CS 130 SOFTWARE ENGINEERING

BUILD MANAGEMENT

Professor Miryung Kim

UCLA Computer Science

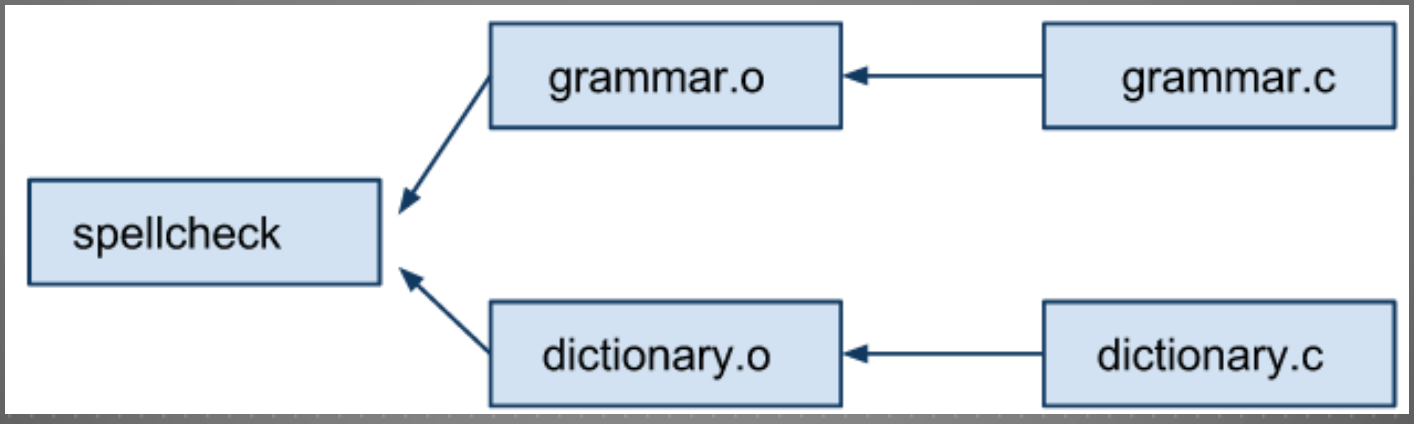
Based on Materials from Miryung Kim, Christine Julien and Adnan Aziz

INTRODUCTION

- A program cannot just be immediately executed after a change
  - It looks like this to you when you use Eclipse, but Eclipse is not doing you any favors by pulling the wool over your eyes…
- Intermediate steps are **required** to **build** the program
  - Must convert source to object
  - Must **link** objects together to create program
- Other important aspects can also be affected by changes
  - E.g., documentation generated from program text
- Definitions:
  - **Source components** are those that are created manually (e.g., Java, Latex)
  - **Derived components** are those created by the computer/compiler (e.g., bytecode, PDF)

AN EXAMPLE

- Imagine a program `spellcheck` that consists of two components: `grammar` and `dictionary`
- Each component starts off in a C source file (`.c`)
  - Each source file is converted to an object file (`.o`) using the C compiler (e.g., `cc`)
  - Object files are linked into a final executable



INCREMENTAL CONSTRUCTION

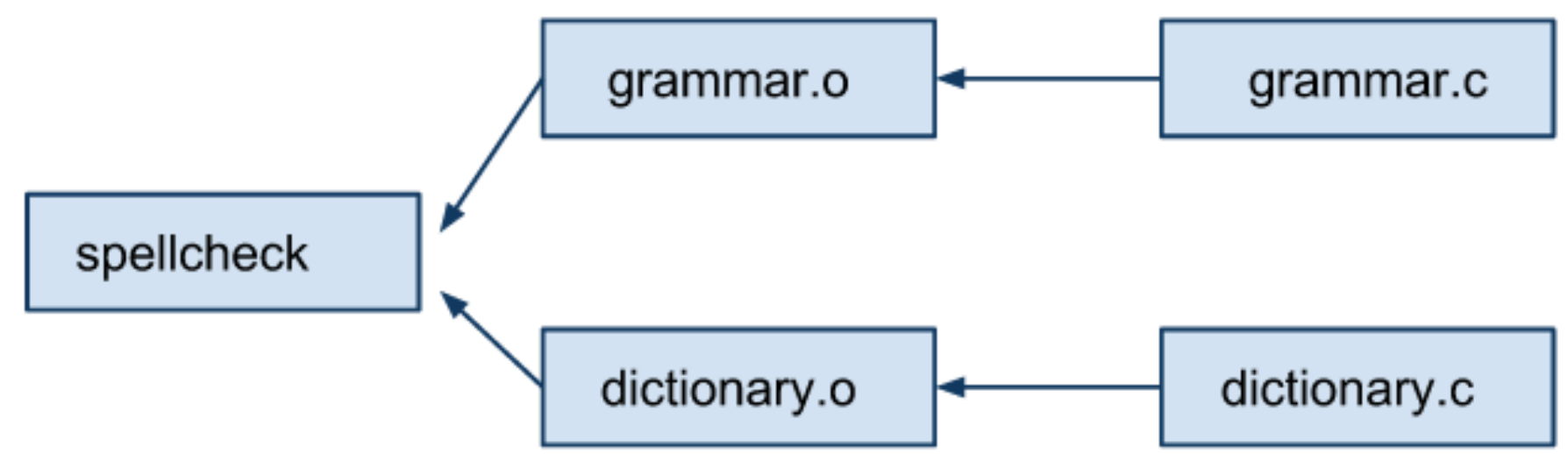
- Easiest approach: create a script
  - `cc -c -o grammar.o grammar.c`
  - `cc -c -o dictionary.o dictionary.c`
  - `cc -o spellcheck grammar.o dictionary.o`
- Problem: all steps are executed every time
  - Even if you just made changes to one of the source files
- Notice:
  - A change in component A can only impact components that are **derived** from component A

INCREMENTAL CONSTRUCTION (CONTINUED)

- **Rebuild Theorem:** Let A be a derived component depending on components $A_1, A_2, \ldots, A_n$. The component A has to be rebuilt if:
  - A does not exist;
  - At least one A_i from $A_1, A_2, \ldots, A_n$ has changed; or
  - At least one A_i from $A_1, A_2, \ldots, A_n$ has to be rebuilt

BACK TO THE EXAMPLE…

- `grammar.o` has to be rebuilt if `grammar.c` has changed
- `spellcheck` has to be recreated because `grammar.o` has to be rebuilt



MAKE

- Created by Stuart Feldman from Bell Labs in 1975
- One of the most influential and widely used software tools
- MAKE realizes incremental program construction using a **system model**
  - A description of software product that lists individual components, their **dependencies**, and steps needed for their construction
  - MAKE's system model is specified in a file (usually called `Makefile`), which consists of a set of rules
  - Rules indicate component dependencies and commands needed to build components

A MAKE RULE

```
target1 target2 … targetn: source1 source2
sourcem
    command1
    command2
    …
    commandL
```

Example

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o
```

A MORE COMPLETE EXAMPLE

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

Suppose `dictionary.c` has changed

```
$ make spellcheck
cc -c -o dictionary.o dictionary.c
cc -o spellcheck grammar.o dictionary.o
```

MAKE ALGORITHM

- Basic algorithm: from `Makefile` and target A_0, calculate dependency graph and run **depth first search** (DFS)
  1. Suppose A is the target component
     - From the graph, determine components $A_1, A_2, \ldots, A_n$ that A depends on
  2. Call algorithm on $A_1, A_2, \ldots, A_n$
  3. If one of $A_1, A_2, \ldots, A_n$ has changed or if A does not exist, rebuild A
- **Question**: how does MAKE know if a file has been changed?
  - Use time stamps from the file system

MAKEFILE:VARIABLES

- MAKE provides a number of properties that increase flexibility and reduce verbosity
- You can use **variables** to store values
  - Refer to variables using $(var) or ${var}

```
CC = cc
OBJECTS = grammar.o dictionary.o
spellcheck: $(OBJECTS)
    $(CC) -o spellcheck $(OBJECTS)
```

```
$ make CC=gcc spellcheck
```

MAKE: MULTIPLE COMMANDS

- In standard practice, MAKE uses the same command to create or update a target, regardless of which file changes
- Consider a library and replacing a portion of its code
- MAKE allows a special form of the dependency, where the action specified can differ, depending on which file has changed:

```
target :: source1
    command1
target :: source2
    command2
```

- If `source1` changes, target is created or updated using `command1`; if `source2` is modified, `command2` is used

ANT

ANOTHER NEAT TOOL

- Rather than specifying commands, an ANT user specifies tasks that realize a specific target. Each task knows which tools and commands to use to realize the target

ANT

- Compilation: Javac (compile a JAVA program), JspC (execute JSP compiler)
- Archives: Jar (create JAR archive), Zip (create Zip archive)
- Documentation: Javadoc (create JAVADOC documentation)
- File management: Checksum (create check sum), COPY (copy files), Delete (delete files), Move (rename files), Mkdir (create directory), Tempfile (create temporary file)
- Test : JUnit (Run tests)

BUILD FILE IN ANT

- A buildfile has exactly one project. Every project has a name and a default target---a target that is used unless some other target is specified.
- Every project has one or more targets. ANT targets are more coarse-grained activities.
- Every target has a name and optional dependencies
- Every target is realized by a number of tasks

ANT BUILDFILE

- A **buildfile** provides the system model, written in XML (`build.xml`)
  - One **project**: name and default **target**
  - Project: one or more targets
  - Target: name and optional dependencies
    - Coarser than MAKE targets (an ANT target refers to an activity)
    - Target can consist of a number of **tasks**
- A great example:
  - The buildfile distributed with Google AppEngine
  - https://developers.google.com/appengine/docs/java/tools/ant?csw=1#The_Complete_Build_File

ANT BUILDFILE EXAMPLE

```
<project name="SimpleProject" default="dist">
  <target name="compile">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes"/>
  </target>
  <target name="dist" depends="compile">
    <mkdir dir="lib"/>
    <jar jarfile="lib/simple.jar" basedir="classes"/>
  </target>
  <target name="clean">
    <delete dir="classes"/>
    <delete dir="lib"/>
  </target>
</project>
```

```
$ ant
Buildfile: build.xml
compile:
    [mkdir] Created dir:
/Users/christinejulien/temp/antexamples/classes
    [javac] Compiling 1 source file to
/Users/christinejulien/temp/antexamples/classes

dist:
    [mkdir] Created dir:
/Users/christinejulien/temp/antexamples/lib
    [jar] Building jar:
/Users/christinejulien/temp/antexamples/lib/simple.jar
BUILD SUCCESSFUL
Total time: 3 seconds
```

UNDERLYING ANT PROCESS

- The default target of `build.xml` is `dist`
- The target `dist` depends on compile, so compile must be realized first
- The compile target is realized by the two tasks `mkdir` and `javac`, which must be executed first
- Now the tasks of `dist` can follow: `mkdir` and `javac`
- All targets are realized an the build was successful

ANT SUMMARIZED

- Like MAKE, ANT works **incrementally**
  - With every new build, only those targets are realized whose dependencies have changed
- Unlike MAKE, incrementally is not part of the tool
  - Instead it is realized by the individual **task** implementations
  - E.g., javac task determines dependencies between Java classes automatically and builds just those classes that must be reconstruted
- ANT is a framework; it can easily be extended by additional tasks
  - By subclassing the Task class
- Can configure at runtime (e.g., replace original javac)

INCREMENTAL CONSTRUCTION

- ANT works incrementally- with every new build, only those targets are realized whose dependencies have changed.
- Incrementality is not built into the tool but must be realized by the individual task implementation.

ANT REMARKS

- Huge benefit: **portability**!
  - `MAKE UNIX: rm -rf classes/`
  - `MAKE Windows: rmidir /S /Q classes`
  - `ANT: <delete dir="classes"/>`
- Use of XML, which is hierarchical, partly ordered, and pervasively cross-linked, can be a barrier to entry
- Backwards compatibility is not high
  - E.g., older tasks such as `<javac>`, `<exec>`, and `<java>` use default values for options that are not consistent with more recent versions
- Provides limited fault handling rules and no persistence of state
  - Cannot use ANT as a workflow tool for any workflow other than build and test

JAVA DOC

SOFTWARE DOCUMENTATION

- Nowadays, a programmer usually spends more than half of her time to simply trying to understand software.
- A line of code is read a hundred times more often than it is written or changed!
- The creation of a program also requires the preparation of the program documentation.
- One of the most unpopular tasks faced by the programmer.

JAVA DOC

- JAVADOC extracts documentation from special comments and converts it into hypertext (HTML).
- The documentation becomes a part of the system and can be maintained and constructed with automated tools.
- A strong cohesion is established between documented parts of a program and their documentation.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute
{@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
@param  url  an absolute URL giving the base location of the image
@param  name the location of the image, relative to the url argument
@return      the image at the specified URL
@see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

getImage

public
Image **getImage**(URL url, String name)

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute URL . The name argument is a specifier that is relative to the url argument.
This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
url - an absolute URL giving the base location of the image
name - the location of the image, relative to the url argument
Returns:
the image at the specified URL
See Also:
Image

TAGS

- @author specifies the author of a class
- @version contains the version of a class.
- @deprecated marks methods, which will be removed in future versions.
- @since indicates the time, when the documented element has been introduced.

TAGS

- @see, @link creates cross-references to other elements
- @param documents the parameter of a method
- @return documents a method's return value
- @exception documents the possible exceptions, which can occur during the execution of a method. @throw is a synonym.

RECAP

- The aim of automatic program construction is to create all derived components from a number of source components.
- If a component has changed, all the components derived from it must be rebuilt.
- MAKE is a tool for building programs, during the course of which existing derived files are reused as much as possible.
- ANT allows specific extensions by the user as well as third parties. ANT is more task oriented.

RECAP

- Bad comments are worse than no comments!
- Keep comments brief and precise
- If the documentation becomes part of the source text, it can be maintained and edited with tools.
- It is easier to keep the documentation synced with source code.

QUESTIONS?