The presentation abstract:

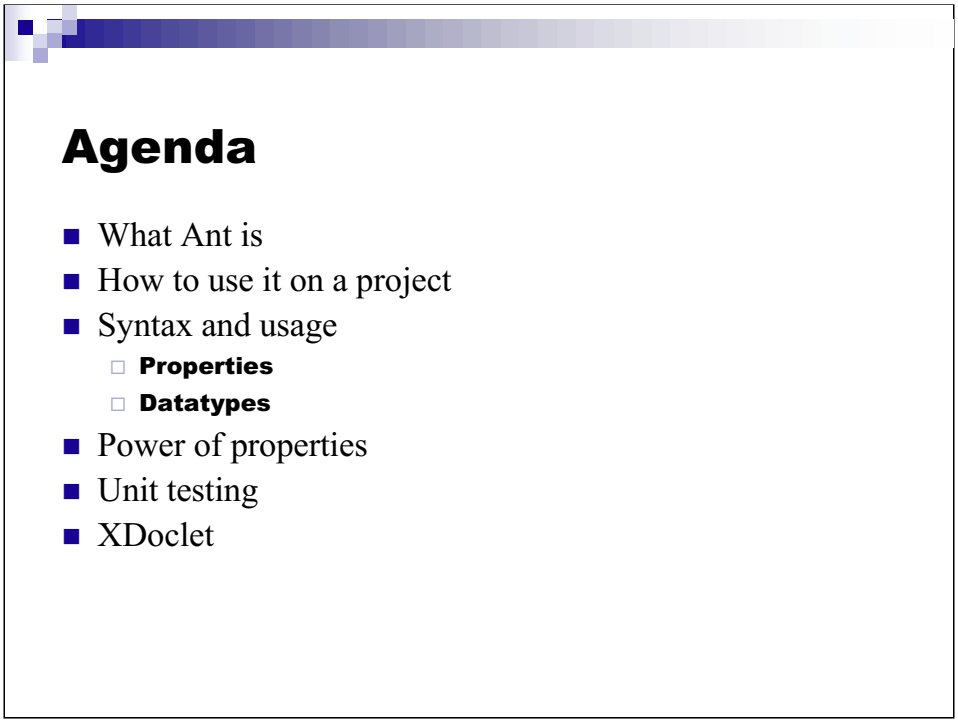
In the Java development world, there is no better example of doing more with less than Ant. It's the Swiss Army knife of Java build tools. Ant has emerged as the de-facto standard Java build tool. All modern Java IDEs integrate with it, most open-source Java projects use it. If you are developing Java software and **not** using Ant, then chances are you're doing things the hard way. It's relatively easy to craft an Ant build file by cutting and pasting pieces from other build files, but we don't want to get into a situation where maintaining the build process is a full-time job. It is important to understand Ant's capabilities in order to avoid hacking at build files. By understanding Ant's basic data types, syntax, and properties and applying some simple best practice techniques, the build process can be easily controlled, extended, and reused. Maintenance of Ant build files is minimal if crafted appropriately - lets learn how!

This presentation will cover:

Who am I?

- Co-author, Java Development with Ant
- jGuru Ant FAQ Maintainer 

- Ant Committer
- XDoclet Committer
- *<insert title here>*, eHatcher Solutions, Inc.



Ant is an Apache Jakarta project.

Unit testing will cover Junit testing and in-container Cactus testing.

XDoclet will cover half of the presentation even though its only one bullet item here.

Introduction to Ant

- What is Ant?
  - **Java-based build tool**
- Why use Ant?
  - **Cross-platform**
  - **Java domain smart**
  - **Fast, extensible, integrated**
  - **Alternatives?**
- Analogy
  - **Factory automation**

Ant is cross-platform in a couple of ways. It runs on all platforms that have a JVM, and path/directory separators can be specified either Windows or Unix format.

Fast - Ant's dependency checking ensures the quickest route to the goal is taken and no more work than necessary is done. Compilation, for example, using <javac> compares date stamps of .class and .java files and only compiles .java that are newer.

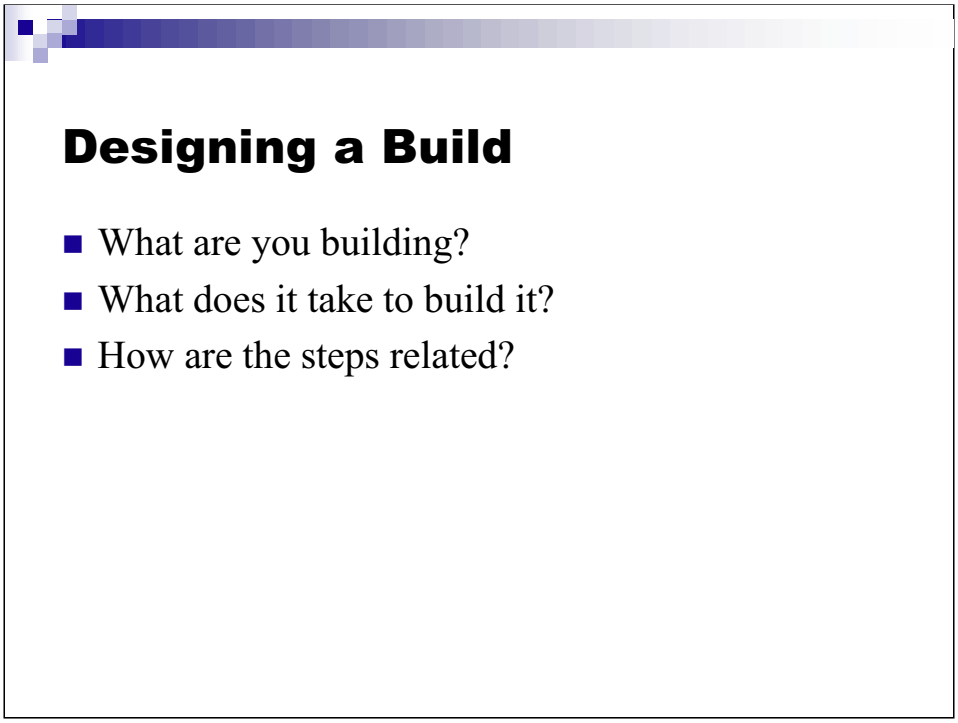
Extensible - Easily add third-party or custom tasks and datatypes. Or custom logging/listening of the build process can be done with simple Java coding. Several other more advanced ways of extensibility as well.

Integrated - every modern IDE has Ant integration built-in.

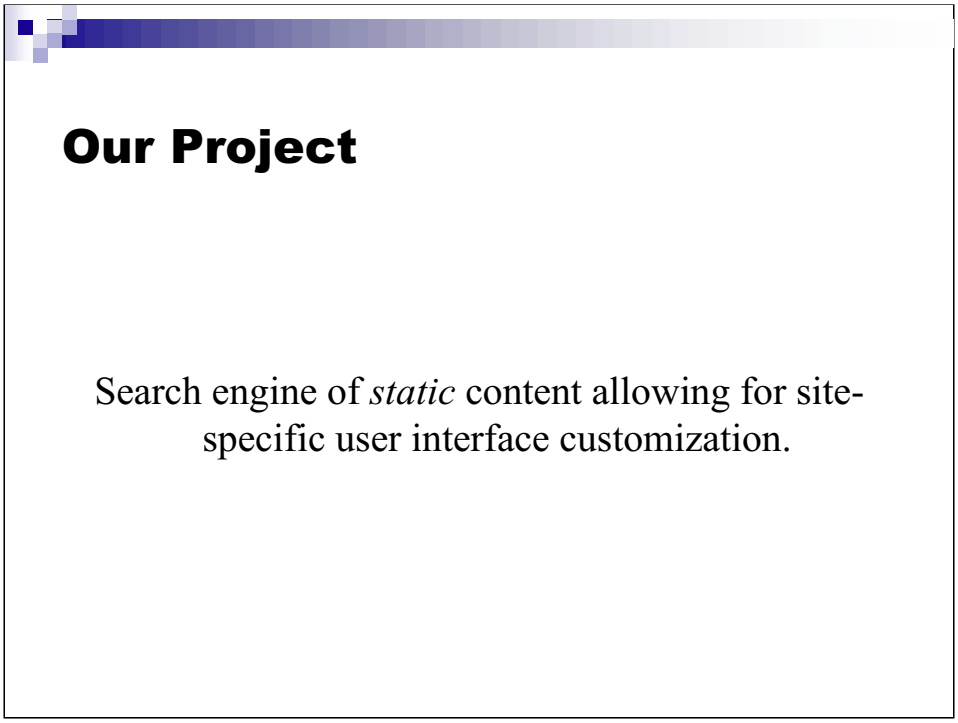
Alternatives - make, but its not even really competition in the Java world because of the features of Ant that make it so much more suitable to Java development. Make differs in dependency checking - it is solelybased on file timestamp checking, whereas Ant has a lot of other mechanisms such as remote server timestamp checking with <ftp>.

Factory automation - view your source code as the raw materials going in one end of the factory, and the end product out the other. All of the tasks done to

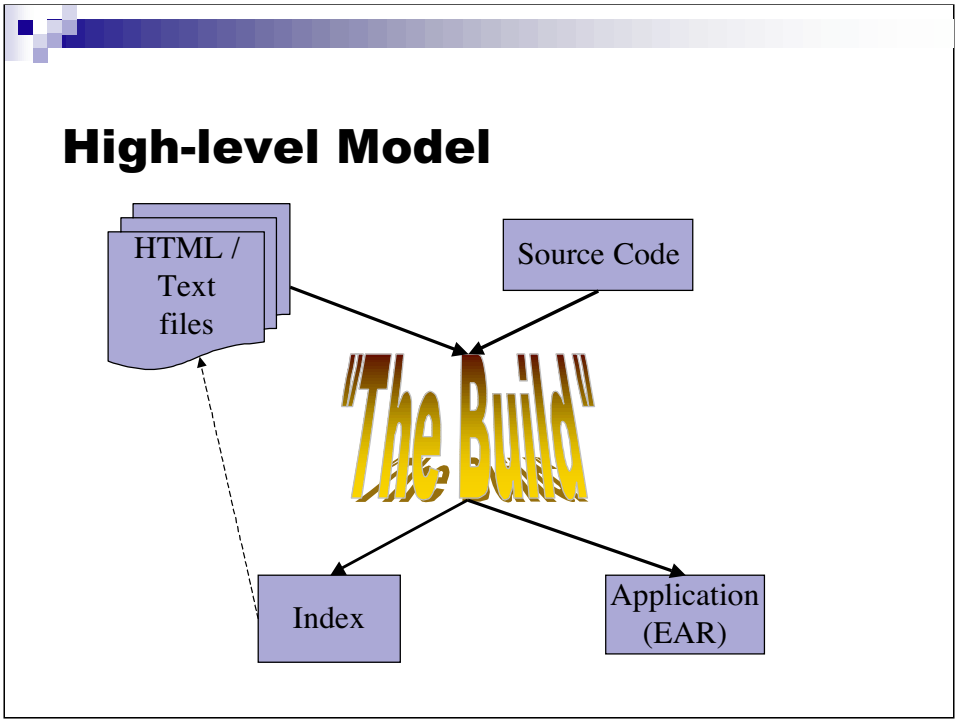
Ant can literally build *anything* desired, not just for Java projects. Using XSL transformations during a build or accessing a database can all be a part of an Ant build process. The typical Java project will at a minimum build .class files, and then create a JAR of them. Web applications are bundled into WAR files and enterprise applications possibly consisting of many web applications are bundled into EAR files. Ant has facilities and specific handling of each of these and much more.



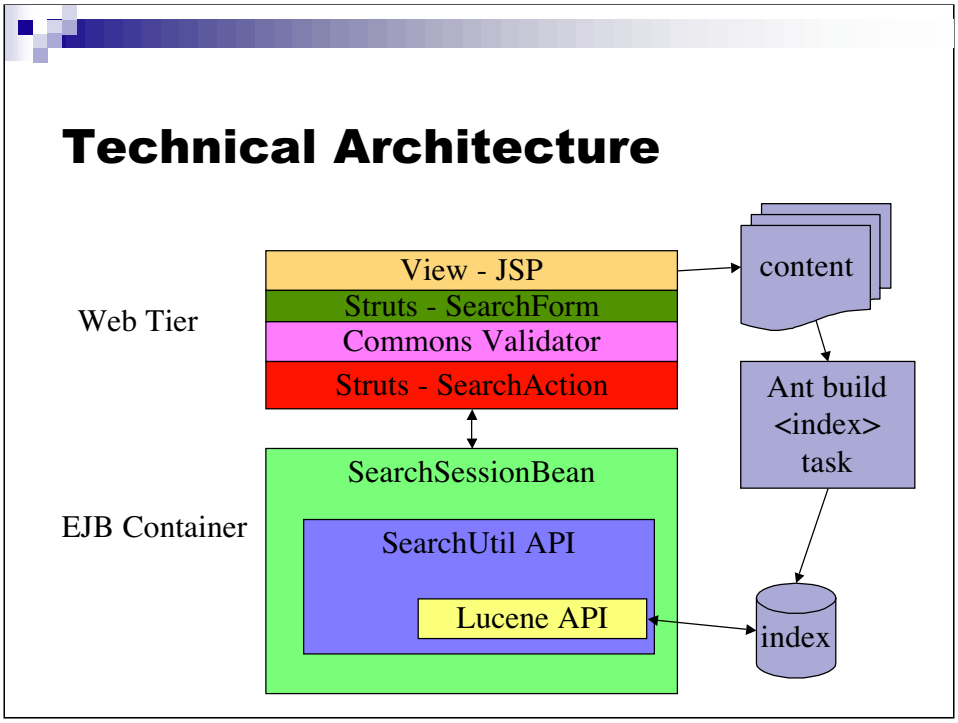
Before delving into the gory syntax details of Ant, first things first. Why have a build process? The end result of our work needs to be considered first and foremost. What is our projects overall objective? Then we step back and figure out what the right tools are. Ant is the right tool for getting all Java projects built. Start with the end result, likely a WAR or EAR, and decide what steps are needed to get it built. How are those steps related to one another? Some steps must come before others and a prescribed order of events is necessary.



So, lets frame this Ant presentation in the context of a real-world example project that we can wrap our heads around and explore how Ant makes life better. The application - a search engine for static content such as (Ant) documentation or a personal scrapbook.



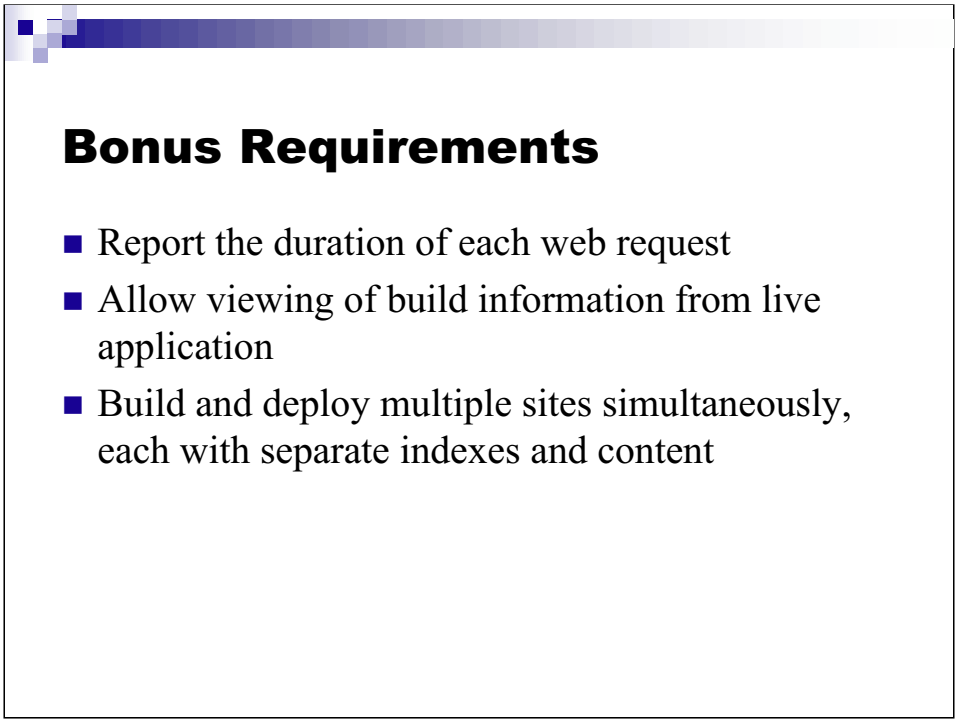
In the big sense, here is the idea of this project. Documentation files are indexed by the build process, source code is compiled, and the application EAR file is built.



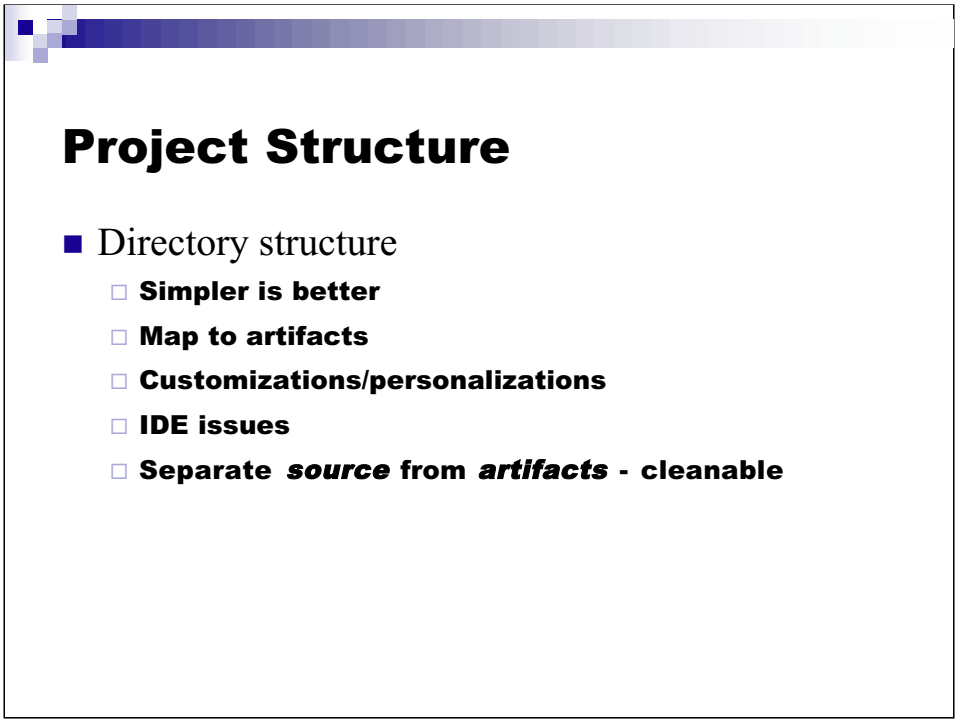
The application is divided into several layers. At the lowest layer, the Lucene API is used for both indexing and searching. The build process indexes the content, and the application accesses the index through a custom wrapper API. The searching API is accessed through a stateless session bean to provide a possible security access layer. At the presentation tier, a very simple Struts configuration is used along with Commons Validator for form field validation at both the client and server sides.

This architecture is relevant to our discussion because of the tricks we'll make Ant do related to our needs.

Note that index and content are separate from the actual application engine - this makes good architectural sense as the content and index can be updated while the site is still live.



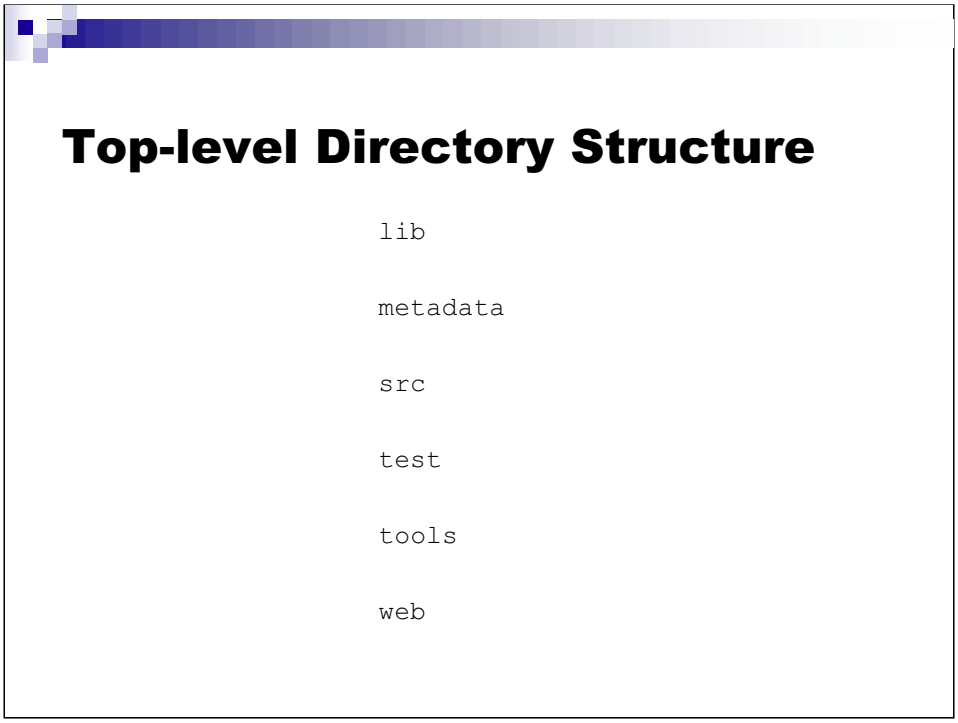
These bonus requirements factor into some of the custom XDoclet generation that will take place and taking advantage of Ant properties.



Now that we've analyzed what we want built and how we'll build it, its time to start coding. But wait, where do we place our source code? Project directory is a very important consideration. Ant is able to handle very bizarre project structure, but it is most maintainable and sensical if things are arranged cleanly. Look to many of the open-source projects to get some ideas on what works well. There is not one best way to do it, but these are some rules of thumb to follow.

For example, in this project, we have a common library which builds into a standalone JAR. There is an EJB JAR, a web application (WAR) and then finally all of it builds into a single EAR.

By choosing the proper structure, customizations such as building with new library versions or site personalizations such as skinning sites uniquely for each customer are made possible.



This project is built upon several third-part librarys (Struts, Lucene, Jtidy, etc) and the default versions of them live in a lib subdirectory. Later you will see that its not just JAR's crammed into a single directory, its much more organized and structured to facilitate the extensive control we'll have on the build process.

Metadata is where deployment descriptor information lives, separated by module underneath.

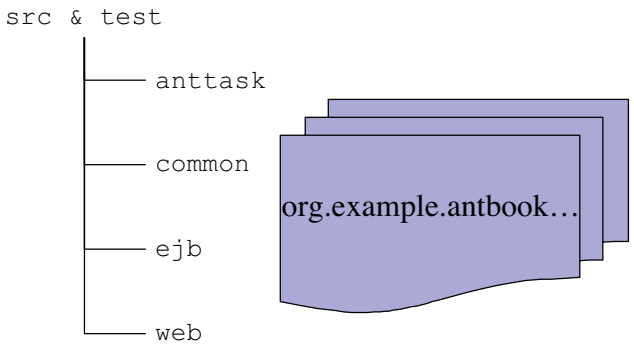
Src - this is our Java source code directory, with subdirectories underneath for each major artifact.

Test - this mirrors our src tree with unit test Java code and even test data.

Tools - Ant is even often maintained in source control systems to precisely version a system and allow it to be rebuilt with the exact version of Ant if needed. Other code generators live here.

Web - this is where JSP's live.

Java Directory Structure



Again, the src and test trees mirror one another and are separated into component trees each being the package root.

Build File

- XML format
- Typically in project root directory
- Default name: build.xml
- Declarative - define steps, not scripty details
- Defines a single *project*
- A project contains *targets*
- Targets contain *tasks*

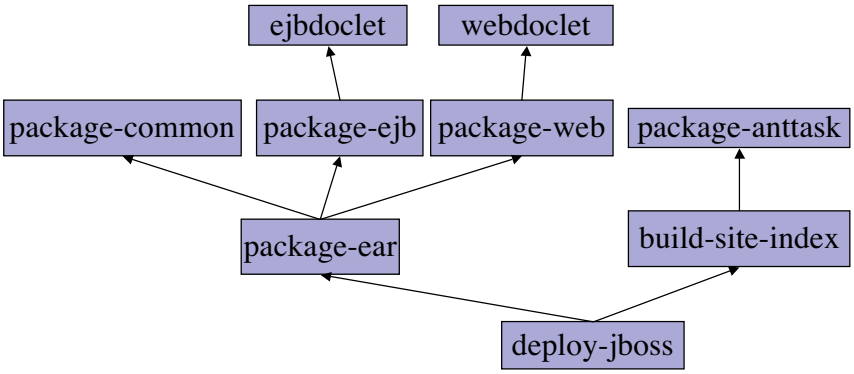
XML format means that entity references and well-formedness apply.

If desired, multiple build files can be created to separate concerns - for example, I typically have a build.xml for building local code, but a cvs.xml to deal with checking the code out for auomated builds, which in turn calls build.xml for building the freshly checked out code.

Declarative - its not a scripting language - the steps are spelled out at a higher level typically.

Targets typically correspond to the artifacts generated, or the individual steps that do work to create the artifacts. Tasks are the lowest level of work - these are the cogs in the machines of our factory automation analogy.

Dependencies



Targets are generally correspond to outputs, or stages.

Our end result is to deploy to an application server for local development. Given the architecture of our system, building the EAR is independent of indexing the content, and this shows in our build dependency graph.

Ejbdoclet and webdoclet are XDoclet targets for code/descriptor generation.

build.xml - Targets

```
<?xml version="1.0" ?>
<project name="antbook" default="default">
    …

    <target name="init">…</target>

    <target name="clean" description="Removes build artifacts">
      …
    </target>

    <target name="package-ear"
            depends="package-common,package-ejb,package-web"
            description="Package EAR">
      …
    </target>


    <target name="deploy-jboss"
            depends="package-ear,build-site-index"
            description="Deploy to local JBoss">
      …
    </target>

    <target name="default" depends="package-ear"/>

</project>
```

Here we see an Ant build file skeleton of the previously shown dependency graph. Of note, the default target, which I recommend to be called "default" for ease of use from the command-line. "clean" and "init" are targets you'll find in most build files. "init" is depended upon at the lowest level targets to do things like store the timestamp and create temporary build directories.

Our default target does not deploy. This is intentional - it just builds, making it friendly for environments where an app. server is not installed and local deployment is not the goal.

Tasks

- Do the work
- Handle dependency checking internally
- Parameterizable
- Commonly used tasks include:
  - `<mkdir>`
  - `<javac>`
  - `<jar>`

Dependency checking by tasks is what separates make from Ant. Make is responsible for dependency checking itself, whereas Ant pushes that responsibility lower into the tasks. Tasks have domain specific attributes allowing them to be parameterized.

<mkdir> makes a directory

<javac> compiles Java code

<jar> builds a JAR file

Running Ant targets

- Execute default target:

```
ant
```

- Execute specific target:

```
ant deploy-jboss
```

- Execute multiple targets:

```
ant test-common generate-test-reports
```

Executing multiple targets on the command-line processes the dependency graph for each target invoked, which can be somewhat unexpected.

Command-line switches

- -projecthelp

```
> ant -projecthelp
Buildfile: build.xml
Main targets:

 checkstyle              Check code style
 clean                   Removes build artifacts
 compile-anttask         Compile anttask module
 compile-common          Compile common module
 compile-web             Compile web module
 deploy-jboss            Deploy to local JBoss
 ejbdoclet               Generate EJB code and descriptors
 generate-test-reports   Generate test reports
 package-ear             Package EAR
 package-ejb             Package EJB JAR
 package-web             Package WAR
 test-anttask            Test anttask module
 test-common             Test common module
 webdoclet               Generate web and Struts descriptors

Default target: default
```

- -verbose / -debug
- -find <*file*>
- -diagnostics
- -logfile <*file*>
- -help
- *... and others*

Document your targets to have -projecthelp output meaningful and useful.

-verbose for troubleshooting, typically. -debug for much more output.

-find - climbs a directory tree looking for the build file specified, and invokes it. This is useful when at a command prompt deep into your project directory tree.

-logfile - captures build output to a log file.

-help - displays command-line syntax

-diagnostics - displays versions of libraries being used and any potential missing tasks (likely to happene for home-built versions of Ant).

Sample Ant Run

```
> ant deploy-jboss -Dsite=personal
Buildfile: build.xml

init:
     [echo] Building for personal site.

…

deploy-jboss:
     [copy] Copying 1 file to
/Users/erik/jboss-3.0.0/server/default/deploy

BUILD SUCCESSFUL
Total time: 18 seconds
```

Build time is an “incremental” build, not clean build of this system.

Note, the -Dsite=personal is a segue into the next section on Ant properties.

Properties

- Ant's parameterizability
- Immutable
- All JVM system properties available
- Useful Ant built-in properties:
  - **ant.project.name**
  - **basedir**
- ${property.name} for property value

Immutable is the keyword here. Its sometimes frustration for new Ant users, but its actually one of the powerful features of Ant, allowing great flexibility in control.

ant.project.name is the <project name="…."> value. basedir is the base directory of the project, full path. Typically this is the directory where build.xml lives, but not necessarily.

<property> task

- Name/value

```
<property name="site" value="sample"/>
```

- Load from properties file

```
<property
        file="sites/${site}/config.properties"
        prefix="site"/>
```

- Load environment variables

```
<property environment="env"/>
```

Simple case is setting a property simply as a value.

Multiple properties can be loaded at once using the 'file' variant, and optionally all properties can be prefixed (note that a "." character is appended to the suffix to separate it from the property names in the properties file).

Environment variables are very important pieces of information, with "env" being the recommended prefix - again a "." is appended giving properties such as env.JBOSS_HOME

Properties set from the command-line (using -D or -propertyfile) are set before the build file is processed, and hence take precedence.

Property for file/directory

- Use the location variant:

```
<property name="build.dir" location="build"/>


<property name="index.dir"
          location="${build.dir}/${site}/index"
/>
```

This Is on a separate slide because its so crucially important to emphasize. Properties representing files or directories should be set using the 'location' variant of the <property> task. The value of the property is resolved to the full path name (from the basedir, if path specified is relative).

This comes into play mostly with larger builds that delegate work to sub-builds and override paths in the process.

Properties Quiz

build.properties
javac.debug=off

```
<property file="build.properties"/>
<property name="javac.debug" value="on"/>
<echo message="debug compilation is turned ${javac.debug}"/>
```

```
[echo] debug compilation is turned ????
```

Remember the most important fact about properties? They are _____ (look back a few slides for a reminder!).

Other Property setting tasks

- <available> - sets if resource present
- <uptodate> - sets if source files older
- <condition> - O/S, checksum, http, socket, files match, and more.
- <tstamp>
- <xmlproperty>

<tstamp> sets a property or properties to date/time stamps, by default setting TODAY, DSTAMP, and TSTAMP.

<xmlproperty> loads properties and names them hierarchically (with dotted syntax) from XML files. This is moderately useful.

<javac>

- Facade over Java compilers
- Performs .java/.class out-of-date checking
- Example of why the docs are important!

```
<javac
   srcdir="src/${module};${additional.src.dirs}"
   destdir="${build.dir}/${module}/classes"
   debug="${javac.debug}"
   classpathref="compile.classpath"
/>
```

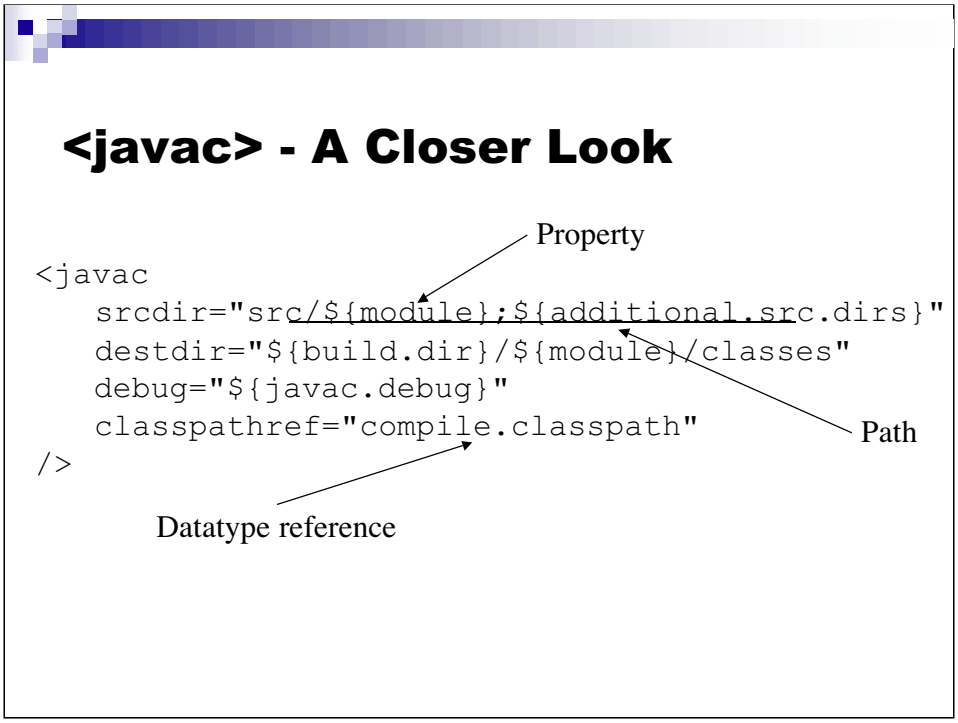
Attributes are associated with a type, not just a raw string. Docs are important because it spells out the types. For example, <javac>'s attributes shown are of these types:

Destdir - file

Debug - boolean

Srcdir - path

Classpath - by datatype reference



To start our discussion of Ant's datatypes, lets use <javac> as an example. Properties are used as simple text replacements. The srcdir attribute corresponds to the "path" datatype. The classpathref attribute is a reference to a previously defined path datatype.

For paths, note that Ant is happy with forward-slashes, semi-colons, colons, and backslashes, regardless of the operating system.

Ant Datatypes

- Provides domain expertise (e.g. paths)
- Reusable
- Commonly used built-in datatypes:
  - **Path**
  - **Fileset**
  - **Patternset**
  - **Selectors**
  - **Filterset**
  - **and others...**

Ant's domain expertise is with concepts such as "paths", "file sets" and other commonly used abstractions in build processes.

Datataypes, particularly paths, are well suited for reuse. Define them once, and reuse them wherever needed.

Path

- Ordered list of path elements
- Analogous to Java CLASSPATH
- Cross-platform path syntax
- Contains files and/or directories
- May contain a fileset, individual elements, or nested paths.

```
<path id="anttask.compile.classpath">
    <pathelement location="${lucene.jar}"/>
    <pathelement location="${jtidy.jar}"/>
</path>
```

Again, cross-platform syntax is fully supported. Note the use of Ant properties in defining this example path. This combination provides a lot of control and benefit, allong the JAR file names to be swapped easily as we'll see soon.

Fileset

- Set of files from a single root directory
- Unordered
- Patternsets / Selectors filter files
- Many special files automatically excluded

```
<fileset dir="web" excludes="search.jsp"/>

<fileset dir="${build.dir}/${site}" includes="*.jsp"/>
```

The awkward note about filesets is that they must be rooted from a specified base directory. Selectors provide incredible selection capabilities, far beyond the simple patternsets we see here.

By default (unless specified otherwise with defaultexcludes="off"), CVS directories and their files are automatically excluded. Visual SourceSafe temporary files, IDE backup files, etc are automatically excluded.

Patternset

- Collection of file matching patterns
- Relative paths, can apply to any root directory
- Exclusions take precedence
- <include>/<exclude> conditional-capable

```
<patternset id="java.files.pattern" includes="**/*.java"/>
<fileset dir="src">
    <patternset refid="java.files.pattern"/>
</fileset>
```

Patternsets group inclusion and exclusion patterns, with exclusions taking precedence.

In this example, the id attribute demonstrates patternset reuse in the <fileset>.

** represents recursive directories.

Selectors

- New to Ant 1.5
- Sophisticated file selection capability
- Available selectors: depth, size, date, present, depend, contains
- Containers: and, or, not, none, majority

```
<copy todir="newfiles" includeemptydirs="false">
  <fileset dir="web">
    <not>
      <present targetdir="currentfiles"/>
    </not>
  </fileset>
</copy>
```

Details of the mentioned selectors:

Depth - allows selecting files only at a certain directory depth from the root directory.

Size - select all files of 1MB or greater, for example.

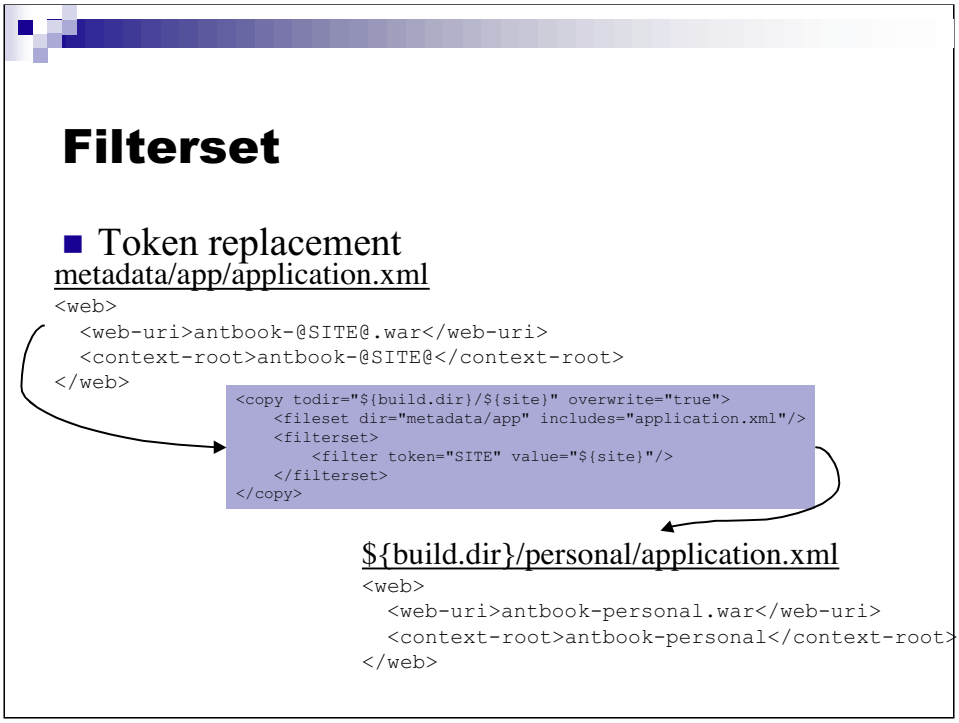
Date - select files modified more than a month ago, for example.

Present - example shown, along with <not> container - all files in the web directory that are not also in the currentfiles directory are copied to the newfiles directory.

Depend - does file date dependency checking, selecting all files that are out of date, for example.

Contains - select files that contains a certain string

Combine any of the above inside <and>/<or>/<not> type logic for even more sophisticated selections.



Filtersets are great for simple templated files, such as this deployment descriptor being parameterized with the <copy> operation shown. Tokens are replaced, and token delimiters can be specified, with "@" delimiters being the default.

FilterChain / FilterReader

- FilterReader: remove/modify text
- FilterChain: ordered group of FilterReaders
- Built-in FilterReaders:
  - **<classconstants>**
  - **<headfilter> / <tailfilter>**
  - **<linecontains> / <linecontainsregexp>**
  - **<tabstospaces>**

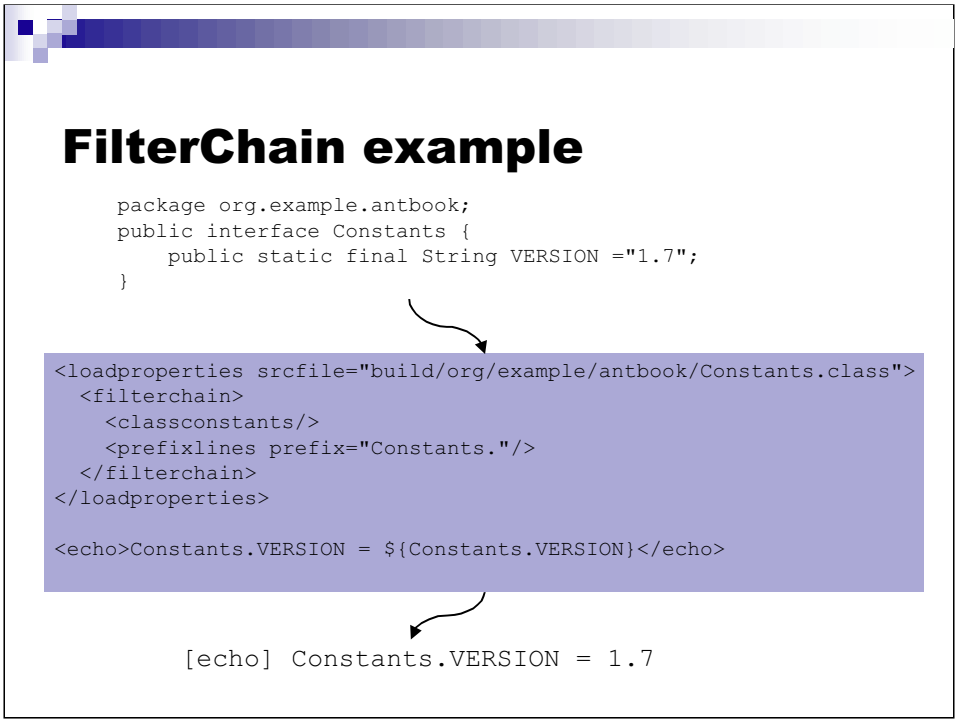
FilterChain's are like the piping of output of one command into the input of another, with each command filtering or altering the data as it goes through.

<classconstants> - example on next slide.

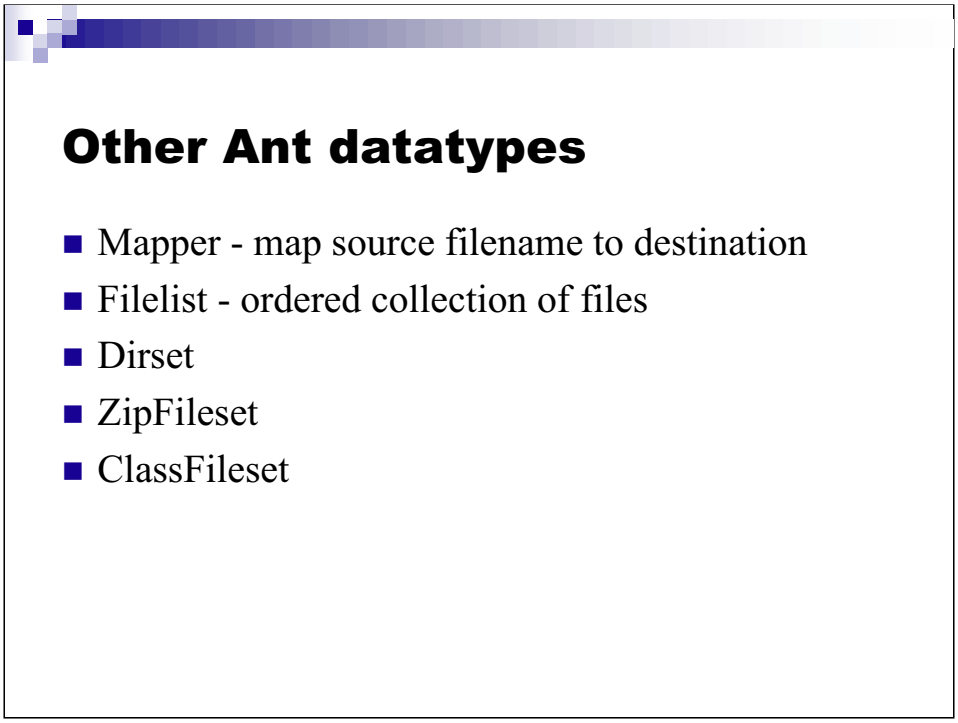
<headfilter>/<tailfilter> - just like the head and tail commands in Unix, outputs the first or last N lines of a file.

<linecontains(regexp)> - filter lines that contain a string or match a regular expression.

<tabstospaces> - just don't run it on your Make files! :)



The <classconstants> FilterReader is a special one in that it reads a .class file (using BCEL). It pulls out any public static final values and outputs them as name/value pairs, in this case it outputs VERSION=1.7. The <prefixlines> then prefixes every line with “Constants.” yielding the output shown.



And there are many other datatypes that can come in handy depending on your needs.

Mapper - rename files during a copy operation appending the .bak to each file, for example, and much more.

Filelist - sometimes order matters, such as with the <concat> concatenate task.

Dirset - same as a fileset, except only contains directories, not files.

Zipfileset - allows an archive to be constructed with the contents of other archives without having to unzip them first. Also allows prefixing of a fileset within an archive to construct the internal structure as desired.

Classfileset - Constructs a set of minimal files that a specified class depends upon by analyzing the class internals (using BCEL).

Overriding properties

- Carefully order <property>

```
<property
        file="${user.home}/.${ant.project.name}-build.properties"/>
<property file="${user.home}/.build.properties"/>

<property file="build.properties"/>

<property environment="env"/>
<property name="jboss.home" location="${env.JBOSS_HOME}"/>
<property name="env.COMPUTERNAME" value="${env.HOSTNAME}"/>

<property name="lib.dir" location="lib"/>
<property file="${lib.dir}/lib.properties"/>
<property file="common.properties"/>
```

- Command-line -D and -propertyfile switches

```
ant deploy-jboss -Dsite=personal

ant -propertyfile nightly-build.properties
```

Not really “override”, but rather “setting first” - but that’s the typical term for it.

Carefully ordering property setting, fine-tuned control can be had. For example, two properties files are initially loaded from the user.home directory (a property provided by the JVM). The first property file is specific to the current project based on the project name, and the second is a general user-specific property file potentially for all projects. This allows users to have their own preferences for a project without having to modify the main build file.

Later, environment variables are loaded. Because Unix and Windows different on the variable used to store the local machine name, a trick is used taking advantage of immutability. On a Unix machine, HOSTNAME is loaded as env.HOSTNAME, and then its set to a property env.COMPUTERNAME (which would not exist on a Unix machine). From then on, ${env.COMPUTERNAME} contains the local machine name for either platform.

Properties can be set on a per-build basis on the command-line using -D or -propertyfile.

Controlling with Properties

- Existence of property is IMPORTANT

```
<junit…
    <batchtest todir="${test.dir}/data" if="testcase">
        <fileset dir="${test.dir}/${module}/classes”
                 includes="**/${testcase}.class"/>
    </batchtest>
    <batchtest todir="${test.dir}/data" unless="testcase">
        <fileset dir="${test.dir}/${module}/classes”
                 includes="**/*Test.class"/>
    </batchtest>
</junit>
```

- Conditional:
  - **<target>**
  - **Patternset <include>/<exclude> elements**
  - **<fail>**

```
<fail unless="site.config.present">
    Site configuration for ${site} not present.
</fail>
```

Properties are either set or not set, if set they have a value. Whether they are set or not is as important as the value of a property. Conditions if and unless take the existence of a property into account, and can be specified at the target and patternset levels, and also on the <fail> task and on <batchtest> on the <junit> task for flexible control over how a build operates.

Datatype References

```
<path id="web.compile.classpath">
    <pathelement location="${dist.dir}/antbook-common.jar"/>
    <pathelement location="${dist.dir}/antbook-ejb.jar"/>
    <pathelement location="${struts.jar}"/>
    <pathelement location="${oro.jar}"/>
    <pathelement location="${commons-digester.jar}"/>
    <pathelement location="${commons-fileupload.jar}"/>
    <pathelement location="${commons-lang.jar}"/>
    <pathelement location="${commons-resources.jar}"/>
    <pathelement location="${commons-validator.jar}"/>
    <pathelement location="${j2ee.jar}"/>
</path>

<path id="web.test.classpath">
    <pathelement location="${junit.jar}"/>
    <path refid="web.compile.classpath"/>
</path>
```

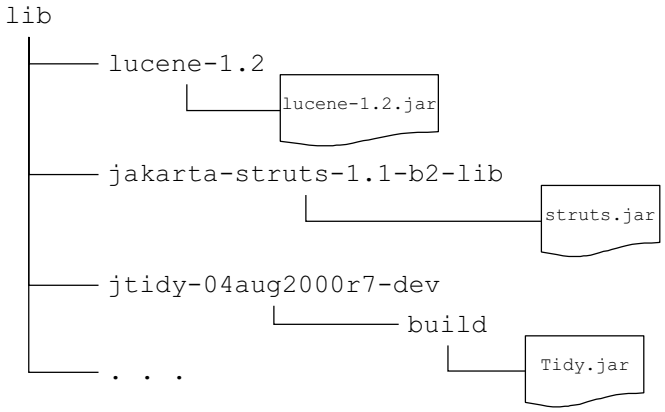
Here is an example of classpath definitions, with the second one taking advantage of reuse by incorporating the first one and adding more to it. Again note the extensive use of Ant properties to represent JAR files - making it easy to “override” their settings.

Library Dependencies

- Easily allow:
  - **Multiple projects to use different library versions**
  - **A developer to build against a new version without affecting the team or local environment**
- Property immutability is the key!

This is an example of the control properties can give to a project. For example, in this project it would be great if upgrading to a new version of Struts or Lucene is trivial to do, and has almost no maintenance involved. This technique accomplishes this.

lib Directory Structure



Under the lib directory of this project, each third-party dependency is extracted as-is from its binary distribution - quirky directory structure and all. Note that Lucene has the version number in both the base directory and its JAR file - we retain that. Jtidy has an intermediate directory. Keeping the directory structure of the libraries in tact allow us to easily replace versions (as long as they don't change the structure!).

lib.properties

lib/lib.properties - Global default library mappings

```
#
# Lucene - http://jakarta.apache.org/lucene
#
lucene.version     = 1.2
lucene.dir=${lib.dir}/lucene-${lucene.version}
lucene.jar=${lucene.dir}/lucene-${lucene.version}.jar

#
# Struts - http://jakarta.apache.org/struts
#
struts.version     = 1.1-b2
struts.dir=${lib.dir}/jakarta-struts-${struts.version}-lib
struts.jar=${struts.dir}/struts.jar
```

Each library has a .version property associated with it, and a .dir property pointing to the base directory of its installation. Each library has its own property mapped as well. The next slilde will show how these can be used to control things tightly.

Swapping Libraries

- Per-build
  ```
  ant -Dlucene.jar=/path/to/mylucene.jar
  ```
- Per-project
  - **Create build.properties with overrides:**
    ```
    lucene.version=1.3
    ```
- Per-developer
  - **Create ~/.antbook-build.properties:**
    ```
    struts.dir=/Users/erik/struts-dev
    site=personal
    ```

From the command-line, a new library version can be tried to ensure it works and all tests pass.

Per-project - perhaps other projects in your system need to depend on different versions of the same library? Easy, use a project-specific build.properties to specify the version. A library repository can be maintained in the source code repository allowing projects to use a common set of libraries with the default versions always being the latest stable versions.

Developers have control over any property without affecting other developers using home directory property files.

Also note that any of the properties (.version, .dir, and .jar) can be overridden allowing for very specific control.

Site Customization

- Controlled by ${site} property
- Artifacts unique to site:
  - **Index and content (of course)**
  - **application.xml (different WAR file names)**
  - **web.xml (different index directory references)**
  - **jboss.xml (different JNDI names for session bean)**
  - **search.jsp (results presentation differs)**

This application uses a ${site} property throughout to customize its output based on site-specific settings and to separate it from other sites being built simultaneously.

Loading site properties

```
<!-- Site configuration -->
<property name="site" value="sample"/>

<property name="site.dir" location="sites/${site}"/>

<property name="index.dir"
          location="${build.dir}/${site}/index"/>

<property file="${site.dir}/config.properties"
          prefix="site"/>
```

sites/sample/config.properties

```
search.link=<a href="file://<bean:write name="item" property="field(path)"/>">
           <bean:write name="item" property="field(title)"/></a>
```

Site-specific config.properties are loaded, with the prefix “site.”. These properties affect the presentation making it custom per-site.

Declarative 'logic'

- Per-site customizations happen without any *explicit* "logic"
- search.jsp customization:

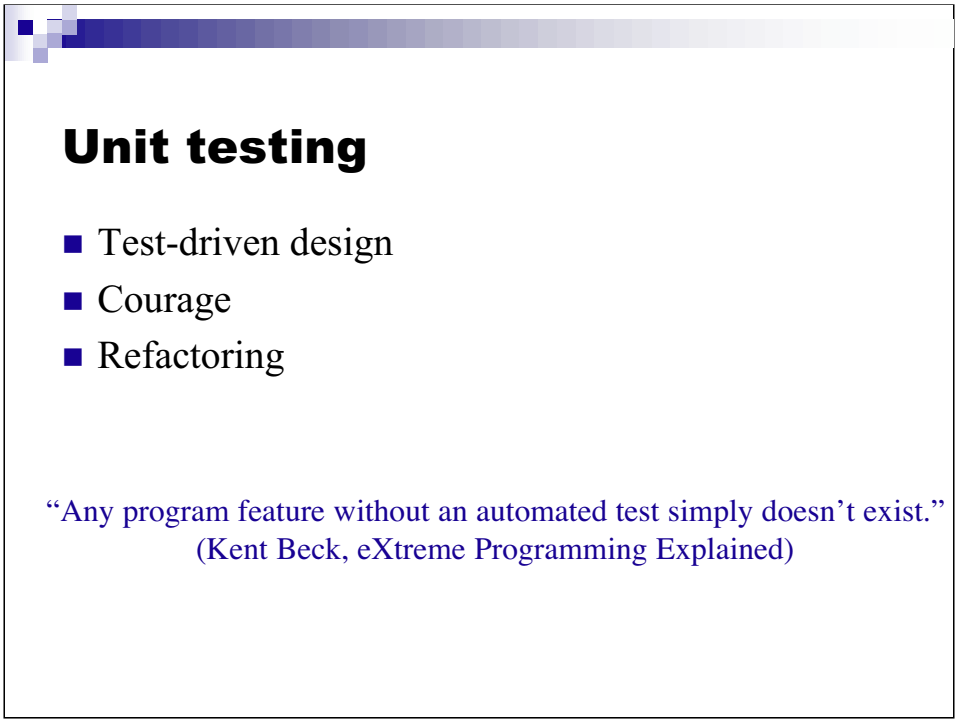
```
<copy todir="${build.dir}/${site}" overwrite="true">
    <fileset dir="web" includes="search.jsp"/>
    <filterset>
       <filter token="LINK"
               value="${site.search.link}"/>
    </filterset>
</copy>
```

- WAR and EAR are named by ${site}:

```
<war destfile="${dist.dir}/antbook-${site}.war"…>
```

Because we load a properties file named by another property (${site.dir}/config.properties) we've implicitly done some logic by "switching" on the site name. This is still declarative (no if statements or branching).

The presentation tier is customized per site using a filtered copy, and the WAR file generated is uniquely named by the site name.

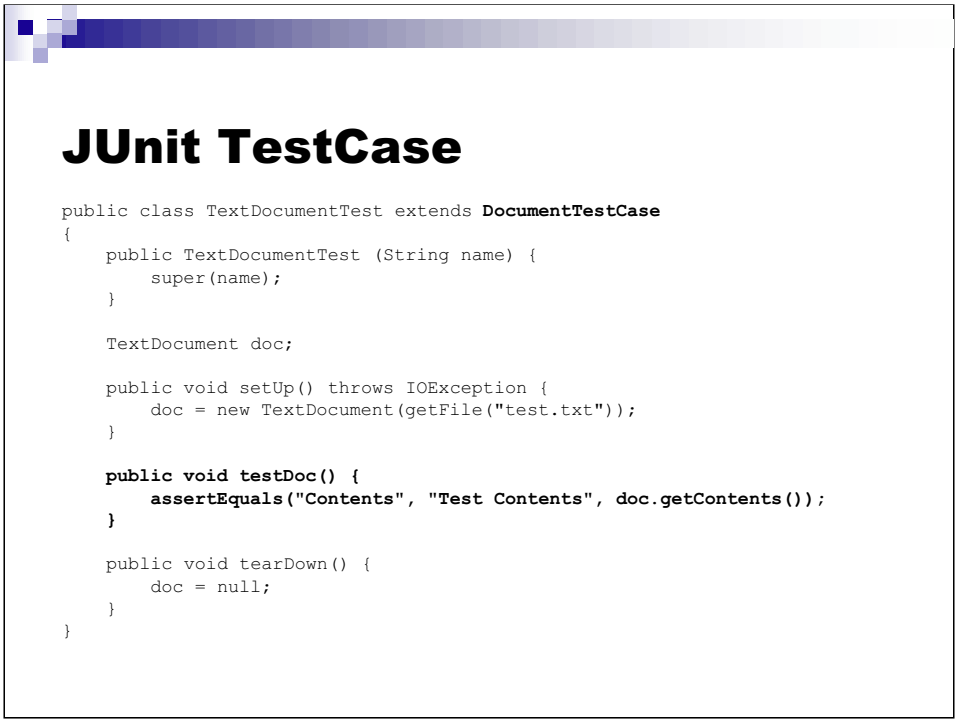


This cannot be emphasized enough - testing is very very important for agile/extreme approaches. Focus on quality workmanship. Ant provides the capability to automate tests, with particularly nice handling of Junit test cases.

With sufficient tests in place, you can have the courage to modify any code in the system (collective ownership) without worrying about breaking something unknowingly - the tests will prove things are ok.

Refactoring is only possible with good tests in place, allowing the code base to stay clean at all times rather than accumulating legacy cruft from developers that are not empowered or courageous enough to modify the code extensively.

Writing tests first is a great way to prove out your API and hone its usage before writing the details. Tests fail at first and then gradually pass as things are fleshed out. This builds confidence.



JUnit's main class to be extended is TestCase. In this example we have a custom subclass that provides a document for testing its parsing. There are a similar and more complicated HtmlDocument and HtmlDocumentTest classes.

<junit> task

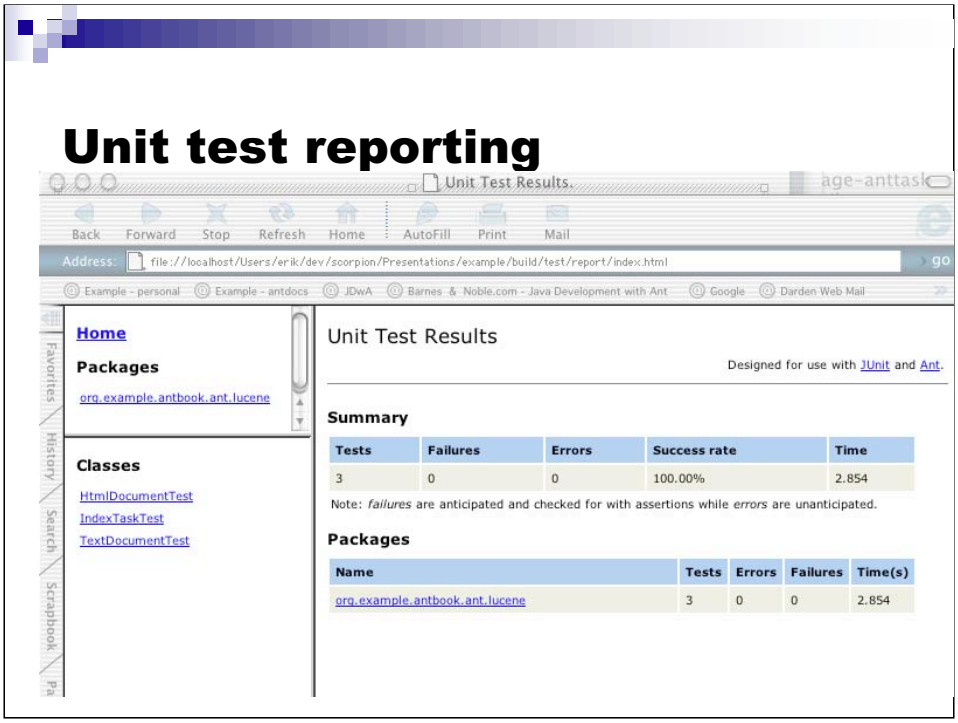
```
 <copy todir="${test.dir}/${module}/classes">
   <fileset dir="test/${module}" excludes="**/*.java"/>
 </copy>

<junit printsummary="no" errorProperty="test.failed"
       failureProperty="test.failed" fork="${junit.fork}">
<!-- . . . -->
  <sysproperty key="docs.dir" file="${test.dir}/${module}/classes"/>
  <sysproperty key="index.dir" file="${test.dir}/index"/>

  <formatter type="xml"/>
  <formatter type="brief" usefile="false"/>

  <batchtest todir="${test.dir}/data">
    <fileset dir="${test.dir}/${module}/classes"
             includes="**/*Test.class"
    />
  </batchtest>
</junit>

<fail if="test.failed">Unit tests failed.</fail>
```

These tests are run using the <junit> task. Things to note - by default <junit> does not fail the build if tests fail. This is a bad default, but its actually more preferred to not fail immediately and have a chance to generate reports or send an e-mail if tests fail.

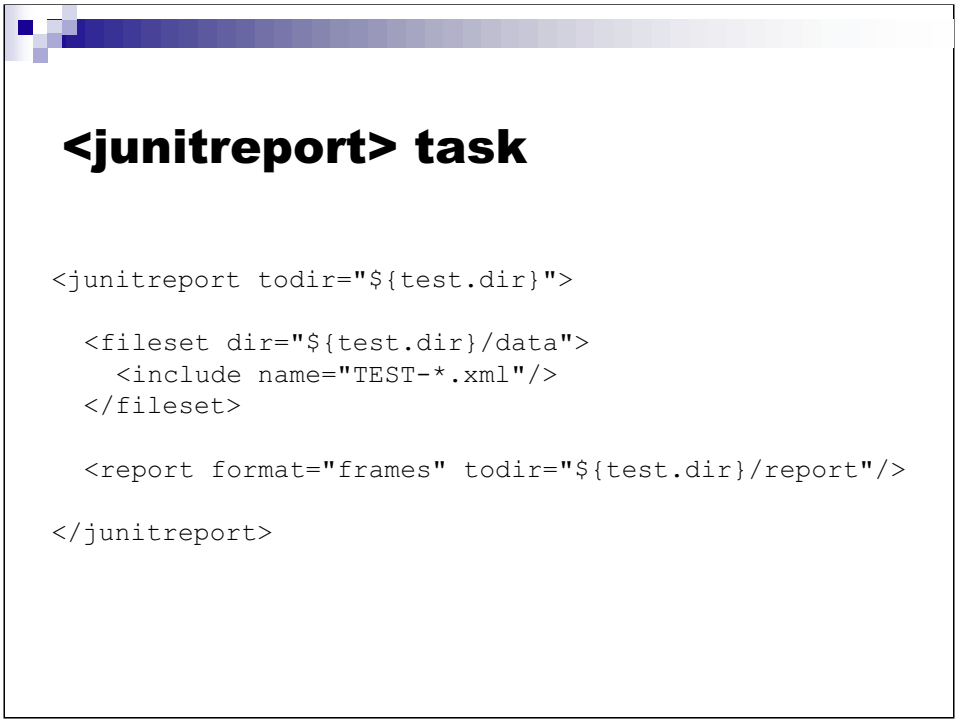
System properties are passed to tests using <sysproperty>, and my DocumentTestCase base class uses docs.dir to access test data files.

Formatters are used to display or capture test results. Brief results are displayed, and complete results captured to XML files.

Naming conventions should be used (**/*Test.java for source code files) to facilitate the running of only concrete tests.



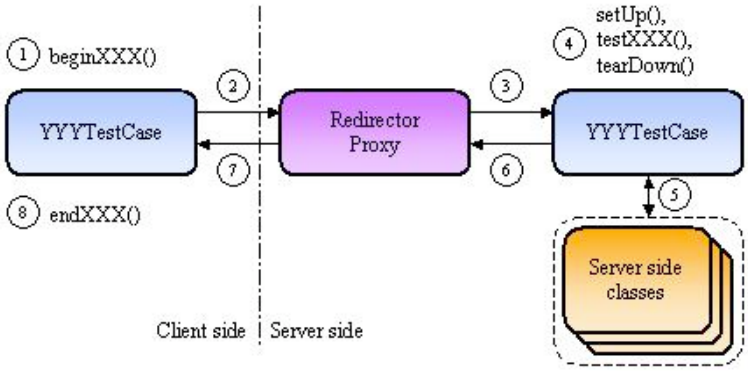
This report generated from the XML captured data. Note the details provided and the easy navigation to the details.



Here is how the report is generated with Ant. This is done using XSLT, and the output can be customized if desired. Because this can add time, its not necessary to generate reports every time you run tests. Depending on your needs, you can pull this into a separate target and only run as needed. The brief console output is usually sufficient for troubleshooting issues.

Cactus

- In-container unit testing



By far one of the best documented Jakarta projects. Here's to Vincent Massol - my hero!

This diagram describes the Cactus test case lifecycle. It adds begin/end methods on the client-side. On the server side the tests run as plain JUnit tests. The client receives and records the result just as if it had run locally.

Cactus example

```
public class SearchFormTest extends CactusStrutsTestCase {
    public SearchFormTest(String s) {
        super(s);
    }

    public void testValidation() {
        addRequestParameter("query","");
        setRequestPathInfo("/search");
        actionPerform();
        verifyActionErrors(new String[] {"query.required"});
        verifyInputForward();
    }
}
```

This test verifies that the Struts validation is working properly for the search form. The query field is required, but it is not being set on the request and thus a validation error occurs and is verified. The CactusStrutsTestCase is actually a base class from the Sourceforge StrutsTestCase project and provides nice Struts-specific capabilities such as verifying errors.

What else?

- Running Java / native programs
- Remote deployment
- Native development
- Custom Ant development
- Maven & Centipede
- Continuous integration
- Last but not least…..XDoclet

whew - this is about the halfway point. The rest of the presentation will cover XDoclet but to whet your appetite for Ant more, here are some other features available.

Ant can launch Java and native programs easily, even allowing Ant to do dependency checking on a set of files and launch a native program for each file (or passing all non-up-to-date files on one command-line).

Remote deployment is possible using <ftp> and <telnet>.

Native development can be done by either launching make through <exec> or using the new <cpp> tasks being developed at the ant-contrib project on Sourceforge.

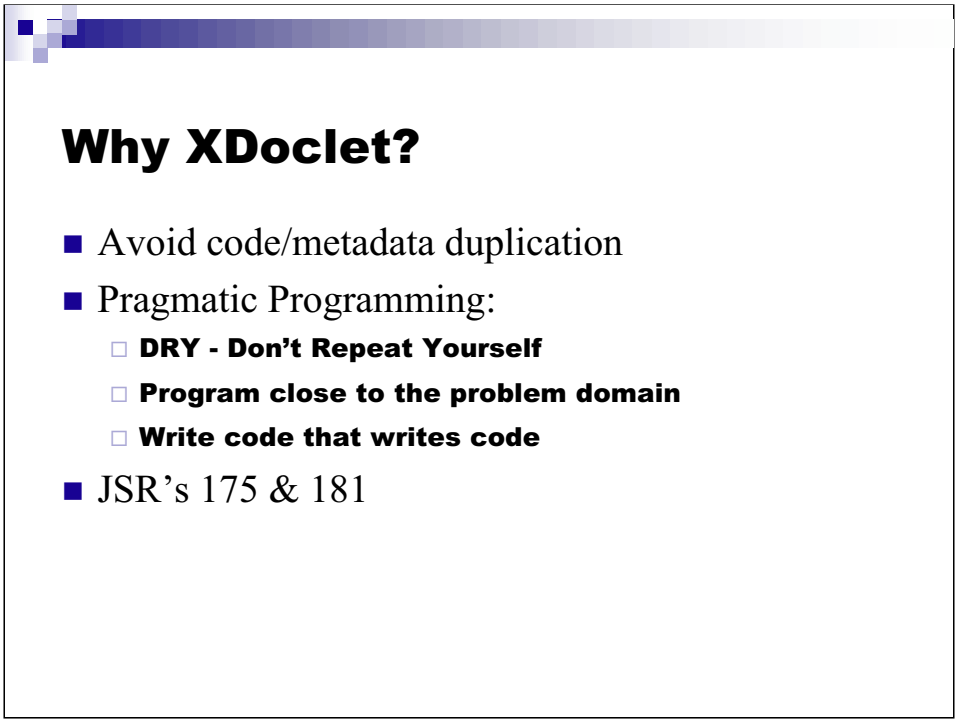
Developing custom tasks is easy (you just need an execute() method, actually), and Ant has other extensibilities such as custom loggers/listeners, mappers, filters, and selectors.

Maven and Centipede are “meta” Ant projects that wrap Ant inside a cleaner project descriptor, taking care of many of the routine gory details automatically such as implicitly providing compilation, archiving, documenting, and even CVS change log and unit testing features.

What is XDoclet?

- Javadoc metadata templating engine
- Attribute-oriented programming
- Outgrown its EJBDoclet roots

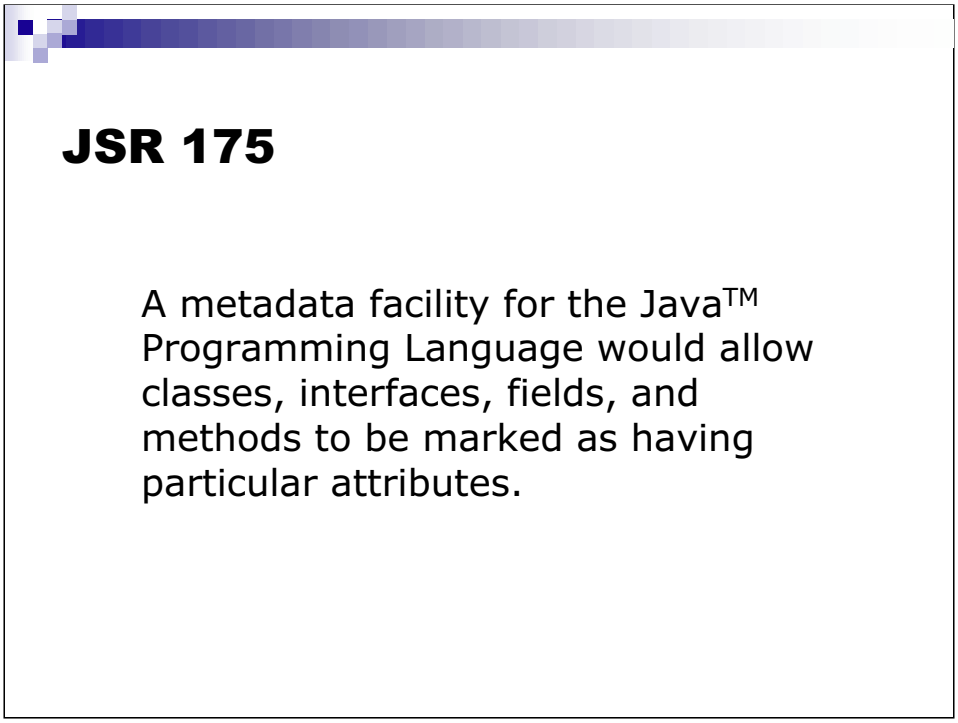
It reads Javadoc tags, either custom or standard, and passes a model of all source code processed to a templating engine. It was designed first for EJB, which has a lot of duplication involved, but its usefulness soon was seen in other areas of code/descriptor generation.



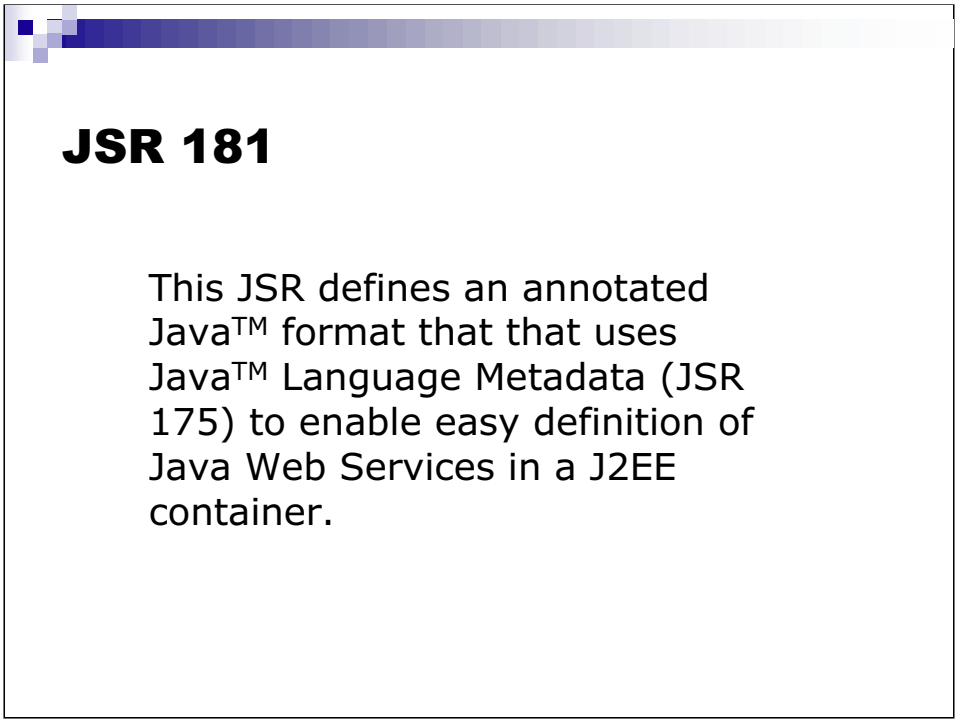
EJB is ugly, real ugly. But XDoclet makes it "easy" by generating all the ugly pieces while you write the one piece of real code that you need to write. Lots of other J2EE deployment descriptors and code can be easily generated, making maintenance headaches much less.

The Pragmatic Programmer book (Dave Thomas, Andy Hunt) espouses several great practices, such as Don't Repeat Yourself (every piece of knowledge must have a single unambiguous authoritative representation), Program close to the problem domain, and write code that writes code. XDoclet is all of those.

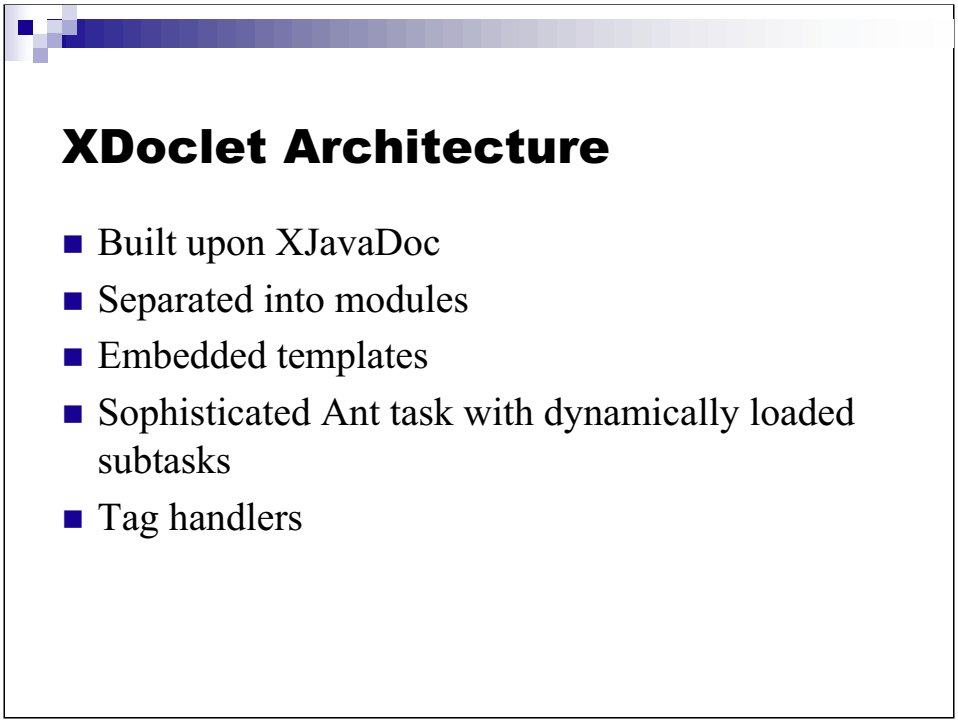
Sun is also embracing this paradigm of metadata coding, details to follow.



This JSR essentially spells out what XDoclet already is, although integrated with the language much tighter (ala C#'s metadata capabilities, perhaps).



And leveraging the previous JSR, web services will automatically supported using this metadata facility. Again, like .NET's capabilities to expose any method as a web service. Scary, but true.



XJavaDoc will be discussed on the next slide - it's the parser and model.

XDoclet has a pluggable architecture allowing additions to be dropped in and auto-discovered. Modules exist for many vendors and frameworks already.

The templates that it uses are built into the JAR files for each module, and can be copied and customized (if needed) easily.

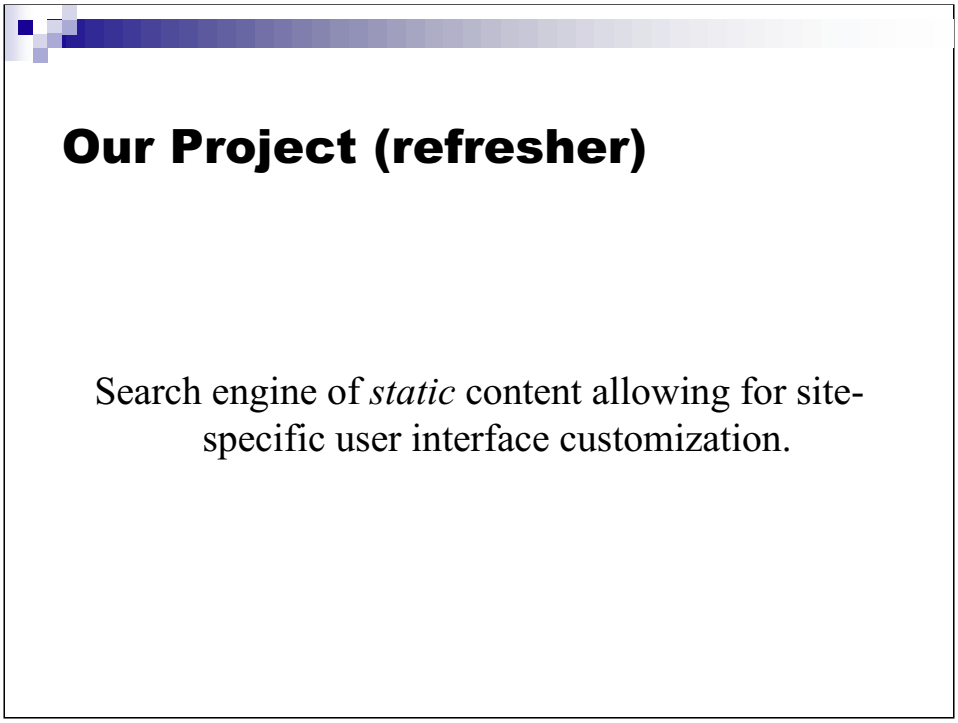
Currently XDoclet (version 1.2) only is runnable through Ant and it has excellent Ant integration and runs *fast* with good dependency checking.

A rich set of "tag handlers" (very much like JSP tag libraries) exist making writing templates quite easy. XDoclet's documentation is very detailed and well organized.

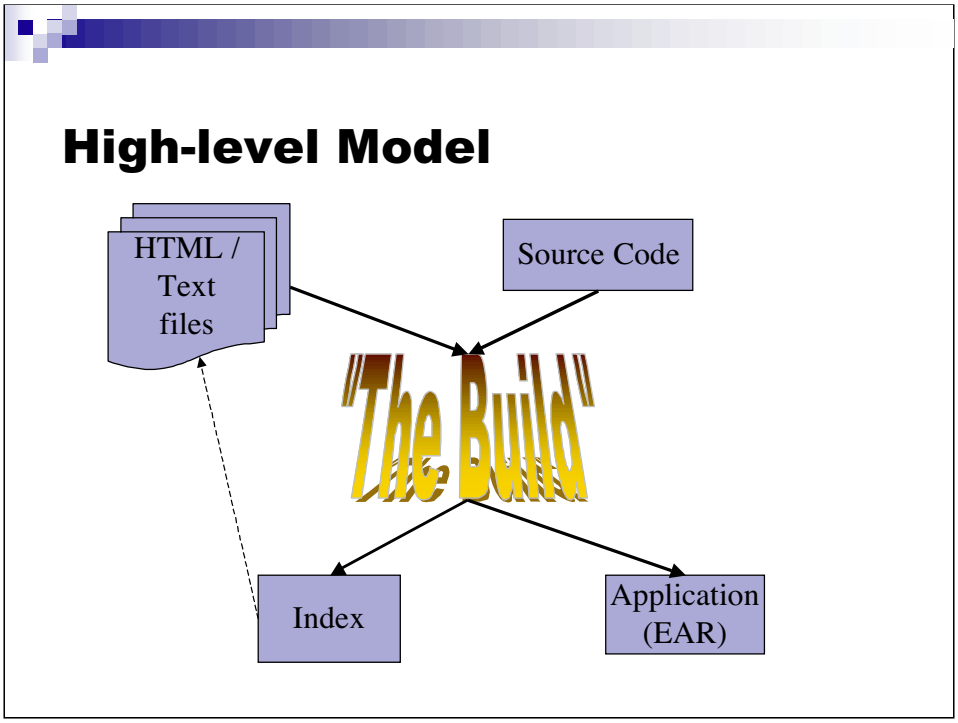
XJavaDoc

- JavaCC-based source code parser
- Builds model of:
  - **Packages**
  - **Classes / inheritance hierarchy**
  - **Methods and parameters**
  - **Member variables**
  - **Constructors**
  - **@tags**
  - **.· 'oh yeah, and Javadoc comments too**

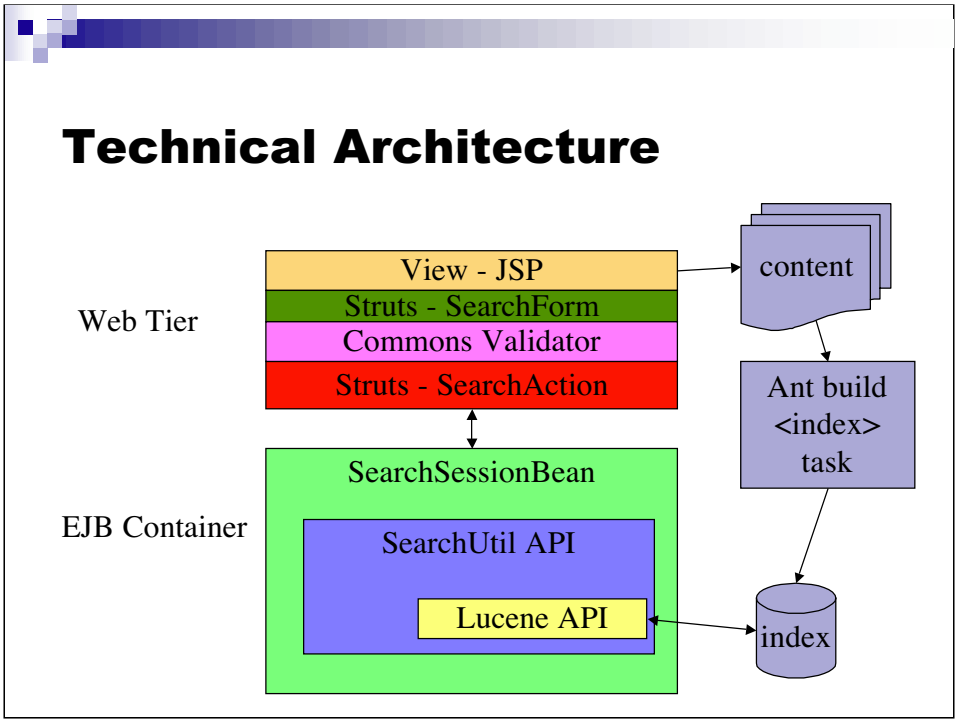
A very fast parser of Java source code was created by the XDoclet wizards - much faster than Sun's javadoc tool. An object model of all source files processed is built, and contains all of the pieces shown here. Templates have access to the entire model.



Just a reminder, quickly, of the sample project we're still working on. This is even more relevant for the XDoclet section because of the massive work it will do for us even in a tiny project.



Again, quick reminder.

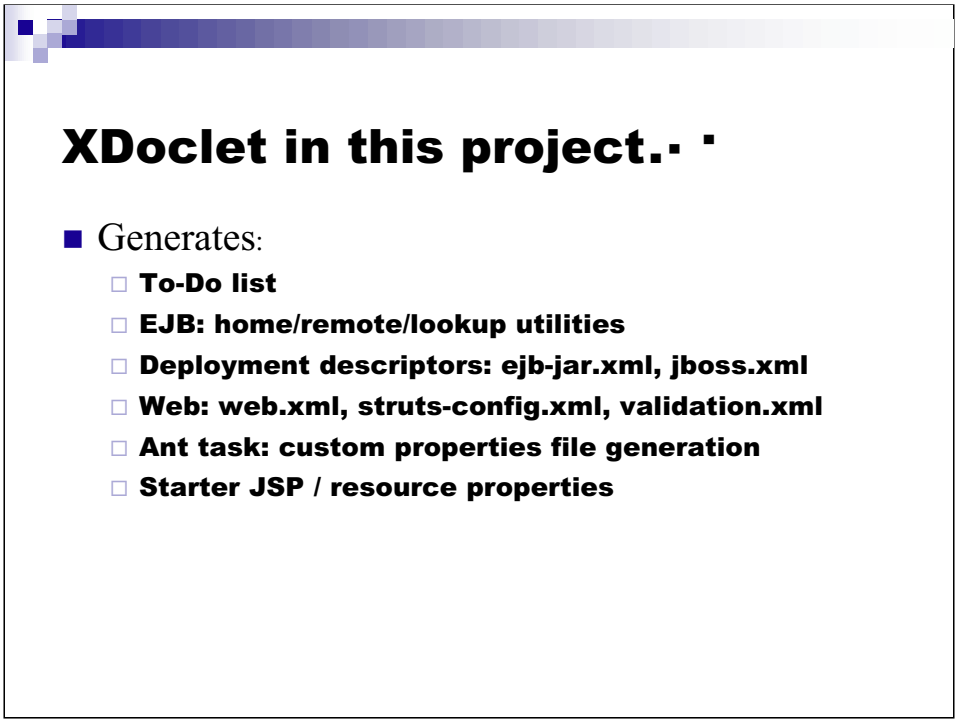


Its important to note here that there are deployment descriptors for EJB (the stateless session bean), the web application, Struts, and the Validator. There are also vendor specific descriptors needed for EJB, typically and cookie cutter code such as home, remote, and lookup utilities that are needed for every EJB.

Bonus Requirements

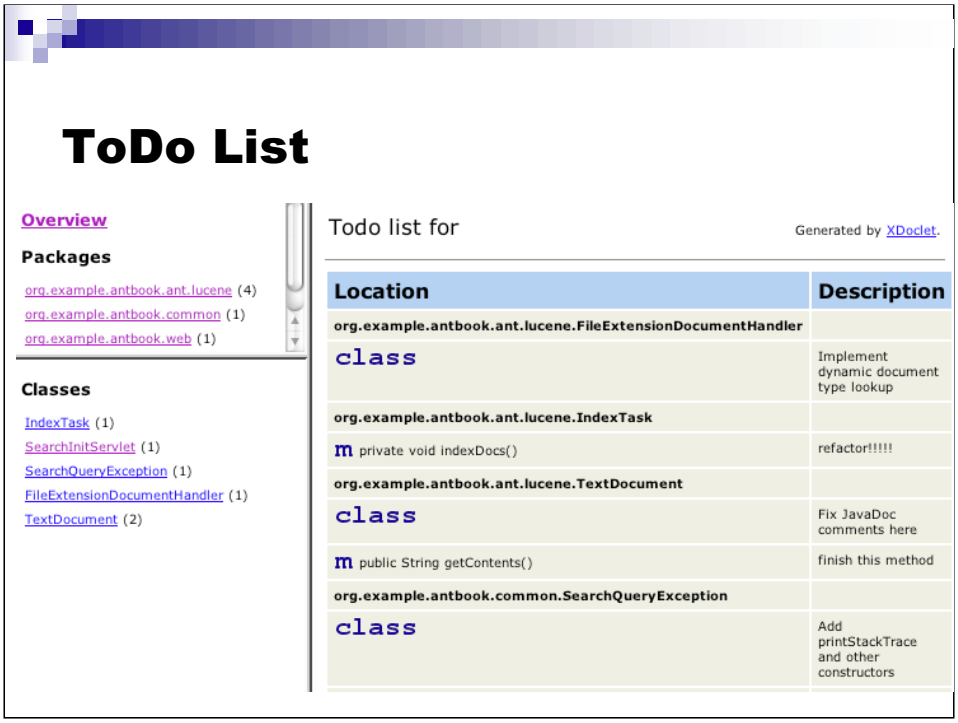
- Report the duration of each web request
- Allow viewing of build information from live application
- Build and deploy multiple sites simultaneously, each with separate indexes and content

Reporting the duration is handled with a servlet 2.3 filter, which needs to be listed in web.xml. Build information is not XDoclet related, but just a reminder. XDoclet deals with our multiple sites requirement by generating site specific information.



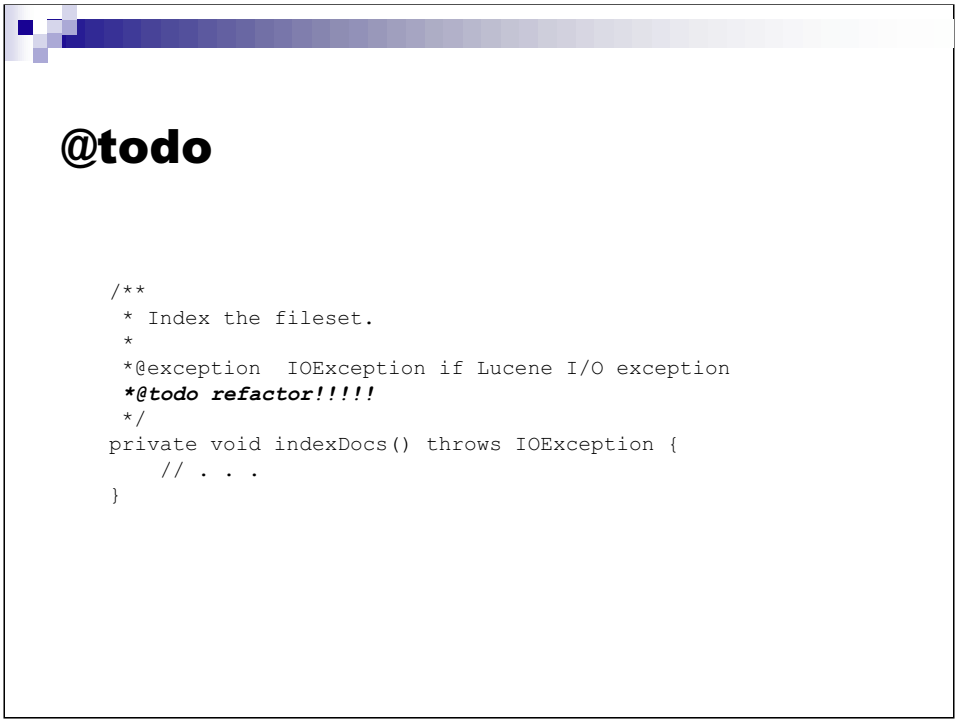
whew - that's a lot of stuff. This really makes life a lot easier and more pleasant for developers.

The examples of XDoclet usage are going to first show the end result of what is generated and then back into how it is generated from the source code and Ant.



Here is an example To-Do list. It looks similar to Javadoc and is a great report to generate nightly or during continuous integration builds to keep the team up-to-date.

How do we mark to-do's in the source code? See the next slide…



Simple - just flag an @todo tag at the method, member variable, or class level.

One thing to note is that JDK 1.4's javadoc tool actually outputs a warning saying this is a reserved tag indicating that Sun intends this to be used in a manner similar to XDoclet's usage at some point in the future.

Ant <taskdef>

```
<taskdef name="todo"
         classname="xdoclet.modules.doc.DocumentDocletTask"
         classpathref="xdoclet.classpath"
/>

<target name="todo">
    <mkdir dir="${build.dir}/todo"/>

    <todo destdir="${build.dir}/todo" >
        <fileset dir="src/anttask"/>
        <fileset dir="src/common"/>
        <fileset dir="src/ejb"/>
        <fileset dir="src/web"/>

        <info/>
    </todo>
</target>
```

Here is what it takes in Ant to generate a to-do list. <taskdef> is how custom or 3rd party tasks are defined in Ant.

The <info/> subtask is what generates the todo list and could do so on any tag, not just @todo.

EJB Deployment Descriptor

```
.
.<!-- ejb-jar.xml -->
.
 <enterprise-beans>

       <session >

         <ejb-name>org.example.antbook.session.SearchSession</ejb-name>

         <home>org.example.antbook.session.SearchSessionHome</home>
         <remote>org.example.antbook.session.SearchSession</remote>
         <ejb-class>
            org.example.antbook.session.SearchSessionBean
         </ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>

      </session>
.
.
.
```

Here is a very typical simplistic EJB deployment descriptor - our single stateless session bean. Note that the classname is listed here, and if we were not generating this file and wanted to move our source code around, the descriptors would break.

Vender-specific Descriptors

```
<jboss>

   <enterprise-beans>

      <session>
         <ejb-name>org.example.antbook.session.SearchSession</ejb-name>
         <jndi-name>
           personal/org.example.antbook.session.SearchSession
         </jndi-name>
      </session>

   </enterprise-beans>

   <resource-managers>
   </resource-managers>

</jboss>
```

Once again, the duplication. Ouch. Fully qualified classname again, along with JNDI lookup information.

Session Bean

```
/**
 * @ejb.bean type="Stateless"
 *         jndi-name="${site}/org.example.antbook.session.SearchSession"
 * @ejb.util generate="physical"
 */
public class SearchSessionBean implements SessionBean {
  //…

    /**
     * @ejb.interface-method
     */
    public Document[] search(String indexDir, String query)
                          throws SearchQueryException,SystemException {
        return SearchUtil.findDocuments(indexDir, query);
    }
}
```

And here is how the source code is tagged to generate the previous two descriptors. Make special note of the Ant property usage in the XDoclet tags. This adds incredible control.

The @ejb.interface-method tag is a marker to generate this method in the home and remote (and/or local) interfaces.

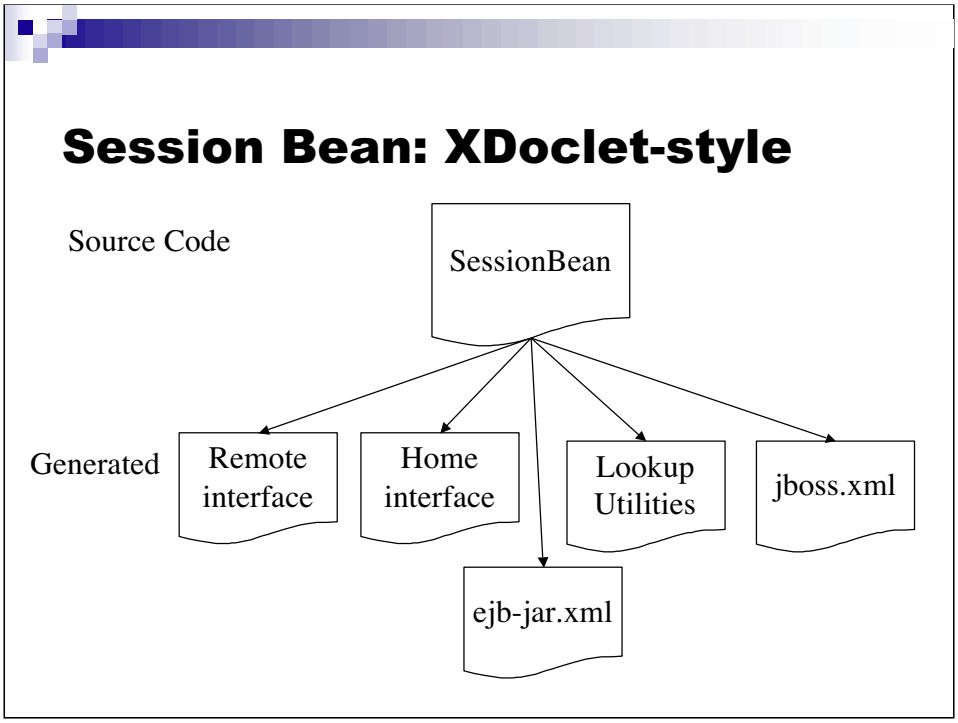
Ant Property Substitution

- Allows build-time control over values generated

```
<property file="${user.home}/.${ant.project.name}-build.properties"/>
<property file="${user.home}/.build.properties"/>
<property file="build.properties"/>

<property name="site" value="sample"/>
```

```
/**
 * @ejb.bean type="Stateless"
 *        jndi-name="${site}/org.example.antbook.session.SearchSession"
 * @ejb.util generate="physical"
 */
public class SearchSessionBean implements SessionBean { . . . }
```

Ant property substitution adds some very special capabilities allowing session beans to have unique JNDI lookups among deployed sites in the same container.



Only the session bean was written. XDoclet did the rest. Wow.

<ejbdoclet>

```
<target name="ejbdoclet" depends="init">
     <taskdef name="ejbdoclet"
                 classname="xdoclet.modules.ejb.EjbDocletTask"
                 classpathref="xdoclet.classpath"/>

     <mkdir dir="${build.dir}/ejb/gen"/>
     <ejbdoclet destdir="${build.dir}/ejb/gen"
                addedtags="@xdoclet-generated at ${TODAY}"
                ejbspec="1.1"
                force="${xdoclet.force}"
                mergedir="metadata/ejb">
          <fileset dir="src/ejb"/>

          <remoteinterface/>
          <homeinterface/>
          <utilobject/>
          <jboss validatexml="true" destdir="${build.dir}/${site}"/>
          <deploymentdescriptor validatexml="true"/>
     </ejbdoclet>
</target>
```

Here's the Ant syntax to do the generation shown. Note that <ejbdoclet> can be forced to regenerate or do dependency checking. Merge directory will be discussed in a moment.

The generation here is restricted to the ejb source code tree.

JBoss' deployment descriptor is site-specific, and so its destination directory is overridden specially.

Servlet Definition

```
<servlet>

    <servlet-name>search-init</servlet-name>
    <servlet-class>
         org.example.antbook.web.SearchInitServlet
    </servlet-class>

    <init-param>
      <param-name>index-dir</param-name>
      <param-value>/Users/erik/. . ./personal/index</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>

</servlet>
```

Now to the web tier generation. We have an initialization servlet with the path to the index in it.

Servlet

```
/**
 * @web.servlet name="search-init" load-on-startup="1"
 * @web.servlet-init-param name="index-dir" value="${index.dir}"
 *
 * @todo Refactor to use JNDI for directory lookup.
 */
public class SearchInitServlet extends HttpServlet {
    public void init() throws ServletException {
        super.init();

        ServletConfig config = getServletConfig();
        getServletContext().setAttribute(Constants.SEARCH_DIRECTORY,
                config.getInitParameter("index-dir"));
    }
}
```

This metadata is easily codified into the servlet.

Let's take a moment to ponder this. We've moved *deployment* descriptor information into our source code. Does that really make sense or have we broken some J2EE best practice by shifting the deployer role to the developer? Well, you decide, but its worth considering because @tags are definitely not the right place for everything. We'll see this more as we go.

Note that Ant property substitution is again being used for the initialization value, making the data in the @tags really come from the build time environment and not hard-coded in the code. Interesting!

Filter Definition

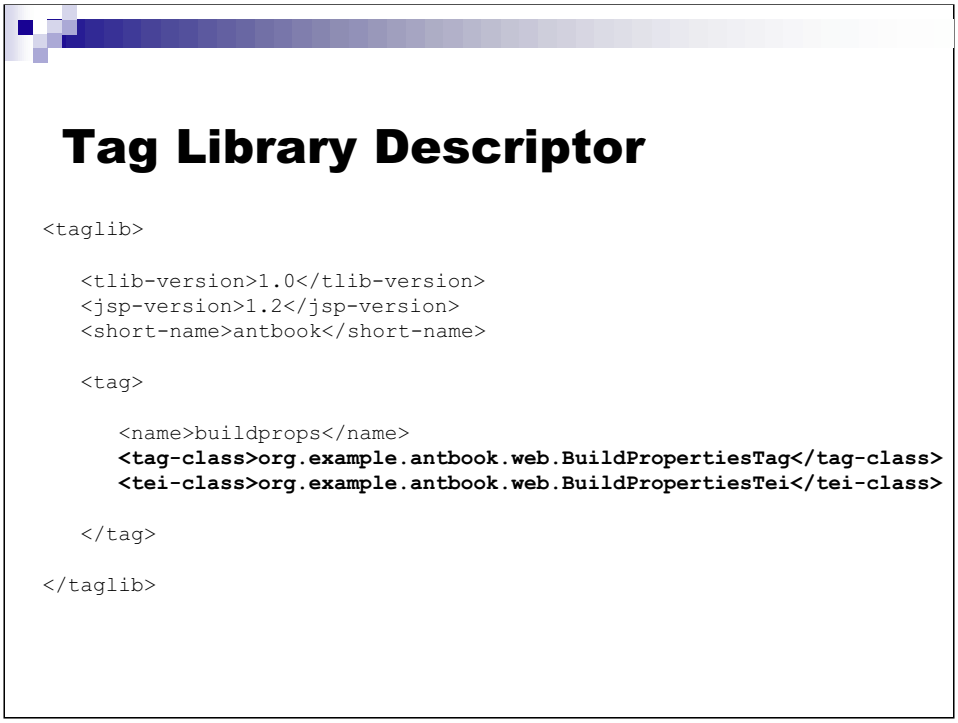
```
<filter>

  <filter-name>TimingFilter</filter-name>
  <filter-class>org.example.antbook.web.TimingFilter</filter-class>

</filter>

<filter-mapping>

  <filter-name>TimingFilter</filter-name>
  <url-pattern>/*</url-pattern>

</filter-mapping>
```

And here's our timing filter being defined in web.xml.

Filter

```
/**
 * Logs duration of each request.
 *
 * @web.filter name="TimingFilter"
 * @web.filter-mapping url-pattern="/*" servlet-name="TimingFilter"
 */
public class TimingFilter implements Filter {
    // . . .
}
```

And our filter source code. The URL pattern information is questionable whether it is ok in the source code. For projects that do not need to be truly sysadmin/deployer configurable it works great this way and that is typically the case. For this filter, the idea is to time every request and no other role needs to control this.

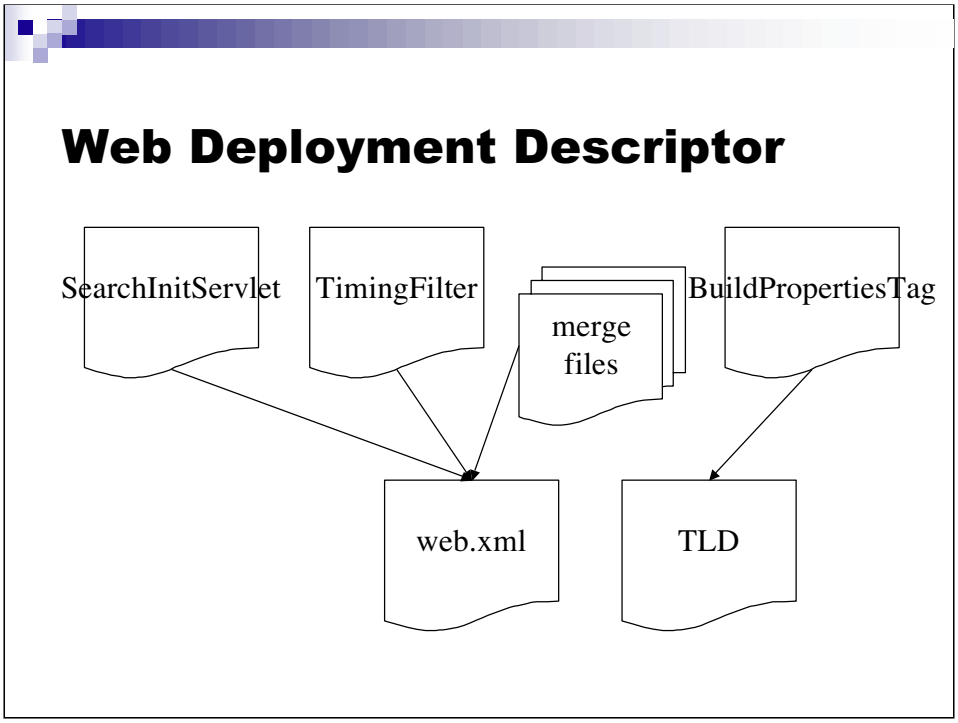


Tag libraries require their own Tag Library Descriptor (TLD) file.

Tag Library

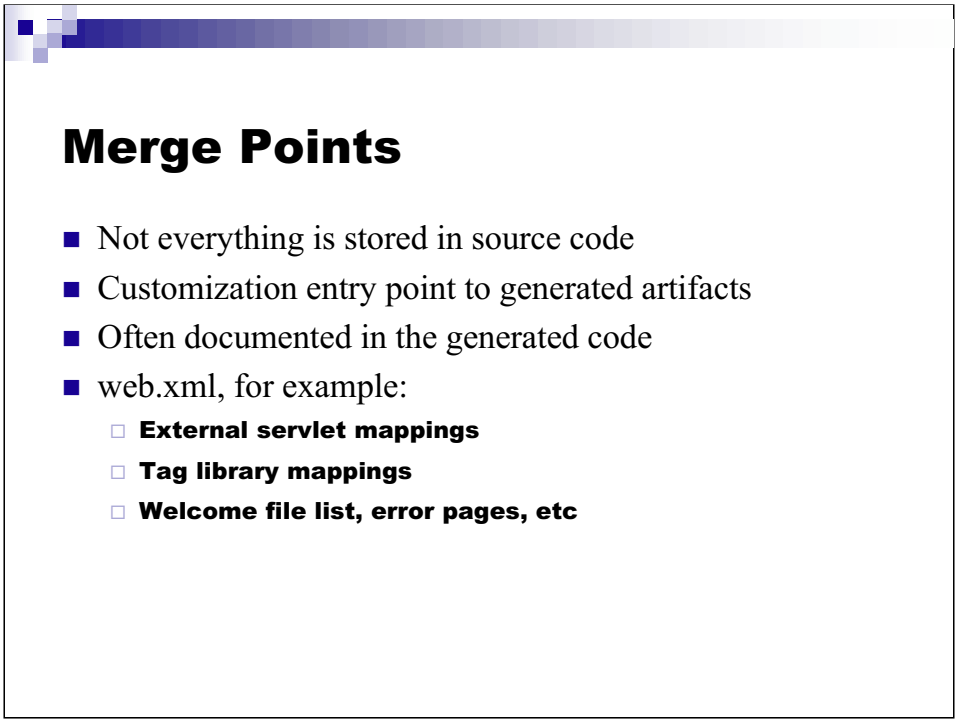
```
/**
 * Iterates build properties and places name/value into
 * page scope.
 *
 * @jsp.tag name="buildprops"
 *          bodycontent="JSP"
 *          tei-class="org.example.antbook.web.BuildPropertiesTei"
 */
public class BuildPropertiesTag extends BodyTagSupport {
    // . . .
}
```

This metadata makes perfect sense to be tied to the source code.



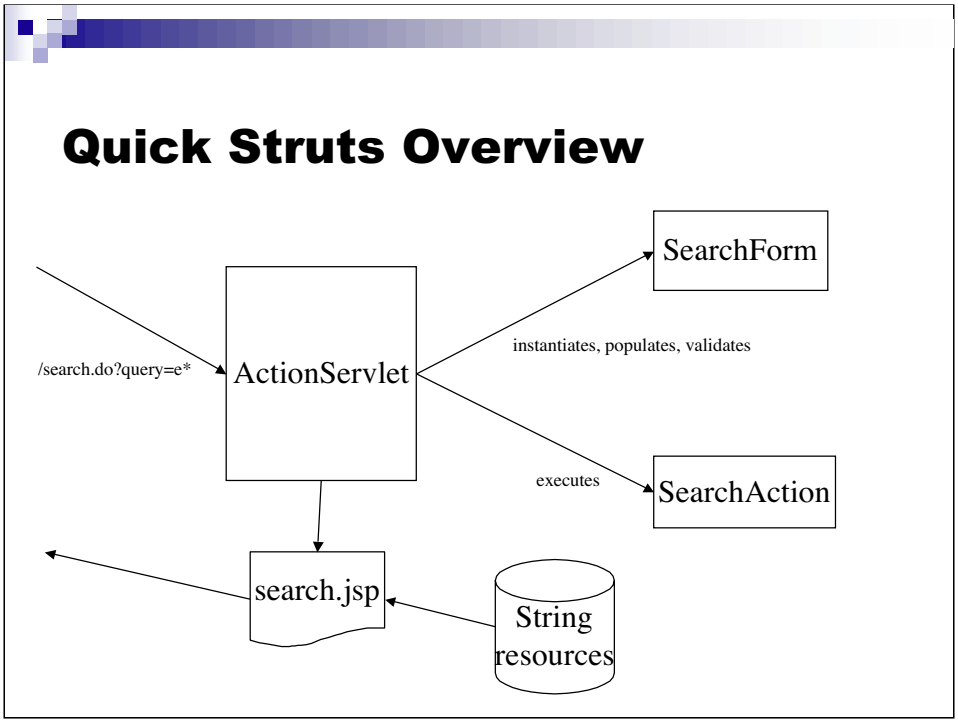
And our web tier generation is upside down from our EJB generation. We have many inputs into a single deployment descriptor.

And here is where merge points make their appearance. Again, not all metadata belongs in the source code and much of it couldn't be there. XDoclet provides a way to feed this information into the templates.



The templates contain “merge points” - places where specific files are pulled in to the template generation process (and even processed as if they were templates, if desired).

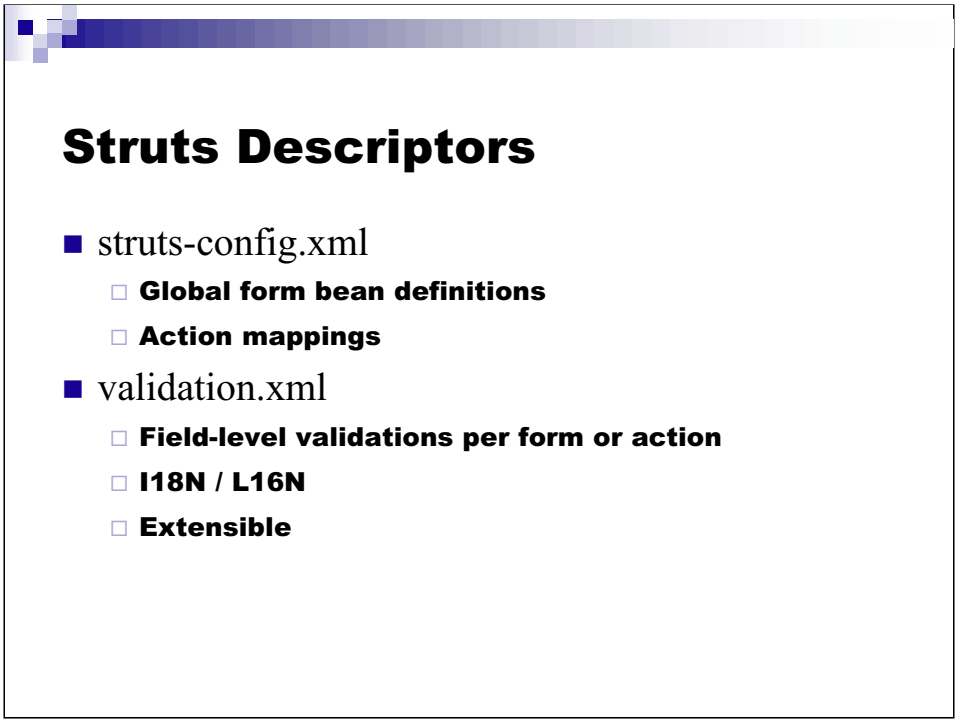
We’ll see an example of the documented merge points in a moment, but essentially if a merge file is not present an XML or Java comment is generated that indicates what filename could be created to inject some merge information.



Because this is a Struts app, and lots of generation occurs at this level, here's a brief overview to see what generation can take place here.

A struts-config.xml defines form beans by name/classname and action mappings. Validation requires a validation.xml defining every field of every form that is to be validated (required, date, etc). These validation rules can be done server-side or even client-side with injected JavaScript.

The JSP and string resources can be passively generated as "starter code" as we'll see shortly.



These are the descriptions of the Struts “deployment descriptors”, as shown from the previous diagram.

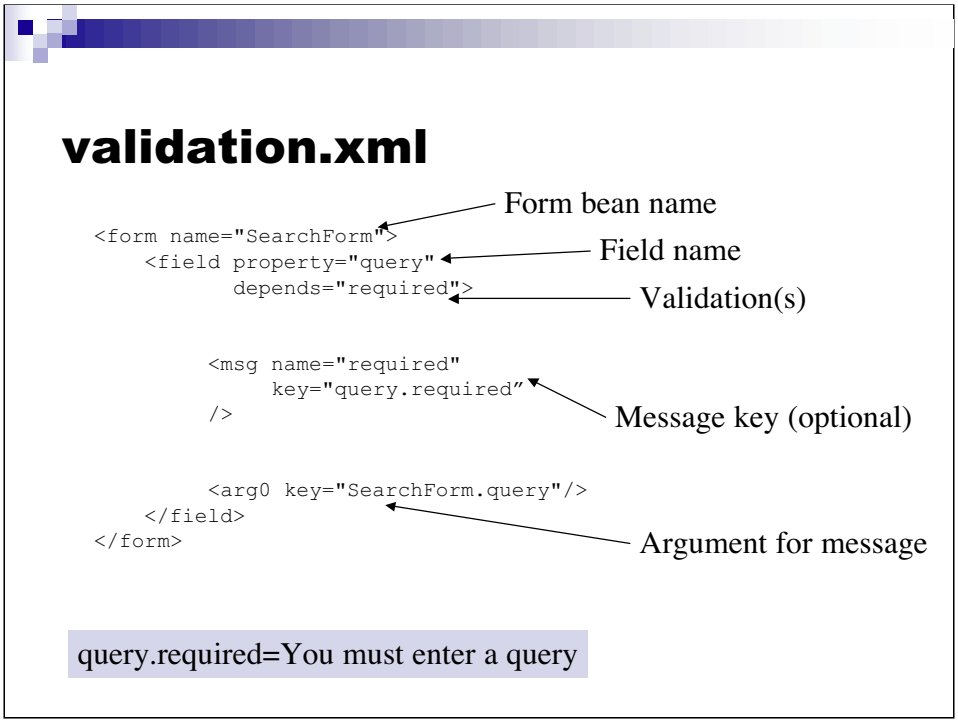
Of great importance is the action mappings for Struts. These can be generated by XDoclet, but I do not believe this particular glue is appropriate for the source code. Multiple forms can be used for a single action class, and multiple actions can process the same form - this glue is not appropriate for source code in my opinion. Your mileage may vary, and certainly for simple applications it might just be the easiest/quickest way to work and do the job successfully.

struts-config.xml

```
<form-beans>
    <form-bean name="SearchForm"
               type="org.example.antbook.struts.SearchForm"
    />

    <!--
         If you have non XDoclet forms, define them in a file
         called struts-forms.xml and
         place it in your merge directory.
    -->

</form-beans>
```

The gist, YADD (yet another deployment descriptor). Name/classname pairs for form beans.

Here, also, is an example of XDoclet's comment about a merge point that could be used to define some external form beans (rarely the case for Struts, but that's beside the point… :)



Validator's deployment descriptor. Details are of course unimportant for the sake of demonstration, but regardless, there are many details here that really belong with our form bean source code.

query.required is a resource key that could be localized and is what appears if the user does not enter a query. The argument (arg0) is a key which refers the label of the field, and could be dynamically placed in the error message using {0} syntax in query.required.

Struts ActionForm

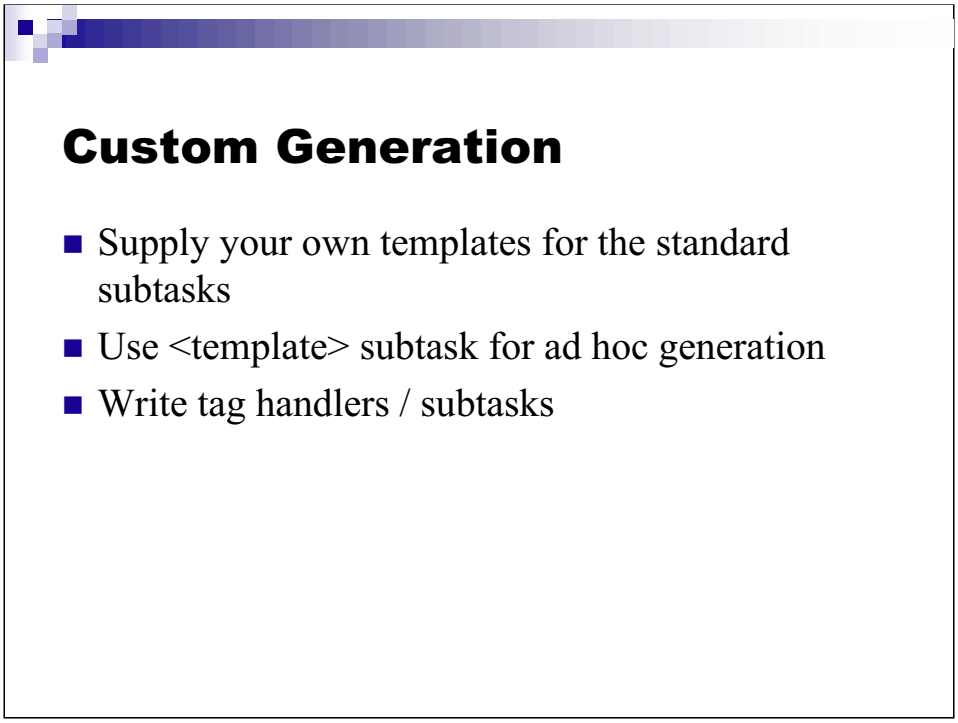
```
/**
 * Search query entry form.
 *
 * @struts.form    name="SearchForm"
 */
public class SearchForm extends ValidatorForm {
    private String query;


    /**
     * Sets the query attribute of the SearchForm object
     *
     * @struts.validator type="required" msgkey="query.required"
     */
    public void setQuery(String query) {
        this.query = query;
    }


    public String getQuery() {
        return query;
    }
}
```

Here is our Struts form bean. The name is provided and also the validations.

<webdoclet>

```
<webdoclet destdir="${build.dir}/web/WEB-INF"
           force="${xdoclet.force}"
           mergedir="metadata/web">

    <fileset dir="src/web"/>

    <deploymentdescriptor validatexml="true"
                          destdir="${build.dir}/${site}"
    />
    <jsptaglib validatexml="true"
               shortName="antbook"
               filename="antbook.tld"
    />
    <strutsconfigxml validatexml="true" version="1.1"/>
    <strutsvalidationxml omitdtd="true"/>
</webdoclet>
```

XDoclet's webdoclet task generates web.xml, antbook.tld, struts-config.xml and validation.xml. Wow, again!



If a bug or enhancement to a template is all that is needed for an existing subtask, simply tweak a custom version of a builtin template file and tell the subtask to use yours instead of the built in one. Its that easy.

Not only can XDoclet do all of the previous generations out of the box, it allows you to build your own templates and generate from source code using a generic <template> subtask.

At the super power user edge of the spectrum, custom tag handlers can be written as well as subtasks. This can be needed for more involved generation (consider the to-do list generation example - it generates many files for the frameset configuration - that's way too much for <template> ad hoc generation).

Ant Task Library

- Declare multiple Ant tasks using a properties file:

```
<taskdef resource="taskdef.properties"
  classpath="${lucene.jar}:${dist.dir}/antbook-
  anttask.jar:${jtidy.jar}"
/>
```

- Name/classname pairs:

```
index=org.example.antbook.ant.lucene.IndexTask
```

- Task(s) available:

```
<index index="${index.dir}"
       overwrite="false">…</index>
```

Custom Ant tasks can be defined by mappings in a properties file in order to define multiple tasks at once and to provide a level of protection against classnames changing and breaking build files. As long as the properties file (which can be embedded in a task library's JAR file) is kept up to date, the build file does not need to change when upgrading. (and yes, XDoclet should use this facility for defining their tasks - I'm working on it!).

And finally, an example of how the index task is used in the build file.

Custom Ant Task Metadata

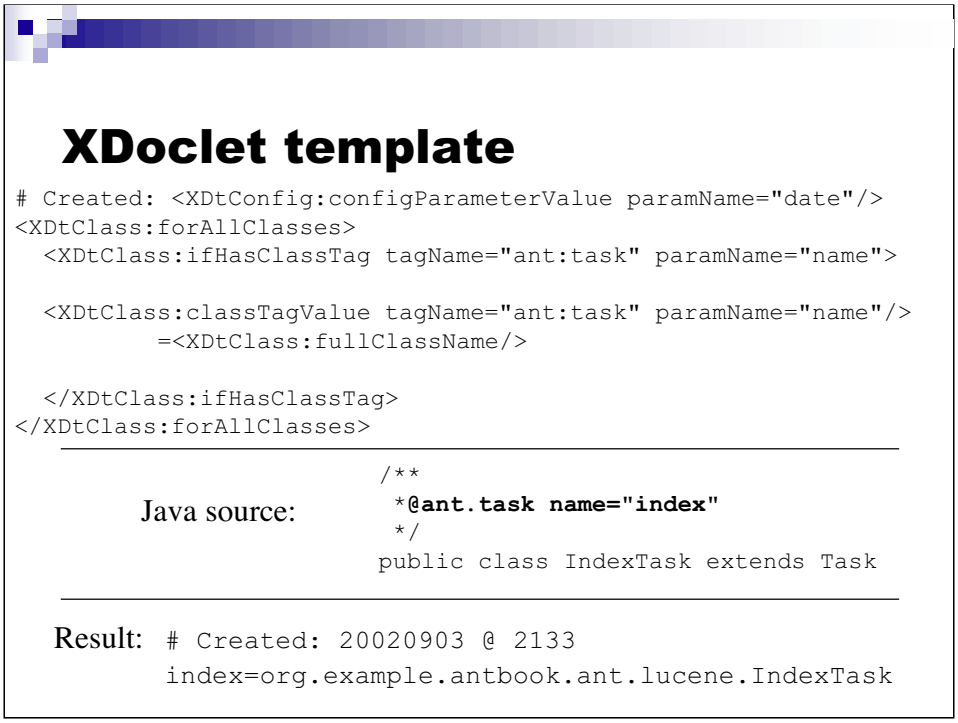
```
/**
 * Ant task to index files with Lucene
 *
 * @ant.task name="index"
 */
public class IndexTask extends Task {
    // . . .
}
```

All our task needs is a name.

<template> subtask

```
<xdoclet destdir="${build.dir}/anttask/classes">
    <fileset dir="src/anttask">
        <include name="**/*.java" />
    </fileset>

    <template templatefile="metadata/anttask/taskdef.xdt"
              destinationfile="taskdef.properties">
        <configParam name="date" value="${DSTAMP} @ ${TSTAMP}"/>
    </template>
</xdoclet>
```

To liven things up, here's an example of generating that properties file, but injecting a configuration (build-time) parameter into the template. This is a date/timestamp.



Shown on top - the XDoclet template used to generate the properties file, then a snippet of the Java source code, and finally the end result.

Notice the tag library syntax of the templates. The configuration parameter is easily available. Tags have a namespace (XDtClass for most of this example). forAllClasses loops over all classes in the model (remember XJavaDoc from the earlier slides). If the class has an @ant:task (or @ant.task, see note below) with a name parameter then a properties line is emitted with that that name, an equals sign, and the fully qualified classname. Instantly our task library "descriptor" will be kept up to date with our source code (if we remember to @tag it, of course).

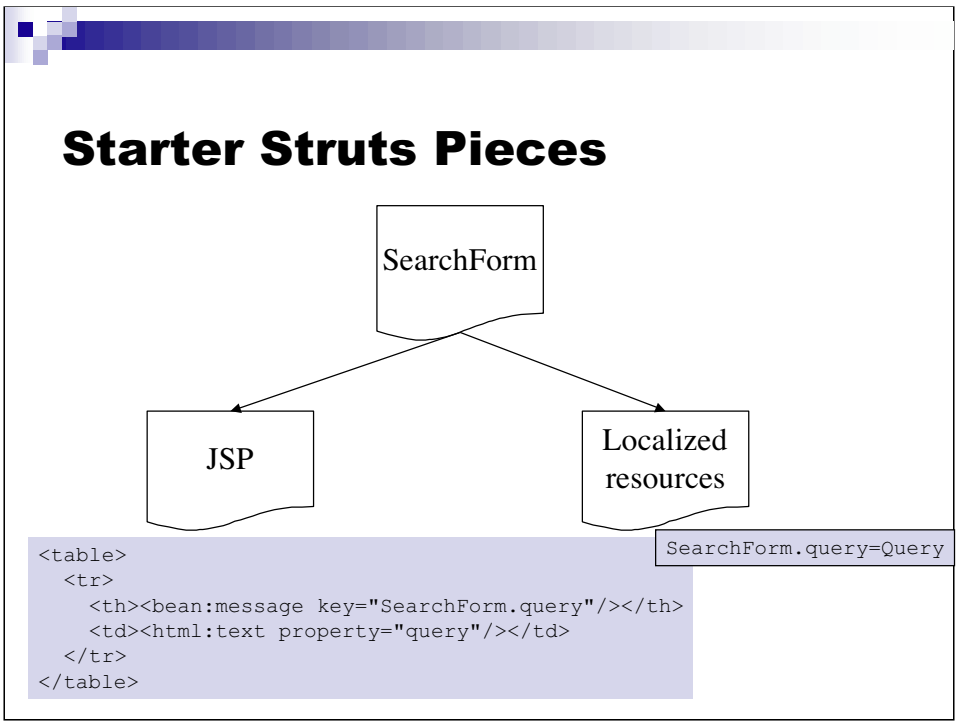
Note: colons and dots are interchangeable as @tag separators currently.

Passive Generators

- Based on metadata from XDoclet
- One-time generation
- Results customized
- Regeneration painful after customizing
- Great for rapid development
- Example:
  - **Starter Struts JSP pages**
  - **String resources**

Sometimes its handy to just generate some quick and dirty “starter” code to then take and tweak. This is a one-time process and not repeated unless the modified files should be completely overwritten and changes lost.

In this example, developers create form beans first and can run a generator to create a JSP file which is automatically localized with generated string resources for field labels.



Very simple (andn ugly) HTML, but it gets the presentation generated quickly which can be later refined by those with true layout skills :)

The string resources are generated with the form name dotted with the field name, with a reasonable representation for a label (first letter of words uppercased and space separated). Struts forms can even be nested JavaBean's, so the dotted syntax could be recursive!

Per-class template generation

```
<xdoclet destdir="${build.dir}”
         excludedtags="@version,@author"
         force="${xdoclet.force}">
     <fileset dir="${struts.src.dir}"
              includes="**/${form.name}.java"
     />
     <template templateFile="src/FormKeys.xdt"
               ofType="org.apache.struts.validator.ValidatorForm"
               acceptAbstractClasses="false"
               prefixWithPackageStructure="false"
               destinationFile="{0}.properties"
     />
     <template templateFile="src/StrutsForm_jsp.xdt"
               ofType="org.apache.struts.validator.ValidatorForm"
               acceptAbstractClasses="false"
               prefixWithPackageStructure="false"
               destinationFile="{0}.jsp"
     />
</xdoclet>
```

.xdt - XDoclet Template - this is the generally used file extension for template files.

Only concrete subclasses of ValidatorForm are processed. The first <template> generates [classname].properties. The second generates [classname].jsp. The file contents are cut and pasted to the proper final place - chances are [classname] is not the desired name of the JSP. The properties are pasted into ApplicationResources.properties (the Struts default resources file).

prefixWithPackageStructure is disabled, but is handy in some cases.

The template for the string resources property file. A custom tag handler was developed to handle the recursive logic and other tricky pieces made easy in Java code, but difficult/impossible with existing template tags.

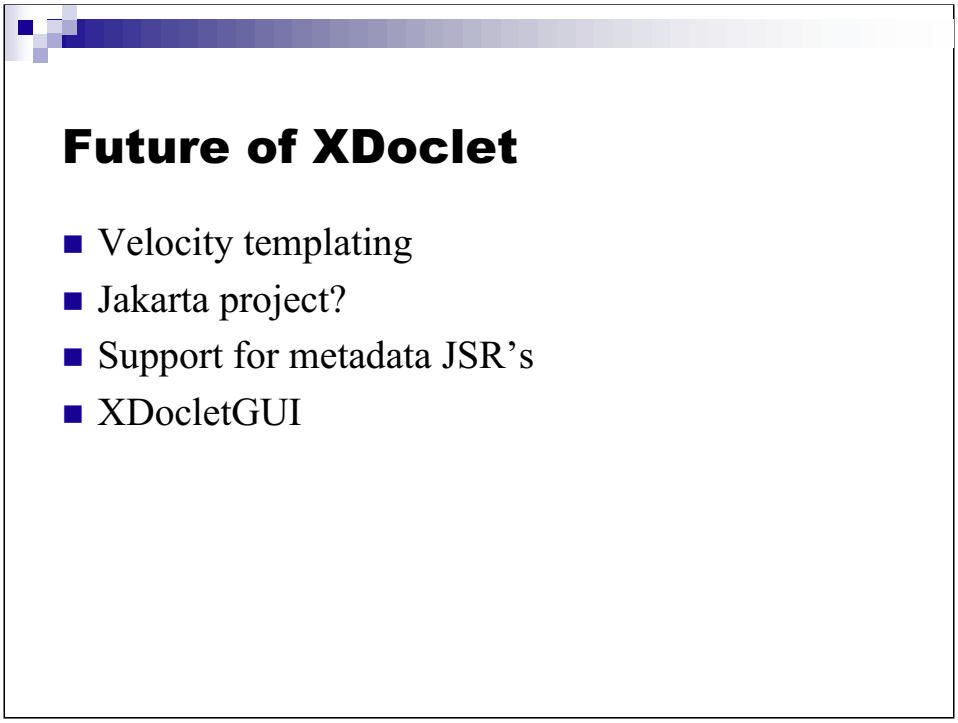
Some tags are "body" tags, and process the body either iteratively or conditionally. Other tags are "content" tags and generate some content directly into the generation process. One builtin tag is used, the other tags are custom.

forAllFields must only iterate over fields that have validations specified. fieldName could potentially be a recursively defined dotted name for nested beans. fieldDescription is just some pretty formatting of the field name (thisIsSomeField would output as "This Is Some Field", for example).

Custom Tag Handler

```
public void forAllFields(String template, Properties attributes)
                                                throws XDocletException {
    XClass clazz = getCurrentClass();
    TreeMap setters = new TreeMap(getFields(clazz));

    for (Iterator iterator = setters.keySet().iterator();
                                                 iterator.hasNext();) {
        curFieldName = (String) iterator.next();

        XMethod field = (XMethod) setters.get(curFieldName);

        setCurrentMethod(field);
        generate(template);
    }
}

public String fieldName(Properties props) {
    return curFieldName;
}
```

Here's the code of two tags. See, its not that bad, but does require some learning of XJavaDocl's API, but good IDE's make this trivial, especially alongside some of XDoclet's own tag handler source code.



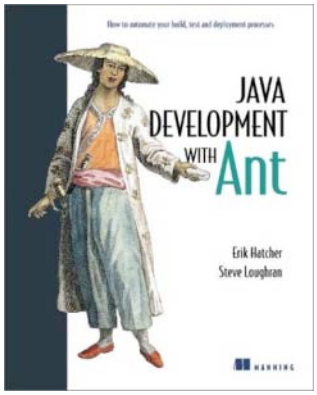
XDoclet 2 is already in progress and Velocity is being actively integrated (probably already done by the time of this presentation). XDoclet to Apache? Quite possible! Its an indispensible tool for J2EE (and even some general Java) developers.

XDoclet team is committed to supporting the metadata JSR's and is already well ahead of the curve for tool support.

XDocletGUI is well under way and plugins for IDEA and jEdit already exist (and probably others by the time of this presentation). It allows slick reading and _writing_ of XDoclet tags into Java code. XJavaDoc is bidirectional!

References

- Java Development with Ant
  - **http://www.amazon.com/exec/obidos/ASIN/1930110588**
  - **http://www.manning.com/antbook/**
- Ant
  - **http://jakarta.apache.org/ant**
- XDoclet
  - **http://xdoclet.sourceforge.net**
- Pragmatic Programmer
  - **http://www.pragmaticprogrammer.com/**
- Agile development
  - **eXtreme Programming Explained, Kent Beck**
  - **Agile Software Development, Alistair Cockburn**



My book - written by two authorities on Ant, both committers and *very* actively involved in the Ant communities.

XDoclet gets an entire chapter in my book.

These three books changed my life. Please read them. Be agile.

The End

- Thank You!