CRD

COMPUTATIONAL RESEARCH DIVISION

Tools and Techniques for Managing Large Scientific Software Projects

Keith Beattie, Chuck McParland, Dan Gunter, Guillaume Egles, Matt Rodriguez

Distributed Systems Department
LBNL

May 13, 2005

Office of Science
U.S. DEPARTMENT OF ENERGY

Schedule

- **Introduction – Keith (10 mins)**
- **Design – Chuck (30 mins)**
- **Coding – Dan, Keith & Guillaume (60 mins)**
- **Break (15 mins)**
- **Release – Dan & Matt (60 mins)**

Software Development Components

- Design
- Implementation
- Version Control
- QA
- Documentation
- Release Eng.
- Distribution
- User Interaction

Software Pollution

- **Write 100% of the code from scratch wherever possible**
- **Ensure LBNL obtains a proper license for non-LBNL code or developers <u>before</u> you invest time & money**
- **Keep a list of all non-LBNL code and developers used**
- **Keep a copy of all license agreements**
- **Contact Tech Transfer for a software "check-up" to ensure code is 100% "clean"**
  - **Tech Transfer will review IP agreements & help resolve IP issues from non-LBL software or funding.**
  - **Seth Rosen: SBRosen@lbl.gov, `http://www.lbl.gov/tt/`**

Design

- History
- Methodologies
- UML
- State Machines

Software Engineering
The *very* brief version

- Software development has not always been considered an engineering activity.
  - IBM creates “programmer” job in ’59.
  - LBL “calculators” were early programmers at lab.
- Need for engineering formalism grew because:
  - Integration with other engineering activities (telecom, “big” science, avionics, etc.)
  - Growth of IT importance in large organizations (how do we manage all these people?..and what is it they do?)
  - Growing experience with which parts of programming task are most important or difficult (design and documentation)

Software Project Elements

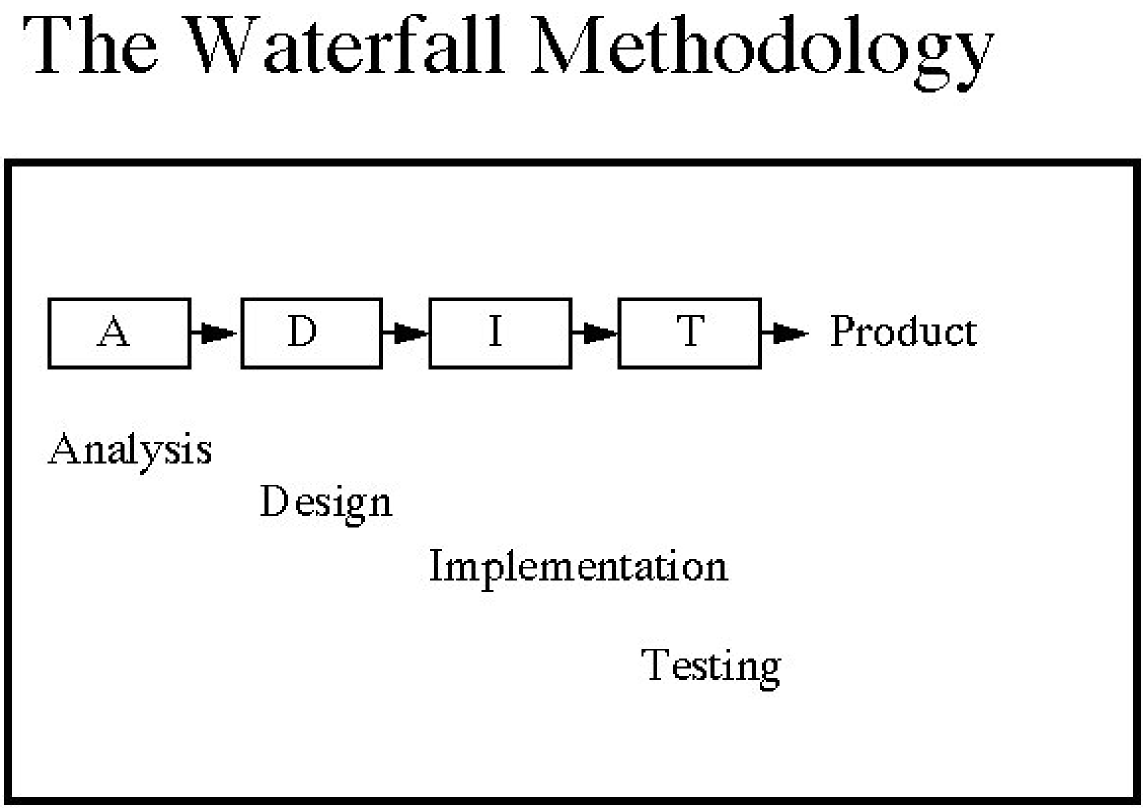
- **Requirements analysis and specification**
- **Design**
- **Implementation**
- **Integration and Test**
- **Maintenance**

➢ ***Different methodologies distribute these tasks along the project timeline in very different ways.***

Waterfall Methodology

Office of Science
U.S. DEPARTMENT OF ENERGY

Software Development Methodologies
Waterfall method



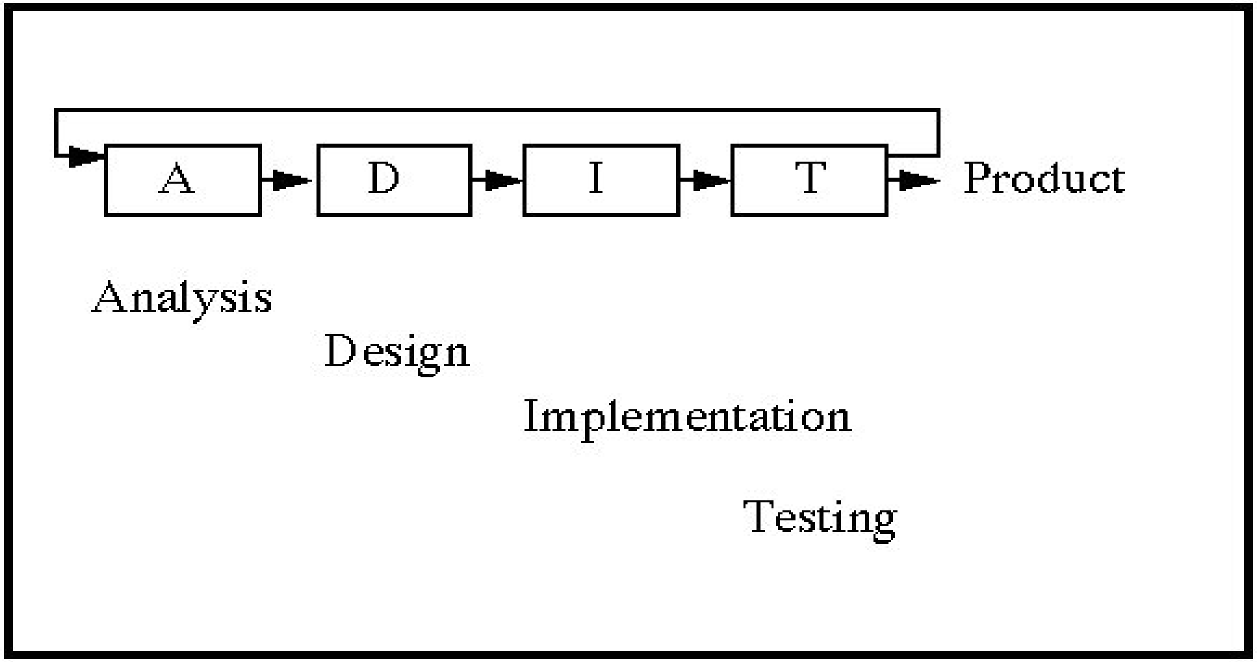
Waterfall method (contd.)

- Major steps:
  - Requirements analysis
  - Design
  - Implementation
  - Testing
  - Integration
  - Maintenance
- Good/Bad points:
  - Well structured/too inflexible
  - Analyze design up front/no chance to revisit design during implementation (i.e. difficult to swim upstream).
- Somewhat “out of fashion” at present.

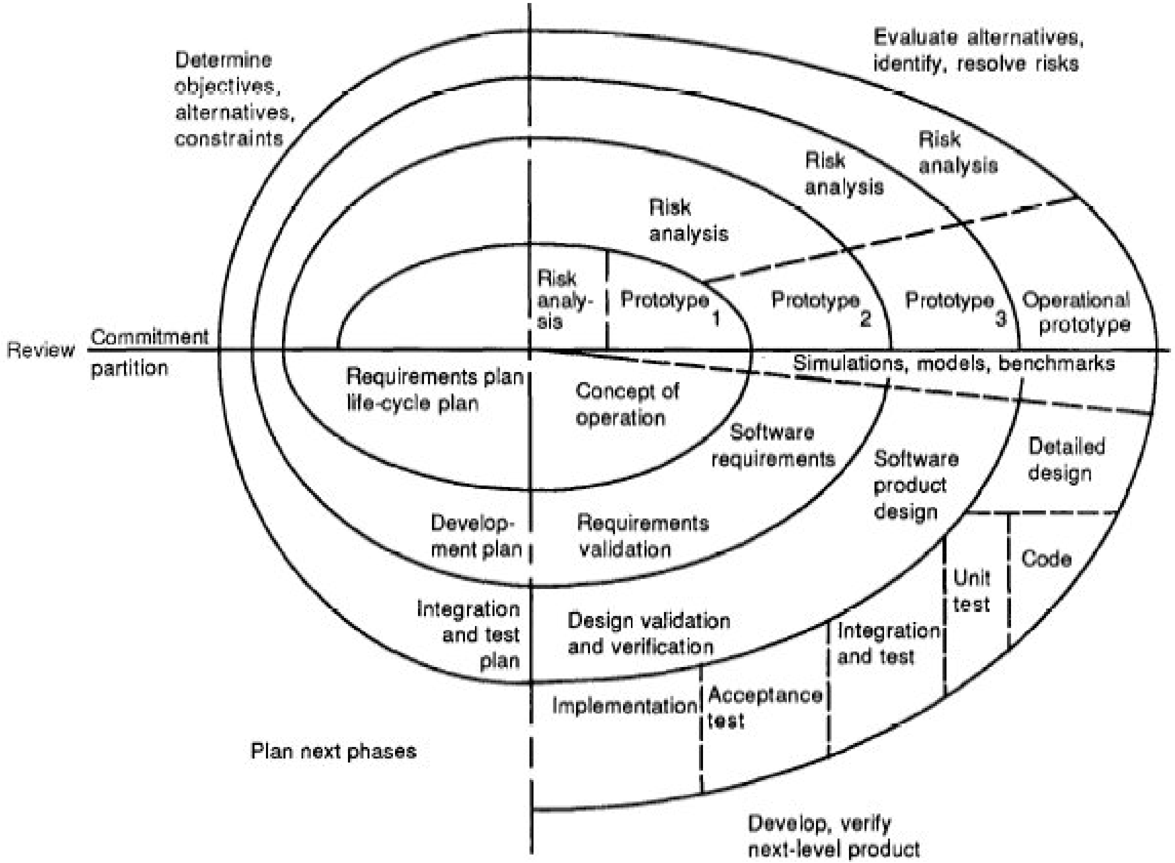
Spiral Methodology

Software Development Methodologies
Spiral method

The Spiral Methodology



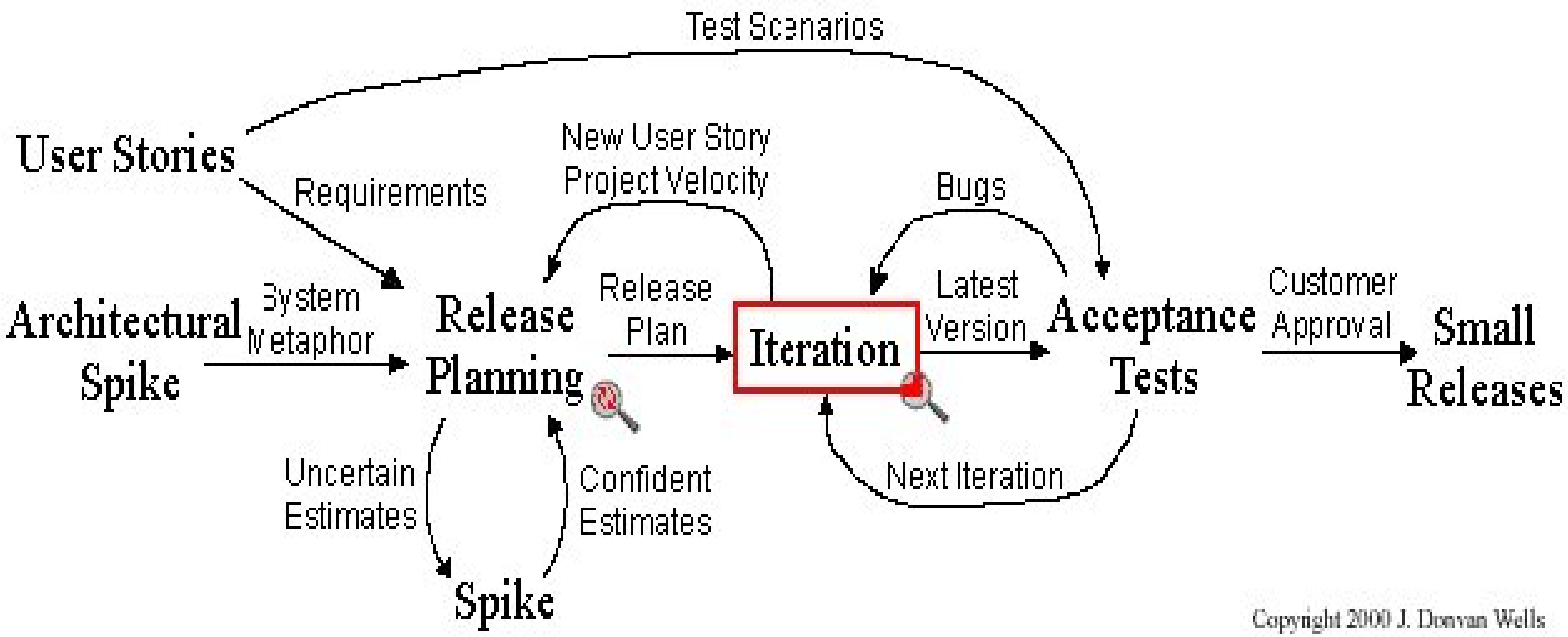
Software Development Methodologies
Spiral method



Software Development Methodologies Spiral method

- Iterative approach allows each task to be revisited each cycle.
  - Requirements can be re-assessed.
  - Code can be re-implemented.
  - Tests can be elaborated and improved.
- Allows “good enough for the moment” implementations.
- Shortens time between implementations and releases.
- Believed by many to be closer match to “real life” programming practice than waterfall model

Software Development Methodologies
Extreme Programming



Software Development Methodologies Extreme Programming

- Guiding principles
  - Small releases
  - Integrated testing
  - Continual refactoring and code improvement
  - Pair programming / collective code ownership
  - Continuous integration (daily)
  - On-site customer to elaborate requirements
  - 40-hour work week (!)

Software Development Methodologies XP variant: Agile

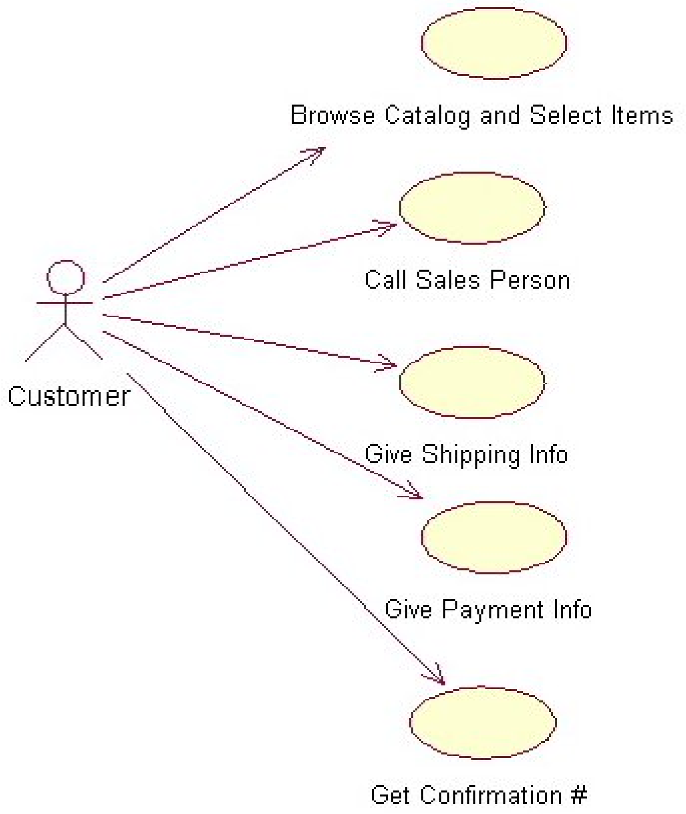
- Guiding principles
  - “Travel Light”
  - Rapid feedback
  - Embrace change
  - Quality work
  - Incremental and continual change
- In practice:
  - VERY customer driven.
  - No requirement is beyond change at any point in the development process.

But, you always need tools. UML-overview

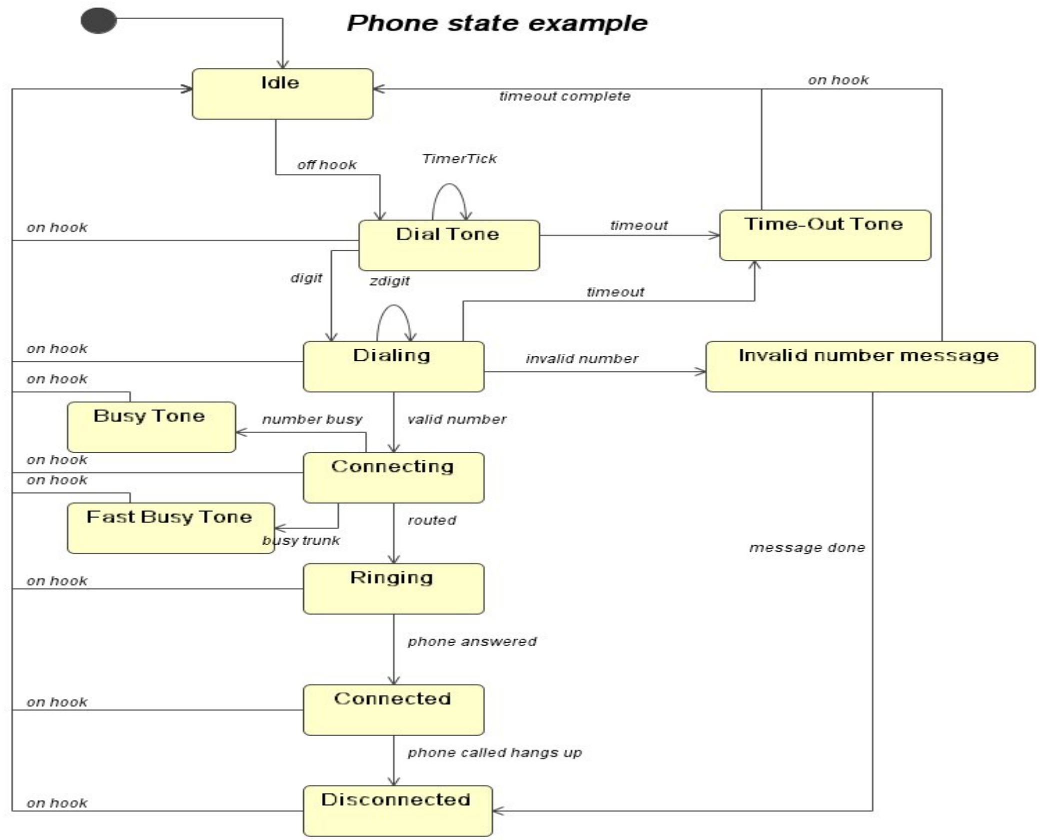
- Unify notations
- UML is a language for:
  - Specifying
  - Visualizing and
  - Documenting the artifacts of a system under development
- Authors (Booch, Rumbaugh and Jacobsen) agreed on notation but not able to agree on a single methodology
  - By itself, not a “unified” development environment
  - This is probably a “good thing”

UML-Use Cases

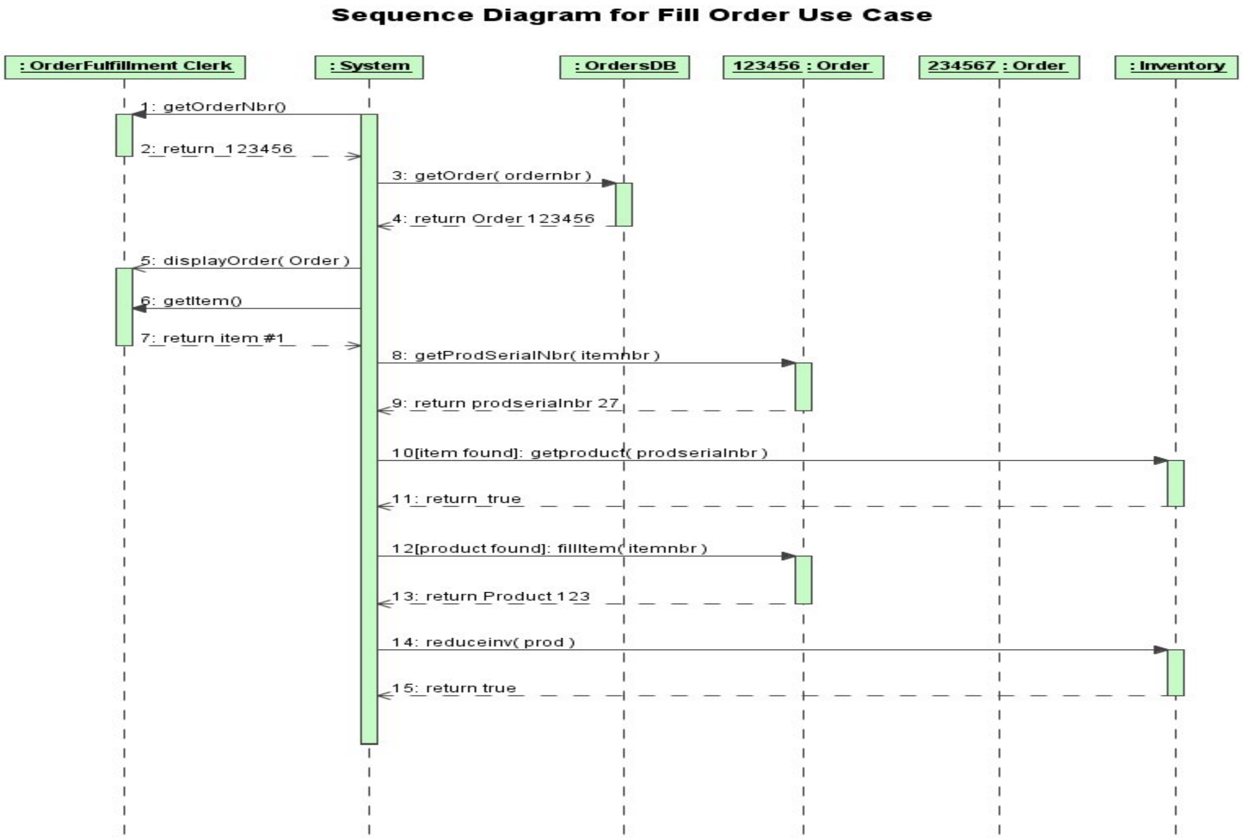
- **Use Case Name:**
- **Description:**
- **Actors:**
- **Assumptions:**
- **Steps:**
- **Variations:**



UML State Diagrams



UML-Sequence Diagrams



Software Methodologies
A practical approach

- Just as in other engineering disciplines, there is no “rote” way to engineer a system.
- Pick a methodology that fits your team and your schedule.
- Make use of available visual and conceptual tools (state diagrams, use cases, etc.)
- Don’t confuse the “map” with the “territory”.
  - Design diagrams and documents often lag behind current implementation.

Software State Machines

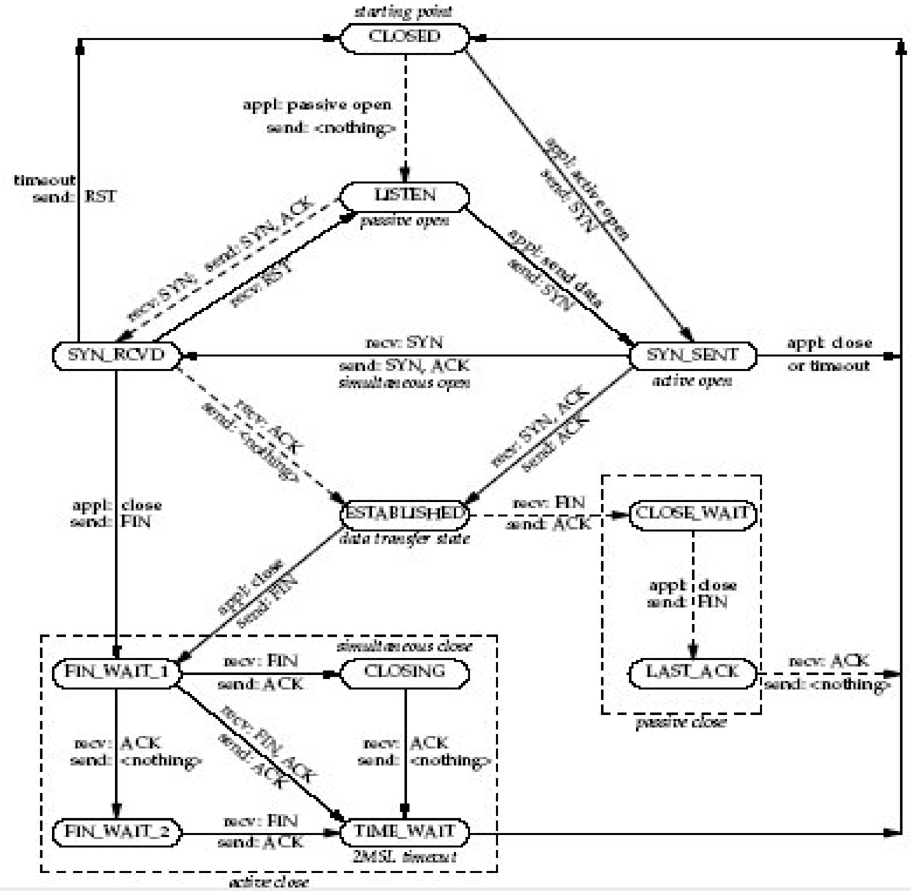
- Tool for describing behavioral modes of system operation (“stopped”, “calibrating”)
- Many variants, but basically:
  - Systems are in only one state at any given time.
  - Transitions between states are instantaneous.
  - Deterministic rules for moving between states.
  - ➢ Knowing a system’s state => knowing what a system is doing.

State machines are everywhere

- Dual heritage:
  - Early automata and math. logic. theory (Turing, et. al.).
  - Mechanical and electronic sequencing machine development.
- Natural fit in early LSI and VLSI electronics (processors, comm. Interfaces, etc.)
- Practical fit in software took hole with comm. protocol stacks.

State machines are everywhere

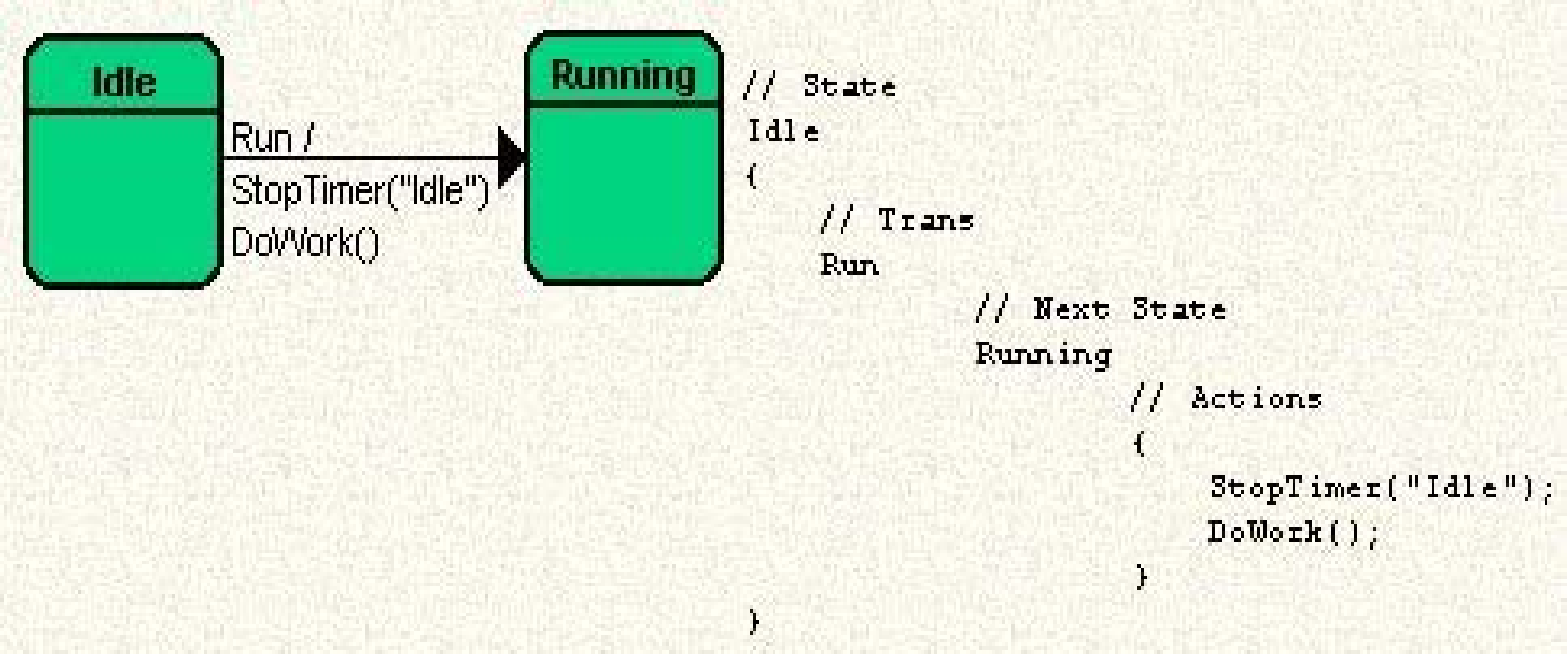
- TCP/IP comm. protocol states.
- Provides clear indication of protocol operational state.



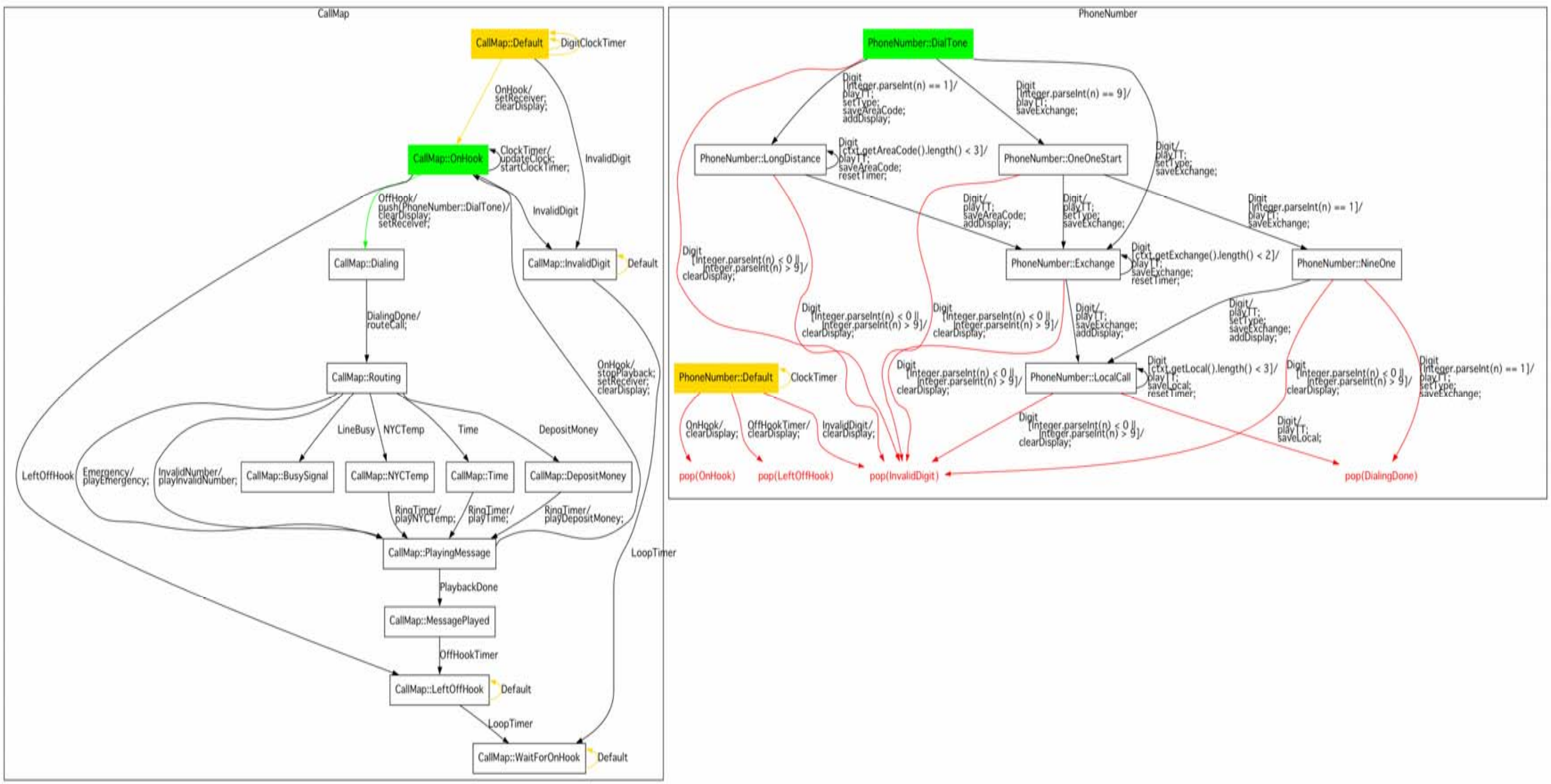
Motivation for state machine code generation tools.

- Separate state machine logic code from specific application code.
- Automate generation of state machine code to eliminate coding and debugging errors.
- Automatically generate documentation and state machine diagrams.

Simple state transitions



Elaborate State Machines



State machines are useful when...

- Describe program behavior in simple, partitioned manner (running, calibrating, etc.).
- Code littered with similar switch/case constructs based on global “state” variable.
- Need to synchronize distributed software components to act as single, integrated system (just what comm. protocols do with state machines.)

Start with XML description of system states and actions

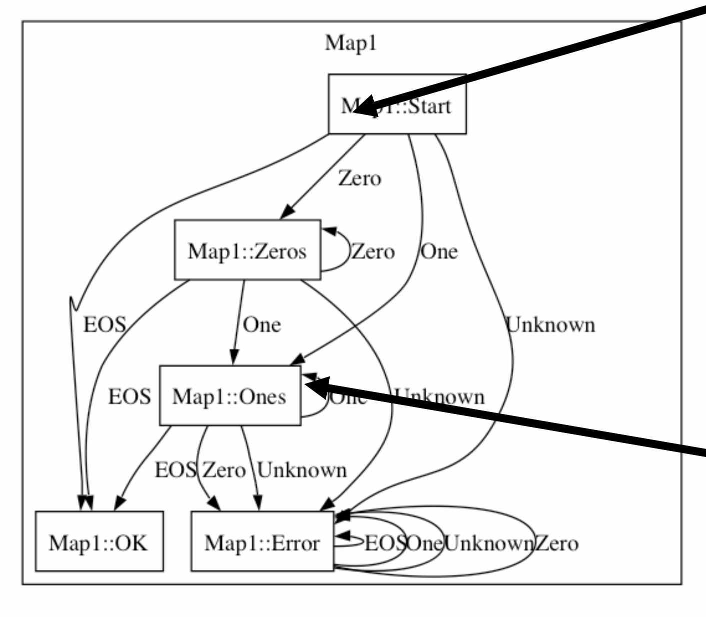
```
<State Name=“Idle">
    <Entry>
      <Action Cmd="enterIdle()"></Action>
    </Entry>
    <Transition Name=“StartSig">
      <NextState Name=“Running">
        <Action Cmd="LoadConfiguration()"/>
      </NextState>
    </Transition>
    <Transition Name=“OfflineSig">
      <NextState Name="Offline"> </NextState>
    </Transition>
```

Generate and execute state machine code

- In this example, we're using SMC (State Map Compiler http://smc.sourceforge.net)
- Pick a target language: C++, Java, Tcl, VB.net, C#
- Generate state machine code.
- Implement specific callback routines for your application.
- Link and execute.
- Give transitions ("events") to running state machine and system responds as designed.

Connecting to the state machine implementation

Application Code();



Application Code();

Benefits

- Simple mechanism for integrating valuable software engineering tool into you application.
- Auto generated graphics files (using Graphviz) provide system documentation.
- Code base easier to maintain….only application specific actions need to be coded.
- Monitor and control of system state more tractable for real time or long running batch applications.

Coding

- **Build Tools**
  - GNU Autotools (Dan: 10 mins)
  - Ant (Keith: 10 mins)
- **Version Control (Keith: 20 mins)**
- **IDE (Guillaume: 20 mins)**

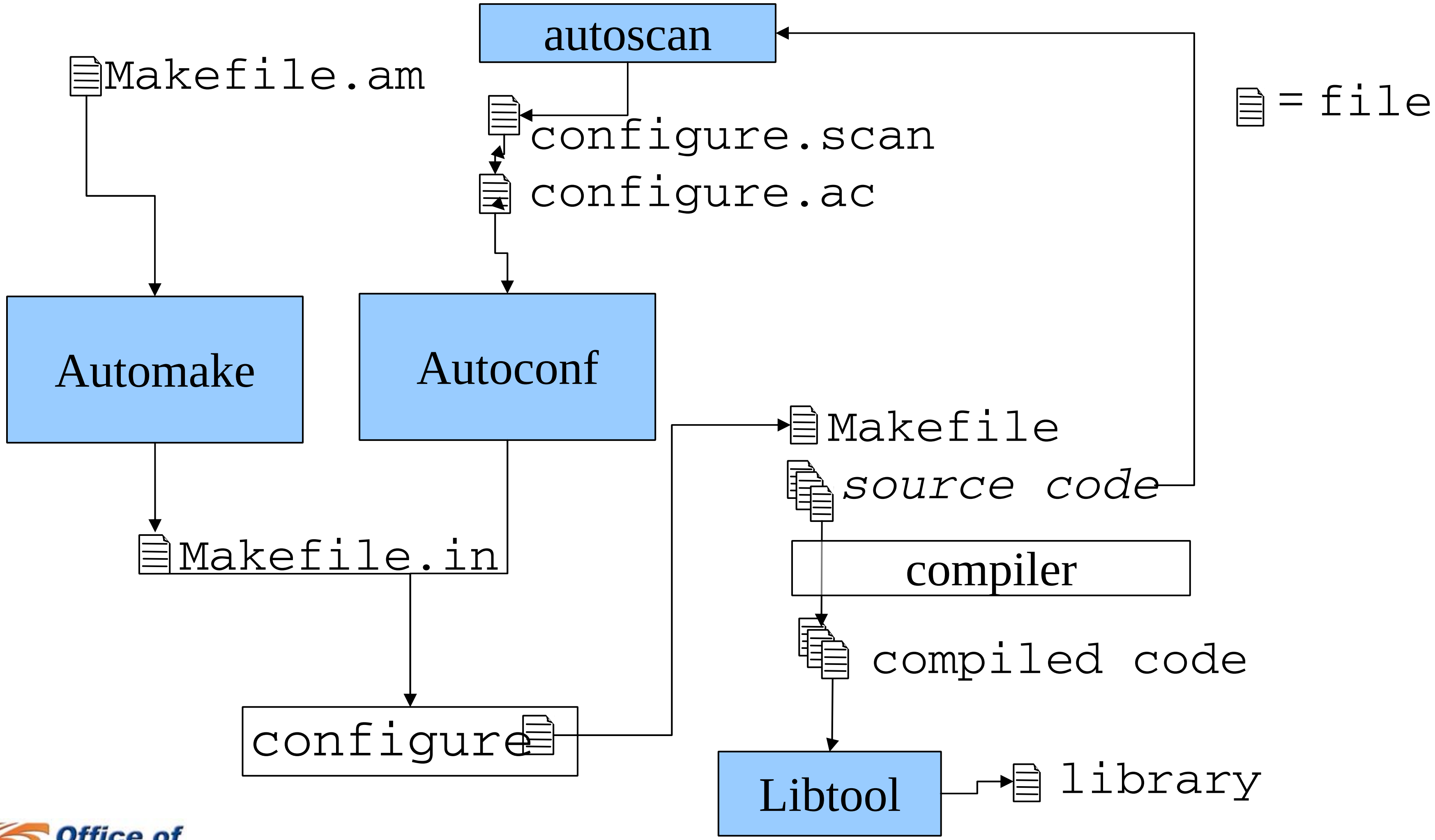
GNU “autotools”

- De-facto standard for portably building C, C++, and Fortran programs
- They support all UNIX platforms as well as Microsoft Win32
- Installation from source, by the user, requires only a Bourne shell and a compiler
- The developer requires other tools, such as Autoconf, Automake, Perl, and GNU m4

Autotools Philosophy

- Instead of enumerating platforms, test for platform characteristics
- Using this information, build (on each target system) an appropriate makefile and a header file, which can be used to build the package natively
- This is superior than writing various "#ifdef" for every system out there, or writing a different makefile for every system

Flowchart for a new Autotools Project



Sample project: files

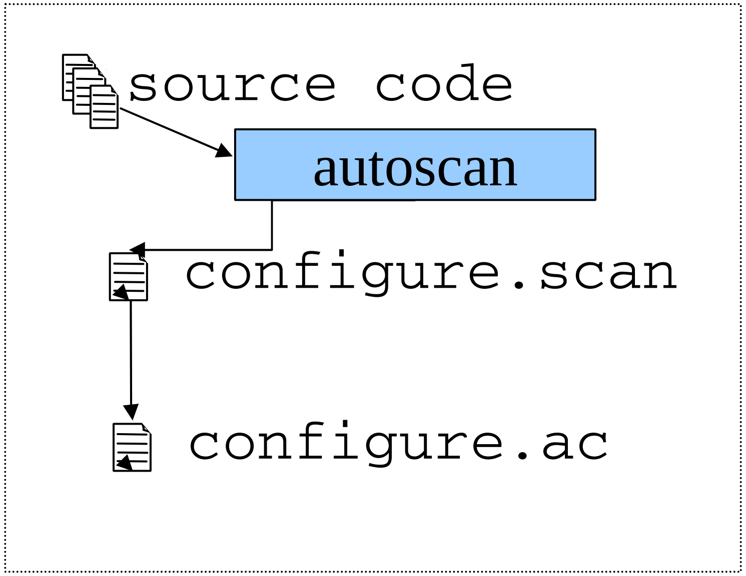
- **TOP**
  - **README**
  - **src/**
    - alg.c alg.h fmt.c fmt.h main.c
    - Makefile.am

1) Run *autoscan to generate a template configure.ac*

```
$ autoscan
autom4te: configure.ac: no such file \
or directory
autoscan: /usr/bin/autom4te failed \
with exit status: 1
```

Ignore this output, and:

```
$ cp configure.scan configure.ac
```

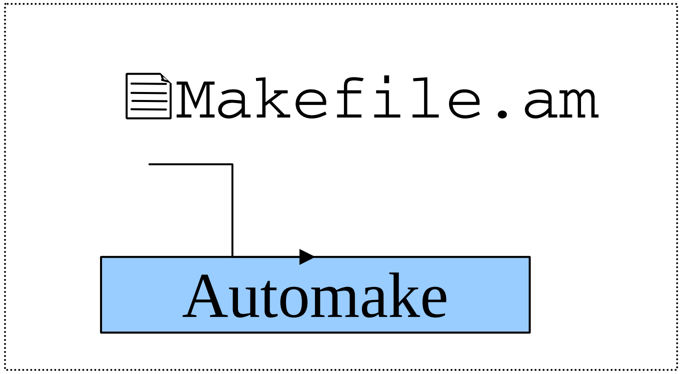


2) Edit generated *configure.ac*


```
AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AM_INIT_AUTOMAKE #add
AC_PROG_LIBTOOL #add
AC_CONFIG_SRCDIR([src/alg.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# FIXME: Replace `main' with a function in `-lm':
AC_CHECK_LIB([m], [main])
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])
# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
# Checks for library functions.
AC_CHECK_FUNCS([sqrt])
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

configure.ac

3) Write Makefile.am's



TOP:

```
SUBDIRS = src
```

TOP/src:

```
lib_LTLIBRARIES =  libsample.la
libsample_la_SOURCES = fmt.c alg.c
```

Build a shared library

```
bin_PROGRAMS = sample
sample_SOURCES = main.c
sample_LDADD = libsample.la -lm
```

Build a program that uses this library

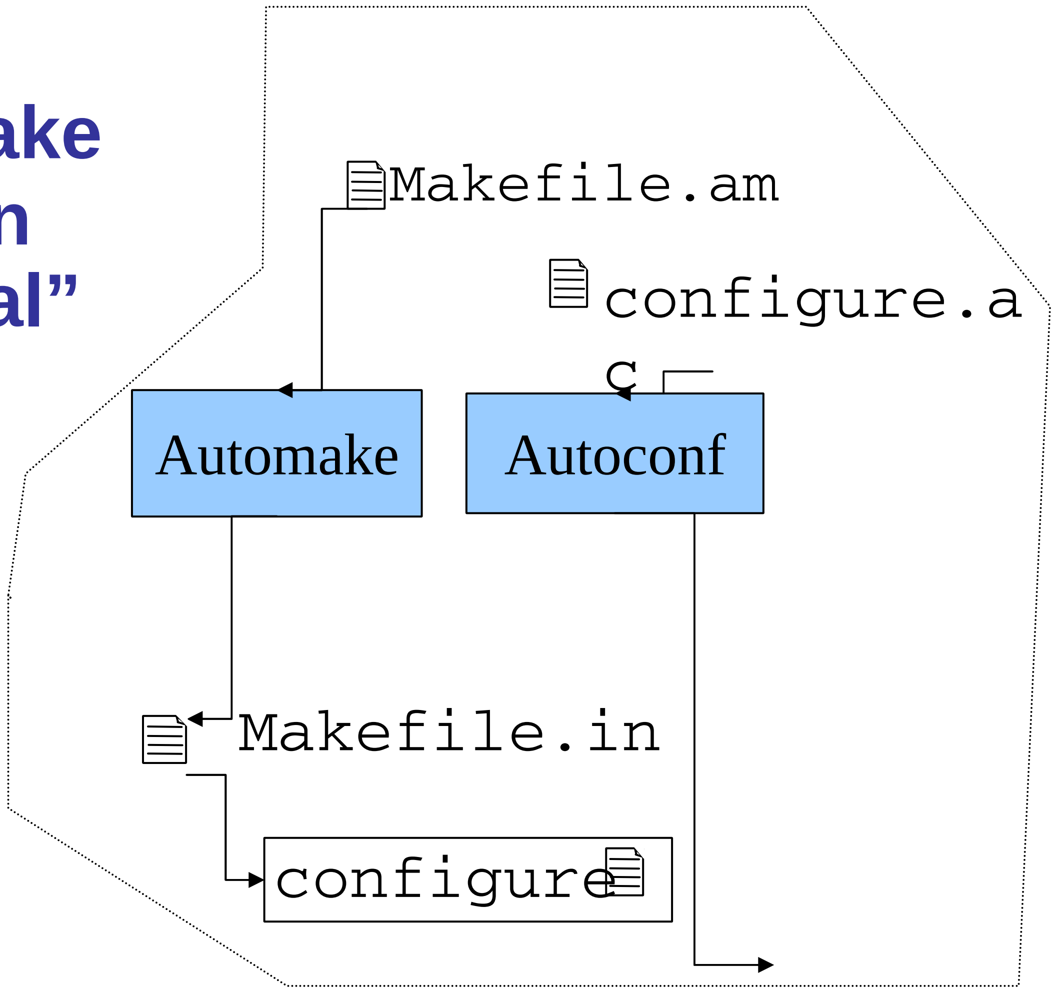
4) Init “helper” files

- `autoheader` – Generate “config.h.in”, the template file for “config.h”, which contains all #ifdefs in one place. Your programs will then just include config.h
- `libtoolize` – Copy over scripts needed to run libtool
- `aclocal-1.9` – Install system-specific macros for use by autoconf

Yes, this is a pain. But you only have to do it this once..

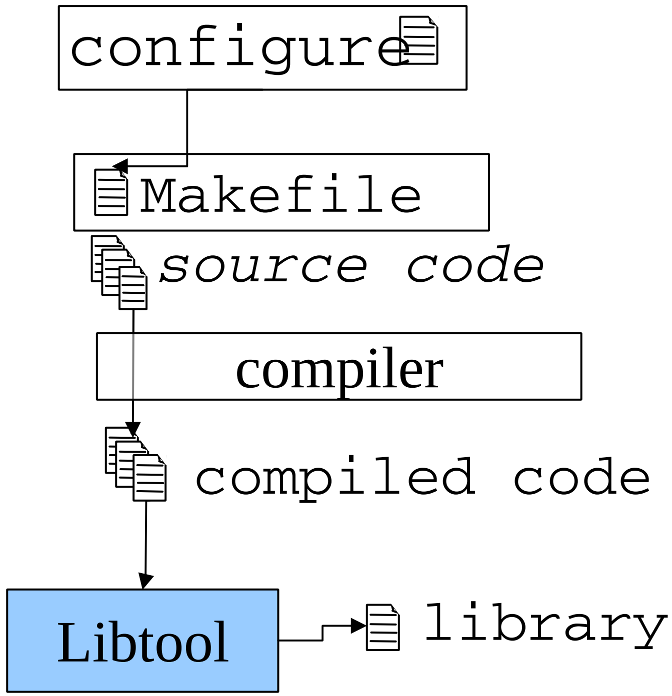
5) Run *autoconf* and *automake*

- **`autoconf`**
- **`automake-1.9` – Make sure that the version matches the "aclocal" version**



6) Build the project!

```
$ ./configure
$ make
$ make install
```



Generated Files

- The contents of your directory will now be full of generated files
- Don't worry, you won't need to understand them..
- Some of them are standard spots for project documentation – use them!
  - README - NEWS
  - INSTALL - AUTHORS
  - ChangeLog - COPYING

defaults to GPL -- change to LBNL license!!

The Payoff

- Building on new platforms is simpler
  - Many compiler / library characteristics automatically handled
  - Shared library differences handled by libtool
- Also, you can now run:
  - make dist – generate .tar.gz file that you users can download
- Finally, changes in “configure.ac” or “Makefile.am” are autodetected, so “make” will reconfigure/rebuild properly

Other languages

- The important point is to use the most standard build and configuration system for your particular language
  - for C/C++ and probably Fortran, this is autoconf
  - for Python, this is distutils
  - for Perl, it's Makefile.PL / CPAN
  - for Java, it's Ant (see next talk!)
- This allows you to leverage other people's experience, and provides the best chance of integrating new people and code in the project

Ant

Keith Beattie

Ant

"*Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make's wrinkles.*"

- **Create a `build.xml` file where tasks (targets) are definied**
- **Run '`ant [flags] target'` to execute tasks**
- **Ant flags:**
  - **h help with ant usage**
  - **p 'project help' list targets in build.xml**
  - **q 'quiet' no need to tell me everything**
  - **v 'verbose' good for debugging build.xml files**

Ant: Hello, World

```
<project name="hello" default="build" basedir=".">
   <property name="src.dir"   value="src"/>
   <property name="build.dir" value="build"/>

   <target name="build" depends="init"
           description="build everything">
      <javac srcdir="${src.dir}" destdir="${build.dir}"/>
   </target>

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

   <target name="clean" description="clean up">
      <delete dir="${build.dir}"/>
   </target>

</project>
```

Ant: Building

```
ksb@fuzz[Hello] 12:38:37 (0)$ ant
Buildfile: build.xml

init:
    [mkdir] Created dir: /home/ksb/Hello/build

build:
    [javac] Compiling 1 source file to /home/ksb/Hello/build

BUILD SUCCESSFUL
Total time: 1 second
ksb@fuzz[Hello] 12:38:39 (0)$
```

Ant: Jar & run

```
<target name="dist" depends="build"
                    description="generate the distribution" >
   <jar jarfile="${dist.dir}/hello.jar" basedir="${build.dir}">
      <manifest>
         <attribute name="Main-Class" value="hello.Hello"/>
      </manifest>
   </jar>
</target>

<target name="run" depends="dist"
                   description="Run the application" >
   <java jar="${dist.dir}/hello.jar" fork="true" />
</target>
```

Office of Science
U.S. DEPARTMENT OF ENERGY

Ant: run

```
ksb@fuzz[Hello] 22:05:03 (0)$ ant run
Buildfile: build.xml
init:
    [mkdir] Created dir: /home/ksb/Hello/build
    [mkdir] Created dir: /home/ksb/Hello/dist
build:
    [javac] Compiling 1 source file to
   /home/ksb/Hello/build
dist:
      [jar] Building jar: /home/ksb/Hello/dist/hello.jar
run:
     [java] Hello, world

BUILD SUCCESSFUL
Total time: 3 seconds
ksb@fuzz[Hello] 22:05:10 (0)$
```

Ant: task library

- Each task is a Java class
- Extensible
- Many, many tasks available
  - cvs
  - junit
  - javadoc
  - rpm
  - ssh

  ...

Ant: JUnit

```
<property name="junit.dir" value="junit-results"/>
<property name="test.class" value="hello.TestHello"/>

<target name="test" depends="build" description="unit test">
   <junit errorProperty="test.failed"
          failureProperty="test.failed">
      <test name="${test.class}" todir="${junit.dir}" />
      <formatter type="brief" usefile="false" />
      <formatter type="xml" />
      <classpath refid="classpath" />
   </junit>
   <fail message="Tests failed: check test reports."
         if="test.failed" />
</target>
```

Ant: JUnit run

```
$ ant test
Buildfile: build.xml

init:

build:
    [javac] Compiling 1 source file to /home/ksb/Hello/build

test:
    [junit] Testsuite: hello.TestHello
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 
   0.006 sec


BUILD SUCCESSFUL
Total time: 2 seconds
```

Version Control

Keith Beattie

Version Control

- **Why?**
  - Backup of source code
  - Central repository
  - History of changes
  - Monitor activity
  - Time Machine
  - Multiple Current Versions

Version Control Models

Check out, Modify, Commit

- Library Model
  - Lock
  - Modify
  - Unlock
- Concurrent Model
  - Copy
  - Modify
  - Update
  - Merge

Library vs. Concurrent

Library

- Pro
  - You 'own' the file
  - Easy to understand
- Con
  - Slow
  - Lock might need to be broken

Concurrent

- Pro
  - Scales
  - Fast
- Con
  - Merge
  - Unintuitive at first

Biased for Concurrent Model

- Conflicts
  - These are relatively rare
  - Not solved by either model
  - Some aren't exposed by the VC system

A VC system is not a replacement for:

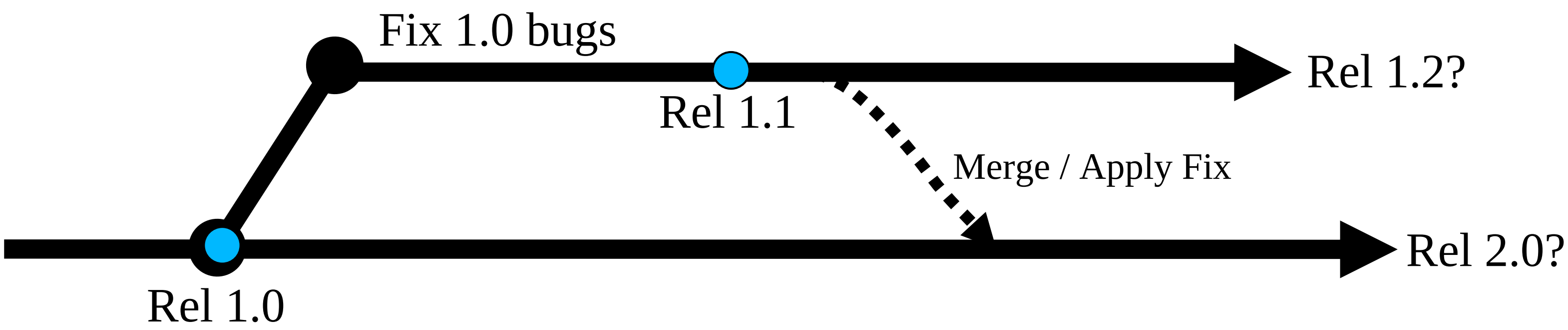
- Developer communication
- System Design
- Project management

Tags

- 'Snapshot' in time
  - Date
  - Name (which may not correspond to a single point in time)
- Known stable points
  - Release candidates
  - Certified Releases
- Easily retrieve old 'snapshots'

Branches

- What happens when an existing release needs a bugfix (and you've already started on the next one)?
  - Branch at release tag (time travel)
  - Work on this release branch, creating new point release
  - Merge into (or fix in) Trunk



Branches 2.0

- Release implies Branch
  - You might never actually branch but the implication & ability is there
- Multiple current lines of development
  - Powerful to allow both supporting existing releases and development of next release
  - As releases diverge, you'll want to 'end of life' old releases
- Project Branches
- Experimental Branches

CVS vs SVN

- Subversion is the CVS replacement
  - Directories, renames & meta-data are revisioned
  - Change sets (rather than file by file)
  - Atomic commits
  - Constant-time tagging & branching
  - Symlinks
  - Apache/WebDAV (http, security)

Integrated Development Environments

Guillaume Egles

The Problem

- Developing a project is not just writing code and compiling it. It's…
  - Writing
  - Debugging
  - Compiling
  - Testing
  - Running
  - Archiving/Versioning
  - Documenting
  - Releasing
  - Redesigning
  - Maintaining
  - …

The manual approach

- **Using separate tools**
  - **editor + compiler + debugger + doc tool.**
- **Advantages:**
  - **Better control**
  - **Flexibility**
- **Disadvantages:**
  - **Have to be an expert with each tool**
  - **Very hard to get the tools to work together**
- **Lots of time wasted AROUND the project instead of ON the project.**

The First Generation IDEs

- Visual Studio, CodeWarrior, KDevelop, …
  - Commercial ones are pricy
  - Free ones are not mature
- Usually only for one/two platform(s)
  - VS only on Win. KDevelop only on UNIX.
- Intrusive
  - Clutter your project with IDE-specific files.
- Limited
  - Provides less than the “separate tools” approach.
- Lack flexibility
  - Does not adapt to the need of your project well.
- It either fits your need perfectly or you are stuck !

The next generation IDE

- 3 forces at play: (Chicken or the egg)
  - Major increase in computer power.
  - New methodologies (XP and Agile) demands for new tools.
  - Java's popularity.
- Characteristics:
  - Cross-platform (usually in Java itself)
  - Comfortable GUI
  - Full-Featured
  - Highly configurable
  - More powerful than separate tools
    - Refactoring, Syntax Highlighting, dynamic documentation.
- Can be a huge gain of time:
  - Lets you focus on your project and not on the tools.
  - Lets you do things you could not do before (at lighting speed).

Side Note: Refactoring

- What is refactoring?
- Martin Fowler: “Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.”
- Formalized so tools (and IDEs) can implement it.
- Major part of XP

State of the Art

- IntelliJ IDEA
  - $499
  - Leading IDE
  - Being passed by eclipse
  - Supports only JAVA
- Eclipse
  - Free
  - Large community (majorly adopted)
  - Building momentum
  - Java has priority but C/C++, XML, python are following
- NetBeans
  - Free
  - Sun’s default IDE

A Good example: The Eclipse Platform

- Quote: "Eclipse is a kind of universal tool platform - an open extensible IDE for anything and nothing in particular"

- This is good and bad:
  - Allows for a lot of flexibility and creates interest for a wider community.
  - Creates a bit of confusion for beginners

Eclipse Features

- CVS/Subversion integration
- Very smart editor
  - Syntax-highlighting
  - Refactoring
  - Dynamic documentation
- Hooks to outside programs
- Ant support / Gnu Make support
- GUI Builder
- Very powerful plugin-system
  - XML / Python / CruiseControl … and 800 more
- Quality software

Eclipse Demo (5-10 mins)

- Check-out a project
- Play with the code
- Write a Unit Test
- Document
- Launch / debug
- Release
- Check-in

Links

- **Eclipse:**
  - **www.eclipse.org**
- **IntelliJIDEA:**
  - **www.jetbrains.com/idea**
- **Refactoring:**
  - **www.refactoring.com**

Break & Release

- Break: 15 Mins
- Logging, Debugging, Tuning (Dan: 20 mins)
- Unit testing (Matt: 20 mins)
- Software Maintenance, Communication, Documentation (Matt: 20 mins)
- Release Engineering (Keith: 5 mins)

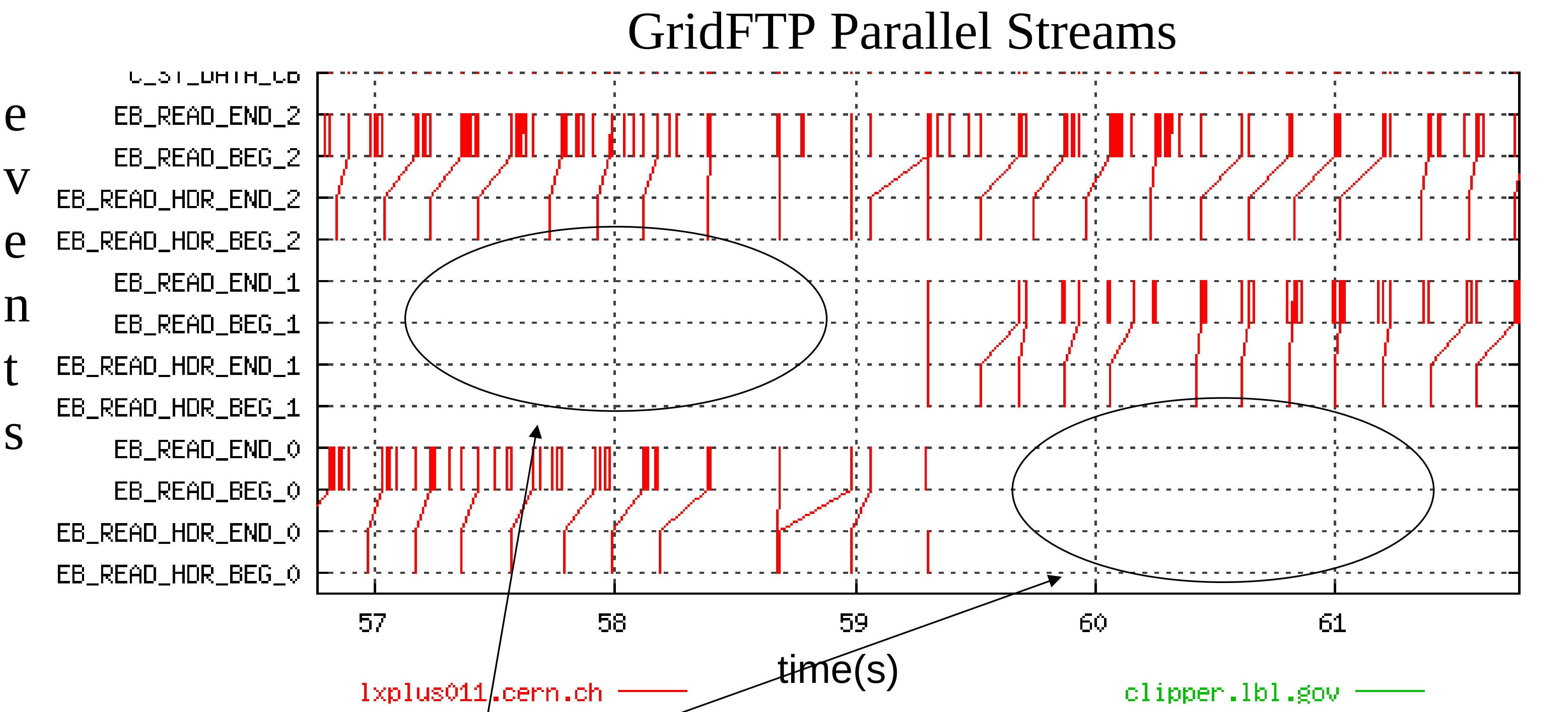
Logging, Tracing, and Debugging

- **The Problem:**
  - Bugs happen, no matter how much we plan to avoid them
- **The Corollary:**
  - Debugging is an essential part of the software lifecycle
- **The Solution:**
  - Ha! We can only really talk about tools...

What is a bug?

- Traditional definition of “bug” is something that makes the program halt or go obviously haywire
  - segmentation fault, hangs indefinitely, etc.
- But distributed computing leads us to recognize other types of “soft failures”, where things work – just more slowly – as bugs
- Detecting these types of failures requires something more than just firing up gdb

Example of "soft failure" bug



turned out to be a select() bug!

The Lesson

- **Instrumentation (logging) should be part of programs, even if they seem to work**
- **Debuggers are an incomplete solution**
  - **Debuggers don't work well with distributed programs**
  - ***Communicating debugging results is harder:***
    - *“debugging statements stay with the program; debugging sessions are transient” - The Practice of Programming, Brian W. Kernighan & Rob Pike*

Types of Debug Statements

- ***Logging***: **Low-volume, readable messages intended for browsing**
- ***Instrumentation*:**
  - ***Tracing***: **Any-volume, timestamped messages intended for automated analysis**
  - ***Profiling***: **Sampled OS/HW/interpreter/program statistics, intended for automated analysis**

Death to *printf()* !

- **Typical debug statement:**

```
#ifdef DANS_DEBUG
   printf(``Got to end of routine!\n``);
#endif
```

- **What's wrong with this?**
  - hard to turn on/off (requires recompile!)
  - non-structured, provides almost no context
  - can't be easily redirected elsewhere
  - completely ad-hoc!

Modern Logging APIs

- Most languages now have at least one *de facto standard API* that can simplify, unify, clarify logging
  - Typically not also good for tracing or profiling, but more on that later
- Feature-rich
- Documented, maintained by someone else!

CRD

Example API: *log4j*

BERKELEY LAB

- Log4j, which came out of the Jakarta project, is the *de facto* standard logging package for Java; similar APIs for C, Python
- Basic concepts:
  - Arrange multiple logging object instances in a hierarchy so properties can be inherited from the root (instead of specified every time)
  - Separate the policy, destination, and format
  - Allow configuration to be serialized into a file

log4j example

```
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
public class MyApp {
  static Logger
     logger = Logger.getLogger(MyApp.class.getName());
  public static void main(String[] args) {
     PropertyConfigurator.configure(args[0]);
     logger.info("Entering application.");
     for (int i=0; i < 100; i++ ) {
         if (logger.isDebugEnabled())
             logger.debug("i="+i);
     }
     logger.info("Exiting application.");
  }
}
```

log4j example (annotated)

```
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
public class MyApp
  static Logger
      logger = Logger.getLogger(MyApp.class.getName());
  public static void main(String[] args) {
     PropertyConfigurator.configure(args[0]);
     logger.info("Entering application.");
     for (int i=0; i < 100; i+
        if (logger.isDebugEnabled())
           logger.debug("i="+i);
     }
     logger.info("Exiting
  }
}
```

Logger instance named for class

Configure from file

extra if/then for efficiency

log4j configuration file

- Java "properties" format (also has XML format)

```
# Set root logger level=DEBUG and only appender to 'A1'
log4j.rootLogger=DEBUG, A1
# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p
   %c %x - %m%n
```

Sample log4j output

- **With “log4j.rootLogger=DEBUG,A1”**

```
0     [main] INFO  MyApp  - Entering application.
4     [main] DEBUG MyApp  - i=0
5     [main] DEBUG MyApp  - i=1
...
31    [main] DEBUG MyApp  - i=99
32    [main] INFO  MyApp  - Exiting application.
```

- **With “log4j.rootLogger=INFO,A1”**

```
0     [main] INFO  MyApp  - Entering application.
4     [main] INFO  MyApp  - Exiting application.
```

- **All that had to change was the config file!**

Beyond “logging” to performance analysis

- When an application gets large and complex, “eyeballing” logs becomes unfeasible
- Free-form user defined log formats make analysis difficult
- In a distributed application, just collecting logs into one place is a pain
- Our solution to this problem, for distributed applications, is called NetLogger

NetLogger Methodology

- **NetLogger is both a** methodology**, and a** set of tools
  - **You can use the NetLogger methodology without using any of our tools.**
- **Methodology:**
  1. **All components must be instrumented to produce monitoring. These components include application software, middleware, operating system, and networks. The more components that are instrumented the better.**
  2. **All monitoring events must use a common format and common set of attributes and a globally synchronized timestamp**
  3. **Log all of the following events: Entering and exiting any program or software component, and begin/end of all IO (disk and network)**
  4. **Collect all log data in a central location**
  5. **Use event correlation and visualization tools to analyze the monitoring event logs**

NetLogger API Sample

- Log API (Python):

```
import sys
from gov.lbl.dsd.netlogger import nllite

log = nllite.LogOutputStream(sys.argv[1])
size = 999.99
log.info("EVENT.START",{"TEST.SIZE":size})
for i in xrange(100):
    if log.debugging():
        log.debug("EVENT.I.START",{"VAL":i})
    # perform the task to be monitored
    if log.debugging():
        log.debug("EVENT.I.END",{"VAL":i})
log.info("EVENT.END",{"TEST.SIZE":size})
```

NetLogger "config file" Sample

- Just a log level in a file whose name is placed in an environment variable (NL_CFG)
- Example usage:

```
$ export NL_CFG=myapp.config
$ echo 5 > $NL_CFG
```

NetLogger Output Sample

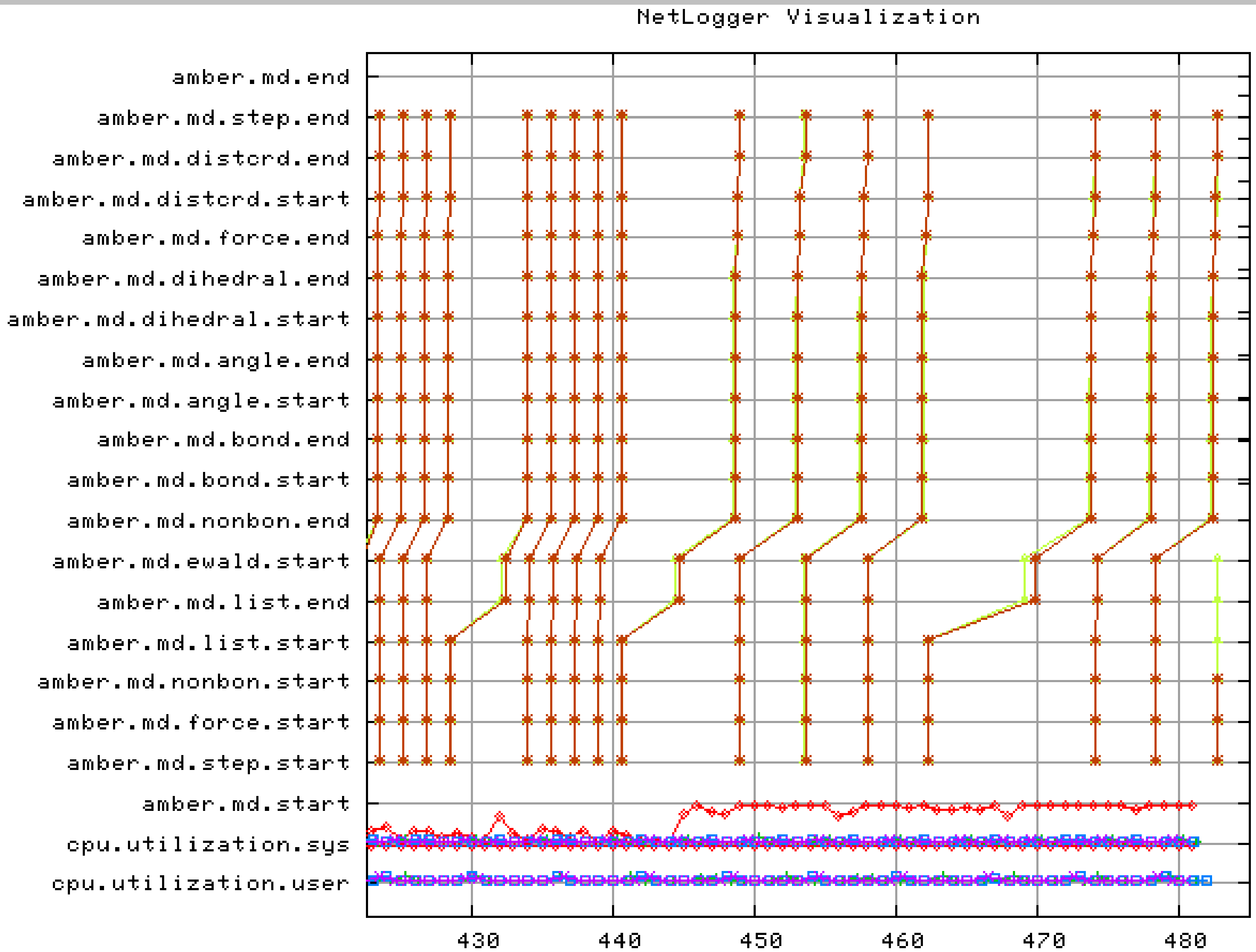
datatype
name
value

```
t DATE: 2005-05-12T07:00:13.57873
s LVL: INFO
s EVNT: EVENT.START
d TEST.SIZE: 999.99

t DATE: 2005-05-12T07:00:13.698969
s LVL: DEBUG
s EVNT: EVENT.I.START
i VAL: 1

t DATE: 2005-05-12T07:00:13.698969
s LVL: DEBUG
s EVNT: EVENT.I.END
i VAL: 1

...

t DATE: 2005-05-12T07:00:13.698971
s LVL: INFO
s EVNT: EVENT.END
d TEST.SIZE: 999.99
```

NetLogger Analysis Concepts

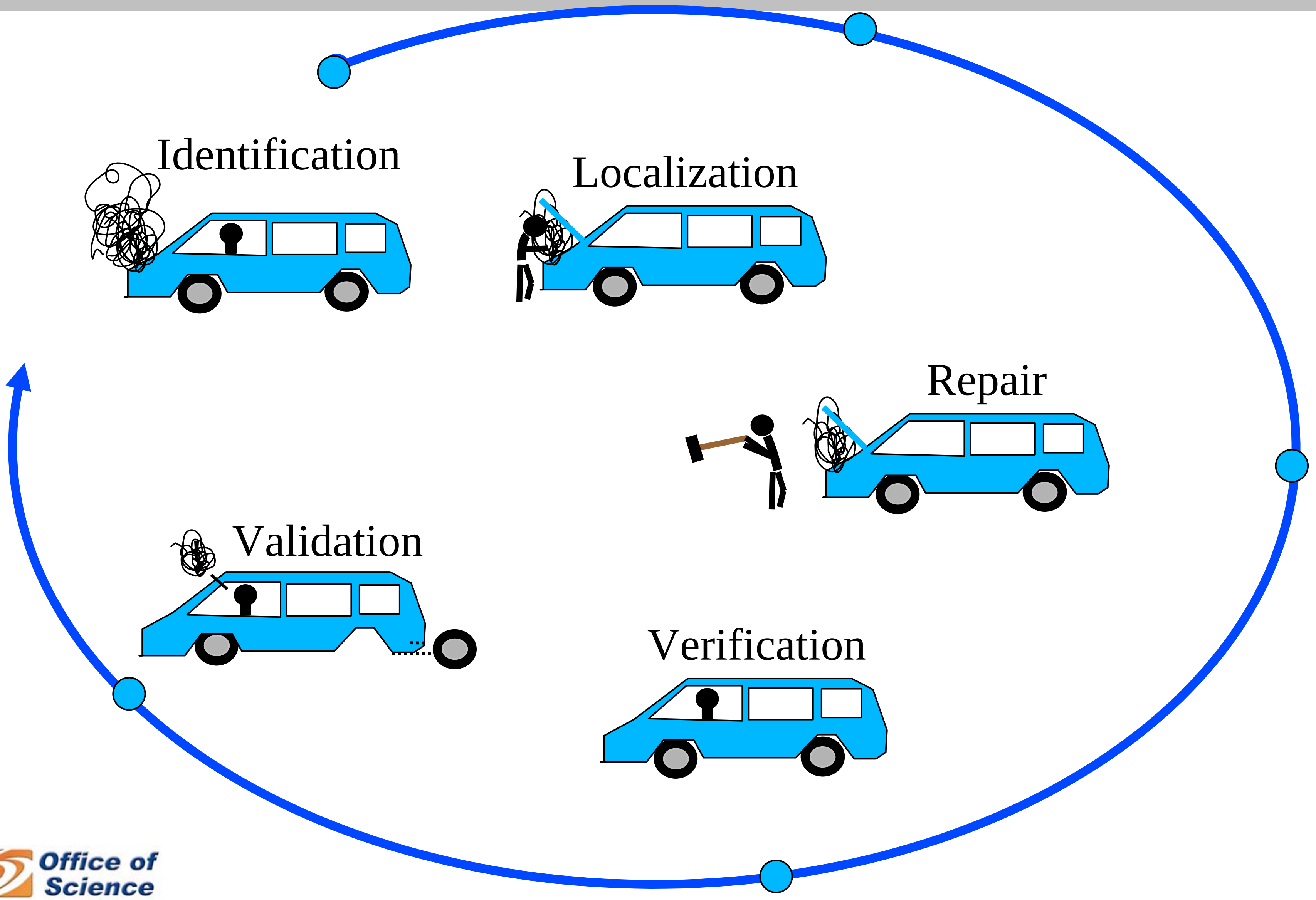
- NetLogger visualization tools are based on time-correlated and object-correlated events.
- If applications specify an "object ID" for related events, this allows the NetLogger visualization tools to generate an object "lifeline"
- To associate a group of events into a "lifeline", you must assign each an "object ID"
- Sample Event IDs: file name, block ID, frame ID, Job ID, etc.

NetLogger Analysis Example



U.S. DEPARTMENT OF ENERGY

Performance/Debugging Process



Resources

- log4j
  - http://logging.apache.org/log4j/
  - http://jakarta.apache.org/commons/logging/
- NetLogger
  - http://dsd.lbl.gov/netlogger/
- NERSC Performance Eval. Research Center
  - http://perc.nersc.gov/main.htm

Unit Test, Software Maintenance & Communication

Matt Rodriguez

Software Maintenance

Techniques for maintaining and developing a software system

Overview

- Techniques for testing your code. (unit testing)
- Using a bug tracking system
- Using automated build and test systems.
- Essential documentation practices for interacting with users

Different Testing Techniques

- Unit testing
- Integration Testing
- Systems Testing
- Regression Testing

Unit Testing

- What is unit testing?
- Tests individual components in a software system.
- Provides a status report of the health of a project
- Unit testing framework can be helpful in discovering problems quickly

Motivation for unit testing

- Unit testing will make your code more stable
- Unit testing will facilitate refactoring and further development
- Unit testing will give you more confidence in your code
- Unit testing will save you TIME.

Unit testing Strategies

- Start from the bottom up, first test classes in isolation, then test instances working together to do common tasks

- Consider writing the tests first or having another developer write the tests after the system has been designed

More unit testing strategies

- Use “mock objects” that pretend to be results from 3rd party services
- Use of drivers and stubs
- A Driver directly tests the software
- A stub is a placeholder that mimics a subsystem that the driver uses

What to unit test?

- Ideally, test each class in isolation
- Practically, test each important method
- Test common tasks
- Test for exceptions
- Test for failures

What not to unit test

- Code that relies heavily on third party services (ie DB)
- Code that gives an non deterministic answer
- Code that requires active user input

Use existing unit test APIs

- Java: JUnit
- C++: CppUnit
- Python: PyUnit, unittest

Most languages have a library that will help you organize and aggregate your test suite.

Build and Test Systems

- Automated systems that routinely build and run the tests in the testing suite
- Can be triggered by cvs commits, or via a web interface
- Can test on all the platforms that your project supports
- Generates html pages to organize and display the results

Motivation

- Testing your software is important but can be tedious
- Catches the introduction of new bugs quickly
- Enables communication between developers

Office of Science
U.S. DEPARTMENT OF ENERGY

When to use an ABTS

- When you are developing on multiple platforms (not in java)
- When you are working with multiple developers
- When you must ensure that your software works with multiple versions of different software libraries

Cruise Control

- Uses ant
- Well supported
- Works with 3rd party tools: cvs, eclipse
- Generates html pages, can send email
- Trigger a build interactively

Tinderbox

- Developed by the mozilla project
- Written in perl
- Not supported very well, not documented very well
- Typically projects use the collection of scripts that come with Tinderbox and tailor them to their needs

Other Build systems

- Dart
- Dart is supported better, written in TCL
- Open source build systems are not as mature as the open source bug tracking systems

Bug tracking systems

- Maintains a history of bug submissions, proposed steps to fix the problem, and resolution
- Enables communication between developers working on the project
- Facilitates interaction with the user community
- Useful tool for managing a project

Free Bug Tracking Systems

- Bugzilla, mantis
- Bugzilla is more widely used today
- From the mozilla project
- Written in perl, uses a MySQL backend DB
- Complaints: People have dislike its user interface, in particular when making queries

Bugzilla features

- Browser user interface
- Interacts with 3rd party tools
- End users can submit attachments when filing a bug report
- Multiple projects can use on bugzilla instance

Configuring your project

- Products- your project
- Components – aspects of your project
- Versions – releases of your project
- Milestones – releases by when certain bugs will be fixed
- Voting – Allows users to pick which bug they want fixed

Filing a Bug Report

- Summary, expected behavior/observed behavior
- Specifically explain how to reproduce the bug
- One bug per report
- Include as much information as possible,

  (OS, version information of relevant tools/components, stack traces)

Integration with 3rd Party Tools

- Bonsai: Web based CVS
- CVS
- Perforce
- Tinderbox

Mantis

- Written in php, supporters feel that it has as better UI than bugzilla
- The Query interface is regarded to be better
- Allows for cvs integration
- Widely used but not as prevalent as bugzilla

Documentation

- API docs
- Project webpage
- Email lists
- README, INSTALL, CHANGELOG

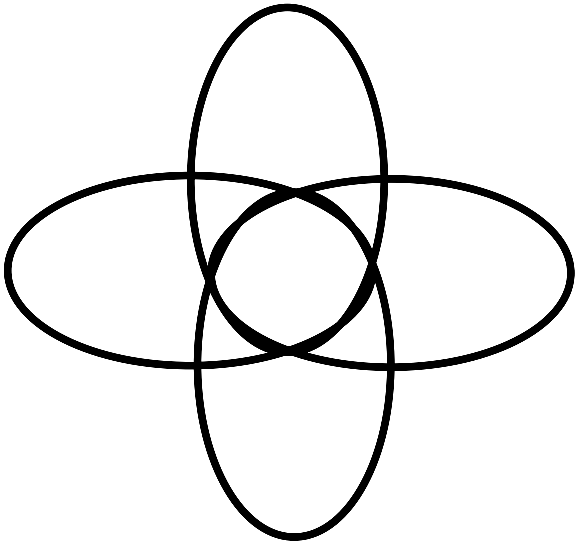
Goal of these practices

- Facilitates development and maintenance of the project
- Discover problems quickly
- Effective communication between your developers and users
- Allows people to use your software without too much hassle

Release Engineering

Intersection of the 4 areas in software dev.

- Engineering
  - Version control
- QA
  - Testing & Certification
- Operations
  - System Administration
- Project Management
  - Scheduling, branch management



Summary

- Design
- Implementation
- Version Control
- QA
- Documentation
- Release Eng.
- Distribution
- User Interaction

CRD

BERKELEY LAB

C O M P U T A T I O N A L   R E S E A R C H   D I V I S I O N

Tools and Techniques for Managing Large Scientific Software Projects

Keith Beattie, Chuck McParland, Dan Gunter, Guillaume Egles, Matt Rodriguez

May 13, 2005

Office of Science
U.S. DEPARTMENT OF ENERGY