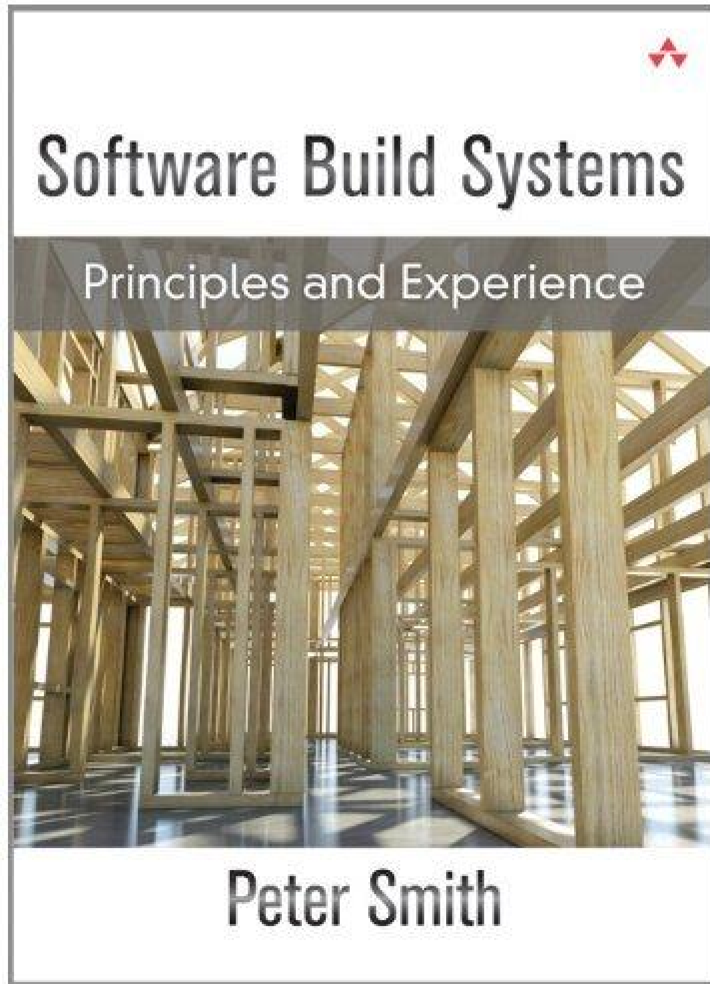
Build Systems

Przemek (Pshemek) Lach
SEng 371

Source



- Slides are based on content from book.
- Examples and diagrams also used from book.
- Software build systems; principles and experience. (2011). Reference and Research Book News, 26(3)

Motivation

Why Build?

- Anytime you want to do more than compile half a dozen files.
- Simplifies software development and testing.
- You want to make a change to your code and hit 'play' which will compile all your code, run all your tests, and maybe even automatically generate documentation and deploy your application.

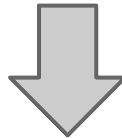
What Can You Build?

- Compilation of software from source to executable.
  - C, C++, Java, C#
- Packaging.
  - Python, JavaScript/Node (Interpreted Languages)
  - Web based applications
    - Combination of compiling source, or hybrid source, along with configuration files.
- Unit and integration testing.
- Automatically generate documentation.

Good, Better, Best

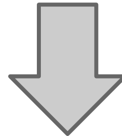
Write the compile statement each time on command line.

```
cc -o sorter main.c sort.c files.c tree.c merge.c
```



Using Make tool you can simply/automate this process.

```
sorter: main.c sort.c files.c tree.c merge.c
        cc -o sorter main.c sort.c files.c tree.c merge.c
```



```
SOURCES = main.c sort.c files.c tree.c merge.c

OBJECTS = $(SOURCES:.c=.o)

sorter: $(OBJECTS)
        $(CC) -o $@ $^
-include $(SOURCES:.c=.d)

%.d: %.c
        @$(CC) -MM $(CPPFLAGS) $< | sed 's#\(.*\)\.o: #\1.o \1.d: #g' > $@
```

Make can be configured to simplify even further and improve performance: only re-compile files that have changed; identify the names of the output files; which compiler to use etc....

An Increase In Complexity

- As a project grows a Make file can become very complicated.
- What usually happens is people start to roll their own build framework hacks.
- The problems really start when your project starts using 3rd party code that also uses a hacked framework; what then?
- Even on small projects (20 people) about 10% of time is spent on build issues.

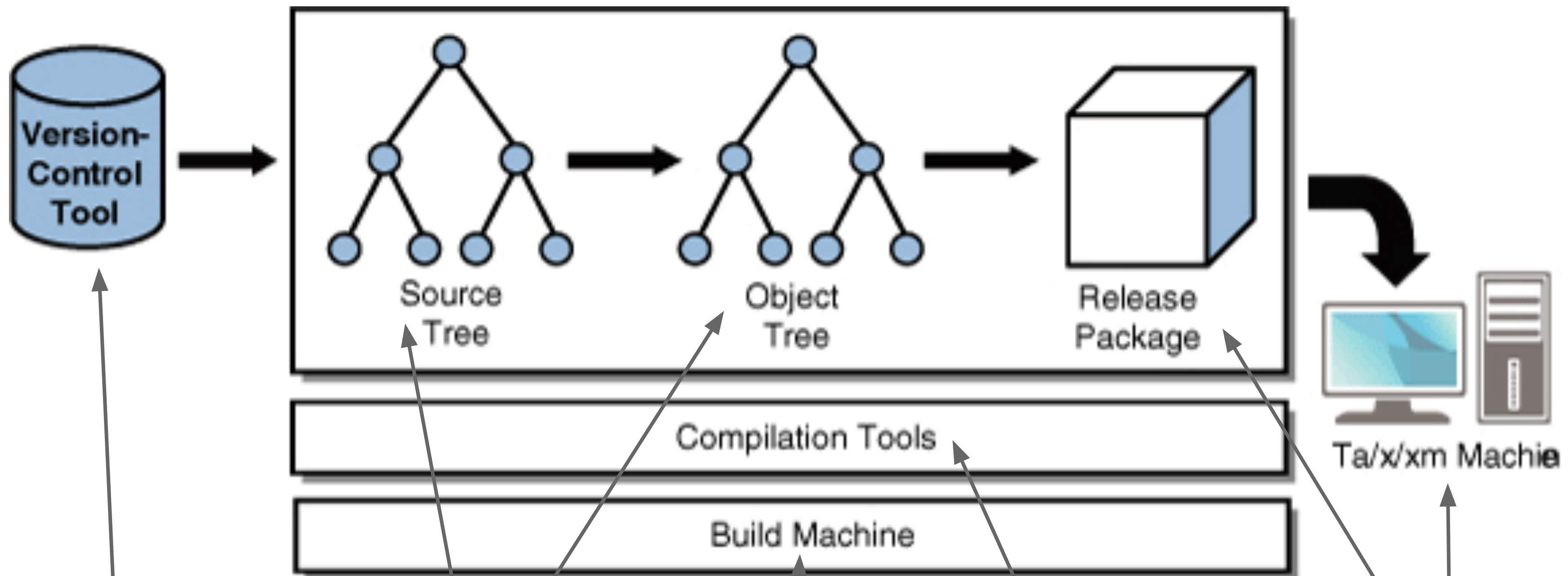
10% On What?

- Bad dependencies resulting in hard to fix compilation errors.
- Bad dependencies resulting in bad software images.
- Slow compilation.
- Time spent updating/fixing build files.
- ... there goes your profit.

Basics Concepts

Build System Workflows and Processes

Compiled Languages



> Git, SVN, Github
> Where all the source is stored and shared amongst developers.

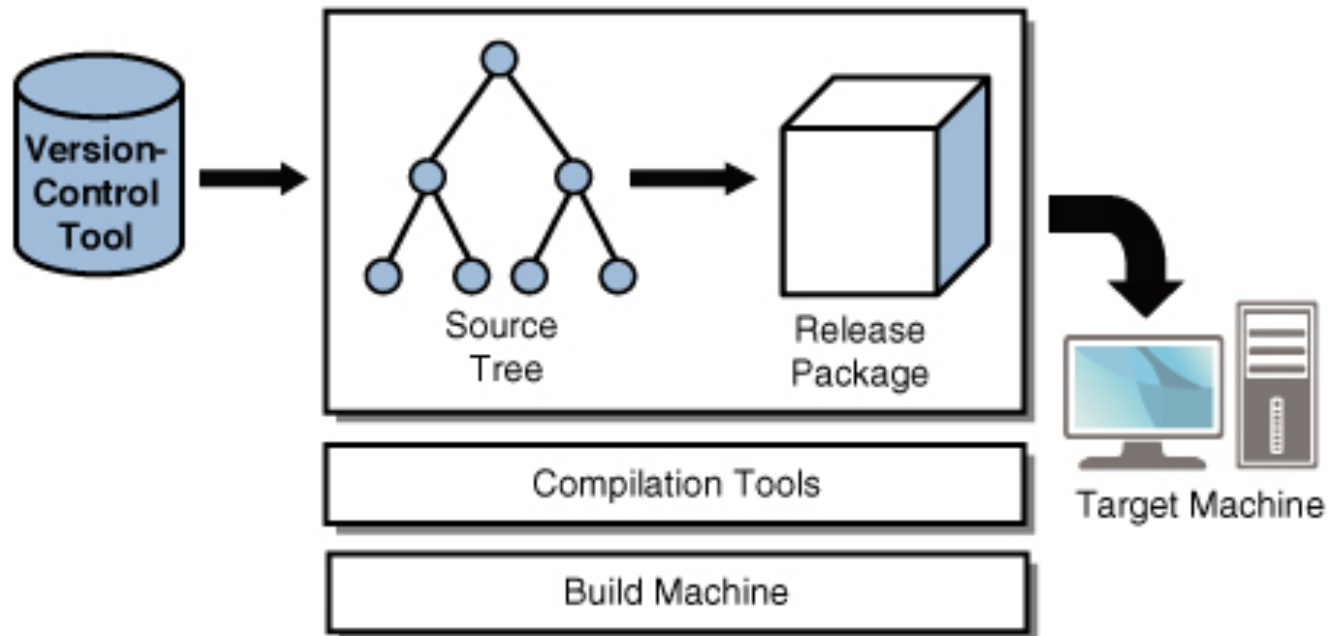
> Compiled objects.
> Each developer has his/her own.

> The physical machines on which the compilation is done.

> Compilers.
> Documentation & unit test generators.

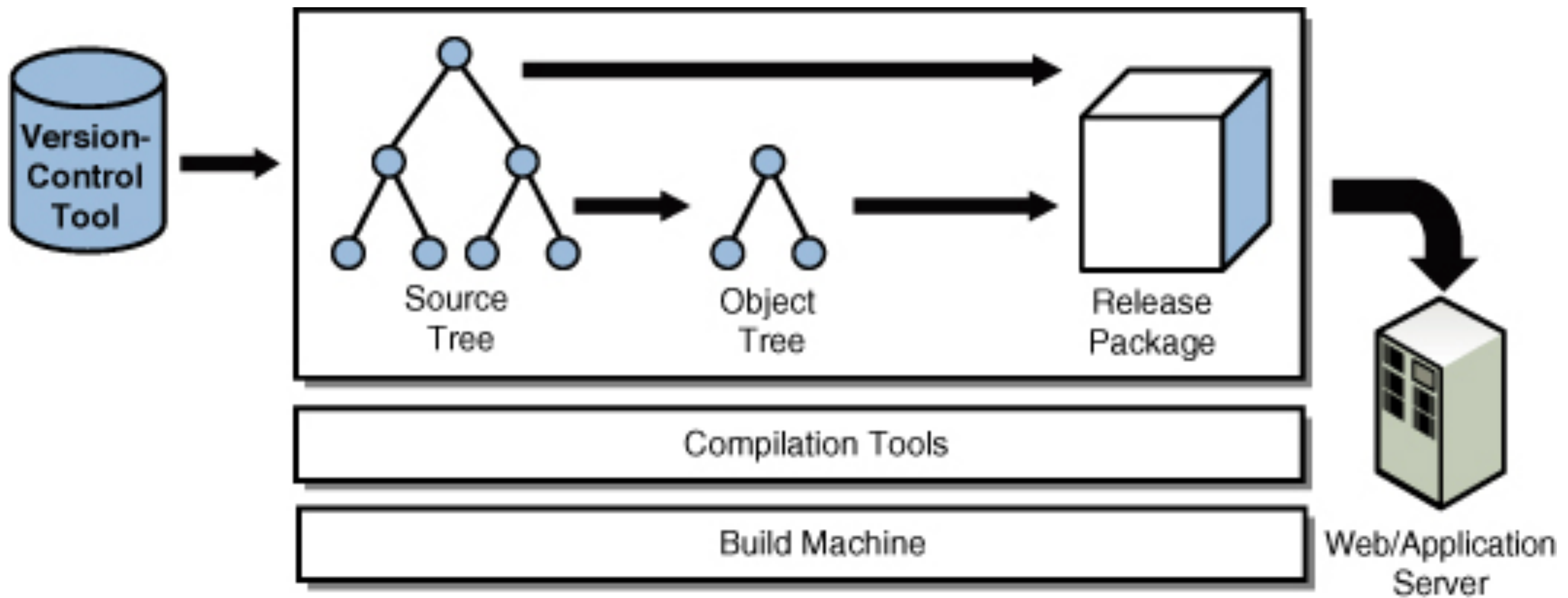
> Method of software packaging, distribution, and installation on client machines.

Interpreted Languages



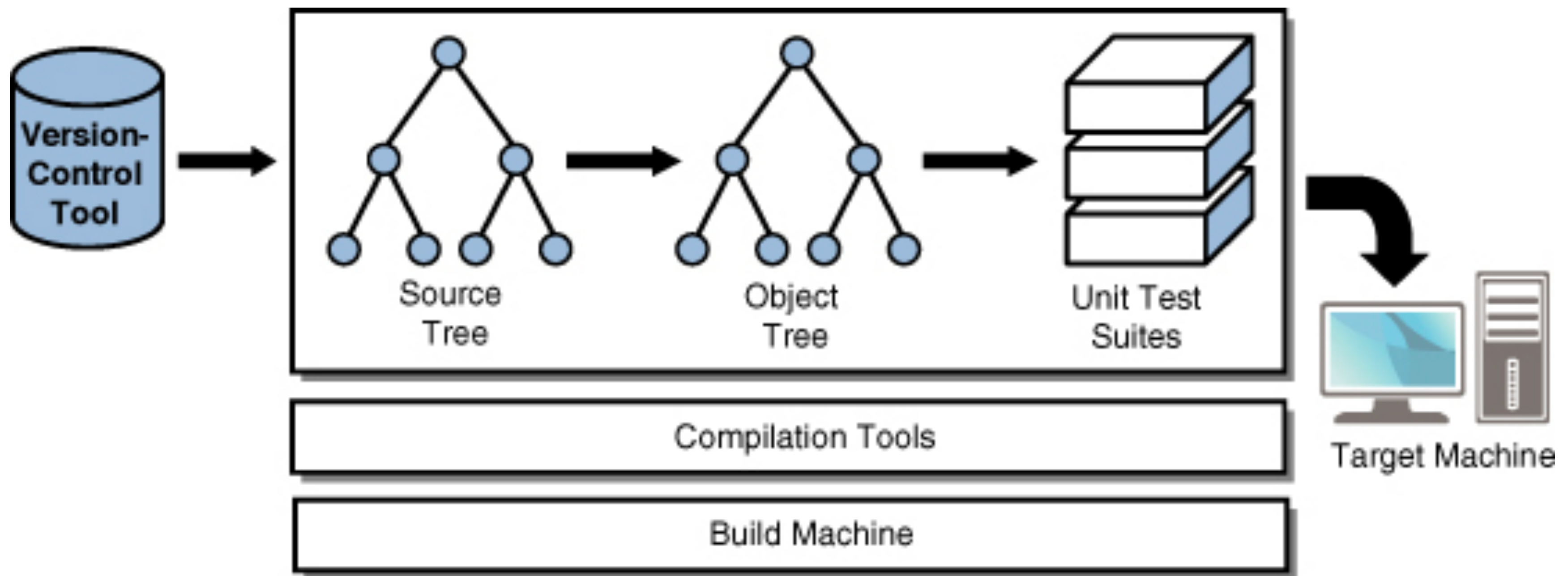
> Similar to compiled except that the source code is not compiled.
> The compilation tools here are used for transforming the source files into packages that are used by the system.

Web Applications



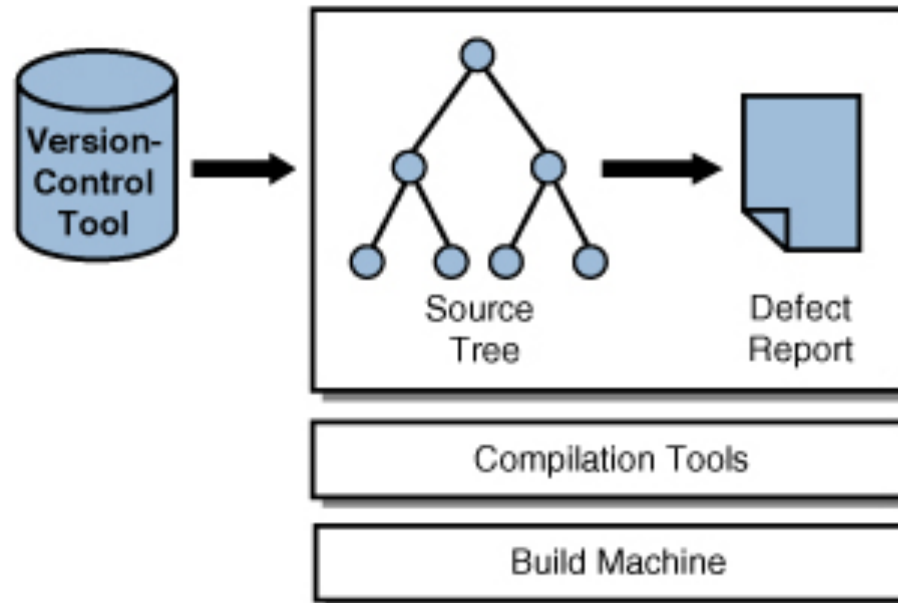
> A combination of compiled, interpreted, and static files (data & configuration).
> Some are copied directly (HTML) others are compiled first (Java).
> Tricky part here is that the users browser is involved in some of the interpretation (JavaScript).

Unit Testing



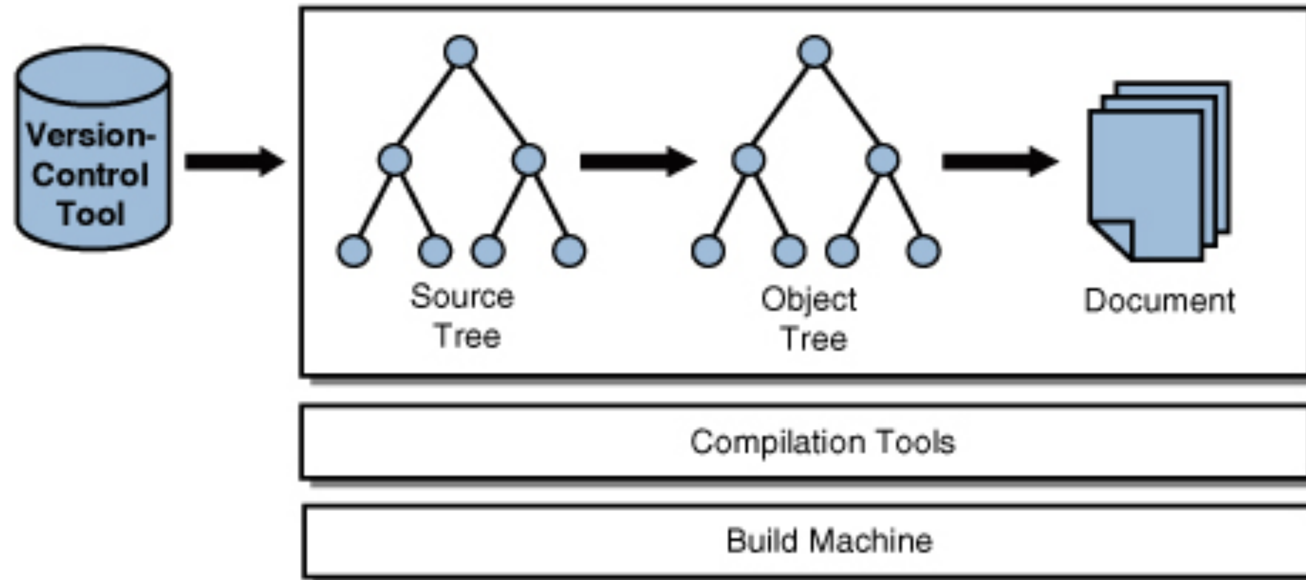
> Similar to what was just discussed except that the build systems produces several unit test suites and runs them.
> Similarly for integration testing. The process of preparing code and setting up the integration tests and test cases is done automatically.

Static Analysis



> Analyze source code to try to identify bugs, security holes, or other hard to detect problems that are not caught by compilation alone.
> Some tools: Coverity Prevent, Klocwork Insight, Findbugs
> No object code produced, just a report from the tool of choice and the source code.

Document Generation

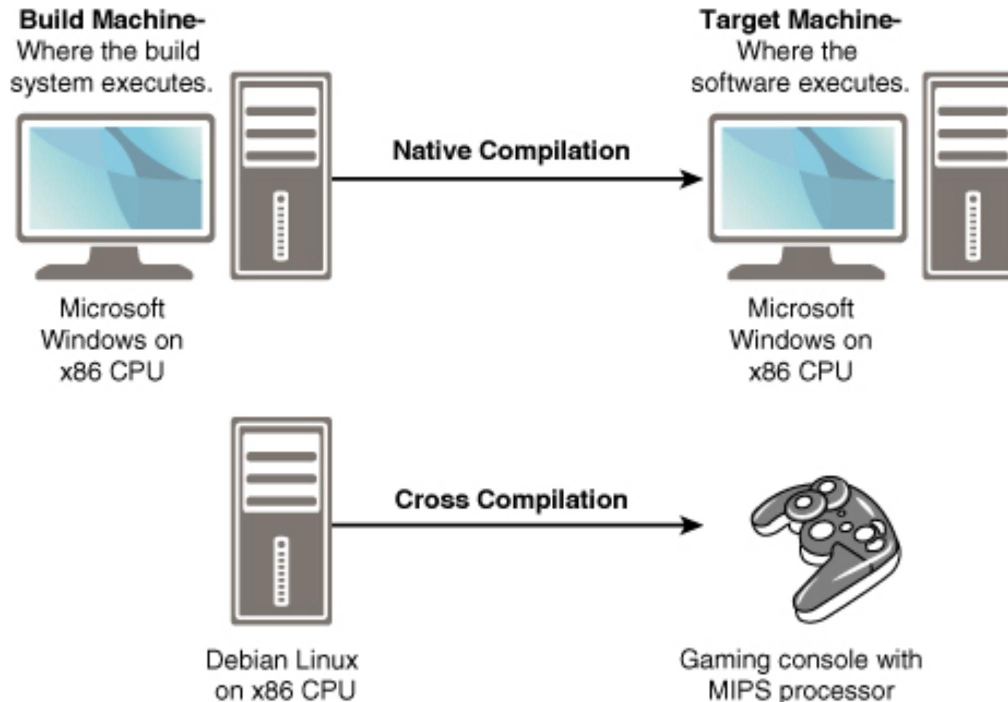


> Produces PDF, HTML and graphical images.
> Output to printer or upload to server

Types of Builds

- **Developer/Private** - checkout source from version control, make changes, compile, make more changes and re-compile.
- **Release Build** - done by release engineers. Create a complete software package for the software testers to test. When testers are happy the same package goes to customer.
- **Sanity Build** - similar to release build but does not go to customer. Automated software error checking done several times a day. a.k.a daily build or nightly build

Build Machines



> Native compilation - the software is executed on a target machine that is identical to the build machine.
> Cross compilation - the software is executed on a target machine that is different to the build machine (CPU, OS). e.g., XNA Game studio

Tools

Properties to Consider

- **Convenience** - how easy is for the people to describe the build process?
- **Correctness** - does it generate all the dependencies correctly, or in certain situations will it miss things?
- **Performance** - how long does it take for the build to complete?
- **Scalability** - as the project grows does the tool scale & can it include other build tools.

Properties to Consider (Cont'd)

- The weight one places on each property varies from developer to developer and project to project; i.e., it really depends on the situation.
- e.g., if you are building an iphone app you may not care about scalability or performance; however, if you're working on a banking application with a large team of developers then you will care.

The Tools

There are many tools out there, too many to cover in one go, so we'll focus on five the cover the different flavours of build tool classes:

- Make
- Apache Ant
- SCons
- CMake
- Eclipse

Make

- Considered the first build tool.
- Most commonly used for C/C++ development.
- If you develop for legacy systems you will most likely have to deal with Make.
- Not recommended for new projects.

Make (Cont'd)

- Created in 1977
- Uses the concept of a *rule* that defines all the dependencies between files for compilation purposes.

```
myprog: prog.c lib.c
    gcc -o myprog prog.c lib.c
```

> myprog is a generated file that is created when running gcc compiler and uses the input files prog.c and lib.c
> make is smart enough to look at file timestamps and re-compile only when necessary.
> a programmer has to write this file by hand (know as makefile). In a program with thousands of files and dependencies this can be a very challenging and error prone task.

GNU Make

- Before GNU Make each OS had its own version of Make. Each had slightly different syntax. This made it obviously difficult for developers.
- GNU Make supports many platforms and therefore makes your life as a developer slightly better. (Xcode)

GNU Make (Cont'd)

- GNU provides a language that can be thought of as three separate languages:
- **File dependencies** - rule-based syntax for specifying dependencies (similar to Make)

```
myprog: prog.c lib.c
```

- **Shell commands** - a list of shell commands enclosed in a rule that is triggered based on some event, like a file change.

```
cp myfile yourfile && cp myfile1 yourfile1
md5 < myfile >>yourfile
touch yourfile.done
```

GNU Make (Cont'd)

- **String Processing** - ability to manipulate GNU Make variables. This means that you can create complex expressions.

```
VARS := $(sort $(filter srcs-% cflags-%, $(.VARIABLES)))
```

GNU Make Pros

- **Widely supported** - mainly because its been around so long.
- **Very fast** - written in C and highly optimized.
- Portable syntax - available on wide range of platforms including Windows.
- **Full language** - if you can write a rule that maps an input file to an output file you can do any compilation you want. (Turing Complete)
- **First tool**.

GNU Make Cons

- **Inconsistent language design** - the language has evolved over a long time. Some features follow a different syntax than others.
- **No framework** - although you get lots of good language support it does not work out of the box.
- **Lack of portability** - although GNU Make is more portable than Make it still has problems: e.g., OS specific commands will not port (ls, grep, dir...)

GNU Make Cons (Cont'd)

- **Difficult debugging** - makefile executing is not guaranteed to be sequential.
- **Ease of use** - even though it's considered a complete language it's not very easy to use especially for beginners. You have to have a deep understanding before doing relatively simple things.
- **Evaluation**:

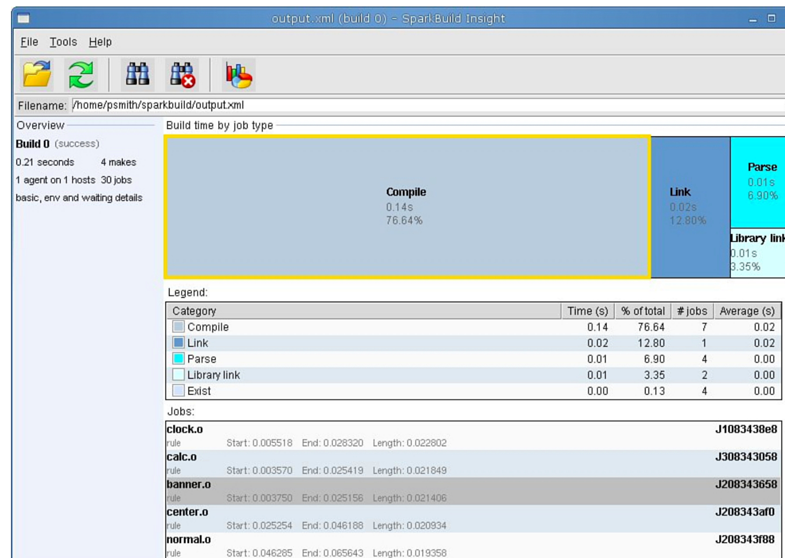
Convenience	Correctness	Performance	Scalability
Poor	Poor	Excellent	Excellent

GNU Make Alternatives

- Berkley Make
  - Berkeley Software Distribution (BSD): developed in mid 1970's.
  - A version of Unix that includes a variant of the Make tool known as Berkeley Make.
- NMake
  - Another variant of Make typically used in Visual Studio.
  - Same syntax as Make or Berkeley Make but shell commands are obviously targeted at Windows.

GNU Make Alternatives

- ElectricAccelerator and SparkBuild
  - Commercial tool made by Electric Cloud Inc.
  - ElectricAccelerator supports cluster based builds and supports GNU Make and NMake syntax.
  - SparkBuild is the feature limited version of ElectricAccelerator
- GUI Tool:



Apache Ant

- Most popular Java build tool.
- Supports compilation and generation of jar files.
- Runs on many platforms: UNIX, Windows, Mac OS.
- Developer specifies what needs to be done via platform independent *tasks*.
  - Don't have to worry about platform specific shell commands.
  - Developers don't't' need to worry about what platform their build is running on.

Apache Ant (Cont'd)

```
...
 3       <target name="all">
 4           <mkdir dir="pkg"/>
 5           <jar basedir="obj" destfile="pkg/prog.jar"/>
 6           <copy file="index.txt" tofile="pkg/index.txt"/>
 7       </target>
...
```

> Line 4: Create directory
> Line 5: Take all object files and package them into a jar.
> Line 6: Copy index file into pkg directory.

Apache Ant Pros

- **Cross-platform support** - no shell specific language components means developers don't have to worry about platform shell support.
- **Hidden dependency analysis** - dependency is handle within each task.
- **Easy to learn** - Simple language constructs and grouping based on tasks. Complexity hidden in tasks.
- **3rd Party Support** - widest range of Java compilation tool and language support.

Apache Ant Pros (Cont'd)

- Critical build system features are standard
  - automatic dependency analysis
  - multi directory support

Apache Ant Cons

- Lack of full programming language - not scriptable and therefore difficult to express complex activities.
- XML - ...
- No shell commands and only for Java.
- No persistent state from build to build, dependency analysis invoked each time tool is run.
- Evaluation:

Convenience	Correctness	Performance	Scalability
Good	Excellent	Good	Excellent

Apache Ant Alternatives

- NAnt - similar to Ant except that it focuses on .NET languages.
- MSBuild - successor to NMake and in some ways very similar to Ant.
  - Visual Studio auto-generates .proj files (equivalent to build.xml for Ant).

SCons

- Uses Python as its description language.
- Version also exist for using Perl (Cons) or Ruby (Rake) as description languages.
- Describe the build process using a sequence of method calls to determine which objects to create and what input files to use.
- Use for C and C++ languages.
- If starting a new C/C++ project consider using SCons rather than Make.

Dev vs. Prod Build Example

```
 1  env = Environment()
 2
 3  if ARGUMENTS.get('production', 0):
 4     env['CFLAGS'] = '-O'
 5     env['CPPDEFINES'] = '-DPRODUCTION'
 6     myLibraryBuilder = env.SharedLibrary
 7  else:
 8     env['CFLAGS'] = '-g'
 9     env['CPPDEFINES'] = '-DDEBUG'
10     myLibraryBuilder = env.StaticLibrary
11
12  myLib = myLibraryBuilder('libcalc',
13         ['add.c', 'mult.c', 'sub.c'])
14
15  env.Program('calculator', 'calc.c', LIBS = [myLib])
```

> Setting flags based on whether the developer wants to build for development or production.

SCons Pros

- **Uses general purpose language** - Python is a syntactically simple language and easy to learn for beginners.
- **Simple build construction** - not much overhead results in quick build definition for simple programs.
- **Builder method portability** - the methods hide the compilation tools. Developer can focus on writing the build instead of worrying about which tools are installed.

SCons Pro

- **100% Python** - everything is done in Python so you don't have to switch languages if you want to do a shell script.
- **Focus on correctness** - one of the main goals of SCons is to be correct; e.g., use md5 file checksums to see if file has changed
- **Active development** - young tool but is actively developed; bugs fixed quickly and new features added on a regular basis.

SCons Pro

- **Debugging** - several debugging options that allow you to more easily narrow down problems in your build.

SCons Cons

- **Slow** - The focus on correctness makes the build process slow. More of a problem when doing incremental builds.
- **Language Support** - good for C/C++ but not so much for Java or C#.
- **Memory footprint** - in certain circumstances uses more memory than other build systems such as Make.
- **Evaluation**:

Convenience	Correctness	Performance	Scalability
Excellent	Excellent	Good	Good

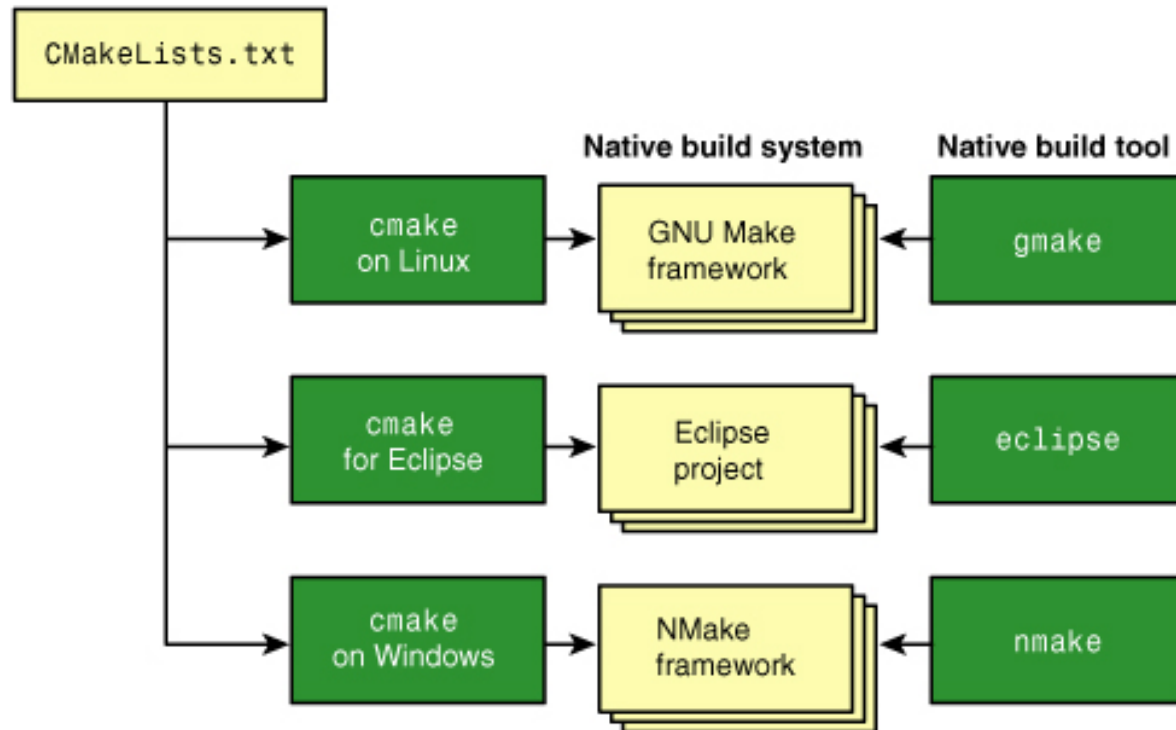
SCons Alternatives

- As mentioned earlier: Cons (Perl), Rake (Ruby)
- Cons was the original inspiration behind SCons.
  - not developed since 2001
  - Cons website recommends to use SCons
- Rake is based on the Ruby language.
  - No automatic dependency analysis.
  - Follows more the GNU Make model where a developer specifies the source, dependencies, and commands to execute.

CMake

- Differs from other tools so far in that it doesn't actually execute the build process.
- Translates a high level build description into a format accepted by other tools.
  - high level description -> GNU Make tool.
- CMake generators are supported by most platforms and languages.

CMake High Level View



> On Linux its default behaviour is to use a makefile based framework.
> If you can tell it to make Eclipse related project files.
> On Windows its default behaviour is to use the Visual Studio Compilers and NMake.

CMake Flavour

```
1  project (basic-syntax C)
2
3  cmake_minimum_required (VERSION 2.6)
4
5  set (wife Grace)
6  set (dog Stan)
7  message ("${wife}, please take ${dog} for a walk")
8
9  set_property (SOURCE add.c PROPERTY Author Peter)
10 set_property (SOURCE mult.c PROPERTY Author John)
11 get_property (author_name SOURCE add.c PROPERTY Author)
12 message("The author of add.c is ${author_name}")
```

> Line 1: a name to uniquely identify build.
> Line 2: min version of CMake required.
> Line 5-6: variable declaration and setting <name> <value>.
> Line 9-10: setting the properties of a file on disk.

CMake Pros

- **Single description file** - generate builds for many build systems and platforms from one description file.
- **Ease of use** - description language syntax is easy to grasp even for beginners.
- **Quality** - the target build system are of high quality (correctness); one primary focus of CMake.
- **Integration** - easy to build end-to-end build systems using CPack (packaging) and CTest (testing).

CMake Cons

- **Limited complexity** - the auto-generated build systems lack some features. If you're goal is a complex build system then doing it natively is recommended.
- **Yet another language** - although the syntax is relatively simply it's another language/framework you have to learn since it doesn't leverage other languages/tools.

CMake Cons

- **Documentation** - not as readable as for other tools. Examples are either difficult to follow are out of date with current version.
- **Cross-platform** - although it does support cross platform development you may still need to tinker with the native build tool.
- **Evaluation**:

Convenience	Correctness	Performance	Scalability
Good	Excellent	Excellent	Excellent

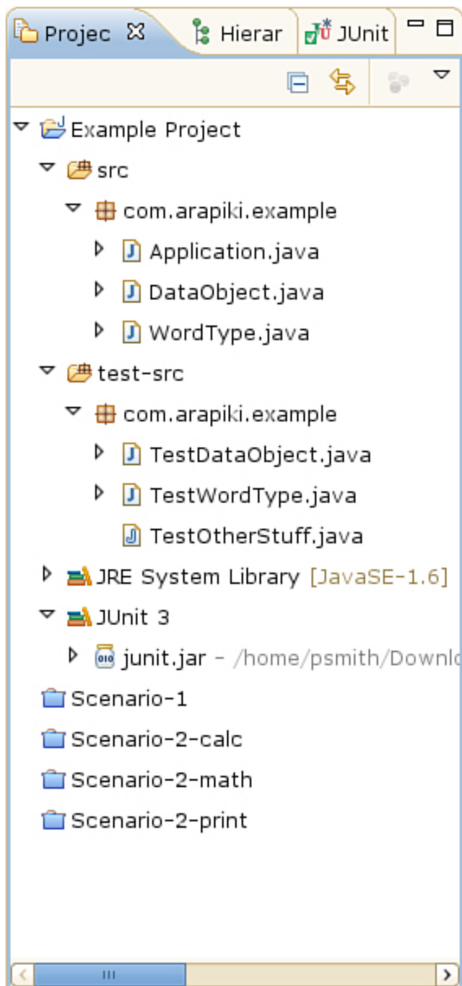
CMake Alternatives

- Automake - part of the Autotools suite.
  - creates a makefile based on a high level description of the build process
  - tightly coupled with GNU development environment -> UNIX type systems only.
- Qmake - part of Qt development environment
  - Qt is designed for cross-platform development and as a result so is Qmake.

Eclipse

- It's an IDE but also a build tool.
  - The build tool is one of its many widgets.
- It can also interface with external build tools if required.
- Works with Java, C/C++, Python, Perl, PHP, UML ...
- Lots of the build aspects are hidden from the developer.
  - The IDE is able to infer the build setup from the structure of the software.

Eclipse Files




```
src/com/arapiki/example/Application.java
src/com/arapiki/example/WordType.java
src/com/arapiki/example/DataObject.java
bin/com/arapiki/example/WordType.class
bin/com/arapiki/example/DataObject.class
bin/com/arapiki/example/Application.class
test-src/com/arapiki/example/TestDataObject.java
test-src/com/arapiki/example/TestOtherStuff.java
test-src/com/arapiki/example/TestWordType.java
test-bin/com/arapiki/example/TestDataObject.class
test-bin/com/arapiki/example/TestWordType.class
.project
.settings/org.eclipse.jdt.core.prefs
.classpath
```

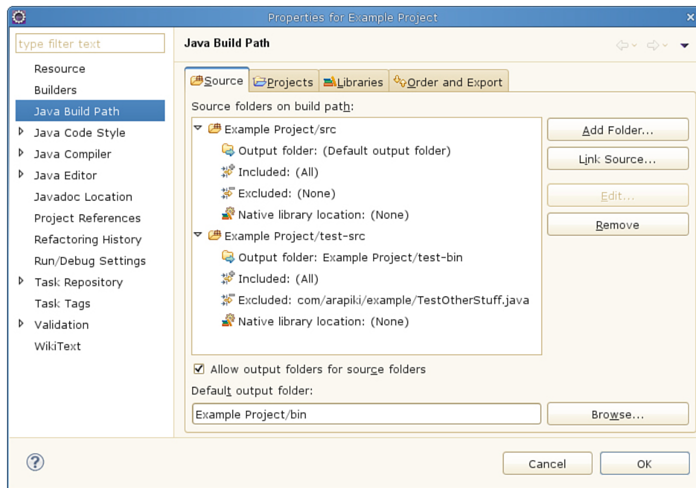
> What you see is not what you get. Eclipse auto-generates folders for the source and folders for the builds.
> This is fine because developers don't care about .class files. They care about the source and execution. Eclipse takes care of the build.

Eclipse Files .project

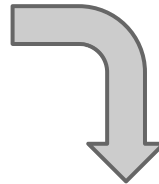
```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
        <name>Example Project</name>
        <comment></comment>
        <projects>
        </projects>
        <buildSpec>
               <buildCommand>
                     <name>org.eclipse.jdt.core.javabuilder</
                     name>
                     <arguments>
                     </arguments>
               </buildCommand>
        </buildSpec>
        <natures>
              <nature>org.eclipse.jdt.core.javanature</
              nature>
        </natures>
</projectDescription>
```

> Auto-generated by Eclipse
> Expresses how the project should be configured.

Eclipse Files .classpath



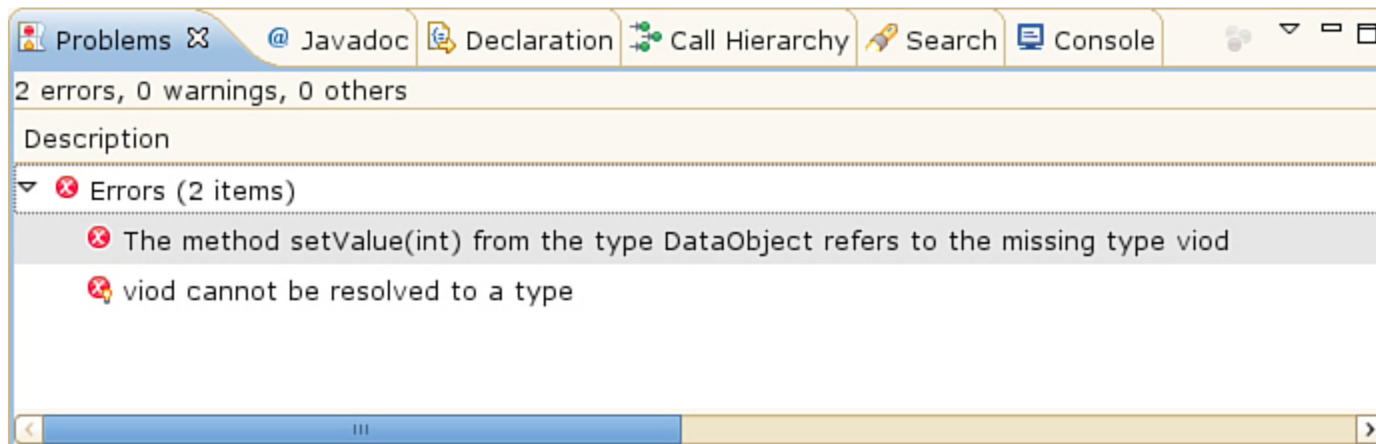
> Again, auto-generated by Eclipse
> Describes how to build the project.
> Manageable via GUI.



```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
   <classpathentry kind="src" path="src"/>
   <classpathentry excluding="com/arapiki/example/TestOther-
Stuff.java"
                  kind="src" output="test-bin" path="test-
                  src"/>
   <classpathentry kind="con"
         path="org.eclipse.jdt.launching.JRE_CONTAINER/org.
         eclipse. \
         jdt.internal.debug.ui.launcher.StandardVMType/Java-
         SE-1.6"/>
   <classpathentry kind="con"
         path="org.eclipse.jdt.junit.JUNIT_CONTAINER/3"/>
   <classpathentry kind="output" path="bin"/>
</classpath>
```

Eclipse Build

- All done through the IDE.
- Every time you save a Java file, the file is compiled and the builder is invoked.
  - This is invisible to you.
- Errors or warnings weather from compilation or build are displayed right away.



Eclipse Pros

- **No description files** - if you use Eclipse JDT everything is done for you and accessible via a GUI.
- **Integration** - compilation and build are integrated into one tool.
- **Wide project support** - many languages and frameworks are supported via plugins. The project plugins are aware of the compilation and build tools required.

Eclipse Cons

- **Complexity** - too many buttons and dialogue boxes; you have to find out where things are hidden.
- **CPU & Memory** - relatively speaking, requires more CPU and memory than other build systems. Not really noticeable on small projects.
- **Build Process** - builds are automatically incremental. Ties the build process tightly with developers workflow.

Eclipse Cons (Cont'd)

- **Hidden** - the build process is hidden from the developer.
- **Build complexity** - more complex build workflows require external tools or additional plugins.
- **Evaluation**:

Convenience	Correctness	Performance	Scalability
Good	Excellent	Good	Poor

Eclipse Alternatives

- CDT for Eclipse
  - C/C++ development and tools
    - compilers, linkers
    - builders
- Other IDE's are also available with varied levels of automation as far as builds are concerned:
  - Visual Studio
- Cloud based IDE's
  - Cloud 9
  - ...

Fin