Configuration
refers to the overall set of elements that comprise a software product ("configuration items")
software components
modules
internal logical files
test stubs and scaffoldings
external documentation
user manual
FAQs
getting started guides
tutorials
internal documentation
requirements
design
test plan

More generally, includes environmental conditions necessary for product function (e.g., external libraries, files, etc)
environmental conditions required to develop the software (e.g., software tools)
These are lists of conditions.

Three concepts of configuration
Configuration of the files.
Configuration of the development environment.
Configuration of the deployment environment.
The basic problem: "things change"
requirements change ->
design changes ->
components change ->
documentation changes

Software configuration management:
manages changes via versions.
creates baselines.
eliminates version skew.
Versioning

create a reference (version) number for each entire configuration.
alternatively: Version 2.4.1 is version 2 of the package, minor release 4, bug fix level 1.

Conventional versioning:
  x.y.z where
  you change x when you make major changes to function.
  you change y when you add minor improvements
  you change z when you add bug fixes (no functional differences).

Also find:
  x.y.z.w = major x, minor y, branch z, revision w.

Releases
  major release number: denotes significant changes in function, e.g., new features.
  minor release number: denotes minor changes in function and/or bug fixes.
Typically, between 2.3 and 2.4, the documentation doesn't change, while between 2.4 and 3.1, the documentation does change.
Configurations form a **version tree:**

```
       root
      /
     /  \
   1   2
  /    /
1.0  1.1 1.2  2.0 2.1 2.2
  |    |    |
2.1.1 2.1.2 2.1.3
```

Linear sequence
A baseline version of the software:

- serves as a basis and reference for further changes.
- has a simple version number, e.g., 2.0
- and sometimes is fully tested and ready for release (whether actually released or not)
- has the property that all components are "in sync" as far as describing the same product.
baselining can be thought of as a funnel:

commitment
process

baseline:
agreed-upon function
Challenge of software configuration management: **version skew**

happens when components must be updated together
caller and callee
component function and documentation
component function and design or requirements
A version skew occurs when two configuration items in one configuration refer to **different versions** of the product.
version branches

Normal development is a linear process:
  1.0, 1.1, 1.2... first version and fixes
  2.0, 2.1, 2.2... second version and fixes
  ...

A **branch** occurs when some entity requests an **experimental or custom change**.
  1.2, 1.2.1.0 (custom), 1.2.1.1, 1.2.1.2, ...

Normal development continues, ignoring the branch
E.g., a commercial company requests a customization "just for them"
Custom branches are a **lucrative source of income** in the open source/free software world.
Custom branches can be **merged** into the main revision tree later, or **orphaned** if no one wants them anymore.

Just because software is "free" doesn't mean programmers should be "poor"!
Free software simplified
  Software is "free" if
    it cannot be charged for,
    its source is available to everyone,
    it cannot be incorporated into any product for which there is a charge.
  See the Gnu Public License.
  So, one cannot "pay" for free software.

Justification for free software
  more people can work on it.
  more people can use it.
  more completely debugged and verified.

Business model for free software
  Software itself is always free, but programmers' time is still worth money.
  So, pay programmers to
    customize free software
    support free software they've written
    write new free software
  In other words, custom branches feed programmers!

Free versus open source
  Free software: source is available, free.
  Open source: source is available for people to review and change, but that may not be free.
Version control refers to:
   ensuring that components change together when appropriate.
   constructing and maintaining new **baselines**
   managing changes to baselines
   ensuring an orderly progress from a change to its verification (through testing)
   tracking the state of a configuration as it is incrementally verified.

Three concepts
   revision control: does a component change?
   version control: does a baseline change?
   change control: is a change reasonable?
Version control
a human process
with plenty of tools available to help.
Main tools: revision control and version control.
Revision control tools

- maintain revision history for a single file.
- store changes, not contents.

**reverse-differential form**: latest version plus changes needed to go backward.

Reference implementation: **Revision Control System (RCS)**

Inside RCS:

```
rcs -l file   # lock a file so only you can change it.
co file      # get a copy of file
co -u1.2 file # get a copy of revision 1.2
co -l file    # get a copy and lock
co -l2.3 file # get a copy of revision 2.3 and lock it.
ci  file      # put a copy into the revision history
ci -u file    # put a copy and retain one for yourself
ci -l file    # put a copy and keep it locked
rlog file     # list revision history
```
RCS properties
  works on single files
  manages the state of a single file.
  very useful for prototyping!
Version control tools

- maintain revision history for a tree of files.
- use revision control on each file, in parallel.
- reference implementations: Concurrent Version System (CVS), SubVersion (SVN), etc.

Inside CVS

cvs checkout dir  # check out a project

cvs commit dir    # check in a version

cvs release dir   # release a version

cvs update dir    # update files that have changed in repository

Main differences between RCS and CVS:

<table>
<thead>
<tr>
<th>RCS</th>
<th>CVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>works on an individual</td>
<td>works on a directory hierarchy</td>
</tr>
<tr>
<td>file</td>
<td></td>
</tr>
<tr>
<td>manages state of one</td>
<td>manages state of the whole</td>
</tr>
<tr>
<td>file</td>
<td>hierarchy as a unit</td>
</tr>
</tbody>
</table>

A word to the wise

Storing a revision is really robust.
Merging revisions is a dirty hack.
Limits of RCS/CVS/SVN version control

designed around text files; versions of binary files are inefficient to record.
line-oriented mechanisms for recording differences; does not work well with auto-formatting.
More modern source control

Web-based.
Upon a subscription model.
Source code is closed.
(And/or the configuration is exceedingly complex)
=> web site can argue for a monopoly.

Examples

github
sourceforge
codeplex

...
Thinking more broadly, the configuration of a piece of software includes the things not written by you that are needed for it to run.

- External libraries
- External files.
- External components/programs.
Basic principle of configuration

You can run, but you can't hide complexity.
   It goes somewhere.
   Into the software.
   Into the configuration.
Deployment:
The process of creating a runnable copy of a software project.
Input: a configuration for the software.
Output: a runnable copy.
Deployment is more than compilation…
Are system requirements met? Check them.
Are other components needed? Install them.

Typical deployment path

**Compile** a version of the software.
**Package** the compiled files (without the sources)
**Install** the package on the target computer.
Packages and dependencies

A "package" is a distributable unit of software.
- Contains a mix of executable code, libraries, files.
- Can optionally contain source code.

A "package manager" is software that installs and maintains packages.

Defacto standard for linux: RedHat Package Manager (RPM). (file format used by many package managers)
- Several managers use the RPM format.

Defacto package manager for windows: MIS Microsoft Software Installer format.
- Several competing formats for "installers"
Inside RPM

A configuration file ('specfile'):
- Lists files to be packaged, and where to install them.
- Lists dependencies between packages, e.g., Package A **REQUIRES** Package B.
- Defines minimum versions for required packages.

Creating an RPM package:
- Edit a specfile for the package.
- Invoke rpmbuild to create the package from source files.
- Distribute the package file created by rpmbuild.

Using an RPM package
- The "rpm --install" command (or a number of variants):
  - Looks up dependencies.
  - Installs them if possible.
  - Installs the package.
  - Runs pre-install and post-install scripts, as needed.
Windows software packages

Many third-party "installers".
Microsoft standard: Windows Installer (.msi) files.

Inside an .msi file:
  - files to install.
  - registry changes to make.
  - scripts to run.

Much simpler than RPMs: no dependency declarations.
Repositories

A **repository** is a resource (usually web based) from which one can download packages. This is used to satisfy dependencies between packages.
Challenges of package-based software deployment

Package conflicts:
One RPM package requires gzip 1.2, the other requires gzip 1.0. Since requirements can't be co-resident, neither can the software.

Cross-package corruption:
RedHat packages can modify files from other packages without consent.
MSI packages can modify the registry key for any other packages.
These are common propagation vectors for spyware.

Sequencing:
Package installation must often be done in a specific order.
E.g., install a language before installing a program written in that language, because that program's installer is written in that language (perl, python, etc)
E.g., append contents to system files in a specific order.
(RPM sequences installs according to dependencies declared in packages.)
Create special-purpose machines on which to install complex software. (e.g., for Cadence)
Create virtual machines rather than installers; run as virtual instances under a host OS (e.g., Hadoop training machines)