Build Engineering

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Describe the concepts of automated build systems
- Describe the build lifecycle
- Understand and apply software build best practices
- Understand and use semantic versioning in your own projects

Inspiration

“Nothing resolves design issues like an implementation.” – J. D. Horton

Build Systems

- In this course we will discuss many areas in which we can automate our processes
  - Such automation is much faster than human-in-the-loop processes
  - These tasks can be re-executed on command
  - This reduces human effort and risk of human error
- Today
  - **Project build automation**: Automating the entire compilation, testing, and deployment process.

Build Systems

- Building software, running test cases, and packaging and distributing the executable are very common, effort-intensive tasks.
- Building and deploying the project should be as easy as possible.
- Build systems ease this process by automating as much of it as possible.
  - Repetitive tasks can be automated and run at-will.

Build Systems

- Build systems allow control over code complilation, test execution, executable packaging, and deployment to production.
- Script defines that can be automatically invoked at any time.
- Many frameworks for build scripting
  - Java: Ant, Maven, Gradle
  - Android: Gradle
  - .NET: MSBuild, NAnt
  - Scala: SBT
  - Ruby: Bundler

Build Lifecycle

- **Validate** the project is correct and all necessary information is available
- **Compile** the source code of the project.
- **Test** the compiled source code using a suitable unit testing framework
  - Run **unit tests** against classes and **subsystem integration tests** against groups of classes.
- Take the compiled code and **package** it in its distributable format, such as a JAR, WAR, or Executable file.

Build Lifecycle

- **Verify** – run system tests to ensure quality criteria are met.
  - System tests require a packaged executable.
  - This is also when tests of non-functional criteria like performance are executed
- **Install** the package for use as a dependency in other projects locally.
- **Deploy** the package to the installation environment.

Best Practices

- Automate everything you can!
  - Build tools can integrate with version control, run scripts, send files, zip files, etc.
  - Use them as a comprehensive project management tool.
- Require all team members to use the same tools
  - Even if different team members use different IDEs or workflow, make them use the same build tool to build the project
  - Require a complete build before checking changes into version control
- Provide a “clean” target
  - All build files need the ability to clean up before a fresh build. Clean should only retain the files in VCS.

Best Practices

Design For Maintenance

- Will your build file be readable in the future?
- Will the file execute on a clean machine?
  - Document the build process
    - Write a text file describing the build and deployment process.
    - Utilize a build management system which manages dependencies.
  - Avoid dependencies on programs and libraries not stored in the project
    - If licensing allows, store external libraries with the project for easier builds.
  - Do not distribute usernames/passwords in the build files. These change and **THIS IS BAD SECURITY**.

Semantic Versioning

Version numbers for releases should follow the Semantic Versioning 2.0.0 approach:

- Each version number is specified as: MAJOR.MINOR.PATCH
- We increment:
  1. MAJOR version when you make incompatible API changes
  2. MINOR version when you add functionality in a backwards compatible manner
  3. PATCH version when you make backwards compatible bug fixes
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format

Are there any questions?