CS11 – Advanced Java

Winter 2011-2012

Lecture 4

Today’s Topics

- No real programming topics today!
- Project management tools:
  - Automating the build process
  - Source-code management tools
- This week’s focus:
  - Get your project into better structural shape
  - Automate your project’s build process
  - Get your sources into a version-control system

The Build Process

- We have multiple steps for our project now:
  - Compile our code, and our unit-test code
  - Run `javadoc` to generate API documentation
  - Run unit-tests and check the results
- This is a lot of work!
  - Automate this process to make it faster and easier
- Current structure is also pretty messy!
  - Program code and test code are in same directory
  - Libraries and generated `.class` files in there too!

Apache Ant

- Ant is a platform-independent build tool
  - Written entirely in Java
  - Takes a **`build.xml`** file describing build process
  - Pluggable architecture with *many* build tasks
    - Compile Java sources
    - Run **`javadoc`**
    - Run JUnit or TestNG test suites and generate a report
    - Perform code-generation steps (e.g. for J2EE projects)
    - Move/copy/delete files, create/remove directories
    - Send e-mails or other notifications
    - Interact with source-code repositories

Example build.xml File

```
<project name="myproject" default="compile" basedir=".">

  <!-- Global properties used in build -->
  <property name="srcDir"   location="src"  />
  <property name="buildDir" location="build"/>
  <property name="buildClassesDir"
            location="${buildDir}/classes"/>

  <target name="-init">     <!-- Initialization target -->
    <tstamp/>
    <mkdir dir="${buildDir}" />
  </target>

  <target name="compile" depends="-init"
          description="Build the project sources.">
    <mkdir dir="${buildClassesDir}" />
    <javac destdir="${buildClassesDir}">
      <src path="${srcDir}" />
    </javac>
  </target>
</project>
```

Use Ant properties to specify config values in one place.

Targets can specify their dependencies. They specify a series of tasks to complete.

Running Ant

- Ant executable is called **`ant`**
- Just type **`ant`** by itself to build default target
    - **`build.xml`** specifies the default build target
- Specify target(s) to run at command-line

```
    ant clean test doc
```

- Can also specify other options
    - Verbose output: **`-v`** or **`-verbose`**
    - Set Java properties: **`-DpropName=value`**
    - Many more!

Ant Properties

- Properties are simple name-value pairs
  - Both name and value are strings
  - Can be specified at top of project file
  - Can be specified inside a build task
  - Use property’s value by wrapping it in **`${propName}`**
- Example:

```
<property name="buildDir"   value="build" />
<property name="codegenDir" value="${buildDir}/codegen" />
```

- Properties can be set once!
  - If specified again elsewhere, it is silently ignored
  - (run **`ant -verbose`** to see details of when properties are set, and when they are “set” multiple times…)

Ant Properties (2)

- A nifty example:

```
<target name="debug" description="Set up for debug build">
  <property name="java.debug" value="on" />
  <property name="java.opt"   value="off" />
</target>

<target name="release" description="Set up release build">
  <property name="java.debug" value="off" />
  <property name="java.opt"   value="on" />
</target>

<target name="compile" depends="debug">
  <javac debug="${java.debug}" optimize="${java.opt}" ... />
</target>
```

  - By default, compilation will use debug settings.
  - To override at command-line, do this:

```
ant release compile
```

Ant Build Targets

- **`<target>`** tags specify the build targets
  - Each target has a name:

    ```
    <target name="compile">
    ```

  - Targets can also be given descriptions

    ```
    <target name="compile"
            description="Compile the sources!">
    ```

  - Names starting with a hyphen cannot be specified on command-line (“internal use only” tasks)

    ```
    <target name="-init">
    ```

Target Dependencies

- Targets can also have dependencies
  - Ant performs dependency analysis at build time
  - Executes all required tasks, in the proper order

```
<target name="-init" />
<target name="clean" depends="-init" />
<target name="compile" depends="-init" />
<target name="test" depends="compile" />
```

  - You run **`ant test`**
  - Ant executes **`-init`**, **`compile`**, and **`test`** targets in that order

Project Help!

- Don’t know what targets are available?

  ```
  ant -projecthelp
  ```

  - Lists all build targets that have descriptions
  - Also prints out any description you specify at top of **`build.xml`** file
- Example:

```
<project name="paint" default="compile" basedir=".">
  <description>
    A simple program for drawing images.
  </description>
  ...
</project>
```

Project Directory Structure

- So far, everything has been in one directory
  - Project sources, test sources, `.class` files, …
- A *much* better approach: Use different directories
  - Source code stays in its own directory structure
  - Generated files (`.class` files, etc.) go somewhere else!
  - Protects sources from overwriting during build process
  - Makes build cleanup easy: Just blow away the build dir!
- Similarly, separate test sources from project sources
  - Shouldn't be part of final package, so keep them separate
- Any other resources, docs, images, etc. also go in their own directories

Example Project Structure

- `src` Project sources
- `lib` Libraries that your project requires
- `test` Test source-code
- `res` Resources: images, grammars, config, …
- `doc` Design documents, manual (not javadocs)
- `build` Generated results go here
  - `codegen` Generated Java sources (if any)
  - `classes` `.class` files generated by `javac`
  - `javadoc` Generated API documentation
  - `tests` Compiled test classes from `javac`
  - `results` Output logs from running test suite
  - Generated jar file(s) can stay in `build` directory

Ant and Project Structure

- Using our Ant properties, specify relevant directories at top of **`build.xml`** file

```
<property name="srcDir"  location="src" />
<property name="buildDir" location="build"/>
<property name="buildClassesDir"
          location="${buildDir}/classes"/>
```

  - Use earlier Ant properties to specify subdirectory paths

- In Ant targets, refer to relevant directories using properties

```
<target name="compile" depends="debug">
  <mkdir dir="${buildClassesDir}" />
  <javac destdir="${buildClassesDir}"
         classpathref="libs.path">
    <src path="${srcDir}" />
  </javac>
</target>
```

Common Concepts and Types

- Ant has several concepts that most tasks use
- FileSet: a group of files within a directory
  - Specified with **`<fileset>`** element
    - Base-directory of FileSet typically specified with **`dir`** attribute
  - Very sophisticated path-matching capabilities
    - Include or exclude files that match specific patterns
- A very simple example:
  - A file-set of all test sources, except those in the subpackage foo, where the sources aren’t ready yet:

```
<fileset dir="${testSrcDir}">
  <include name="**/Test*.java" />
  <exclude name="**/foo/**" />
</fileset>
```

`**` wildcard matches zero or more directory levels

  - Many Ant tasks can function as a FileSet as well

Common Concepts and Types (2)

- Path-Like Structures:
  - A mechanism for constructing sophisticated classpaths and other sets of paths
  - Frequently used on tasks that compile or run Java code
- Example: classpaths

```
<classpath>
  <pathelement location="${libDir}/foo.jar" />
  <pathelement location="${buildClassesDir}" />
</classpath>
```

  - Can also contain FileSets:

```
<classpath>
  <fileset dir="${libDir}" includes="*.jar" />
</classpath>
```

Common Concepts and Types (3)

- Can also create path-references
  - For defining several paths that refer to each other
- Example:
  - One path for running the project itself, and a separate path for running the project's tests

```
<path id="libs.path">
  <fileset dir="${libDir}" includes="*.jar" />
  ...
</path>

<path id="test.path">
  <path refid="libs.path" />
  ...
</path>
```

  - Tasks refer to these with attributes like:

```
<javac classpathref="test.path" ... >
```

Ant Summary

- Ant is used extensively for many Java projects!
- Many powerful techniques
  - Conditional compilation based on the current OS
  - Ant tasks that are implemented in a scripting language
  - Configuration loaded from properties files
  - Update build numbers and substitute values into code
  - Perform version-control tasks
  - Update website details
  - Perform SSH/FTP tasks
  - …
- http://ant.apache.org

Source Code Management

- You are working on a large software project…
- Problem 1: You break the code
  - Need to roll back to a previous version that works
- Problem 2: Other people also working on project
  - …perhaps on the *exact* same source files
- Problem 3: Centralized source of project info?
  - Maybe a website that shows current test pass-rate, most recent API docs, etc.
- A source code management system can solve all of these problems, and many more

Managing the Source Code

- Basic idea:
  - Store all project files in a repository
  - Repository keeps track of all changes to any file
  - Copies of the project are “checked out” from the repository
  - Developers are isolated from others’ changes
  - Changes to project files are “checked in” or “committed” back to the repository, when ready.
  - Multiple changes to the same file are merged
    - Automatically, if possible; otherwise, manually!

Distributed Version Control

- A new trend in version control systems:
  - Don’t use a central repository server!
- Distributed version control systems
  - Each user has a local repository
  - Users work against their own local repository
    - Check out a working copy, make edits, then check in
  - Users can synchronize with other repositories very easily
- Great for widely distributed software development
  - Open-source software, for example
- Used less often in commercial development teams
  - Software companies prefer to have a single central server
  - Can still use DVCS in a centralized manner, though

Version Control Systems

- Commercial centralized version control systems:
  - Perforce, Visual SourceSafe, BitKeeper, …
- Open-source centralized version control systems:
  - Subversion – written as a replacement for CVS
- Open-source distributed version control systems:
  - Git – written by Linus Torvalds
    - Used for Linux kernel development, Eclipse, PostgreSQL, …
  - Mercurial (`hg`) – distributed VCS written in Python
    - Used by Python project, vim, OpenOffice, GNU Octave, …
  - Bazaar – also written in Python
    - Used by Ubuntu project, GNU Emacs, MySQL, …

Using Subversion

- Two main commands in Subversion:
  - **`svn`**
    - Program used by developers to access the repository
    - Can check out files, check in, move, delete, etc.
  - **`svnadmin`**
    - The repository administration tool
    - Used rarely, by repository administrator
- Both programs take commands
  - Example: **`svn checkout ...`**
  - Both have a help command:
    - **`svn help`** or **`svn help`** ***`command`***

Setting Up a Repository

- Start by creating a repository
  - Repository contains all the config and data files
  - Command:

    ```
    svnadmin create /path/to/repository
    ```
  - Can be an absolute or relative path
- Can create your repository on the CS cluster

  ```
  svnadmin create ~/cs11/advjava/svnrepo
  ```
- Subversion can use different storage layers
  - Filesystem storage, or BerkeleyDB
  - Default is filesystem – use that!

Accessing the Repository

- Subversion uses URLs to refer to repositories
  - Supports access via HTTP, if needed
- For local access, use a **file://** URL
  - On CS cluster: **file:///home/user/cs11/advjava/svnrepo**
- Subversion also supports remote access
  - **svn://…** URL for use of Subversion’s server
  - Or, **svn+ssh://…** URL for accessing over SSH
- For accessing CS cluster repository remotely: **svn+ssh://user@login.cs.caltech.edu/home/user/cs11/advjava/svnrepo**

Importing Source Code

- Need to import initial project source into repository
  - **`svn import`** does this
  - Recursively adds a whole directory tree to repository
- Lay out your repository in a reasonable way
  - Each project (or subproject) should have its own directory
  - Create subdirectories based on good project structure
- For Boggle project:
    - **`boggle/src`**
    - **`boggle/test`**
    - etc. (<u>not</u> **`boggle/build`** !)
- Subversion lets you move files/directories later, too
  - Just in case you make a mistake…

Importing Source Code (2)

- Go to directory with your source files
  - Clean up **`*.class`** files, **`*~`**, etc.
  - Don’t want to import those!
- Import the directory tree into the repository
  - Usually want to specify a subproject to use

    ```
    svn import file:///home/user/cs11/advjava/svnrepo \
      boggle
    ```

  - Subversion will add all files in the local directory (and subdirectories!) into a **`boggle`** subdirectory of your repository

Working On Your Project

- Now, repository is the central store of all versions of all files
  - Can check out any version of any file
  - Usually want the most recent version to work with
- Retrieve a working copy of your project
  - A local copy of a particular version of the files
  - You can make changes in isolation
  - Can periodically sync up with other changes that have occurred
  - Once your local copy works properly, check it in!

Checking Out Files

- To check out files:
  - **`svn checkout`** ***`url`***
  - URL specifies both repository location, and directory within repository
- Example: to get Boggle project from repository:

  ```
  svn checkout \
     file:///home/user/cs11/advjava/svnrepo/boggle
  ```

  - Will create a local directory named **`boggle`**, with project files in it
- To update local working copy:

  ```
  svn update
  ```

  - If performed within working copy, no URL needed!

Working with Local Files

- Can add new files using **`add`** command
  - From within working copy:

    ```
    svn add path1 path2 ...
    ```

  - Can add whole directories
    - Subversion will recurse through directory's contents
- Can delete files using **`delete`** command
  - Again, within working copy:

    ```
    svn delete path1 path2 ...
    ```

- Can move files using **`move`** command

  ```
  svn move frompath topath
  ```

Committing Changes

- Changes to working copy must be committed before they are visible to anyone else
  - Includes add/delete/move operations
- Subversion makes sure your local working copy is up to date first
  - Can’t commit until you have latest version incorporated
- Issue **`commit`** command

  ```
  svn commit
  ```

  - Can specify files to commit, if desired
  - By default, commit operation is *recursive*

Commit Logs

- Subversion will prompt you for a commit log message
  - Describes changes you made in that particular commit
- Always give a descriptive commit message, even for small changes!
  - Other people need to know what you have done
  - You may need to be reminded, too
- Subversion client will start an editor for you
  - Can specify which editor to use with the **`SVN_EDITOR`** (or **`EDITOR`**) environment variable
  - For short messages, use the **`-m`** command-line option to specify the commit message

Discarding Changes

- Use **`svn revert`** to discard local changes
  - Subversion keeps a local copy of original files, so operation doesn’t require actual repository access
  - Can’t actually revert *every* local change (e.g. can’t restore deleted directories)
- Another option:
  - Simply delete working copy and fetch a new one
  - *Does* require repository access, so a little slower than using **`svn revert`**

Repository Code

- Always compile and test your code before checking it in
  - Your mistakes will affect other people badly.
  - Repository version of code should *always* compile, and ideally, work well too.
- Keep your working copy updated with latest version of repository code
  - Avoids big headaches from getting out of sync with other development progress

Subversion Documentation

- Subversion website:
  - http://subversion.tigris.org
- The Subversion Book (very useful!)
  - http://svnbook.red-bean.com
  - Subversion v1.6 available on CS cluster – use version of Subversion Book for that version
- Don’t forget **`svn help`** too!

This Week’s Assignment

- Lay out your project in a cleaner directory structure
- Create an Ant build script for your Boggle program
  - Create tasks for:
    - Cleaning up all build artifacts
    - Compiling your source code
    - Compiling your tests
    - Running your tests
    - Generating Javadoc documentation
- Check your source code (and build script) into a Subversion repository
- No coding for this week! ☺