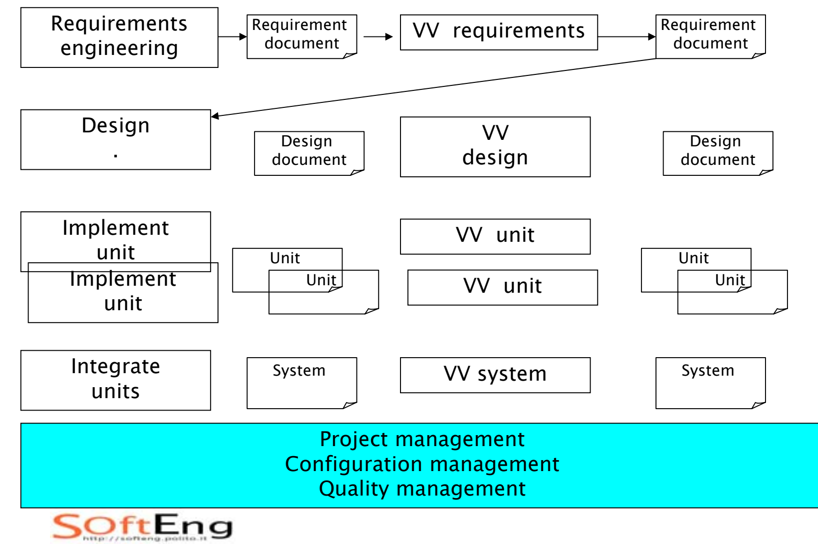
Software Configuration Management (SCM)

The whole picture



Software

- Made of many parts
  - Documents
  - Programs
  - See Heating control system
- Thousands of separate documents are generated for a large software system

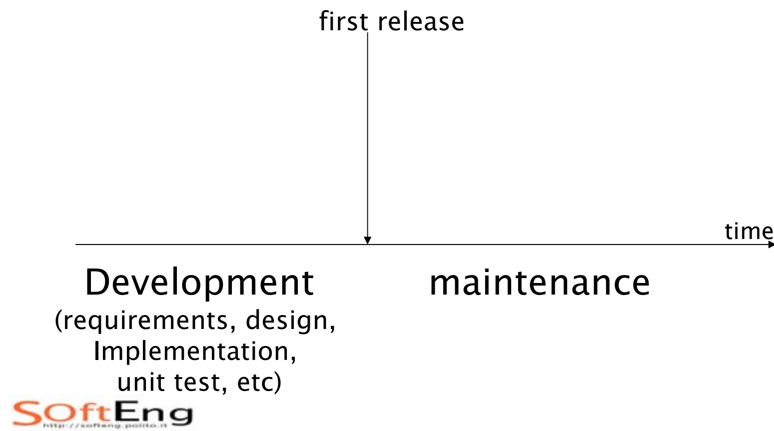
Change is inevitable

- A software system changes
  - Different instantiations of software for different customers
  - Same software changes over time

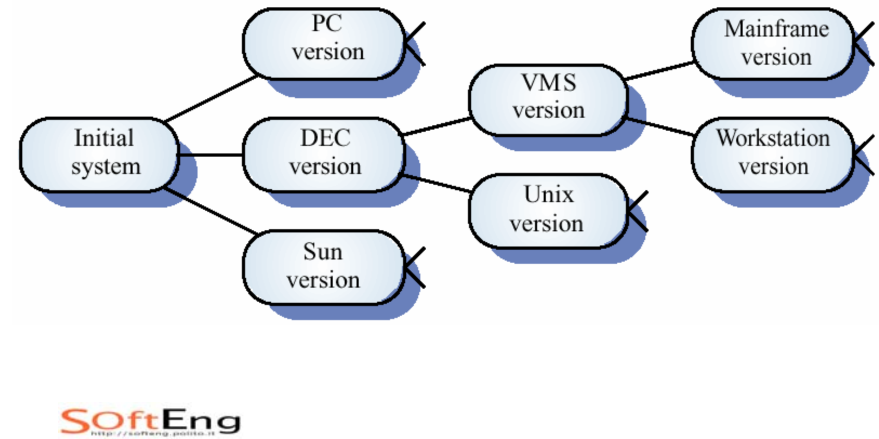
"No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle." [Bersoff et al., 1980]

- Parts of software must be kept consistent over changes
  - Configuration management

Phases



Different instantiations



Time and space dimensions

- Space
  - System made of many parts
  - System (and parts) adapted for many situations
- Time
  - Parts, and system, change over time

Typical situation

- Team develops software
- Many people need to access parts of software
  - Common repository, all can read/write documents/ programs

Scenarios

- One
  - John changes module A to fix bug
  - Meanwhile Linda downloads old A and links it
- Two
  - John and Jack download A to fix bugs
  - John uploads A
  - Jack uploads A (John's changes are lost)
- Three
  - Module A is used in system X
  - John fixes bug in A
  - Nobody is notified to relink A in X

Problems

- Concurrent access
  - What if two people access same document at same time?
- Concurrent modification
  - What if two people access same document and both modify it?
- Notification of changes
  - Document is changed, how all are informed?

Goals of CM

- Identify and manage parts of software
- Control access and changes to parts
- Allow to rebuild previous version of software

Outline

- Version management
- Change Control
- Build
- Configuration management planning
- Tools

Version management

Terms

- Configuration item (CI)
- Configuration Management aggregate
- Configuration
- Version
- Baseline

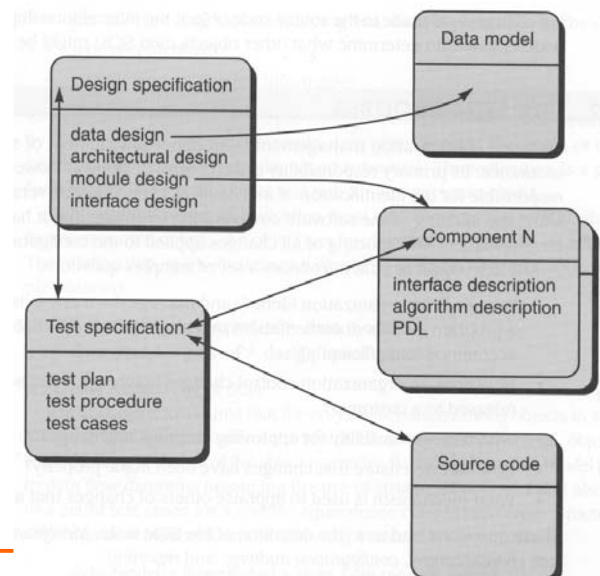
Configuration Item

- Unit for the CM system

  a work product or piece of software that is treated as a single entity for the purpose of configuration management.

  - May correspond to one/more document(s), one/more programs
    - Simple example of CIs
      - Requirement document
      - Design document
      - Source code module
- CM aggregate: composition of CIs

Links between CIs



- Choices of CM system
  - What parts of software system become CIs
  - (not all documents may become CIs)
- Changes to CI are subject to procedures defined by CM system
  - Typically, change must be approved and recorded
  - New version of CI must be generated

Version

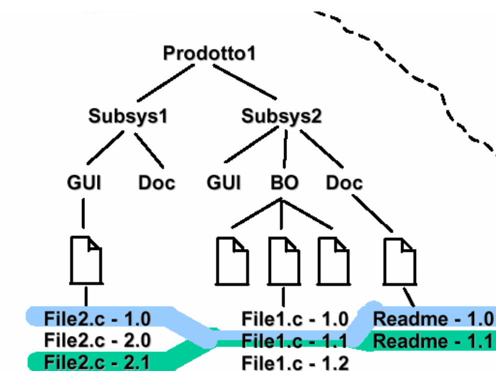
- Instance of CI
  - Ex Req document 1.0
  - Req document 1.1

Configuration

- Snapshot of software at certain time
  - Various CIs, each in a certain version
  - Same CI may appear in different configurations
  - Also configuration has version

Ex.

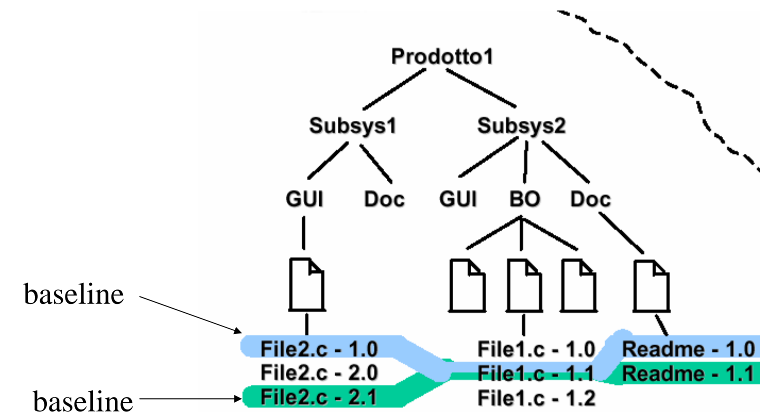
- System 1.0 (configuration 1.0)
  - File2.c 1.0 + File1.c 1.0 + Readme 1.0
- System 1.1 (configuration 1.1)
  - File2.c 1.0 + File1.c 1.1 + Readme 1.0



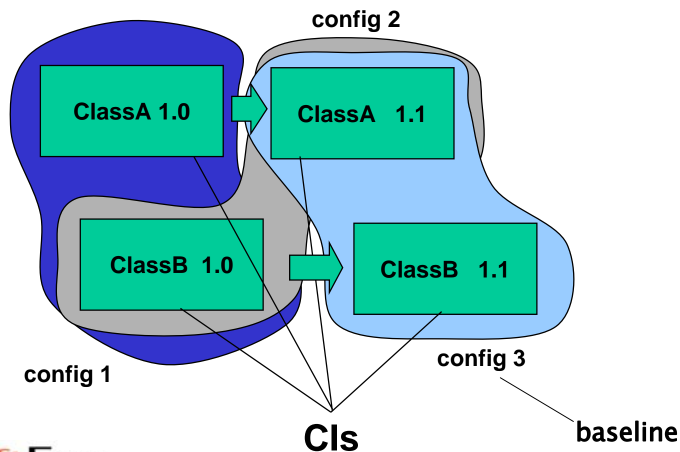
Baseline

- configuration in stable, frozen form
  - Not all configurations are baselines
  - Any further change / development will produce new version(s) of CI(s), will not modify baseline
- Types of baselines
  - Development – for internal use
  - Product – for delivery

Ex.



Ex.



Identification of CI

- Two levels
  - Logical
    - May or may not reflect structure of product
  - Physical
    - Name of file(s) containing CI
  - Rules to pass from one level to other
- Problem: id must be unique

Ex.

- Requirement Specification
  - Logical name -1
    - <office id> - <project id> - <version>
    - Torino1-P00A72-101
  - Logical name -2
    - <title>/<document type>/<project id>/<version>
    - RequirementDocument/REQ/P00A72/101
  - Physical name
    - Same as logical? Problems of length and special characters

Ex.

- C++ source code
  - Logical
  - <name of class>/<project id>/<version>
  - Physical (2 files needed)
    - <name of class>.h <name of class>.cpp
    - Version and project number can appear in file name?

Identification

- Logical identification
- Physical identification subject to constraints from file system and tools
  - Compilers, linkers, ..
- Passage must be supported by CM tool

Derivation history

- Record of changes applied to a document or code component
- Should record, in outline, the change made, the rationale for the change, who made the change and when it was implemented
- May be included as a comment in code. If a standard prologue style is used for the derivation history, tools can process this automatically

Component header info

```
// PROTEUS project (ESPRIT 6087)
//
// PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE
//
// Object: PCL-Tool-Desc
// Author: G. Dean
// Creation date: 10th November 1998
//
// © Lancaster University 1998
//
// Modification history
// Version       Modifier  Date        Change      Reason
// 1.0     J. Jones    1/12/1998   Add header  Submitted to CM
// 1.1     G. Dean     9/4/1999    New field   Change req. R07/99
```

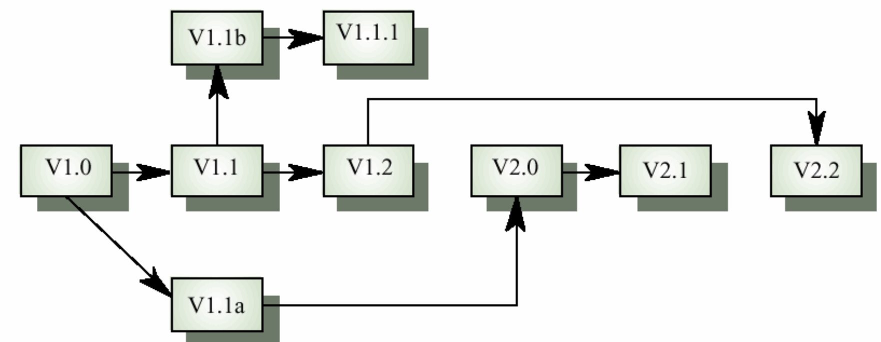

Version identification

- Procedures for version identification should define an unambiguous way of identifying component versions
- basic techniques for component identification
  - Version numbering
  - Attribute-based identification

Version numbering

- Simple naming scheme uses a linear derivation
  e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.
- Actual derivation structure is a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better

Version derivation structure



Parallel evolutions (1.1.1), possibly merged back together (1.2)

Attribute-based identification

- Attributes can be associated with a version with the combination of attributes identifying that version
- Examples of attributes are Date, Creator, Programming Language, Customer, Status
- More flexible than an explicit naming scheme for version retrieval; Can cause problems with uniqueness
- Needs an associated name for easy reference

Attribute-based queries

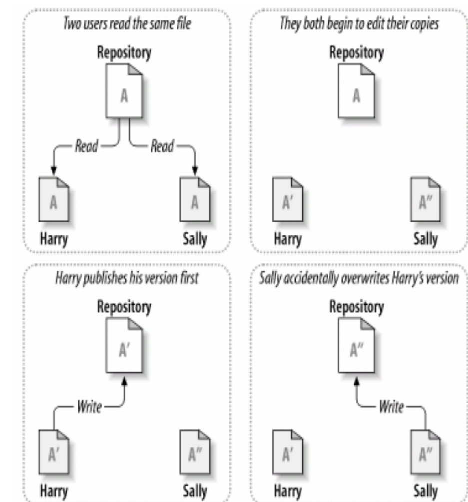
- An important advantage of attribute-based identification is that it can support queries so that you can find 'the most recent version in Java' etc.
- Example
  - AC3D (language =Java, platform = NT4, date = Jan 1999)

Change control

Change control

The problem:

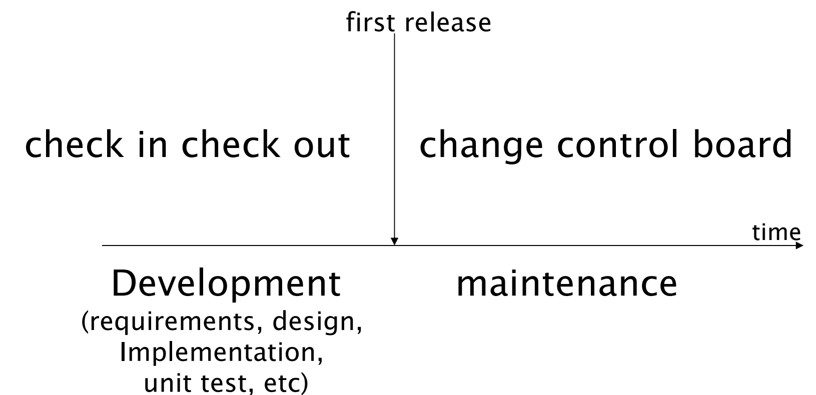
allow users to share information, but prevent them from accidentally stepping on each other's feet?



Change control

- Changes must be disciplined
  - Who controls
  - What is controlled
  - How control is implemented
- Approaches
  - CCB
    - Best for maintenance process
      - Ex Windows Vista after jan 2007
  - Check in – check out model, Workspaces
    - Best for development process
      - Ex Windows Vista before jan 2007

Phases



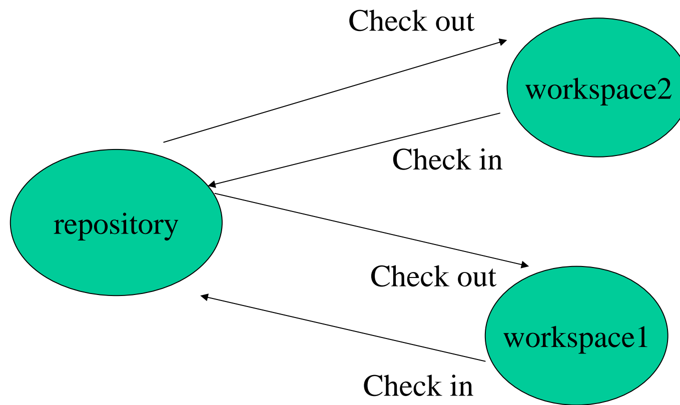
Check-in check-out

- Check-out
  - Extraction of CI from repository
    - with goal of changing it or not
    - After checkout next users are notified
- Check-in
  - Insertion of CI under control

Workspace

- ‘Private’ space where developer has full control

Workspace and check in/out



Checkin checkout vs. file system

Check in /out	File system
▪ CIs are in repository	▪ Files are in shared directory
▪ To rd/wr CI user needs to do check out	▪ Any user can get copy of file, or work on original
▪ After checkout next user knows that CI is used by someone else	▪ Users can work on copies of file without knowing that others are doing the same

Check in/out – choices

- Who can do check in/out
- Checked-out CI is locked or not
  - If locked, one writer, many readers
    - One only can modify
- Checked-in CI increments version or not
  - If not, old version is lost

Check in / check out – scenarios

- Lock modify unlock (or serialization)
  - One can change at a time
- Copy modify merge
  - Many change in parallel, then merge

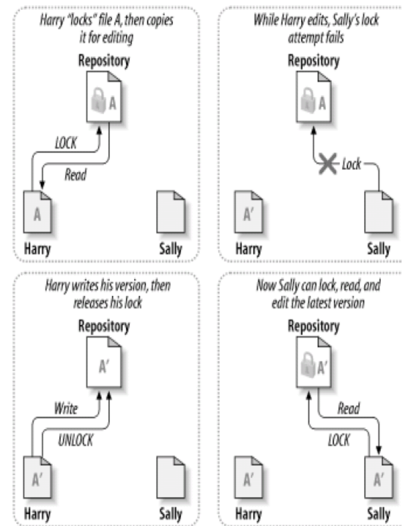
Lock-Modify-Unlock

Only one person to change a file at a time; good for editing of bynary files, but for others:

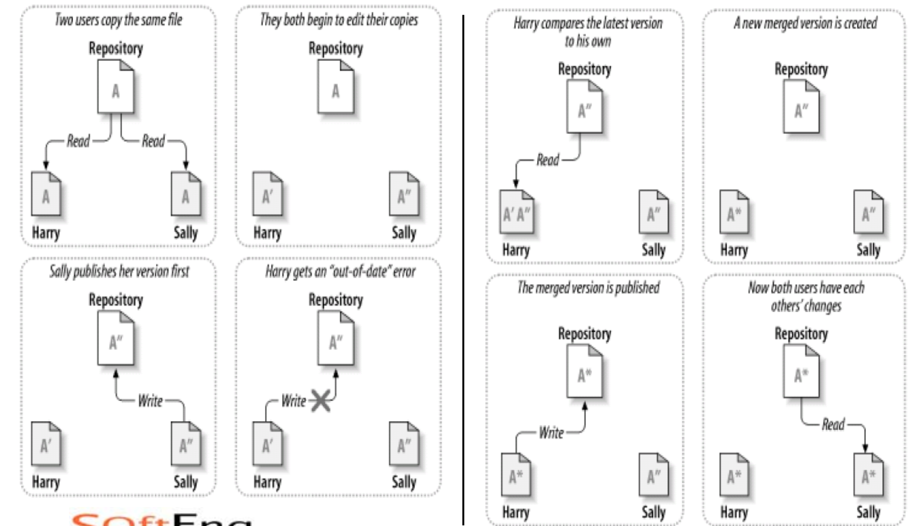
Locking may cause administrative problems.

Locking may cause unnecessary serialization.

Locking may create a false sense of security (dependiecs among files locked by different persons).



Copy-Modify-Merge



CCB

- Configuration Control Board
  - Authorizes changes to a baseline
    - Corrective maintenance
  - Defines what will be in next baseline
    - Perfective maintenance

Change control board

- Changes should be reviewed by an external group who decide whether or not they are cost-effective from a strategic and organizational viewpoint rather than a technical viewpoint
- Should be independent of project responsible for system. The group is sometimes called a change control board
- May include representatives from client and contractor staff

Change procedure

```
Request change by completing a change request form
Analyze change request
if change is valid then
  Assess how change might be implemented
   Assess change cost
  Submit request to change control board
  if change is accepted then
     repeat
     make changes to software
     submit changed software for quality approval
    until software quality is adequate
  create new system version
 else
  reject change request
else
  reject change request
```

SoftEng

Change request form

- Definition of change request form is part of the CM planning process
- Records change required, suggestor of change, reason why change was suggested and urgency of change (from requestor of the change)
- Records change evaluation, impact analysis, change cost and recommendations (System maintenance staff)

SoftEng

Change request form

Change Request Form

Project: Proteus/PCL-Tools **Number:** 23/94
Change requester: I. Sommerville **Date:** 1/12/98
Requested change: When a component is selected from the structure, display the name of the file where it is stored.

Change analyser: G. Dean **Analysis date:** 10/12/98
Components affected: Display-Icon.Select, Display-Icon.Display

Associated components: FileTable

Change assessment: Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.

Change priority: Low
Change implementation:
Estimated effort: 0.5 days
Date to CCB: 15/12/98 **CCB decision date:** 1/2/99
CCB decision: Accept change. Change to be implemented in Release 2.1.
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments

SoftE

CM Planning

SoftEng

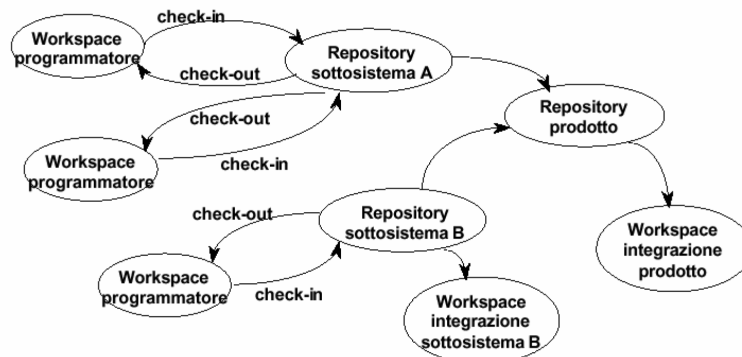
CM planning

- Starts during the early phases of the project
- Must define (CM plan document)
  - CIs
    - Identification, versioning
  - Baselines
  - Change control rules, roles, responsibilities
  - Tools used

Example

- The product
  - Several subsystems, each subsystem an executable and several source files (modules)
  - Hierarchy
- The team
  - One person responsible per module
  - One person responsible per subsystem
- The repository
  - One repository per subsystem
  - Check in/out
  - Workspace per person

Example

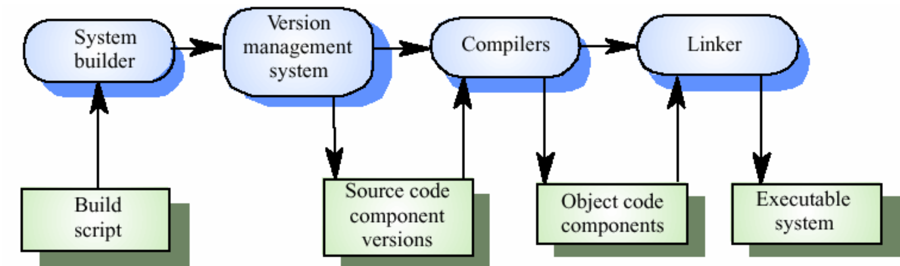


Build

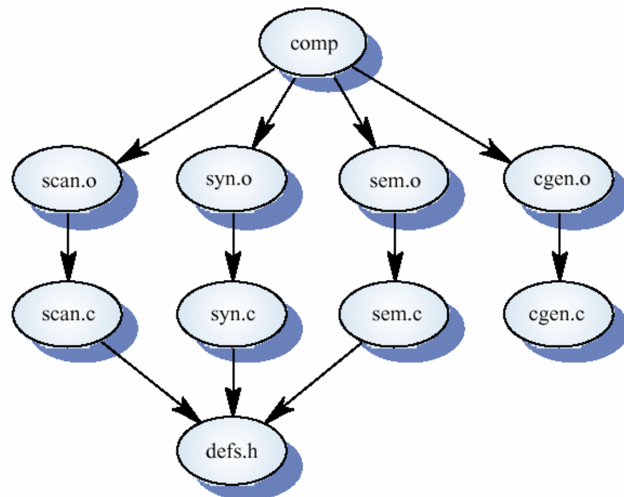
System building

- The process of compiling and linking software components into an executable system
- Different systems are built from different
combinations of components
- Invariably supported by automated tools that are driven by 'build scripts'

System building



Component dependencies



System building problems

- Do the build instructions include all required components?
  - When there are many hundreds of components making up a system, it is easy to miss one out. This should normally be detected by the linker

- Is the appropriate component version specified?
  - A more significant problem. A system built with the wrong version may work initially but fail after delivery
- Are all data files available?
  - The build should not rely on 'standard' data files. Standards vary from place to place

SOftEng

System building problems

- Are data file references within components correct?
  - Embedding absolute names in code almost always causes problems as naming conventions differ from place to place
- Is the system being built for the right platform
  - Sometimes must build for a specific OS version or hardware configuration

SOftEng

- Is the right version of the compiler and other software tools specified?
  - Different compiler versions may actually generate different code and the compiled component will exhibit different behaviour

SOftEng

System representation

- Systems are normally represented for building by specifying the file name to be processed by building tools. Dependencies between these are described to the building tools
- Mistakes can be made as users lose track of which objects are stored in which files
- A system modelling language addresses this problem by using a logical rather than a physical system representation

SOftEng

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o insert.o  search.o files.o
    utils.o
main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
insert.o : insert.c defs.h buffer.h
  cc -c insert.c
search.o : search.c defs.h buffer.h
  cc -c search.c
files.o : files.c defs.h buffer.h command.h
  cc -c files.c
utils.o : utils.c defs.h
  cc -c utils.c
clean :
  rm edit main.o kbd.o command.o display.o insert.o  search.o files.o utils.o
```


Tools

Functions

- Change management
- Version management
- Build

Tools

- CM + VM
  - RCS
  - CVS
  - SCCS
  - PCVS
  - Clearcase
  - Subversion
  - BitKeeper
- Build
  - Make
  - Ant
  - Maven

Change management tools

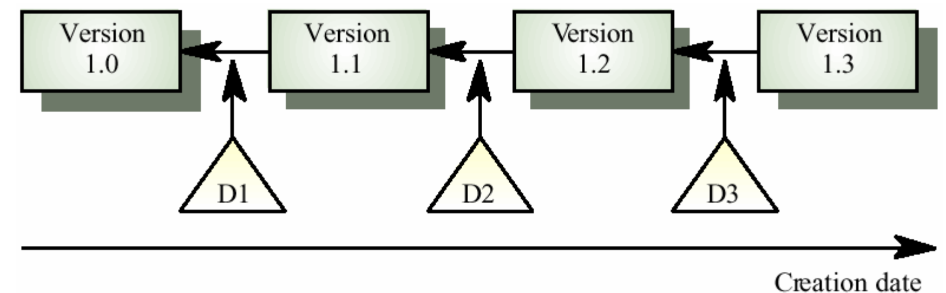
- Change management is a procedural process so it can be modelled and integrated with a version management system
- Change management tools
  - Form editor to support processing the change request forms
  - Workflow system to define who does what and to automate information transfer
  - Change database that manages change proposals and is linked to a VM system

Version management tools

- Version and release identification
  - Systems assign identifiers automatically when a new version is submitted to the system
- Storage management
  - System stores the differences between versions rather than all the version code

- Change history recording
  - Record reasons for version creation
- Independent development
  - Only one version at a time may be checked out for change. Parallel working on different versions

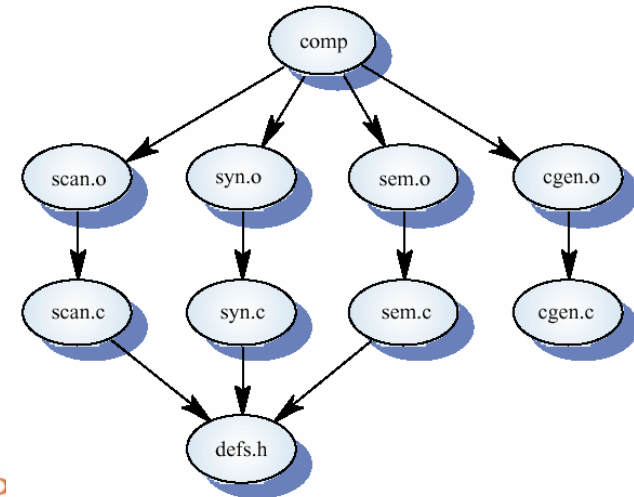
Delta-based versioning



System building

- Building a large system is computationally expensive and may take several hours
- Hundreds of files may be involved
- System building tools may provide
  - A dependency specification language and interpreter
  - Tool selection and instantiation support
  - Distributed compilation
  - Derived object management

Component dependencies



RCS

- Unit is file
- Baseline
- Check in check out
  - CI command
    - Inserts file in baseline
    - Associates comment explaining the change
    - Associates new version number (automatically or not)
  - CO command
    - Extracts file, in Rd or Wr mode

- Ident command
  - Associates name to file, starting from attributes (name author version)
- Rlog
  - Extracts from baseline description
    - List of composing files
    - Comments attached to files

- Storage of versions based on delta
  - Storage space saved
  - Check in / out can be slow
- Lock mechanism (default)
  - At checkout file is locked
  - Checkin possible only if user did checkout

CVS

- Built on top of RCS
- Client server
- Unit is file or directory
- Same commands as RCS (if applied to directory they are applied to all contained files)
- Check out with lock or not
  - Concurrent work on file possible
  - Reconciliation at checkin (semi automatic)

PCVS

- Client server
- Concepts
  - Project = set of files + directories
  - Archive = set of all versions of file
  - Revision = version of file
- Suite of tools
  - Version manager
  - Configuration builder (to support creation of release)
  - Tracker to support change request
  - Notify (via email) to notify changes

Functions

- Create project
- Browse project
- Check out (w w/out lock)
- Check in
- Reports
- Branch merge management

Svn – subversion

- See slides

Make

- Part of Unix
- Allows to describe components and dependencies among components
- Allows to describe operations to build system from components
- Builds system – recompiles only if component was changed (using data tag)


```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o insert.o  search.o files.o
    utils.o
main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
insert.o : insert.c defs.h buffer.h
  cc -c insert.c
search.o : search.c defs.h buffer.h
  cc -c search.c
files.o : files.c defs.h buffer.h command.h
  cc -c files.c
utils.o : utils.c defs.h
  cc -c utils.c
clean :
  rm edit main.o kbd.o command.o display.o insert.o  search.o files.o utils.o
```


ANT

- A build tool like make
- Open source
  - from the Apache Jakarta project
  - http://jakarta.apache.org/ant
- Implemented in Java
- Used to build many open source products
  - such as Tomcat and JDOM

Why Use Ant Instead of make?

- Ant is more portable
  - Ant only requires a Java VM (1.1 or higher)
  - make relies on OS specific commands to carry out it's tasks
- Ant targets are described in XML
  - make has a cryptic syntax
  - make relies proper use of tabs that is easy to get wrong
    - you can't see them
- Ant is better for Java-specific tasks
  - faster than make since all tasks are run from a single VM
  - easier than make for some Java-specific tasks such as generating javadoc, building JAR/WAR files and working with EJBs

How Does Ant Work?

- Ant commands (or tasks) are implemented by Java classes
  - many are built-in
  - others come in optional JAR files
  - custom commands can be created
- Each project using Ant will have a build file
  - typically called build.xml since Ant looks for this by default
- Each build file is composed of targets
  - these correspond to common activities like compiling and running code
- Each target is composed of tasks
  - executed in sequence when the target is executed
  - like make, Ant targets can have dependencies
    - for example, modified source files must be compiled before the application can be run

How ..

- Targets to be executed
  - can be specified on the command line when invoking Ant
  - if none are specified then the default target is executed
  - execution stops if an error is encountered so all requested targets may not be executed
- Each target is only executed once
  - regardless of the number of other targets that depend on it, ex:
    - the "test" and "deploy" targets both depend on "compile"
    - the "all" target depends on "test" and "deploy"
    - but "compile" is only executed once when "all" is executed
- Some tasks are only executed when they need to be
  - for example, files that have not changed since the last time they were compiled are not recompiled

Build file example (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Web App." default="deploy" basedir=".">

  <!-- Define global properties. -->
  <property name="appName" value="shopping"/>
  <property name="buildDir" value="classes"/>
  <property name="docDir" value="doc"/>
  <property name="docRoot" value="docroot"/>
  <property name="junit" value="/Java/JUnit/junit.jar"/>
  <property name="srcDir" value="src"/>
  <property name="tomcatHome" value="/Tomcat"/>
  <property name="servlet" value="${tomcatHome}/lib/servlet.jar"/>
  <property name="warFile" value="${appName}.war"/>
  <property name="xalan" value="/XML/Xalan/xalan.jar"/>
  <property name="xerces" value="/XML/Xalan/xerces.jar"/>
```

relative directory references are relative to this

target that is run when none are specified

Some of these are used to set "**classpath**" on the next page. Others are used in task parameters.

Where possible, use **UNIX-style paths** even under Windows. This is not possible when Windows directories on drives other than C must be specified.

Build file example (2)

```
<path id="classpath">
  <pathelement path="${buildDir}"/>
  <pathelement path="${xerces}"/>
  <pathelement path="${xalan}"/>
  <pathelement path="${servlet}"/>
  <pathelement path="${junit}"/>
</path>

<target name="all" depends="test,javadoc,deploy"
 description="runs test, javadoc and deploy"/>
```

used in the compile, javadoc and test targets

means that the test, javadoc and deploy targets must be executed before this target

doesn't have any tasks of its own; just executes other targets

Build file example (3)

```
<target name="clean" description="deletes all generated files">
  <delete dir="${buildDir}"/> <!-- generated by the prepare target -->
  <delete dir="${docDir}/api"/> <!-- generated by the javadoc target -->
  <delete>
    <fileset dir=".">
      <include name="${warFile}"/> <!-- generated by the war target -->
      <include name="TEST-*.txt"/> <!-- generated by the test target -->
    </fileset>
  </delete>
</target>

<target name="compile" depends="prepare"
 description="compiles source files">
  <javac srcdir="${srcDir}" destdir="${buildDir}" classpathref="classpath"/>
</target>

<target name="deploy" depends="war,undeploy"
 description="deploys the war file to Tomcat">
  <copy file="${warFile}" tofile="${tomcatHome}/webapps/${warFile}"/>
</target>
```

means that the prepare target must be executed before this target

compiles all files in or below srcDir that have no .class file or have been modified since their .class file was created; don't have to list specific file names as is common with make

makes the servlet available through Tomcat; Tomcat won't expand the new war file unless the corresponding webapp subdirectory is missing

Build file example (4)

```
<target name="dtd" description="generates a DTD for Ant build files">
  <antstructure output="build.dtd"/>
</target>

<target name="javadoc" depends="compile"
 description="generates javadoc from all .java files">
  <delete dir="${docDir}/api"/>
  <mkdir dir="${docDir}/api"/>
  <javadoc sourcepath="${srcDir}" destdir="${docDir}/api"
   packagenames="com.ociweb.*" classpathref="classpath"/>
</target>

<target name="prepare" description="creates output directories">
  <mkdir dir="${buildDir}"/>
  <mkdir dir="${docDir}"/>
</target>
```

generates a DTD that is useful for learning the valid tasks and their parameters

generates javadoc for all .java files in or below srcDir.

can't just use a single * here and can't use multiple *'s

creates directories needed by other targets if they don't already exist

Build file example (5)

```
<target name="test" depends="compile" description="runs all JUnit tests">
  <!-- Delete previous test logs. -->
  <delete>
    <fileset dir=".">
      <include name="TEST-*.txt"/> <!-- generated by the test target -->
    </fileset>
  </delete>

  <taskdef name="junit"
   classname="org.apache.tools.ant.taskdefs.optional.junit.JUnitTask"/>
  <junit printsummary="yes">
    <classpath refid="classpath"/>
    <batchtest>
      <fileset dir="${srcDir}"><include name="**/*Test.java"/></fileset>
      <formatter type="plain"/>
    </batchtest>
  </junit>
</target>
```

runs all JUnit tests in or below srcDir

junit.jar must be in the **CLASSPATH** environment variable for this to work. It's not enough to add it to <path id="classpath"> in this file.

** specifies to look in any subdirectory at any depth

Build file example (6)

```
<target name="undeploy" description="undeploys the web app. from Tomcat">
  <delete dir="${tomcatHome}/webapps/${appName}"/>
  <delete file="${tomcatHome}/webapps/${warFile}"/>
</target>

<target name="war" depends="compile" description="builds the war file">
  <war warfile="${warFile}" webxml="web.xml">
    <classes dir="${buildDir}"/>
    <fileset dir="${docRoot}"/>
  </war>
</target>

</project>
```

makes the servlet unavailable to Tomcat

creates a web application archive (WAR) that can be deployed to a servlet engine like Tomcat

contains HTML, JavaScript, CSS and XSLT files

Download

- http://ant.apache.org
- http://ant.apache.org/manual

Commands

- ant [*option*s] [*target-name*s]
  - omit target-name to run the default target
  - runs targets with specified names, preceded by targets on which they depend
  - can specify multiple target-names separated by spaces
  - -D option specifies a property that can be used by targets and tasks
    - *-Dproperty-name=property-value*
- ant -help
  - lists other command-line options

Core tasks (some)

- Chmod
- Concat
- Copy
- Cvs
- Delete
- Exec
- Java
- Javac
- Javadoc
- Mail
- Mkdir
- Move
- Sleep
- Sql
- Tar
- Zip
- Unzip

References and Further Readings

- “Software configuration management: A roadmap”, J.Estublier, Proc. INt. Conf.onSoftware Engineering, 2000, IEEE Press.
- IEEE STD 1042 – 1987 IEEE guide to software configuration management
- IEE STD 828-2005 Standard for Software Configuration Mangement Plans
- “Configuration Management Principle and Practice”, A.M.J.Hass,2002, Addison Wesley