Software Tools & Methods
Class 3

Lecturer: Greg Bolcer, greg@bolcer.org

Summer Session 2009

ELH 110 9am-11:50am

Overview

- Homework
- Last Class
  - Reverse Engineering and Refactoring
  - Process Models
- This Class
  - Code Reading
  - Revision and Version Control
- Next Class
  - Software Architecture Models

Notes

- Homework 2 is due today
  - http://checkmate.ics.uci.edu/ is now working
- Homework 2 Lab should be finished
- Homework 3 is due on Thursday
- Homework 3 Lab is due on Thursday
- Reading finished by today chapters 4, 14, 15.3.2 of van Vilet
- Today + 2 more classes until midterm

Software Methods & Tools

CODE READING

Step 1: Acquire Domain Knowledge

- Start with what the program is supposed to do
  - Don’t start with the code
  - Things to try
    - Asking experts
    - Reading documentation or comments
    - Running the program
    - Running test cases
- Whiteboard Exercise, should we use only domain experts in requirements and designing of a software system?

Step 2: Start Looking at the Code

- Where to start
  - Main
  - Specific UI parts
  - Consider the roles of components in the framework
  - Do not just start reading code from top to bottom

Code Reading Strategy

- Use a methodical approach
  - Use program structure and component relationships as a guide
  - IDEs help with this

Code Reading Tactic

- Make use of the style and formatting
  - e.g. indentation, and naming convention
  - But don't let it fool you

```
….
If (meetCondition)        \\1
    a.doSomething().      \\2
    b.doSomething().      \\3
…..
```

- If meetCondition is *false*, will statement 3 be executed?
  - Yes

More Tactics

- Code and comments (or other documentation) may not agree
  - Even names of variables/methods might not agree with their purposes
- Consider the possibility that the programmer did not know what he was doing
- Be wary of objects that have the same identifier but with different scopes
- Be wary of dead code
- …

Software tools & methods

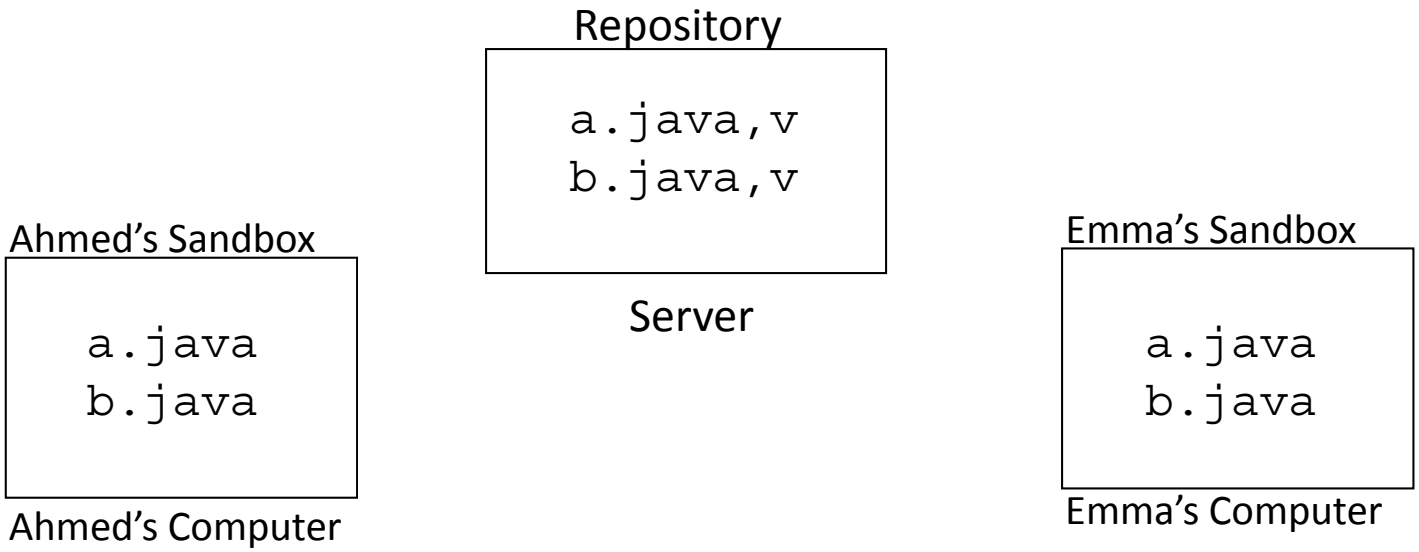
REVISION & VERSION CONTROL

Version Control Tools

- Provide a complete history of a file
  - All current and previous versions available- nothing ever goes away
- Other names
  - Revision Control System
  - Source Control
  - Configuration Management
- Idea:
  - Keep files in repository
  - Also keep a history of the changes to the file
  - Use the history to recover any version of the file
  - Also record times: "What did these files look like at noon last Friday?“
  - Can also tag groups of files

How Version Control Works

- Place the official version of source code into a central repository, or database
- Programmers check out a working copy into their personal sandbox or working copy
- When finished and fully tested, programmers check in their code back to the repository



Subversion

- subversion
  - Open Source
  - Available since 2000 as a successor to CVS
  - Suitable for individuals or medium-sized teams, though large teams are using it too
  - Runs on Linux, Solaris, Mac OS X, Windows, and others
  - http://subversion.tigris.org/

- On command line, check in and check out accomplished by typing commands
- GUI front ends/clients available
  - TortoiseSVN, Eclipse plug-in subclipse

Subversion Commands

- checkout
  - Retrieving a file from the repository
- commit
  - Putting changes back into repository
- update
  - Refresh the local or working copy copy with any changes since checkout

http://www.cs.put.poznan.pl/csobaniec/Papers/svn-refcard.pdf

Control Regimes

- Pessimistic Model
  - File becomes locked on check out
  - Nobody else can make changes
  - Not widely used, most software components have higher granularity than file-based checkout
  - Cost of locking outweighs reduction in team productivity
- Optimistic Model
  - File not locked on check out
  - Conflicts or collisions are managed at check in
  - Leverages tools and best practices
  - Software projects with many smaller components have lower collision of work being performed by different developers
  - Assigning curator or module owners has helped, coordination and communication

Control Regimes

- Default in subversion is optimistic model, but pessimistic model possible
- Change in code check-in and checkout policies largely has reduce use of pessimistic model
- Practice shows, keep your local copies up to date, checkout files only when you are ready to work on them or submit your changes
- Also better RCS tools for sparse branching
  - Diffs, mergetools, automated refactoring, etc.

Conflict Detection & Management

- Most current tools poll repository and provide visual cues
  - to others that have the file opened for edit or
  - when a submission has occurred over the top of your checkout
- Depending on the volatility of the code line and discipline of the teams
  - most conflicts are reduced by use of best practices.
  - If code is being refactored, it should be isolated through
    - Sparce branching practices
    - Isolated branches
    - Late and Short-lived branching of few files

Conflict Detection & Management

- Resolution of conflicts are handle in many case by the tools themselves
- In some cases, out-of-band blending of the code may occur
  - Use 3rd party visual comparison tools
  - Leverage developer or module owner for contextual decisions during code-blending
- Checking in the blended result revision of the artifact breaks the pure geneology of the artifact
  - But at a small cost to keeping a successfully building code line

Conflict Detection &Management

- On check in, official repository copy is compared with new copy
  - Check version number
    - If repository and working copy versions are the same, accept the changes
    - If repository version is newer, reject commit operation.
- Need to update working copy before check in
  - Includes a synchronization step to merge changes from two files (new repository version and modified file from working copy)

Conflict Detection &Management

- Merge algorithm
  - Line by line comparison
  - Changes to different lines are OK
  - Changes to same lines labeled as a conflict
    - Both versions written to the working copy copy
    - Choose which line by editing manually
  - No guarantee of code correctness after merge
- On a successful check in, save new version of files with a set of backwards references to changes
- Can apply detection selectively
  - Binary files only use version numbers or timestamps
  - No conflict detection applied to files that are ignored

Operations are Atomic

- Checkout, update, and commit operations are like database transactions, they are all or nothing
  - In case of network error, machine crash, etc.
  - Avoids partial operations
- Operations work on directories
  - Recursively applied to files
- Every commit transaction is assigned a number, indicating a version of the directory
  - In other words, Version N and Version M of a file may be the same

Updates

- Update command is applied to the directory
- Three possible outcomes for each file
  - U - updated, repository version newer than working copy
  - G - merged, changes don’t overlap
  - C - conflict, you need to resolve
- After resolving, tell subversion client that you have done so

Software Methods & Tools

PROGRAMMING PRACTICES WITH VERSION CONTROL

What to check in

- Code that compiles cleanly and has been tested
- Don't check in files that are automatically created from others
  - e.g. .class files
- Do check in:
  - Your own little test programs
  - And their expected output
  - Readme files, notes, build logs, etc.
  - Anything else you created by hand

When to check in

- Version control is not a backup system
  - Your computer should have one of those
- Don't check in just because you're taking a break
  - Check in files when they are stable
  - e.g. After adding a new feature
- Or when you have to switch machines
  - e.g. From home to school or vice versa

Comments

- Upon check in, you will have the opportunity to add a comment
  - USE THIS FEATURE!
- You're going to wish you did when you try to revert back to an earlier version

More Uses for Version Control

- Protecting you from yourself
  - Backing out changes
  - Finding where errors were injected
- Working with a team
  - Simultaneous file sharing
  - More complex products
    - Multiple versions, platforms
- Recording an audit trail
  - Hey boss, I've been working…
  - Linus Torvalds vs. SCO

Branching and Merging

- Using a particular version as the baseline for a series of versions
- Reasons for branching
  - Variations on a theme
  - Experimental code
  - Bug fix chains
- Happens at the repository, not your working copy

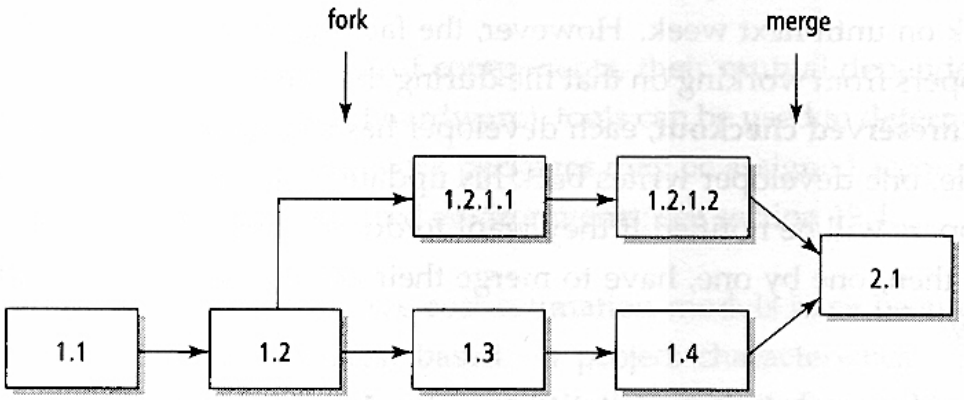
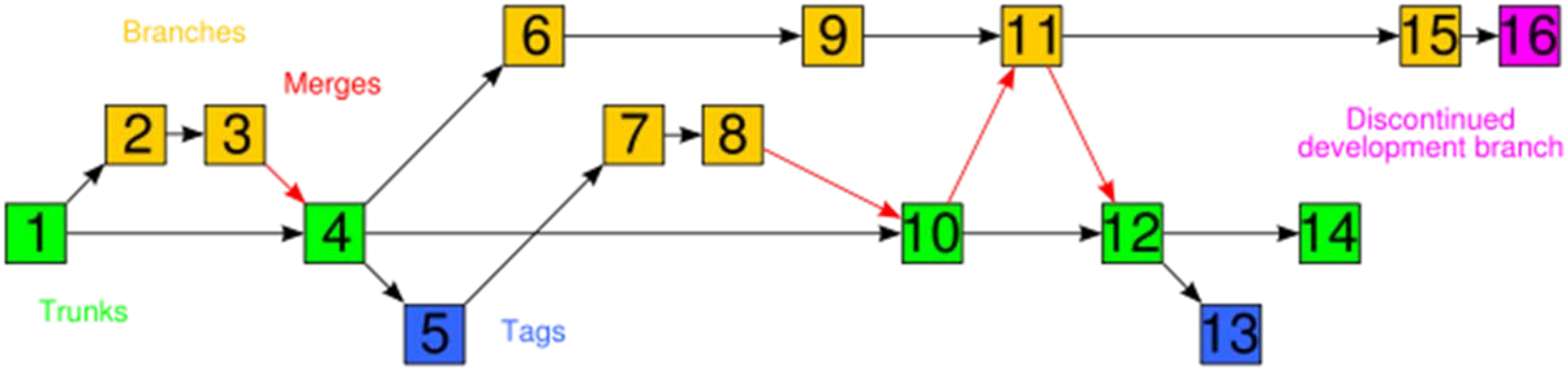


Figure 19.6 Forking and merging of development paths

Tagging

- Use tags to label a group of files
  - Makes it easy to check out a release or configuration



Software Tools & Methods

SOFTWARE CONFIGURATION MANAGEMENT

Whiteboard Exercise

- Name an example where the development, testing, deployment, and runtime configuration changes

Whiteboard Exercise

- Name an example where the development, testing, deployment, and runtime configuration changes
- Example mentioned in previous class, software was an appliance
  - Development, testing done in Vmware
  - Deployment done with hardware box
  - Live updates delivered over network
    - Had different stages of servers

Software Configuration Management

- SCM centers around these three things:
  - Practice
  - Policy
  - Process
- Determining the best tool chain for organization or project is a tough question.
- Trade-offs and constraints can be pretty broad and very passionate point of discussion
- However these three P's transcend over the tool chain choices

Software Configuration Management

- Always have a documented plan and stick to it
- Define a mechanism for determining the degree to which procedures and policies have been followed to maintain consistency and history of versions and variants
  - Has the specified change been made?
  - Has the technical correctness of the change been assessed?
  - Has the software process been followed and standards been applied?
  - Have the SCM procedures for noting the change, recording it, and reporting it been followed?
  - Have all related SCIs been properly updated?

Practice

- Discipline software process
- Accountability or ownership of code lines, modules or components
- Organizational influences, software model being used-- Agile, Waterfall, proto-typing

Policy

- Code line Check-in policies
- Build Frequencies
- Test cycle costing
- When to branch
  - Why to branch
  - Cost of branching

Process

- Complexity of tool chain can increase cost of ownership for the project
- Integrating the disparate tool chain into what appears to be one seamless assembly line that produced identifiable, reproducible testable code
- Promotional model, lowers testing cost, specially when certify smaller components of code
- Identify, use of labels and tags to

Tool Chains

- Choosing a tool chain across teams of developers and organization divisions is a challenge
- A good rule of thumb to assists in the tool selection is to consider these items:
  - Target environment, where will the software be deployed and executed
  - Total cost of ownership of the tool chain: licensing, integration with other tools, local expertise or community resources in managing and using the tool
  - Type of software, shrink wrapped product, embedded system-- like appliance, web site
  - Developers OS and IDE preferences

Tool Chains

- High Level divisions of tools:
  - **OS**: determine the OS that best fits the team or the product target
  - **IDE**: IntelliJ, Eclipse, NetBeans, MS Visual Studio.
  - **Version control system**: CVS, SVN, Perforce, StarTeam, git, and the list goes on
  - **Defect Track**: ExtraView, Bugzilla
  - **Project Management**: Rally, GreenHopper plugin for JIRA
  - **Build Management system**: AntHill, Bamboo, Parabuild, Microsoft Team foundation Server
  - **Continuous Integration system**: Bamboo, Hudson, Continuum
- Choosing a set of tools involves nearly all the software techniques
- Performance, usability, access to support, licensing model, administrative cost-- or cost of ownership.
- In the end it is a huge matrix of constraints and trade-offs

Tool Chains

- Can’t please all the developers or divisions on the project
  - no silver bullet on tool chain choice and how they fit together
  - Someone always thinks their time is more important, do impose process or tools upon me.
  - Any tool chain that ends up being selected to produce your product will be under continuous scrutiny
  - Trade-offs and many factors lead up to the final choices on the tools.
  - The biggest downside to any software project is lack of communication and discipline, ultimately accountability.
- In the end, if you never show up to the race with your product, doesn't matter how cool it is, no one will know

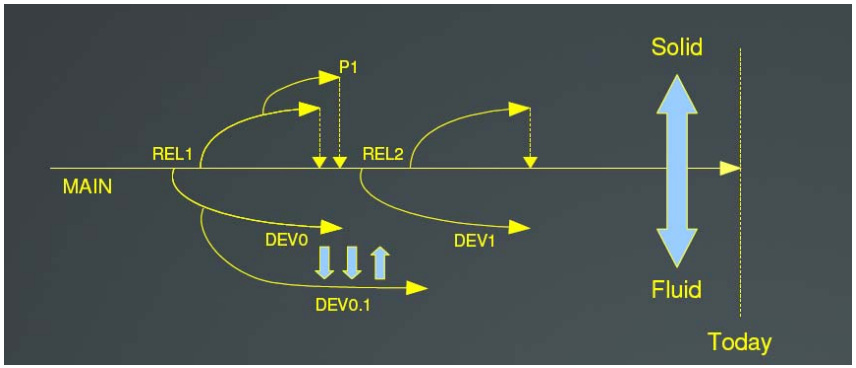
Tool Chains Case Study

- Maven works better with Continuum works better and directly, so we should get rid of Anthill
- Once the tools are picked, it is hard to make a change.
- But also note, you should not throw out the entire chain of tools for some feature that isn't present
- Internal tool was built in 10 hours, that now takes and reads the Maven project file and basically creates CI build tracks
  - Without discarding months of domain expertise in the existing tool chain.
- Counterpoint: Do not latch onto a tool because it exist and is in wide acceptance
- Proof of concept, normally showing lower cost of ownership is a good way to go. Prove the case in correctness, but also man hours saved

Configuration Control Board

- Technique for preventing ad hoc changes to configuration, release processes, tool chains, or software being developed
- Can control policies, practices, procedures in addition to:
  - What tools are acceptable?
  - Who can modify parts of system?
  - Who can modify requirements
  - Who decides what issues get cut or added to released?
  - Who decides what issues get escalated?
  - How are decisions communicated?

Example SCM Policy



- Main code line is pristine.
- Release code lines are Firm.
- General Availability (GA) quality merged into Main
- Development code lines are fluid, high volatility
- Code flows into and out of code line freely

Example SCM Policy

Promotional Model

- Development (DEV) build
  - Owner: Development.
  - Frequency: Daily, By Request.
- Integration (INT) build
  - Owner: Development.
  - Frequency: By Request.
- Quality Assurance (QA) build
  - Owner: Quality Assurance.
  - Frequency: By Request.
- Release Candidate (RC) build
  - Owner: Quality Assurance
  - Frequency: By Request
- General Audience (GA) build
  - Owner: Quality Assurance
  - Frequency: By Request

Example SCM Policy

Code Line Procedures

- Code line owner, curator determines usage policies.
- Configuration Manager oversees and advises owner of best practices
- Code Checkin Policies
  - DEV lines are open and free for checkin early in cycle.
  - Developer is responsible for build breakage
    - Development environment is available to preview and catch bad checkins
  - Softlocking: Late Cycle, Near Milestone, checkin policy determined development lead or code line owner.
  - Hardlocking: CM locks codeline by direction of code line owner.
    - Checkins allowed by committee approval, Change Control Board.
- Defect Tracking
  - Linkage to ExtraView Issue via Perforce Job artifacts

Release Management

- Release Management is proactive technical support focused on the planning and preparation of new services and software
- Some of the benefits are:
  - The opportunity to plan expenditure and resource requirements in advance
  - A structured approach to rolling out all new software or hardware, which are efficient and effective
  - Changes to software are ‘bundled’ together for one release, which minimizes the impact of changes on users
  - Testing before rollout, which minimizes incidents affecting users and requires less reactive support

Example RM Policy

Product Artifacts

- Code line Labels and Tags
  - Each build type results in identification:
    - Schema: ProductName_<Version Details>_BuildType
- Notification
  - Build Results, success or failure, are sent to Engineering teams, determine by code line curator
- Archiving
  - Locally: RAID 5 critical system, and system mirrors
  - Offsite: IronMountain Data Services
- Storage Duration
  - DEV builds 14 days.
  - INT, QA, RC duration of Cycle.
  - GA keep indefinitely.