Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Software Configuration Management

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/
Slides stolen from Eelco Dolstra

Department of Information and Computing Sciences, Universiteit Utrecht

September 26, 2008

Overview

Software configuration management

Scenarios

Software configuration management areas

1. Software configuration management

What SCM isn't

What it's not about:

- How to configure your printer driver for duplex printing in Windows XP.
- How to configure Doom 3 to use anti-aliasing.

Configuration management

- Webster: *configuration*, noun, from Latin verb *configurare*, "relative arrangement of parts or elements".
- General CM: "the discipline of controlling the evolution of complex systems" (Tichy).
- Necessary for complex systems since:
  - Design evolves.
  - *System families* (or *product lines*).

Software product lines/application families 1 §1

- Software systems are configured/specialized versions of a generic system.
  - Exploit similarities, configure the differences
- The generic system is
  - usually tailored for a specific domain and
  - has a fixed architecture
- The same application for multiple platforms.
- Examples: internet banking applications.
  - Quinity, Enterprise Resource Planning systems (SAP)
- ERP (deployment time configuration): fixed number of freedoms of configuration.
  - If you need more, make sure you bring some money.
- Quinity (design time configuration)
  - Customer never notices that the system was not built from scratch.

Software product lines/application families 2 §1

- Specializing to a software product line gives many advantages:
  - stability
  - speed of development due to less work
  - speed of development due to familiarity
- Of course, there should be enough demand for the type of application.
- Rather like a monolithic, very specialized application framework.

Software configuration management

- SCM is CM applied to software systems.
- Required since software products aren't atomic, non-evolving, non-composable entities.
- Difference to general CM:
  - Software evolves much faster.
  - Potentially more automatable.

2. Scenarios

Scenario: multiple developers, multiple workspaces

§2

- Alice and Bob are working on some software product.
- They stay synchronised by emailing each other their changes.
- Problems:
  - Manual labour. Which files change?
  - What if they forget a file? Loss of synchronisation.
  - What if they both changed the same file?

Scenario: multiple developers, single workspace §2

- Alice and Bob now use a network drive and work on the *same* files simultaneously.
- No longer any sync issues, but...
- Problems:
  - What if they are editing the same file at the same time?
  - Alice is making big changes to support component X. Bob is working on application Y that uses X. But while Alice is changing X, Bob is stuck because his stuff doesn't compile.

Scenario: supporting old releases

- Alice and Bob have released version 2.0 to the customers and start working on version 3.0.
- A bug is discovered in 2.0 which necessitates a version 2.1.
- So now there is development on two *branches*: the 2.x stable branch, and 3.x development branch.
- Problem:
  - They fix the bug relative to 2.0 and release 2.1...
  - But forget to port the fix to the ongoing 3.0 development.
  - Thus, a *regression* occurs.

Scenario: supporting old releases

Like the previous one, but:

- The source code for 2.0 has disappeared.

Scenario: supporting old releases

Like the previous one, but:

- The source code for 2.0 has disappeared.

Or:

- The compiler has disappeared.

Scenario: supporting old releases

Like the previous one, but:

- The source code for 2.0 has disappeared.

Or:

- The compiler has disappeared.

Or:

- The people who know how to build the system have disappeared.

Scenario: supporting old releases

Like the previous one, but:

- The source code for 2.0 has disappeared.

Or:

- The compiler has disappeared.

Or:

- The people who know how to build the system have disappeared.

Or:

- The hardware necessary to build the system has disappeared.

Real example: Skylab reactivation mission §2

- Attempt to keep Skylab in space for a longer period (it was slowly falling back to earth due to atmospheric drag).
- This required uploading new control software.
- *"When IBM began to make preparations to modify the software, it discovered that there was almost nothing with which to work. The carefully constructed tools used in the original software effort were dispersed beyond recall, and, worse yet, the last of the source code for the flight programs had been deleted just weeks beforehand. This meant that changes to the software would have to be hand coded in hexadecimal, as the assembler could not be used—a risky venture in terms of ensuring accuracy. Eventually it became necessary to repunch the 2,516 cards of a listing of the most recent flight program, and IBM hired a subcontractor for the purpose."*

Real example: Skylab reactivation mission (cont'd)

§2

- *"What happened after the manned Skylab program demonstrated the need for foresight and proper attention to storage of mission-critical materials until any possibility of their use had gone away. [...] The destruction of the flight tapes and source code for the software by unknown parties was inexcusable. A single high-density disk pack could have held all relevant material."* (NASA, *Computers in Spaceflight—The NASA Experience*)

Scenario: component integration

- To prevent all the sync overhead, every team member works on just one component of the system.
- At the end (just before the deadline), they put everything together (*big-bang integration*).
- Nothing works—expected and actual interfaces are subtly different from the design documents, many bugs surface only now in the complete system, and so on.

Scenario: manual building

- For very small systems it may be doable to build the system by hand.

```
$ javac *.java
```

- Doesn't work anymore when we have
  - many build steps,
  - different languages,
  - many source files,
  - lots of build time configuration options.

```
$ gcc -g -DDEBUG=1 -c foo.c
$ yacc parser.y
$ gcc -g -DDEBUG=1 -c parser.c
$ gcc -o app foo.o parser.o
```

Scenario: building is slow

- So we could just make a build script.
- What if we have a system consisting of 10,000s of source files and tens of millions of lines of code? $\Rightarrow$ slow.
- And developers should rebuild/test the system after every change!
- Result: developers will skip testing.

Scenario: incomplete rebuilds

- Solution: just rebuild the things that "changed".
- If `foo.c` changes, rebuild only `foo.o` and `app`.
- But if `bar.h` changes, and `foo.c` includes that file...
- And what if the compiler changes?
- *Dependencies* are very tricky.
- If done manually: easy to forget things ⇒ binaries not consistent with sources.
- If done automatically: is the dependency information correct?

Scenario: packaging

- The product is done and you want to release/ship it.
- Should be installed automatically.
- But at installation time, it turns out that you forgot to add a required file to the distribution.

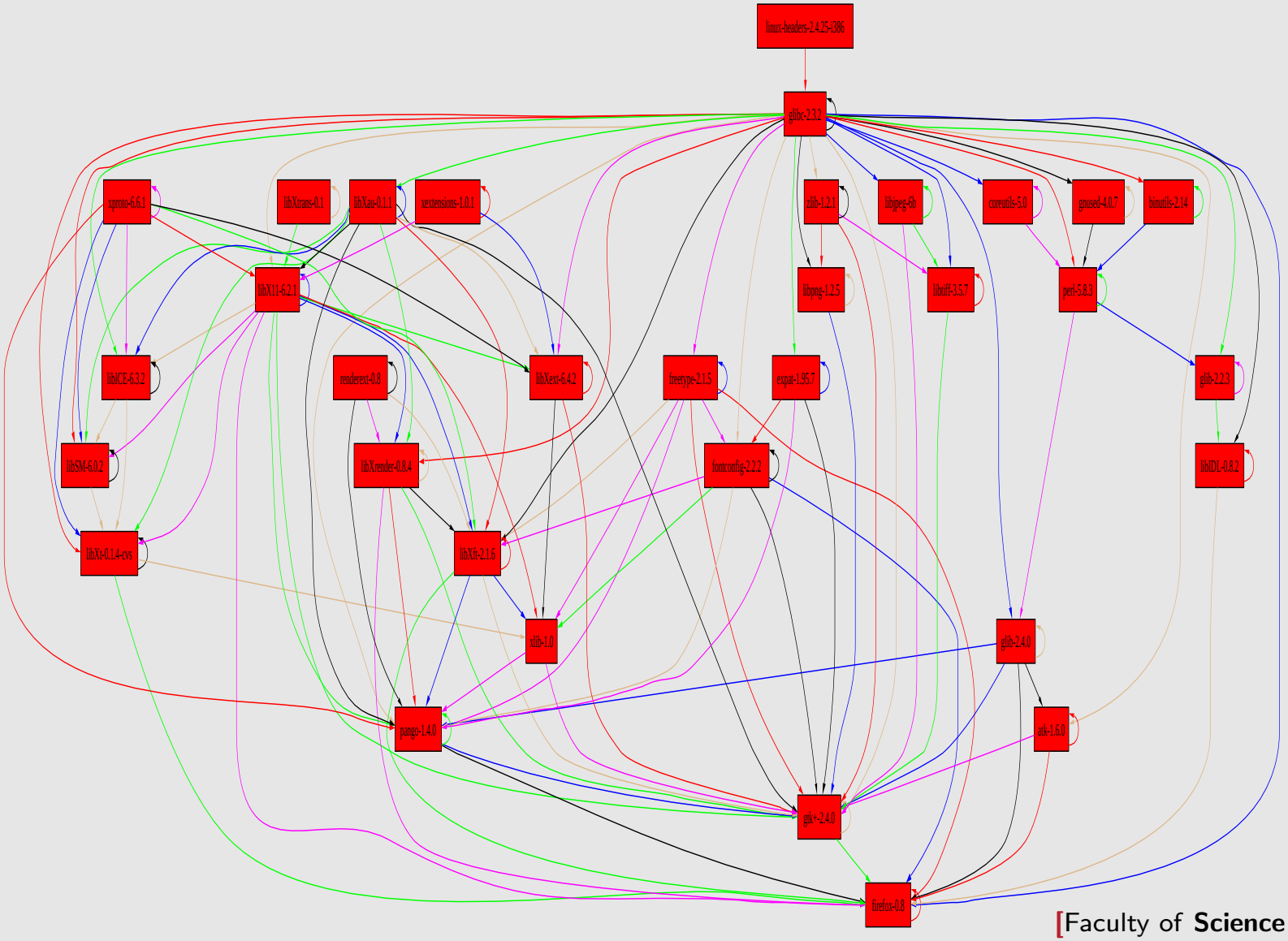
Scenario: DLL Hell

- The user installed your application, but the installation breaks some other application because you overwrote some common DLL file in `C:/Windows/System32` with an incompatible version.

Scenario: missing components

- The user tries to run the application, but gets an error message about `foo.dll` missing.
- Problem: your application required third-party `foo.dll`, but you forgot to ensure that it's installed.
- Testing didn't reveal this since `foo.dll` happened to be present on your machine.

Example: Firefox dependencies



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Scenario: uninstallation

- The user wants to get rid of the application. How to do that?
- Manually delete files: dangerous.
- Automatically delete files: need complete manifest of files belonging to the application.
- What about shared files?

3. Software configuration management areas

SCM areas

- Support the evolution of source code ⇒
  version management (a.k.a. source revision control).
- Control building of derivative artifacts ⇒
  build management.
- Manage transmission and installation of software ⇒
  software deployment / *package management*.
- Support (continuous integration) testing and releasing ⇒
  continuous build systems (a.k.a. build farms).

Related:

- *Issue tracking systems*.

Version management

Goals:

- Provide safe storage for source code (a *repository*).
- Store history:
  - Source code at all (or many) points in time.
  - Reason for the change, and who made the change.
- Allow parallel lines of development.
- Allow flow of changes between developers and branches.
  - *Merging of changes*.
  - Resolve *conflicts*.
- Provide identification.
  - Give names to releases/branches.

Build Management

Goals:

- Build *derivates* (e.g., binaries) automatically from sources.
- Efficiency: prevent unnecessary rebuilds.
- Correctness: do rebuild if necessary.
- Support variability
  - Multiple platforms, debug on/off,...

Deployment

Goals:

- Get software from the producer site to the consumer site.
- Manage installations (install, upgrade, uninstall).
- Correctness: ensure that software works the same at producer and consumer sites.
- Deal with dependencies / interferences.

Continuous builds

Integration of version management and automated builds (and maybe deployment).

Goals:

- Verify after every source change that the system builds / passes automatic tests.
- Portability testing: build on multiple platforms.
- Report build errors.
- Link back to version management ($\Rightarrow$ "blame" facility).
- Maybe build and/or deploy packages automatically.

Bug tracking

- Should be integrated with version management system.
- E.g., so we can query: "is bug X fixed in branch Y?"

More information

- Paper: Walter Tichy, *Software Configuration Management Overview*.
- Book: Stephen P. Berczuk, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*.