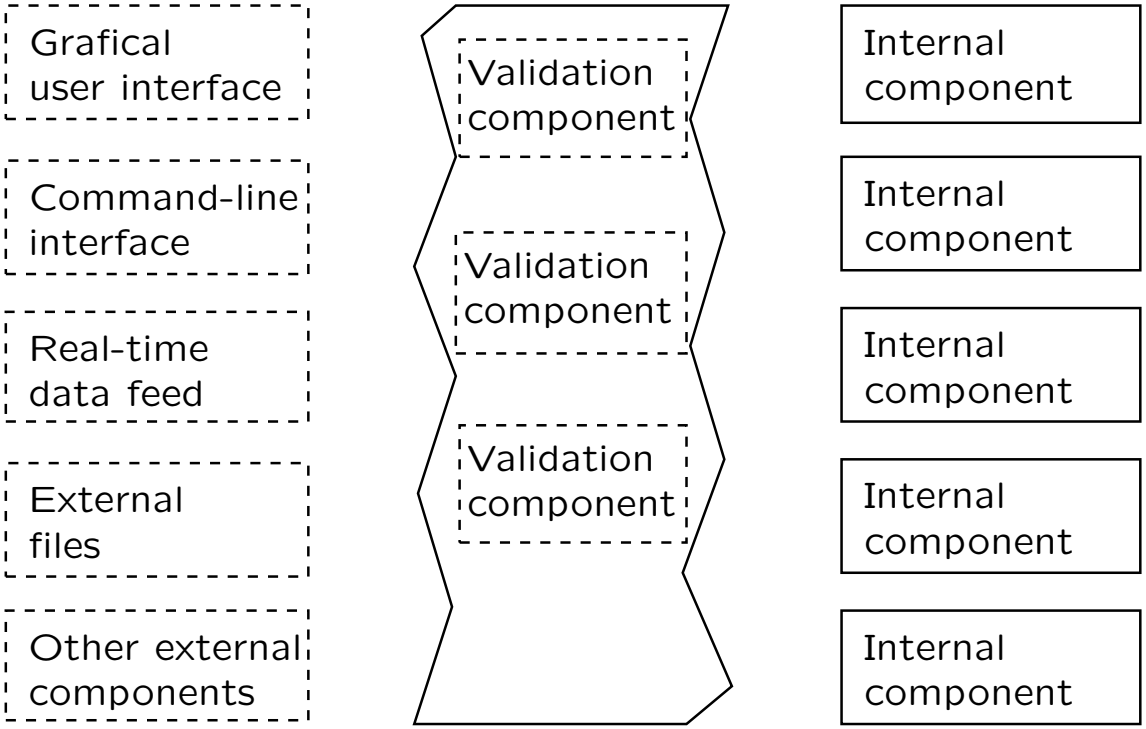
Guidelines for using assertions

- Use assertions to validate function arguments.
- Strip undefined behavior from your code, or use assertions to catch illegal uses of undefined behavior.
- Don't waste people's time. Document unclear assertions.
- Either remove implicit assumptions, or assert that they are valid.
- Use assertions to detect impossible conditions.
- Don't hide bugs when you program defensively.
- Use a second algorithm to validate your results.
- Don't wait for bugs to happen; use startup checks.

[Steve Maguire, Writing solid code, §2]

Barricades

Barricades are similar to firewalls in a building. A building's firewalls prevent fire from spreading from one part to another.



[Steve McConnell, Code complete, §8.5]

Offensive programming

- Make sure **asserts** abort the program.
- Completely fill any memory allocated so that you can detect memory allocation errors.
- Be sure the code in each **case** statement's **default** or **else** clause fails hard (aborts the program) or is otherwise impossible to overlook.
- Set up the program to e-mail error log files to yourself so that you can see the kinds of errors that are occurring in the released software, if that's appropriate for the kind of software you're developing.

[Steve McConnell, Code complete, §8.6]

Removing debugging aids

- Maintain both ship and debug versions of your program.
- Use version-control tools to build different versions of a program from the same source files.
- Use build tools like `make` to include all the debug code in development mode, and exclude any debug code in production mode.
- Use built-in preprocessor to include or exclude debug code at the flick of a compiler switch.
- Or write your own preprocessor for including and excluding debug code.

[Steve McConnell, Code complete, §8.6]

Pair programming

When **pair programming**, one programmer types in code at the keyboard and the other programmer watches for mistakes and thinks strategically about whether the code is being written correctly and whether the right code is being written.

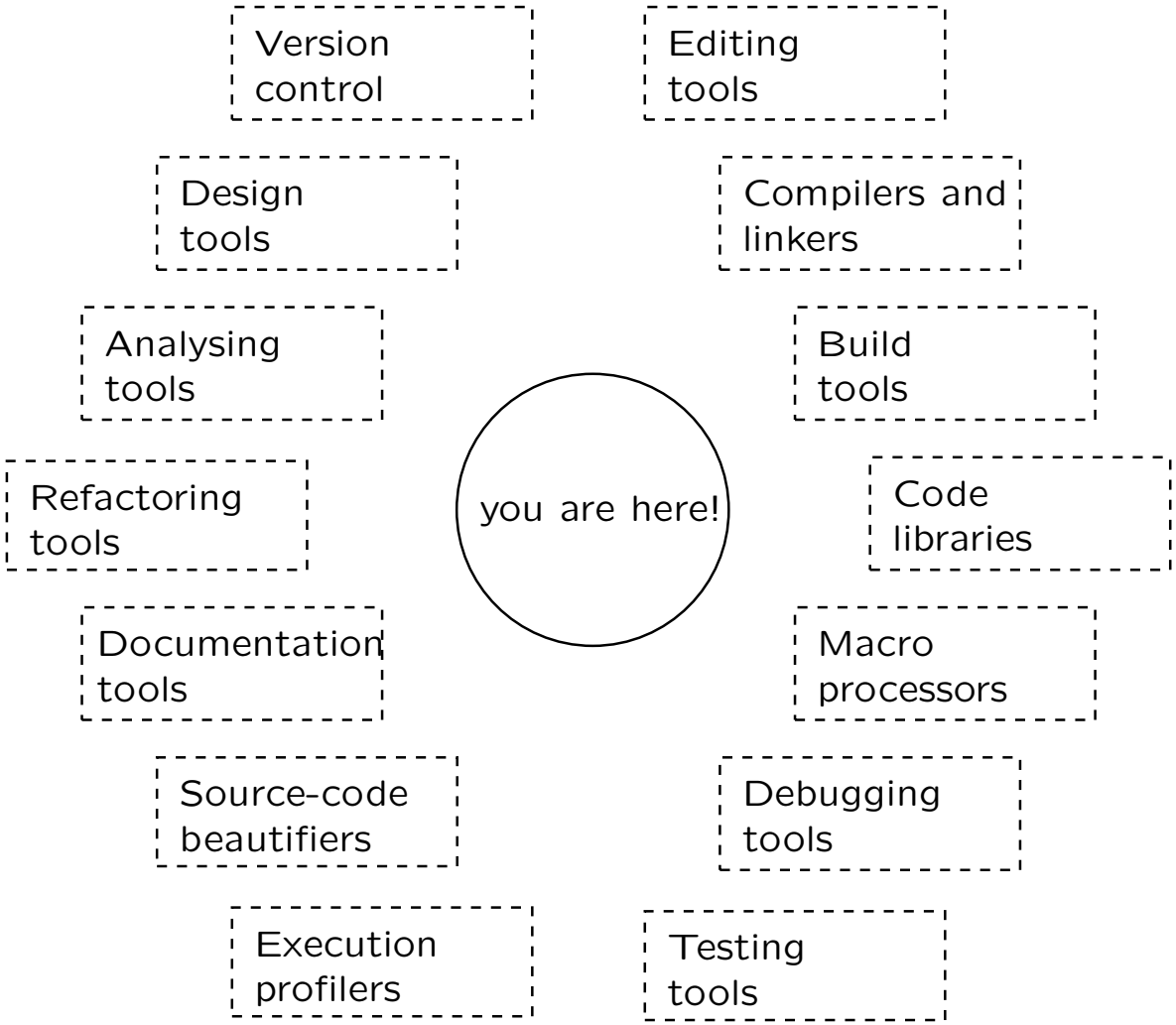
Compare this to **pair writing**; most of my recent joint work (with Christopher Derek Curry and Claus Jensen) is done using pair writing. That is, the technique can be used for non-construction activities as well.

Key to success with pair programming

- Support pair programming with coding standards
- Do not let pair programming turn into watching
- Do not force pair programming of the easy stuff
- Rotate pairs and work assignments regularly
- Encourage pairs to match each other's pace
- Make sure both partners can see the monitor
- Do not force people who do not like each other to pair
- Avoid pairing all newbies
- Assign a team leader

[Steve McConnell, Code complete, §21.2]

Programming tools



Debugging tools

Compiler warning messages: Set your compiler's warning level to the highest and fix the errors/warnings it reports.

Test scaffolding: Use any of the numerous test frameworks available to provide scaffolding for your programs (CppUnit, UnitTest++).

Execution profilers: Recall the folk theorem that less than 4% of a program usually accounts for more than 50% of its run time.

Interactive debuggers: Recall that a debugger isn't a substitute for good thinking. But, in some cases, thinking isn't a substitute for a good debugger either.

[Steve McConnell, Code complete, §§23, 30]

Testing tools

Automated test frameworks: E.g. CppUnit and UnitTest++

Automated test generators: Topic for a master's thesis!

Coverage monitors: E.g. logic analysers and trace monitors

System perturbers: E.g. memory fillers, memory shakers, selective memory failers, memory-access checkers

Diff tools: E.g. for comparing data files and captured output

Defect-injection tools: E.g. for testing exception safety

Defect-tracking software: E.g. error databases

[Steve McConnell, Code complete, §§22, 30]

Specific tools relevant for your assignment

IDE: Eclipse, `emacs`

Scripting: Python, `make`

Shell commands: `diff`, `grep`, `find`

Version control: `cvs` (includes diff, merge, and log facilities)

Concept correctness: Boost concept archetypes

Debugging: `ddd`

Memory-usage monitoring and profiling: `valgrind`

Benchmarking: `benz`

20% of the tools tend to account for 80% of the tool usage.

Read manuals!

Manual phobia is rampant among programmers. Manuals tend to be poorly written and poorly organized, but for all their problems, there's much to gain from overcoming an excessive fear of paper products. Manuals contain the keys to the castle, and it's worth spending time reading them.

[Steve McConnell, Code complete, 1st edition, p. 760]

Summary

- Programmers sometimes overlook some of the most powerful tools for years before discovering them.
- Good tools can make your life a lot easier.
- Tools are readily available for editing, analysing code quality, refactoring, version control, debugging, testing, and code tuning.
- You can make many of the special-purpose tools you need.
- Good tools can reduce the more tedious aspects of software development, but they can't eliminate the need for programming, although they will continue to reshape what we mean by *programming*.

[Steve McConnell, Code complete, §30]

Discussion: Are your tools in order?

- Do you have an effective IDE?
- Does your IDE support integration with source-code control; build, test, and debugging tools; and other useful functions?
- Do you have tools that automate common refactorings?
- Are you using version control to manage source code, content, requirements, designs, project plans, and other project artifacts?
- Are you making use of an interactive debugger?
- Do you use `make` or other dependency-control software to build programs efficiently and reliably?
- Does your test environment include an automated test framework, automated test generators, coverage monitors, system perturbers, diff tools, and defect-tracking software?

[Steve McConnell, Code complete, §30]