Build and Provision: Two Sides of the Coin We Love to Hate

Ed Merks
Eclipse Modeling Project Lead

8/31/2009
The Software Pipeline

• Software artifacts flow from developer to developer and ultimately to the clients in pipeline fashion
  – Each intermediate stage involves provisioning the developer’s environment and building that environment’s contents
  – The final stage involves provisioning the client’s execution environment
• The pipeline is fundamental to the smooth flow of goods from producer to consumer
• All its stage are amenable to automation
Build

• Build is the process of converting source artifacts into target (typically binary) artifacts
• In order to build we must establish the environment in which the build operates, i.e., we must
  – Provision the source artifacts that are to be built
  – Provision the binary artifacts against which to build
• Builds are like the weather
  – We want it to be nice
  – More often than we’d like, it’s unpredictably horrible
  – We love to complain about it
  – But is it really beyond our control?
Provisioning

- Provisioning is the process by which source and binary artifacts are found and retrieved to make them available for local use
- In order to provision we must establish well known locations where the artifacts can be found
  - Source and binary repositories
- Provisioning is an issue both for developers and for the clients of what they produce
The Growth of the Software Pipeline

- Software is becoming increasingly complex
  - Modularity helps manage this complexity
  - But modules increase the length and therefore fragility of the software pipeline
  - So module dependencies must be carefully managed
Release Engineering

• A release engineer is the domain expert who specializes in managing the pipeline
  – It’s a thankless job
  • When things go wrong, it’s all your fault
  • When things go right, you’re completely taken for granted
  • It’s considered menial labor, despite the technical challenge

• The release engineering task is often assigned to junior developers as an entry level task
  • High turnover leads to hacked designs
  • Change is motivated primarily to address symptoms
  • The result is a constant source of recurring problems
Release Engineering Technology

• Release engineering tools and technologies tend to be of stone age quality

• There is a virtual tower of Babel of scripting languages
  – Let’s use an XML syntax and call it Ant, that will solve all the problems!

• A generous helping of spit and glue helps keep it all from falling apart
  – Let’s just poke it here and kick it a few times there and hope it works after that!
  – Better yet, let’s use chewing gum and duct tape instead!
Why Are Builds Always Broken?

- It’s ironic that builds are always broken because developers are doing them constantly all day long just to do their jobs.
- A key problem is that the automated build on the build machine is not the same as the local build done by the developers.
- So there are really two builds
  - One that mostly works because the developers use it constantly and would otherwise sit idle
  - One that always breaks because it’s done in some other apparently more fragile way by someone else.
Why Are Builds So Hard to Set Up?

• Provisioning the development environment is tedious and error prone
  – The more dependencies, the more tedious and error prone the problem becomes
  – Developers must follow a mysterious poorly documented golden path to set up everything correctly
  – It’s so much work they are understandably reluctant to repeat the process as often as they should
    • So they don’t pick up new dependencies regularly
    • And so they don’t notice new upstream problems until long after they’ve been introduced
Why Does the Output of the Software Pipeline Fail to Function Properly?

• It’s caused by yet another provisioning problem
  – In a highly modular system, it’s difficult to ensure that all the appropriate versions of the modules are installed
  – Installing a new module can break other modules
  – The environment itself often has differences, e.g., an inappropriate JVM
Problems, Problems, and More Problems

• How do I reproduce a build?
  – In open source to be truly open conducive to participation and contribution it’s important that anyone can reproduce your builds

• Provisioning is a moving target
  – The things we need constantly change, including their dependencies

• How do we solve these problems?
  – Let’s make someone else responsible
  – Let’s steer clear of the whole mess
  – No! Take responsibility!!
Build Solutions

- Declarative data is the key
  - Describe what’s in a module
  - Describe how modules depend on other modules
  - Describe what needs to be built rather than detail how to build it
  - Drive the build process directly from the description

- Eclipse declares what’s needed with
  - MANIFEST.MF
  - plugin.xml/feature.xml
  - build.properties

- Much of this information is needed at runtime as well as at build time
Provision Solutions

• Declarative data is the key
  – Describe what’s available
  – Describe dependencies between them
  – Describe where these things are located

• Eclipse describes such things today with
  – Project set files for source artifacts
  – Installable units (p2) for binary artifacts
  – Buckminster provides a more complete provisioning solution that builds on these
Equinox and OSGi

- Equinox is Eclipse’s implementation of an OSGi runtime
  - A container for bundles with well defined dependencies
- Bundles
  - The fundamental modular unit of OSGi
  - Contains code
  - Describes dependencies on bundles or packages
- Plug-ins
  - Eclipse bundles that can define extension points or contribute extensions to extension points
- Features
  - A grouping of plugins that are expected to be installed together
- Fragments
  - A special type of bundle that augments some host plug-in
- Product
  - A top level grouping mechanism
p2 Installable Units

- Eclipse’s provisioning technology
- Installable Units are the central concept
  - Identified by namespace and version
p2 Repositories

- Metadata Repositories
  - Information about the installable units available
- Artifact Repositories
  - The actual contents of the installable units
- Typically they are collocated at a given URI
p2 Profiles

- Installable units are fetched from repositories and installed into a profile
  - Maintains history of what’s been installed
  - Supports rollback
p2 in Action

Metadata fetched and constraints analyzed
Provisioning operation requested
IU install, uninstall, update operations
IUs configured into runtimes
Runtimes

Data transfer
Http/Https
File system
Volume

Transports

Mirroring
Artifact availability and mirroring
Profile updated
Profiles

Repositories
Update Site

Engine
Eclipse/OSGi
Native/OS

Director

Profiles

Runtimes

8/31/2009 © Ed Merks | EDL V1.0
Eclipse’s Development Environment

• **Workspace**
  – A container for a set of projects, i.e., source plug-ins, source features, and so on
  – Typically provisioned from one or more team project set files that extract the source from a repository

• **Target Platform**
  – A set of plug-ins and features used when building projects in the workspace and when running the build result in launched processes
Eclipse’s Java-centric tools for developing plug-ins/bundles

- Manage the classpath based on MANIFEST.MF dependencies
- Editing support for MANIFEST.MF, plugin.xml, build.properties
- Generate build.xml ANT scripts for driving headless PDE build
Buckminster

• Automated provisioning of Eclipse installations, workspaces, and target platforms
• Exploits workspace build directly even for headless/automated builds
• Exploits knowledge about dependencies between modular units to ensure that the entire dependency closure is materialized
Buckminster Repository Support
• A proposed incubator project focused on simplifying the build and assembly process

• A declarative model-based approach for build definition and execution that unifies developer builds and automated builds
  – Clear separation between description and execution
  – Direct execution of build models

• Expand upon PDE’s declarative data and Buckminster’s provisioning support
The Eclipse Developer

- Developer
- Workspace
- Test
- Target
- Process
- Check-out
- Commit
- Provision
- Src
- Bin
The Eclipse Software Pipeline

- Process
- Check-out workspace
- Target
- Test
- Headless
- User
- Install
- Provision
- Aggregate
- Src
- Bin

8/31/2009  © Ed Merks | EDL V1.0
Conclusion: Reuse Technology

- Reuse the information already captured to describe the contents and dependencies of modular units for use in a componentized runtime to also drive a streamlined, robust software build and assembly pipeline
- Reuse modeling technology to describe declaratively all aspects of the build process
- Reuse the same build infrastructure exercised continuously by the developers to drive the headless automated build
- Reuse provisioning technology to drive both development and deployment
References