Software Engineering

Configuration Management

Topics covered

- Change management
- Version management
- System building
- Release management

Configuration management

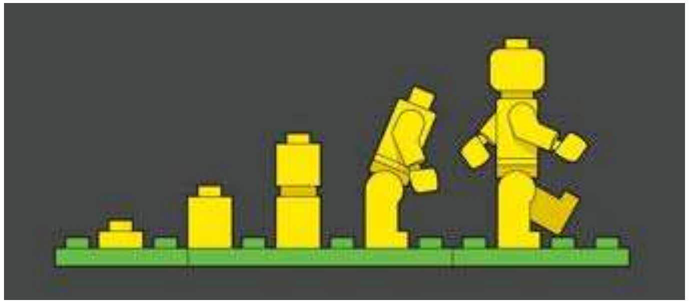
- Software changes frequently,
  - systems, can be thought of as a set of versions - each of which has to be maintained and managed.
- Software versions implement:
  - proposals for change, corrections of faults, adaptations for different hardware and operating systems.
- Configuration management (CM)
  - policies, processes and tools for managing changing software systems.
  - it is easy to lose track of what changes and component versions have been incorporated into each system version.

Configuration management activities (1 of 4)

- **Change management**
  - Keeping track of requests for changes to the software from customers and developers,
  - working out the costs and impact of changes,
  - and deciding the changes should be implemented.

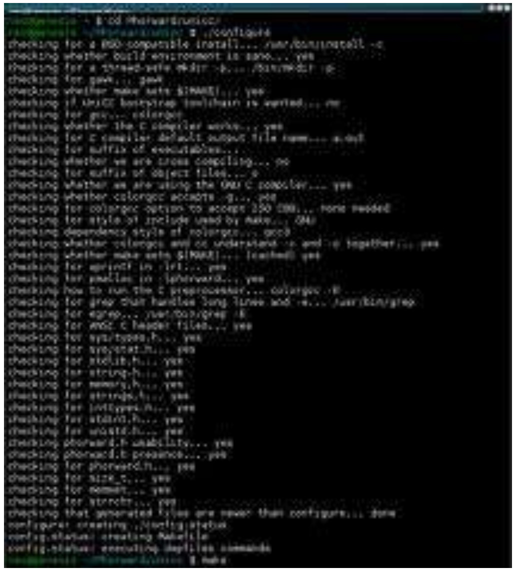
Configuration management activities (2 of 4)

- **Version management**
  - Keeping track of the multiple versions of system components,
  - and ensuring that changes made to components by different developers do not interfere with each other.



Configuration management activities(3 of 4)

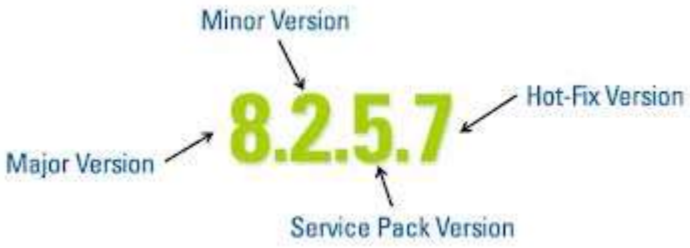
- **System building**
  - The process of assembling program components, data and libraries,
  - then compiling these to create an executable system.



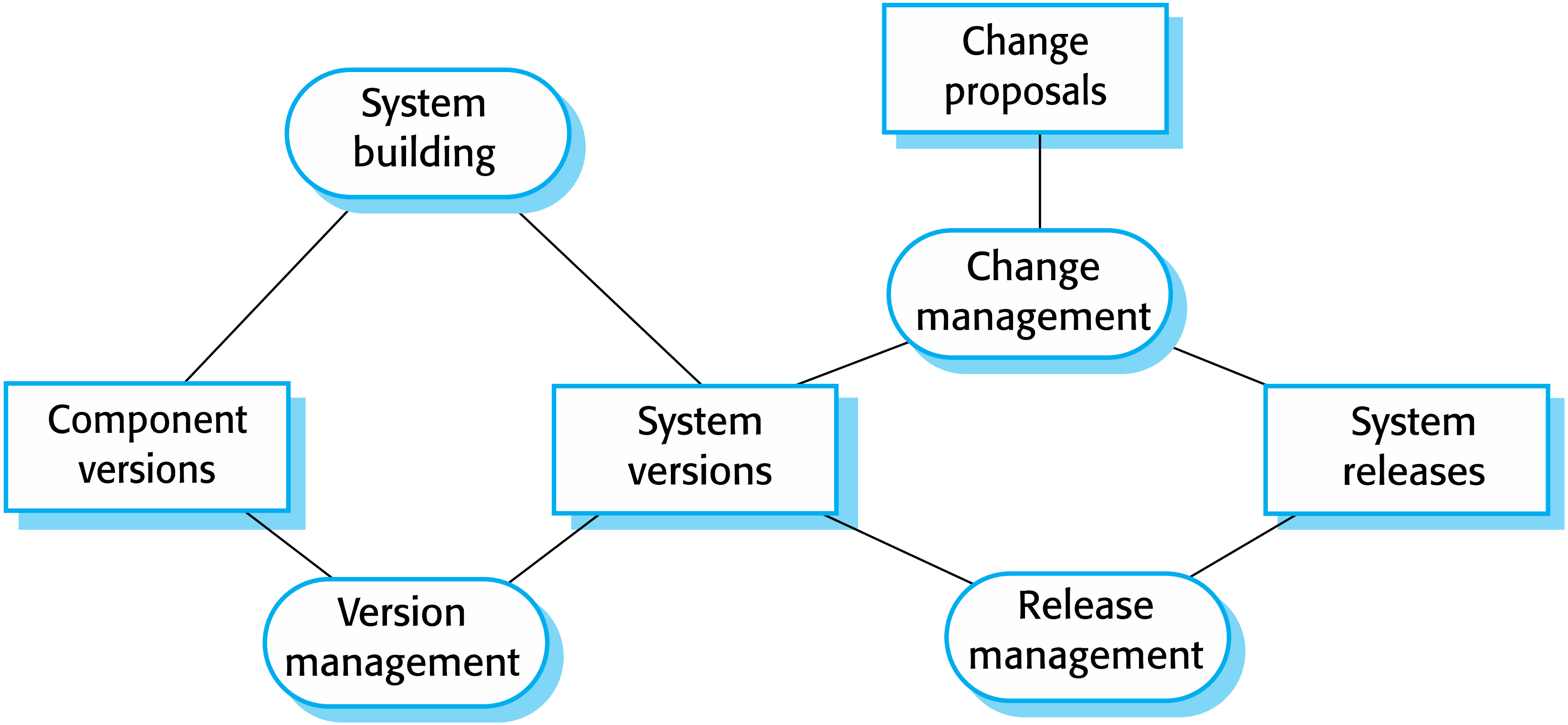
```
[illegible] ~ $ cd [illegible]
[illegible] [illegible] $ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking if UniCC bootstrap toolchain is wanted... no
checking for gcc... colorgcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether colorgcc accepts -g... yes
checking for colorgcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of colorgcc... gcc3
checking whether colorgcc and cc understand -c and -o together... yes
checking whether make sets $(MAKE)... (cached) yes
checking for sprintf in -lrt... yes
checking for pmalloc in -lphorward... yes
checking how to run the C preprocessor... colorgcc -E
checking for grep that handles long lines and -e... /usr/bin/grep
checking for egrep... /usr/bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking phorward.h usability... yes
checking phorward.h presence... yes
checking for phorward.h... yes
checking for size_t... yes
checking for memset... yes
checking for strrchr... yes
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
[illegible] [illegible] $ make
```

Configuration management activities(4 z 4)

- **Release management**
  - Preparing software for external release,
  - and keeping track of the system versions that have been released for customer use.



Configuration management activities



Agile development and CM

- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.
- They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.

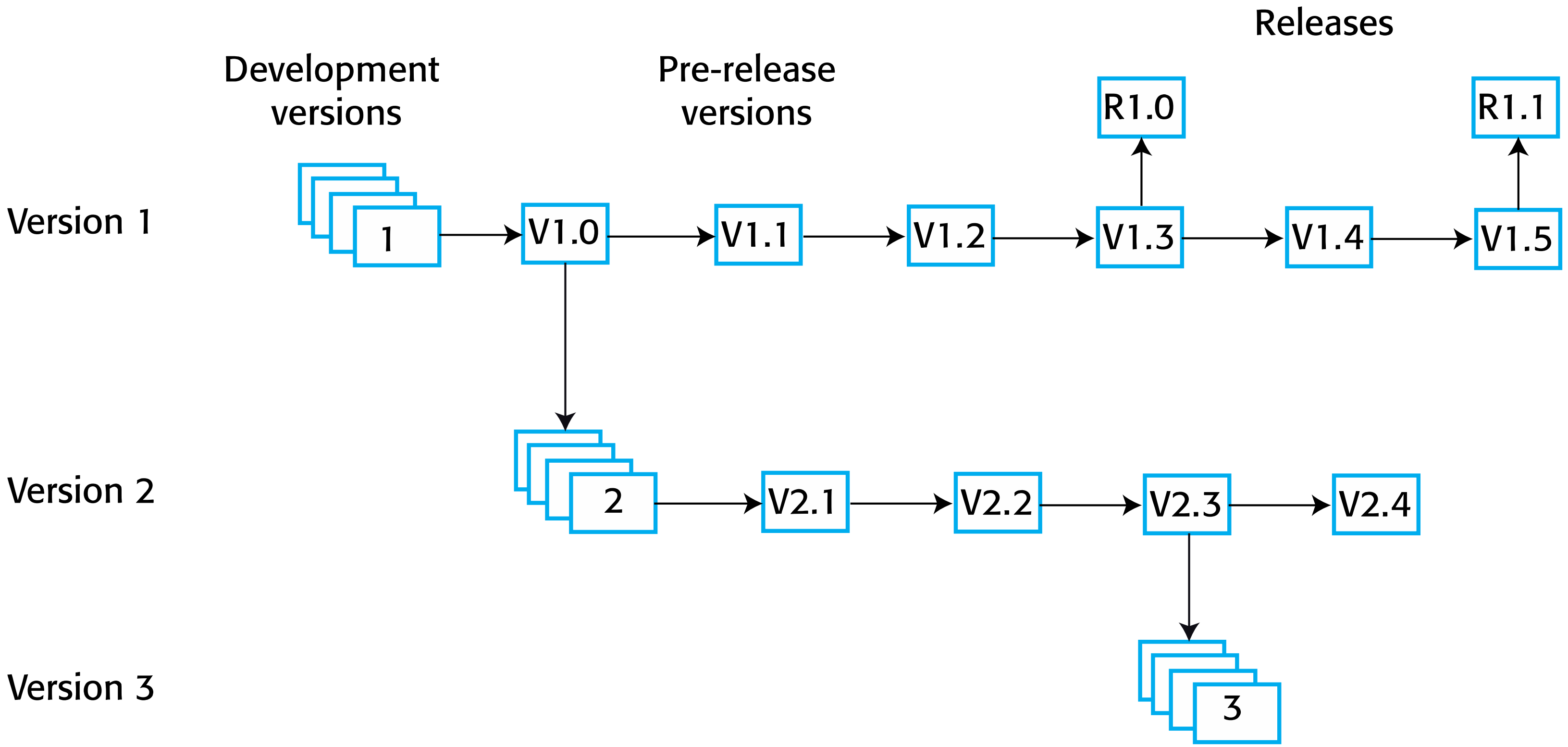
Development phases

- A development phase where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- A system testing phase where a version of the system is released internally for testing.
  - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- A release phase where the software is released to customers for use.
  - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

Multi-version systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

Multi-version system development



CM terminology (1 of 2)

Term	Explanation
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to recreate a baseline from its constituent components.

CM terminology (2 of 2)

Term	Explanation
Mainline	A sequence of baselines representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

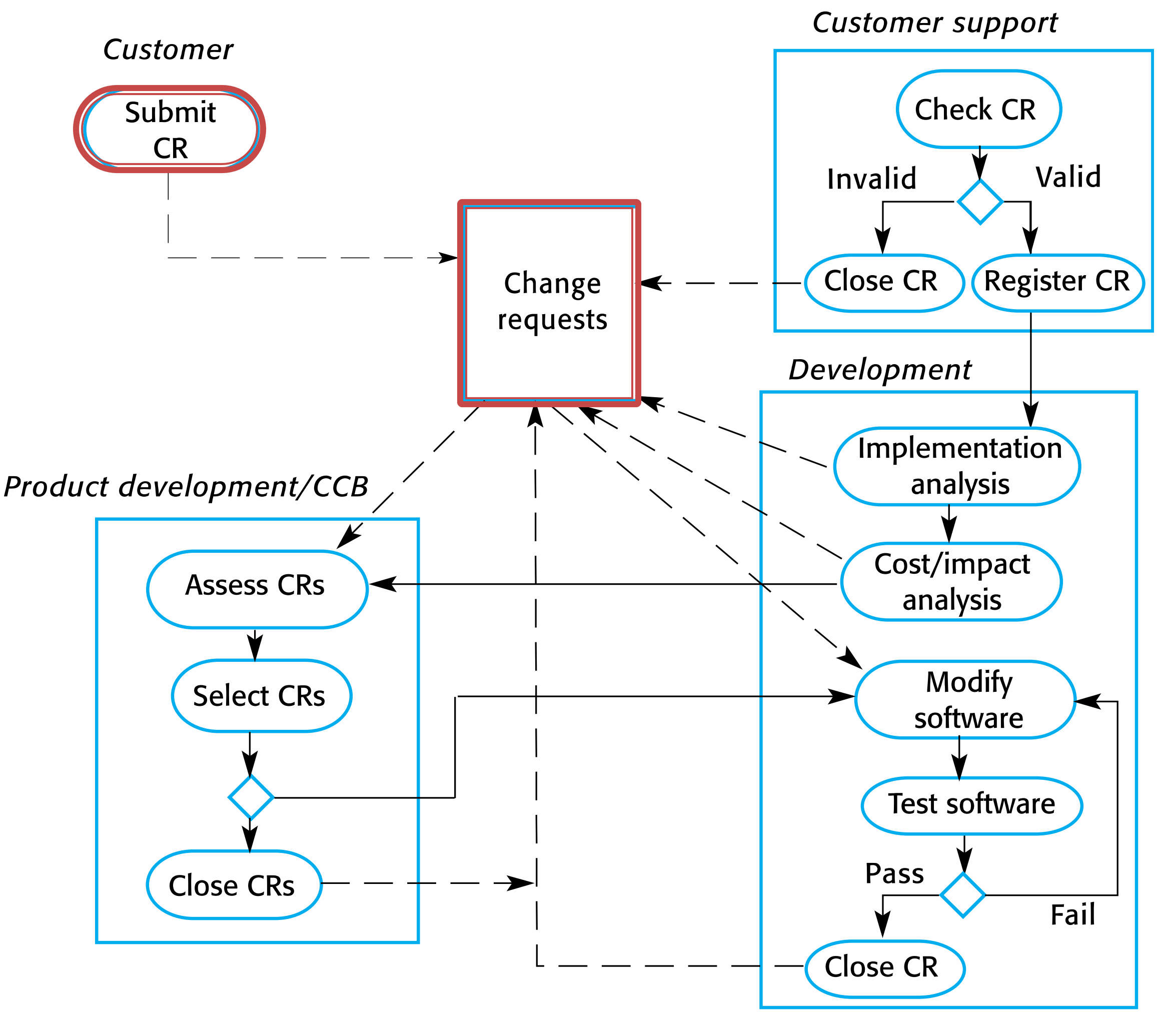
Change management

Configuration management

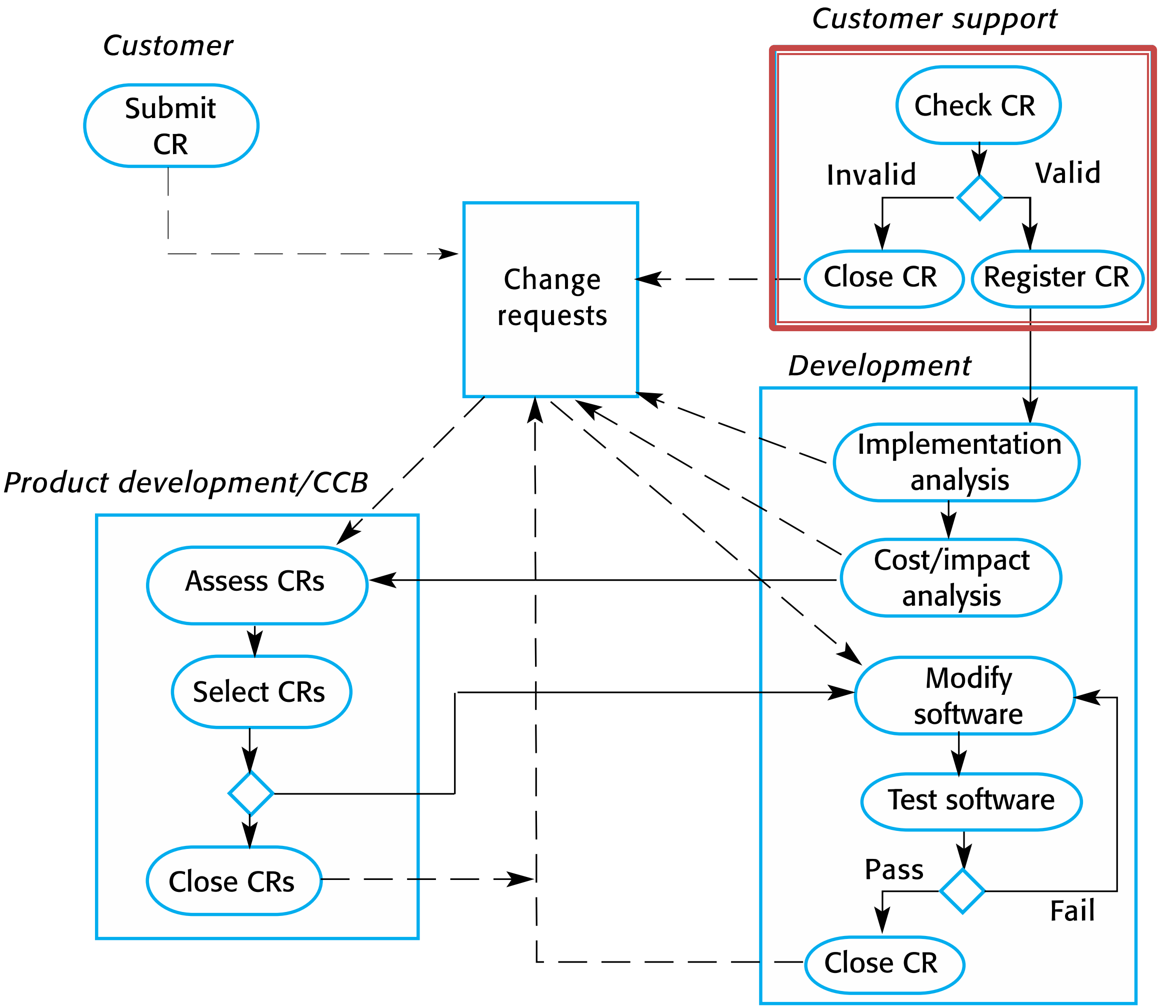
Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with:
  - analyzing the costs and benefits of proposed changes,
  - approving those changes that are worthwhile,
  - tracking which components in the system have been changed.

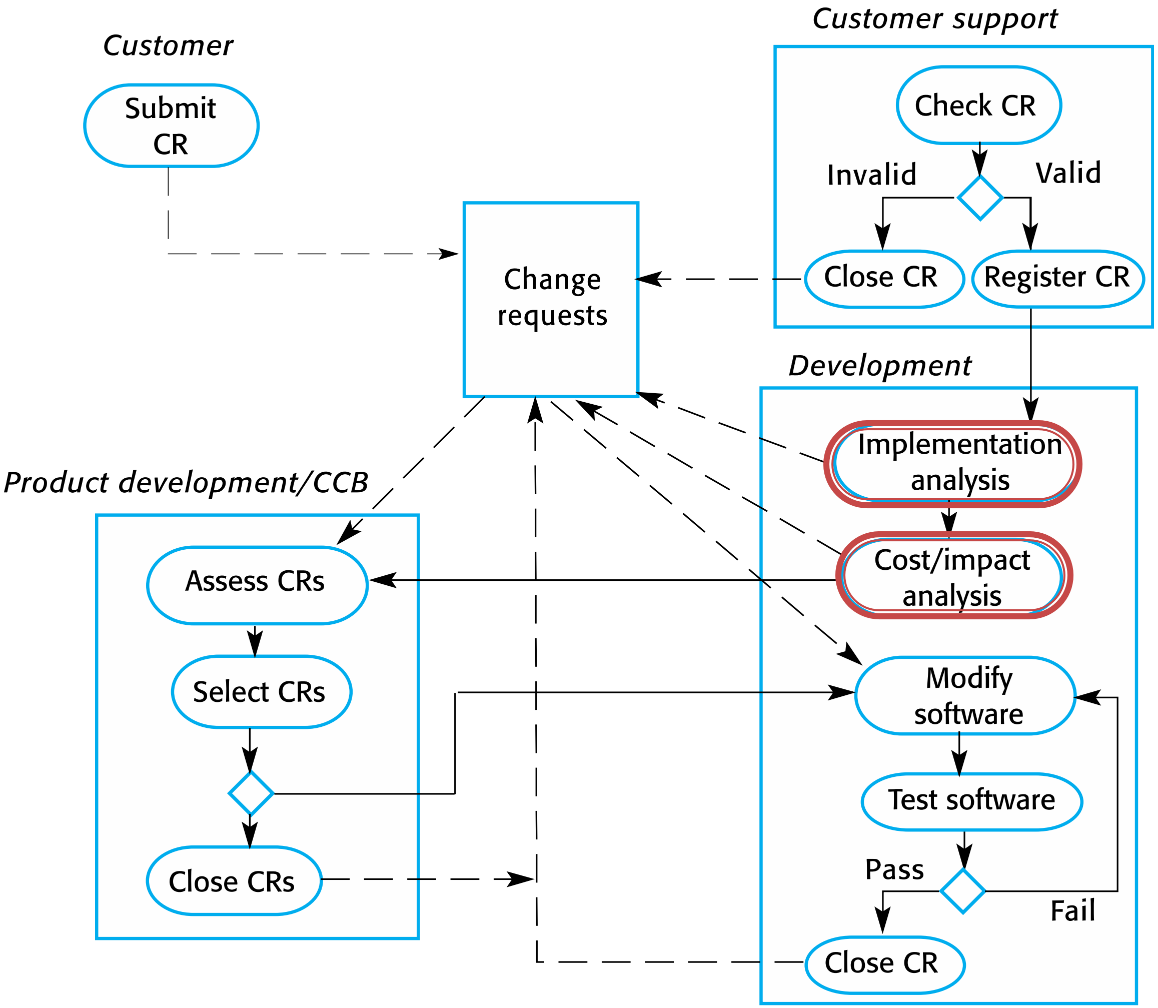
The change management process



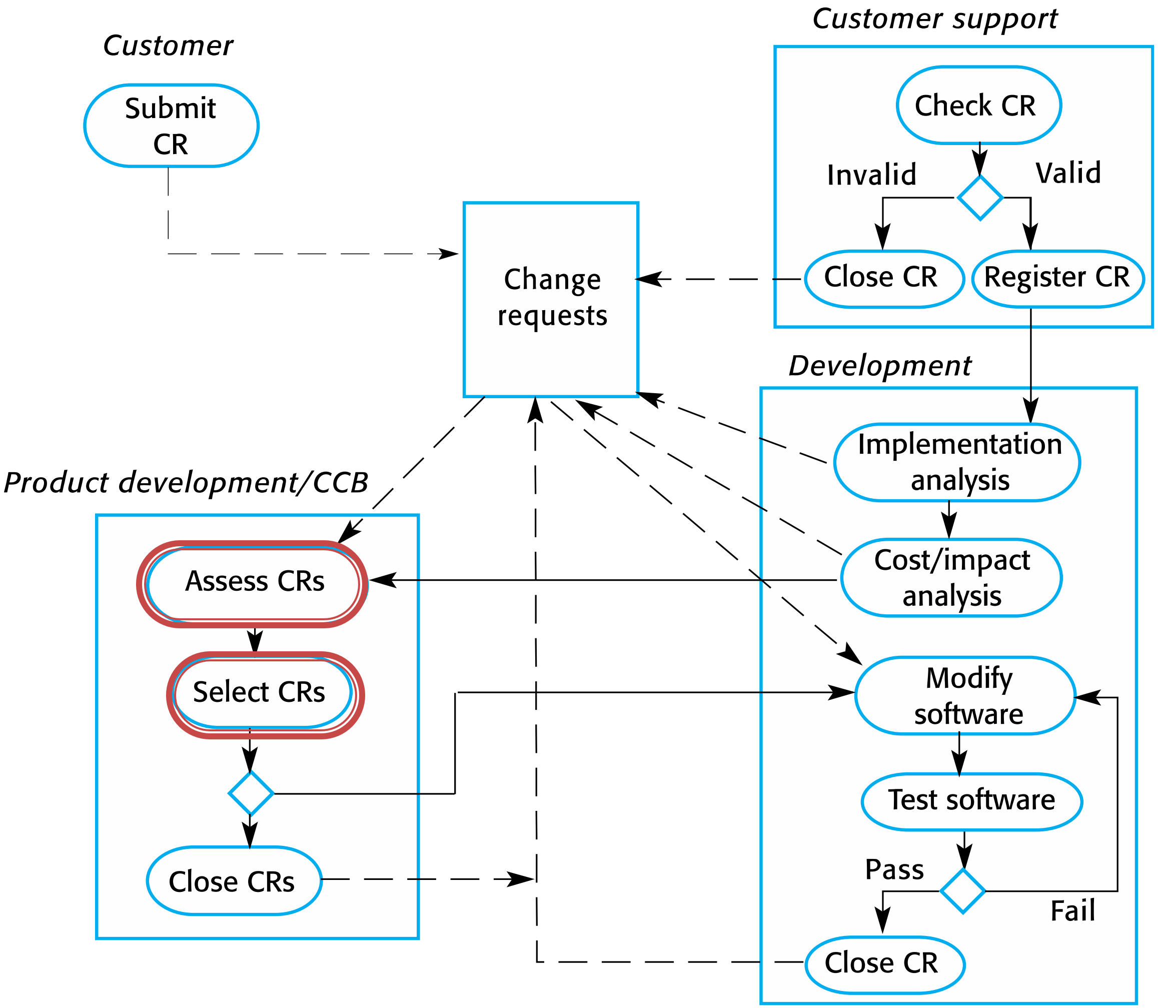
The change management process



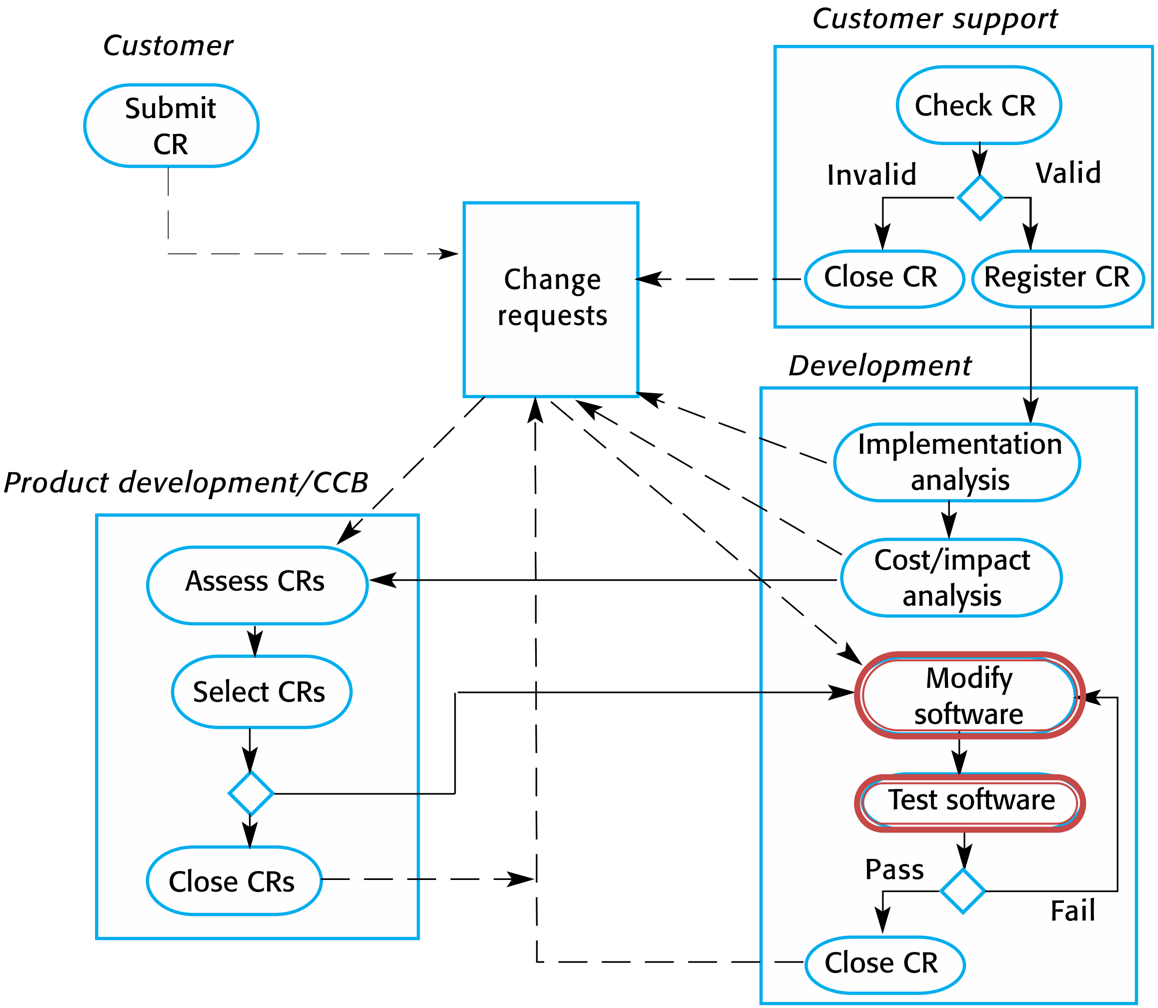
The change management process



The change management process



The change management process



A partially completed change request form (a)

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

A partially completed change request form (b)

Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments:

Factors in change analysis

1. The consequences of not making the change
2. The benefits of the change
3. The number of users affected by the change
4. The costs of making the change
5. The product release cycle

Change management and agile methods

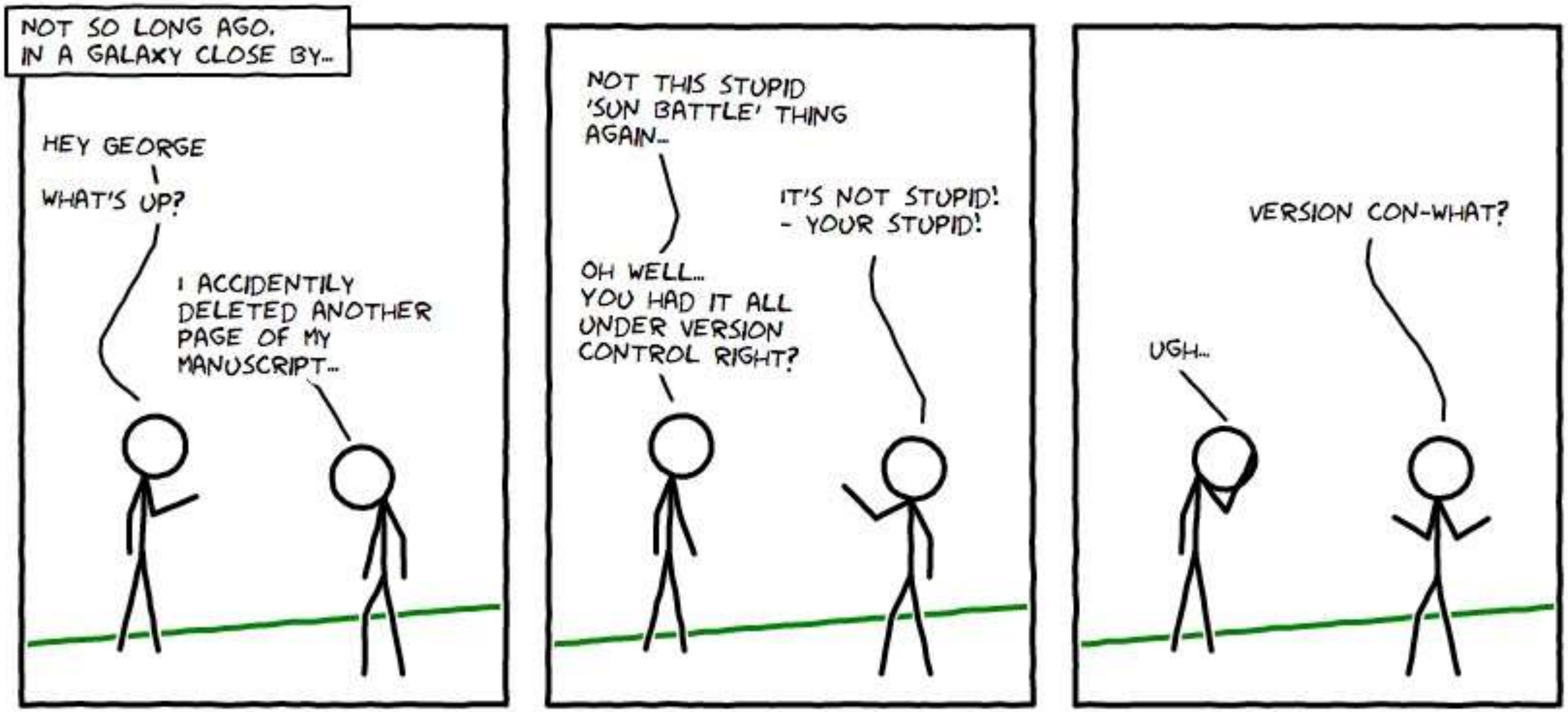
- In some agile methods, customers are directly involved in change management.
- The propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

Derivation history

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2009
//
// © St Andrews University 2009
//
// Modification history
// Version Modifier   Date          Change       Reason
// 1.0     J. Jones   11/11/2009    Add header   Submitted to CM
// 1.1     R. Looek   13/11/2009    New field    Change req. R07/02
```

Change management tools

- ***bug/defect tracking, issue-tracking***
  - Bugzilla,
  - Trac,
  - Redmine,
  - JIRA,
  - Zentrac,
  - Pivotal Tracker,
  - FogBugz,
  - Lighthouse


NOT SO LONG AGO.
IN A GALAXY CLOSE BY...
HEY GEORGE
WHAT'S UP?
I ACCIDENTILY DELETED ANOTHER PAGE OF MY MANUSCRIPT...
NOT THIS STUPID 'SUN BATTLE' THING AGAIN...
OH WELL... YOU HAD IT ALL UNDER VERSION CONTROL RIGHT?
IT'S NOT STUPID! - YOUR STUPID!
VERSION CON-WHAT?
UGH...

Return to Zero

WHAT'S THIS CUP OF USB DRIVES?
AH, THAT'S HOW I VERSION-CONTROL MY FILES...
TYPE
TYPE
TYPE
STAN'S DESK!
IT'S BRILLANT REALLY. JUST MAKE A NEW COPY ON A NEW ONE EVERYDAY.
WHAT HAPPENS WHEN YOUR CUP FILLS UP?
EASY! I DUMP THEM IN HERE!
USB

EEWeb.com

Version management

Configuration management

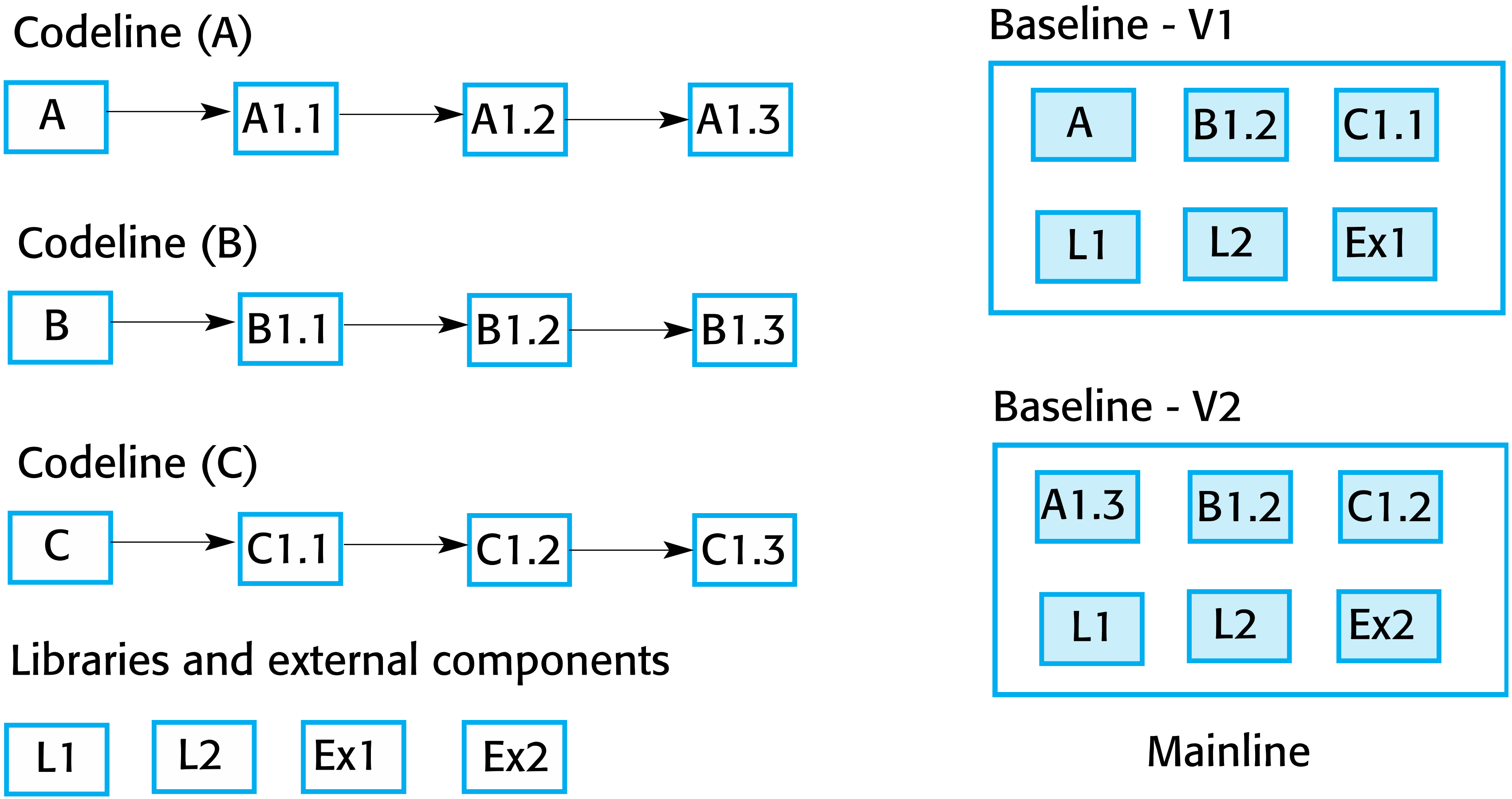
Version management

- Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.

- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.

- Therefore version management can be thought of as the process of managing codelines and baselines.

Codelines and baselines



Codeline (A)

A → A1.1 → A1.2 → A1.3

Codeline (B)

B → B1.1 → B1.2 → B1.3

Codeline (C)

C → C1.1 → C1.2 → C1.3

Libraries and external components

L1 L2 Ex1 Ex2

Baseline - V1

A B1.2 C1.1

L1 L2 Ex1

Baseline - V2

A1.3 B1.2 C1.2

L1 L2 Ex2

Mainline

Baselines

- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system.
  - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

Version management systems

1. Version and release identification
   - Managed versions are assigned identifiers when they are submitted to the system.
2. Change history recording
   - All of the changes made to the code of a system or component are recorded and listed.
3. Support for Independent development
   - The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

Version management systems

4. Project support
   - A version management system may support the development of several projects, which share components.
5. Storage management
   - To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.

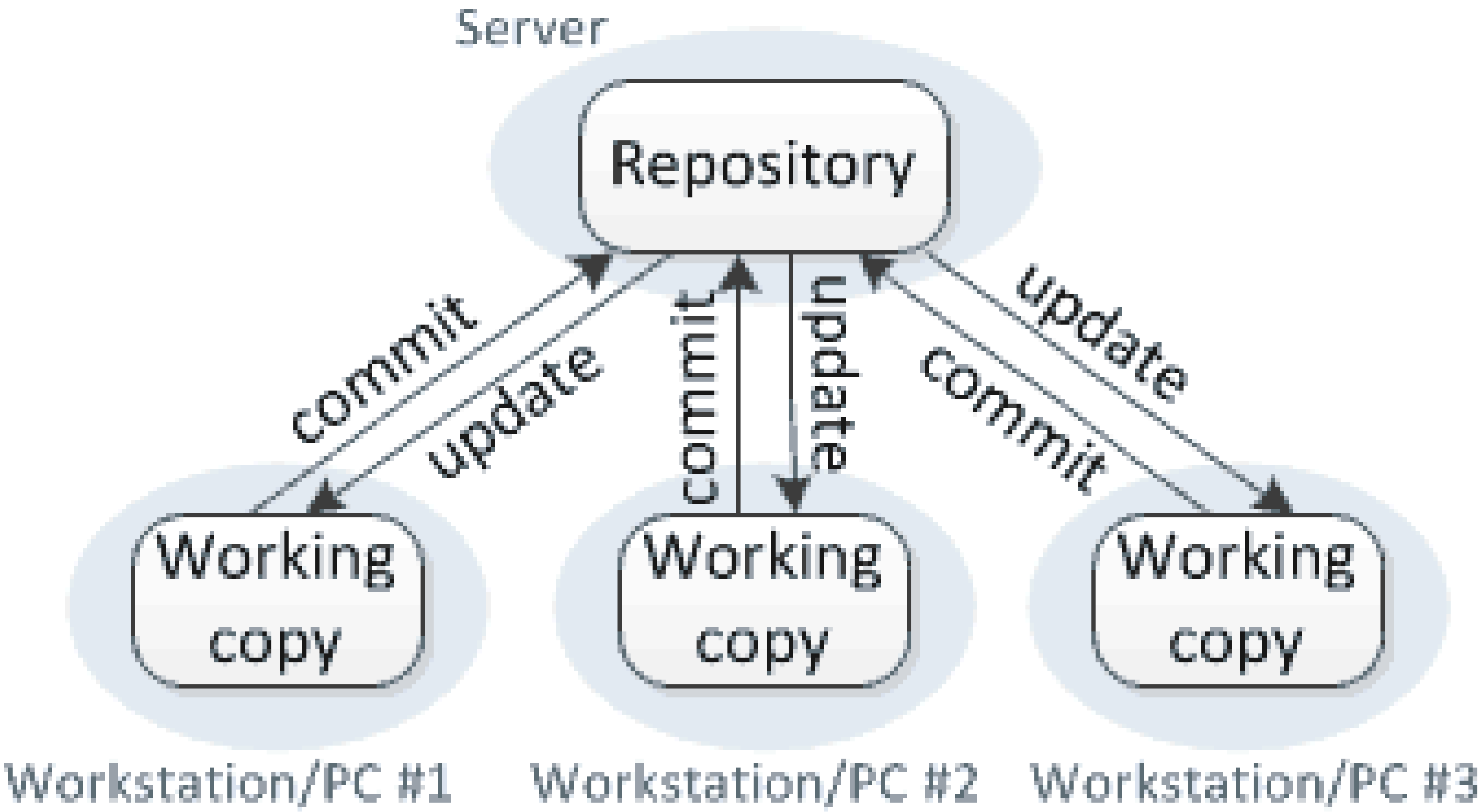
Public repository and private workspaces

- To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
- The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
- When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- When they have finished their changes, the changed components are returned (checked-in) to the repository.

Modern types of version management systems (1)

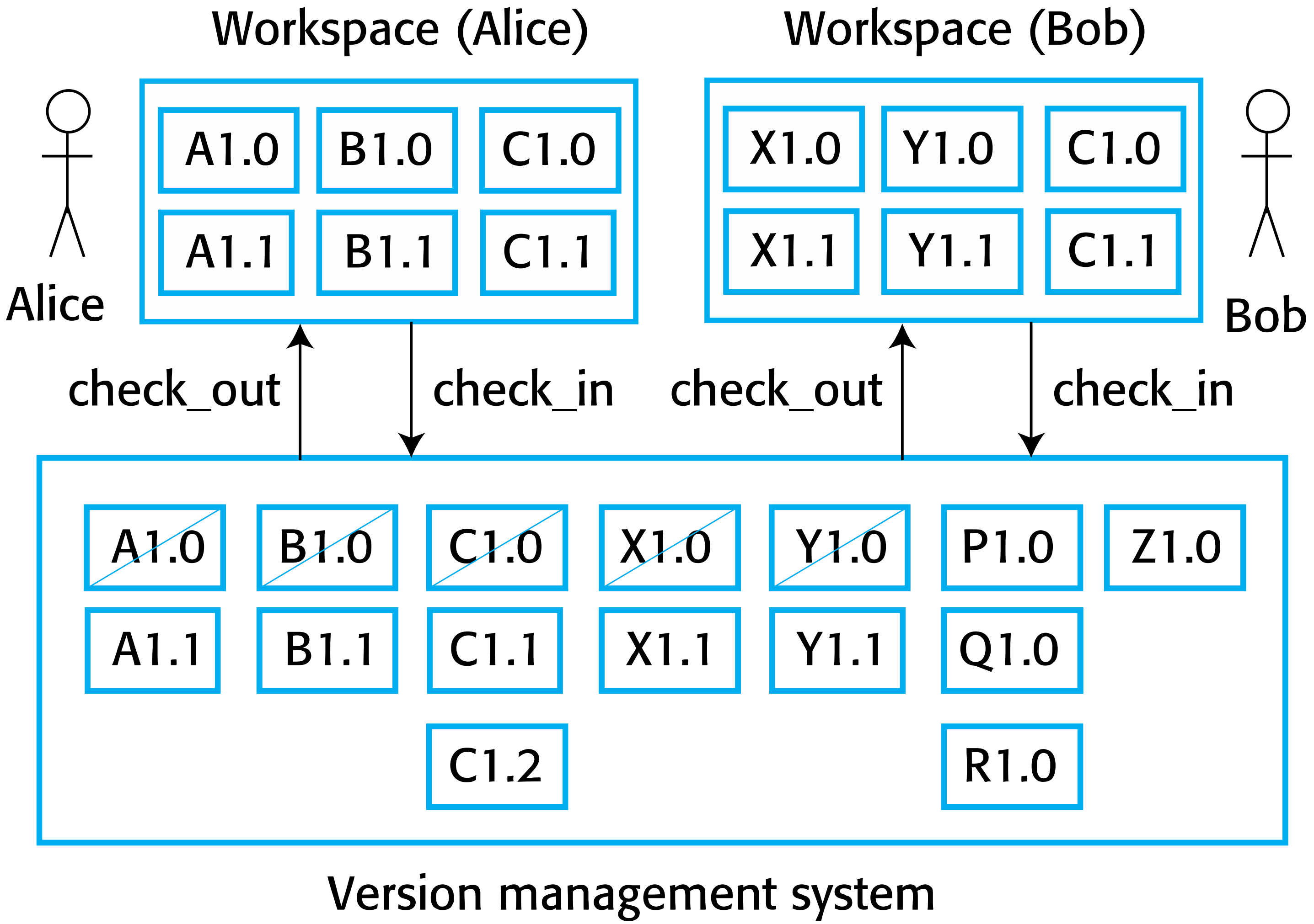
- **Centralized** Version Control Systems
  - components from the project repository into their private workspace and work on these copies in their private workspace.
  - When their changes are complete, they check-in the components back to the repository.

Centralized version control



e.g. CVS, Subversion, Perforce

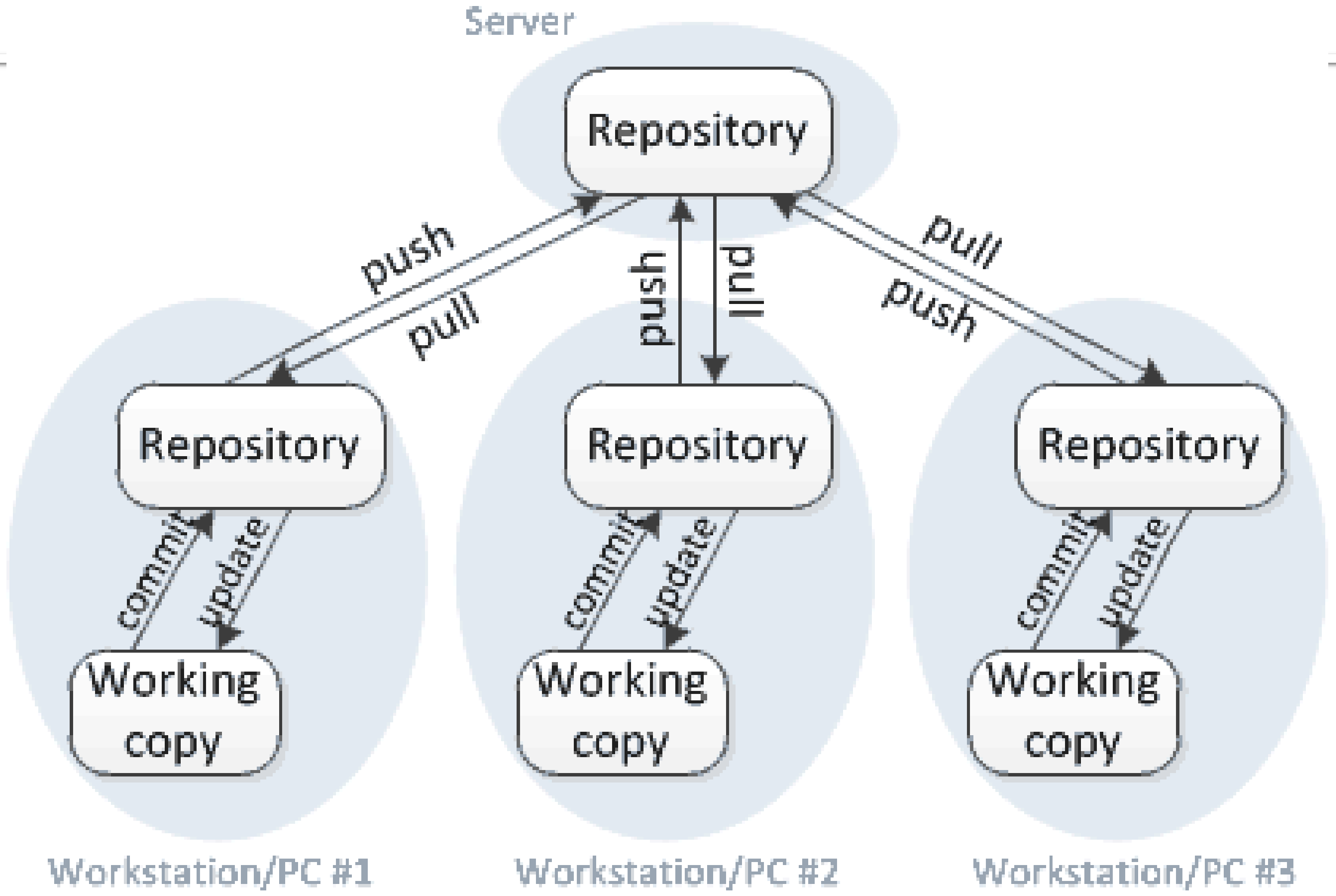
Check-in and check-out from a version repository



Modern types of version management systems (2)

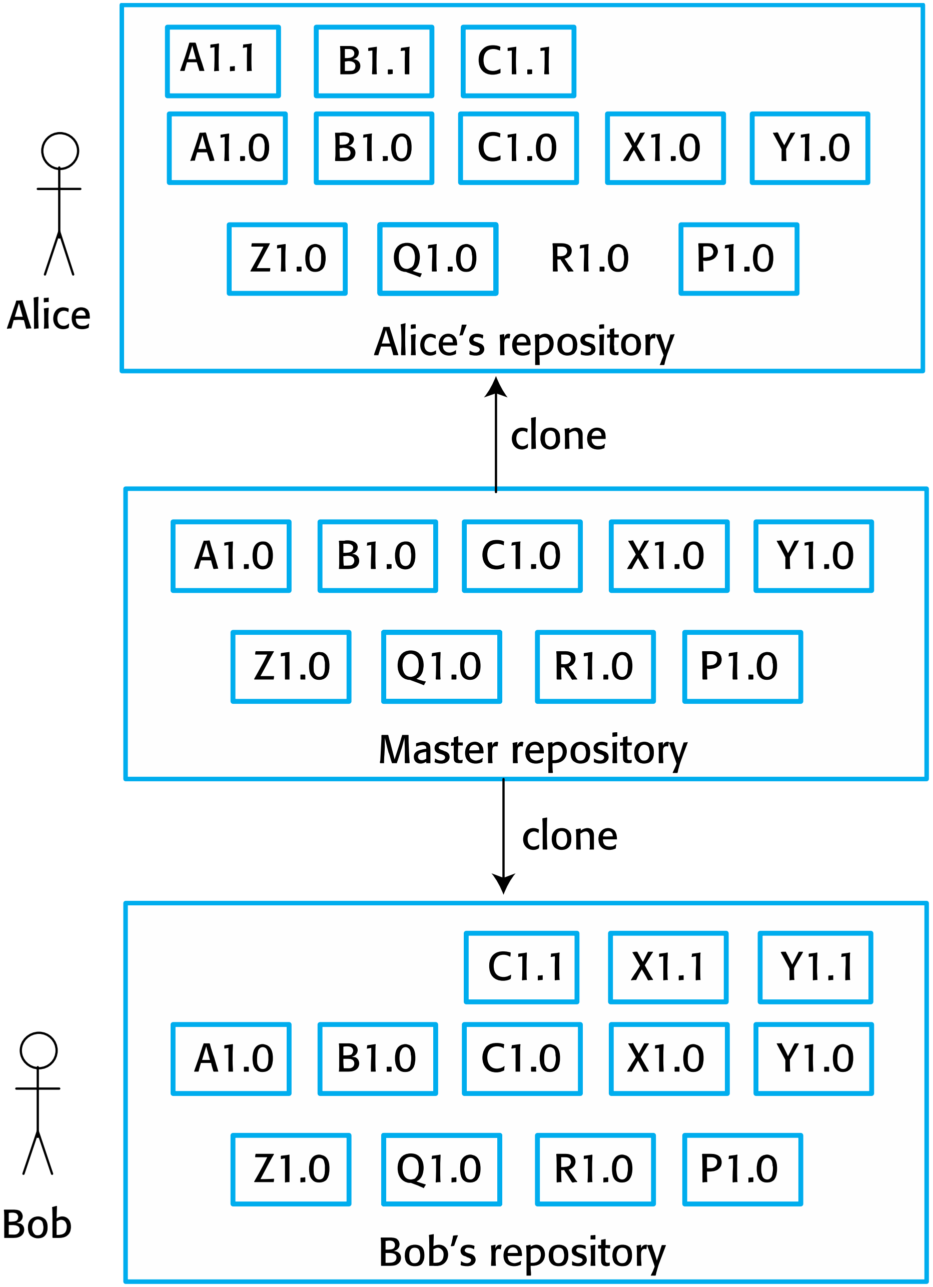
- **Distributed** Version Control Systems
  - A 'master' repository is created on a server that maintains the code produced by the development team.
  - Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
  - Developers work on the files required and maintain the new versions on their private repository on their own computer.
  - When changes are done, they 'commit' these changes and update their private server repository. They may then 'push' these changes to the project repository.

Distributed version control



e.g. Git, Mercurial, Bazaar, Darcs

Repository cloning



Alice

A1.1 B1.1 C1.1

A1.0 B1.0 C1.0 X1.0 Y1.0

Z1.0 Q1.0 R1.0 P1.0

Alice's repository

clone

A1.0 B1.0 C1.0 X1.0 Y1.0

Z1.0 Q1.0 R1.0 P1.0

Master repository

clone

Bob

C1.1 X1.1 Y1.1

A1.0 B1.0 C1.0 X1.0 Y1.0

Z1.0 Q1.0 R1.0 P1.0

Bob's repository

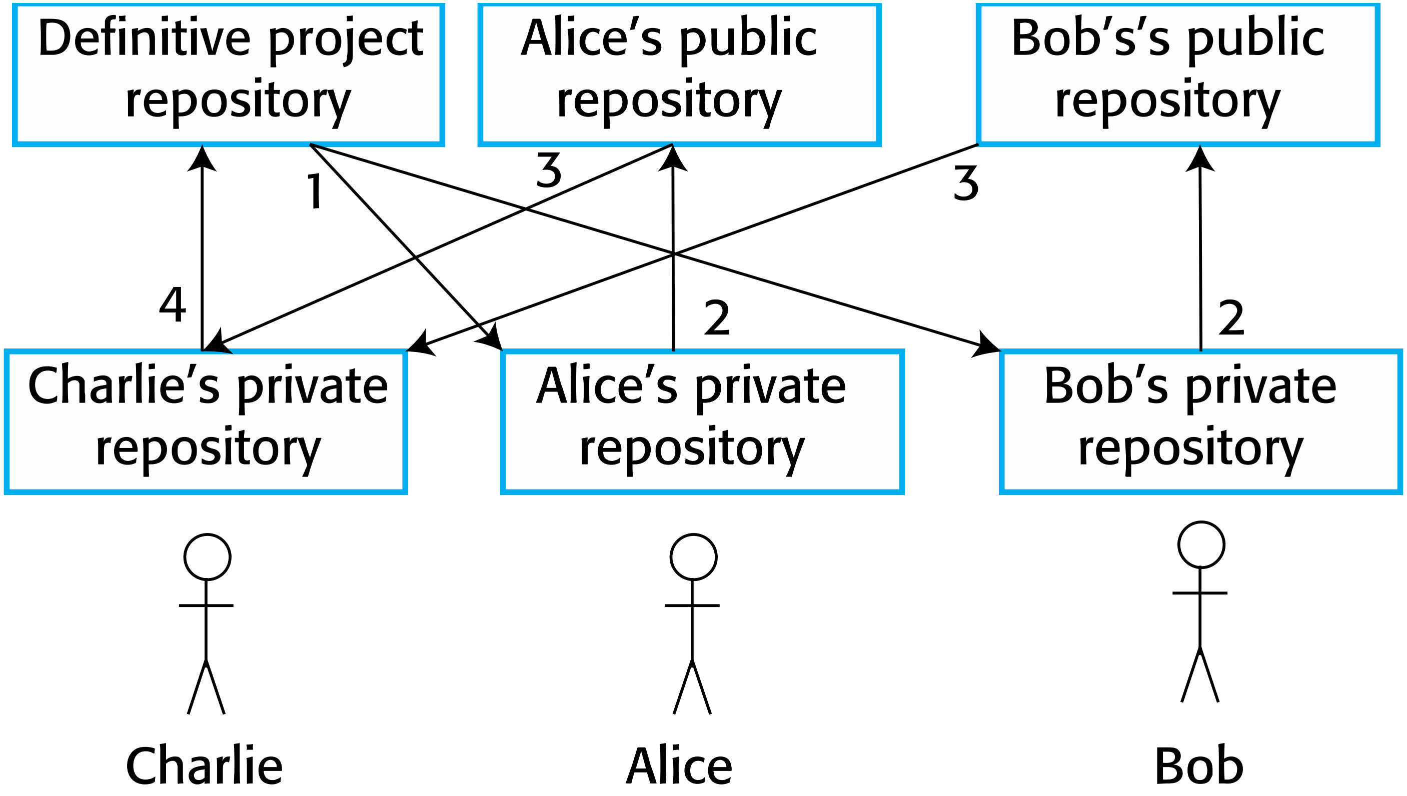
Benefits of distributed version control

- It provides a backup mechanism for the repository.
  - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- It allows for off-line working so that developers can commit changes if they do not have a network connection.
- Project support is the default way of working.
  - Developers can compile and test the entire system on their local machines and test the changes that they have made.

Open source development

- Distributed version control is essential for open source development.
  - Several people may be working simultaneously on the same system without any central coordination.
- As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

Open-source development


Definitive project repository
Alice’s public repository
Bob’s’s public repository
1
3
3
4
2
2
Charlie’s private repository
Alice’s private repository
Bob’s private repository
Charlie
Alice
Bob

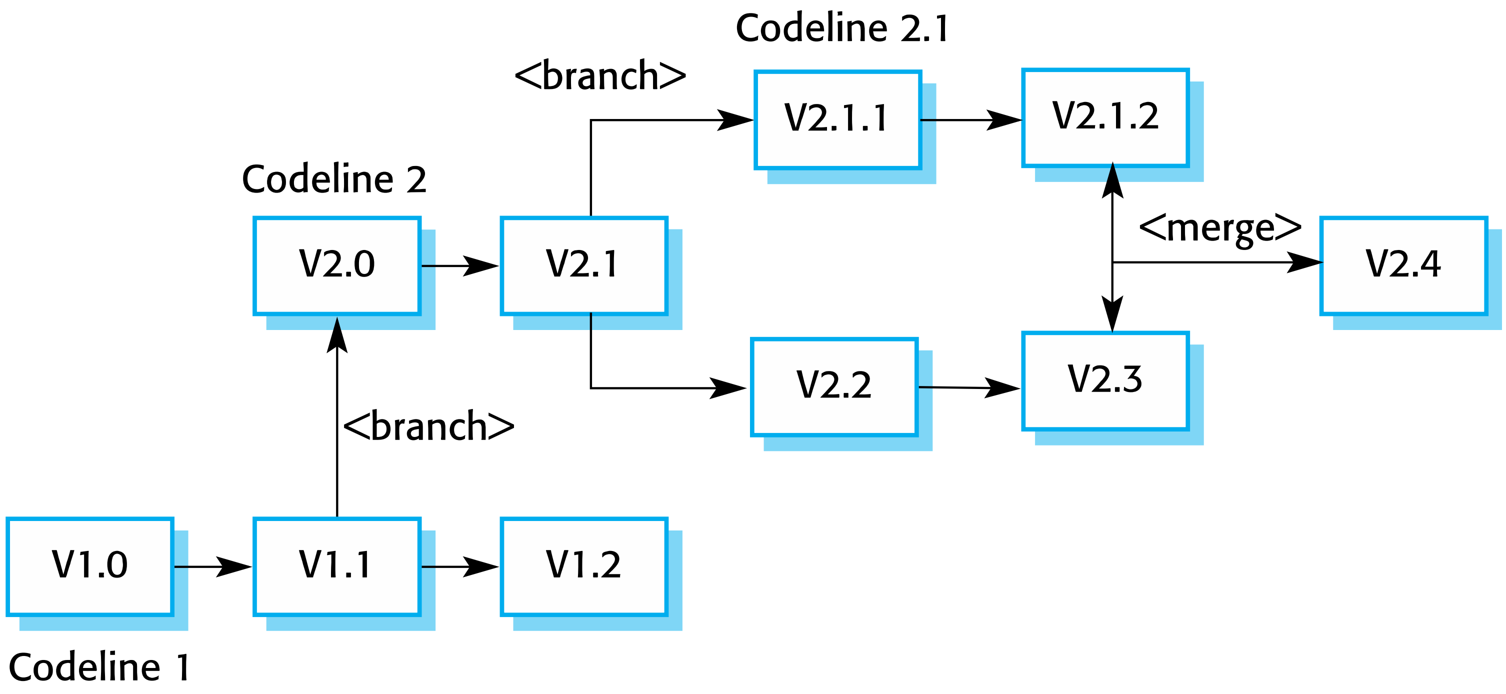
Version control systems as a service

- Github (Git)
- Bitbucket (Git, Mercurial)
- Launchpad (Bazaar)
- CloudForge(SVN, Git)

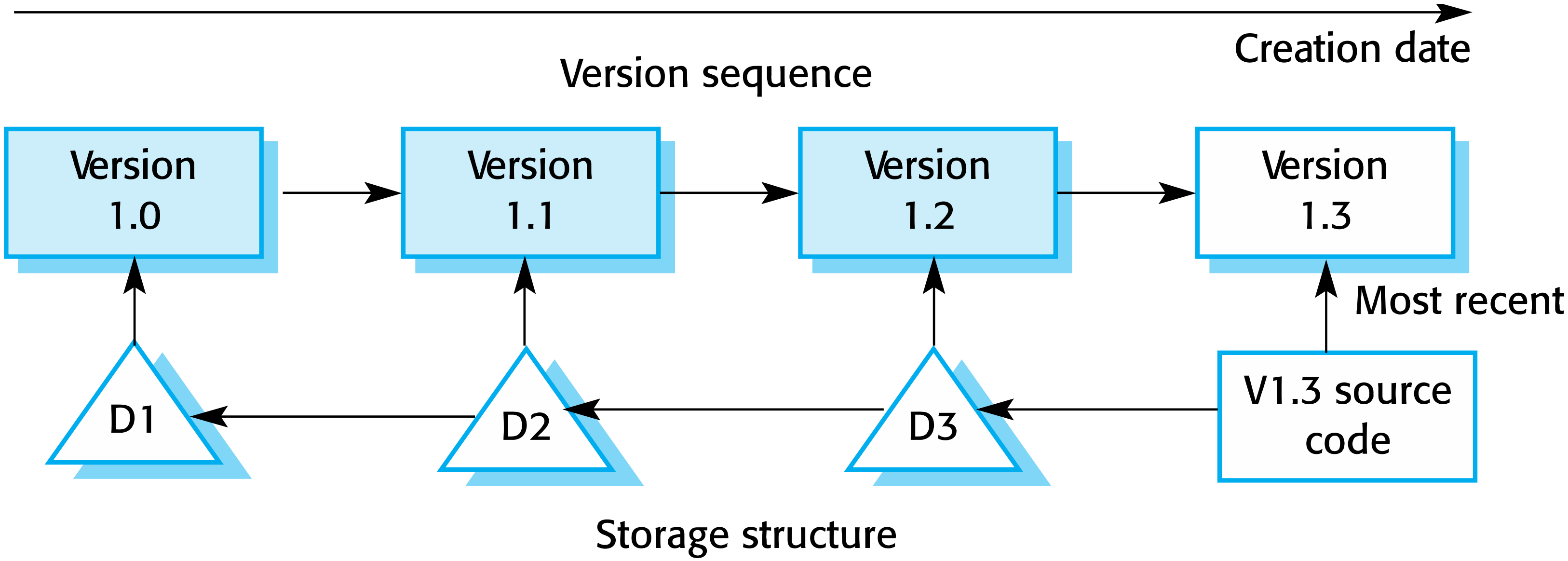
Branching and merging

- Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences (**branches**).
  - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- At some stage, it may be necessary to **merge** codeline branches to create a new version of a component that includes all changes that have been made.
  - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

Branching and merging



Storage management using deltas



Storage management in Git

- As disk storage is now relatively cheap, Git uses an alternative, faster approach.
- Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
- It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

System building

Configuration management

System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The configuration description used to identify a baseline is also used by the system building tool.

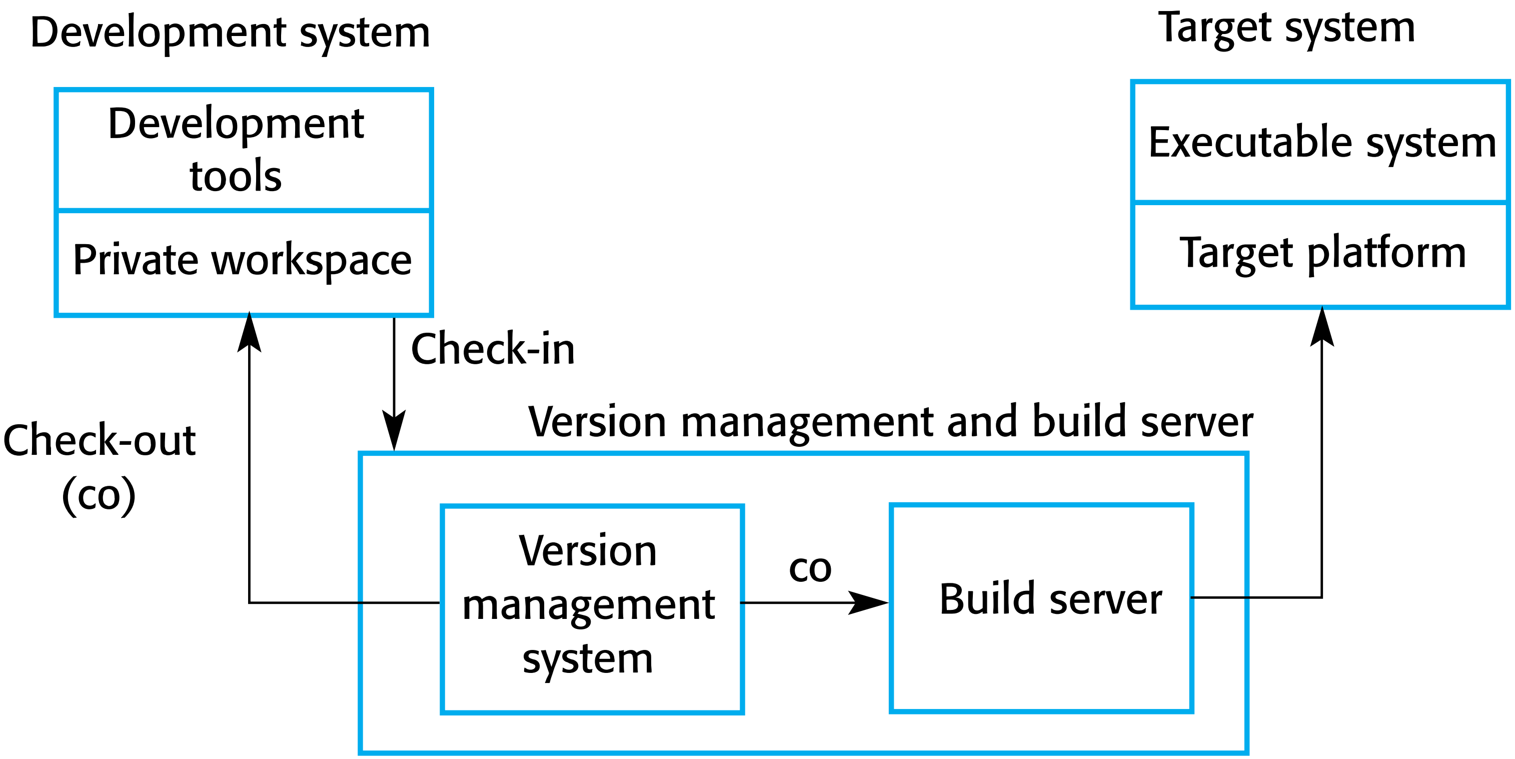
Build platforms

1. The development system, which includes development tools such as compilers, source code editors, etc.
   - Developers check out code from the version management system into a private workspace before making changes to the system.

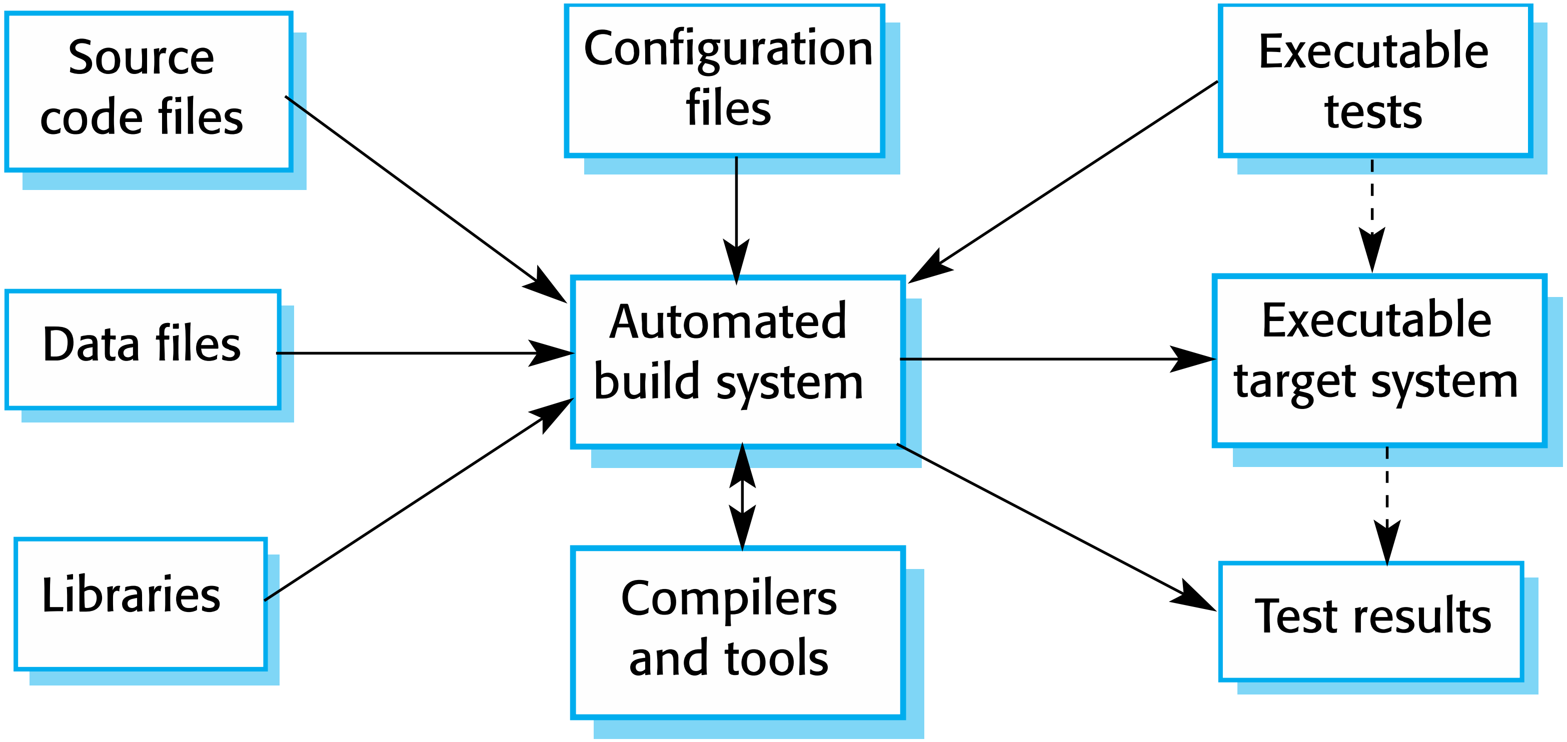
2. The build server, which is used to build definitive, executable versions of the system.
   - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.

3. The target environment, which is the platform on which the system executes.

Development, build, and target platforms



System building



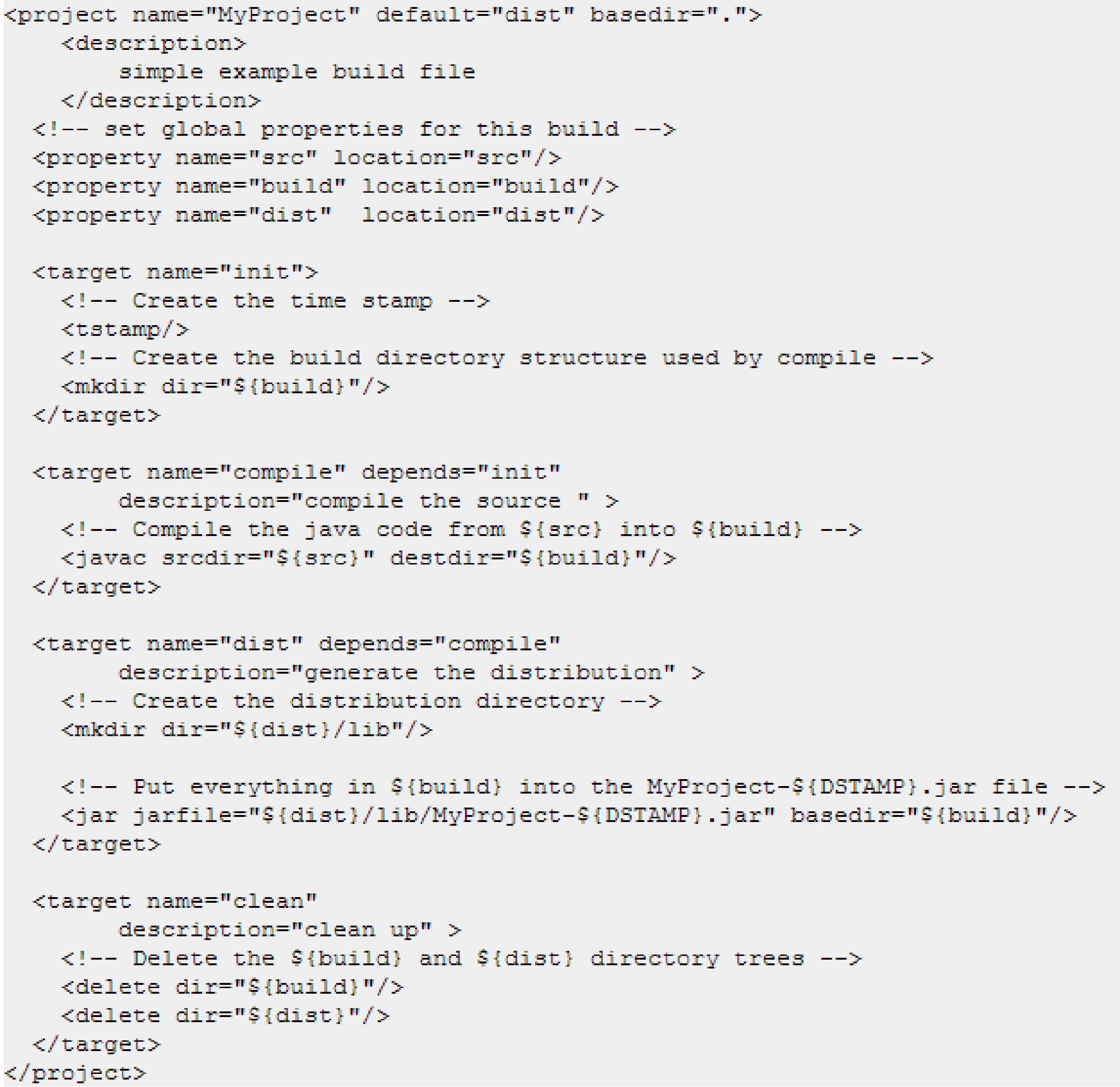
Build system functionality

1. Build script generation
2. Version management system integration
3. Minimal re-compilation
4. Executable system creation
5. Test automation
6. Reporting
7. Documentation generation

The development of building automation systems

- command line
- scripts
- dedicated scripting languages (GNU Make) - the beginning of automation
- automation of activities before and during construction (invoking multiple scripts).
- Minimizing self-scripting
- Construction type - Continuous Integration - incremental build characterized by very frequent references to the compilation process.
- construction distribution (accelerating software development by separating the compilation process into multiple locations)
- processing distribution - every step of the construction process can be done on a different machine.
- Automatic dependency detection

Build script

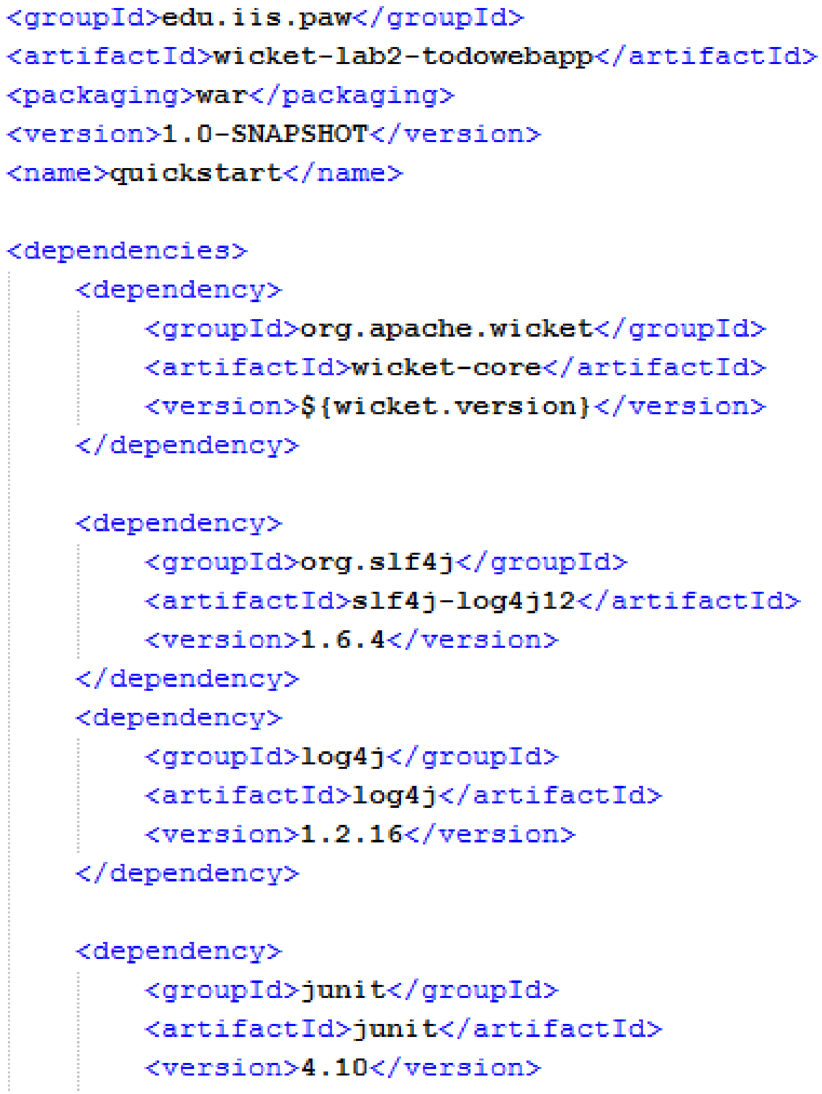


```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Examples of tools for automating the building process

- Apache Ant (java), Nant (.NET) – automation based on script files.
- Apache Maven (java), Gradle (Groovy)
- Scons (Python)
- Phing (PHP)
- Rake (Ruby)
- AnthillPro (CI, deployment automation, test automation)

Dependencies management



```xml
<groupId>edu.iis.paw</groupId>
<artifactId>wicket-lab2-todowebapp</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>quickstart</name>

<dependencies>
    <dependency>
        <groupId>org.apache.wicket</groupId>
        <artifactId>wicket-core</artifactId>
        <version>${wicket.version}</version>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.6.4</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.16</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.10</version>
```

Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.

- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.

- A unique signature identifies each source and object code version and is changed when the source code is edited.

- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.
  - Modification timestamps
    - The signature on the source code file is the time and date when that file was modified.
  - Source code checksums
    - The signature on the source code file is a checksum calculated from data in the file.W

Agile building

- Check out the mainline system from the version management system into the developer's private workspace.

- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.

- Make the changes to the system components.

- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

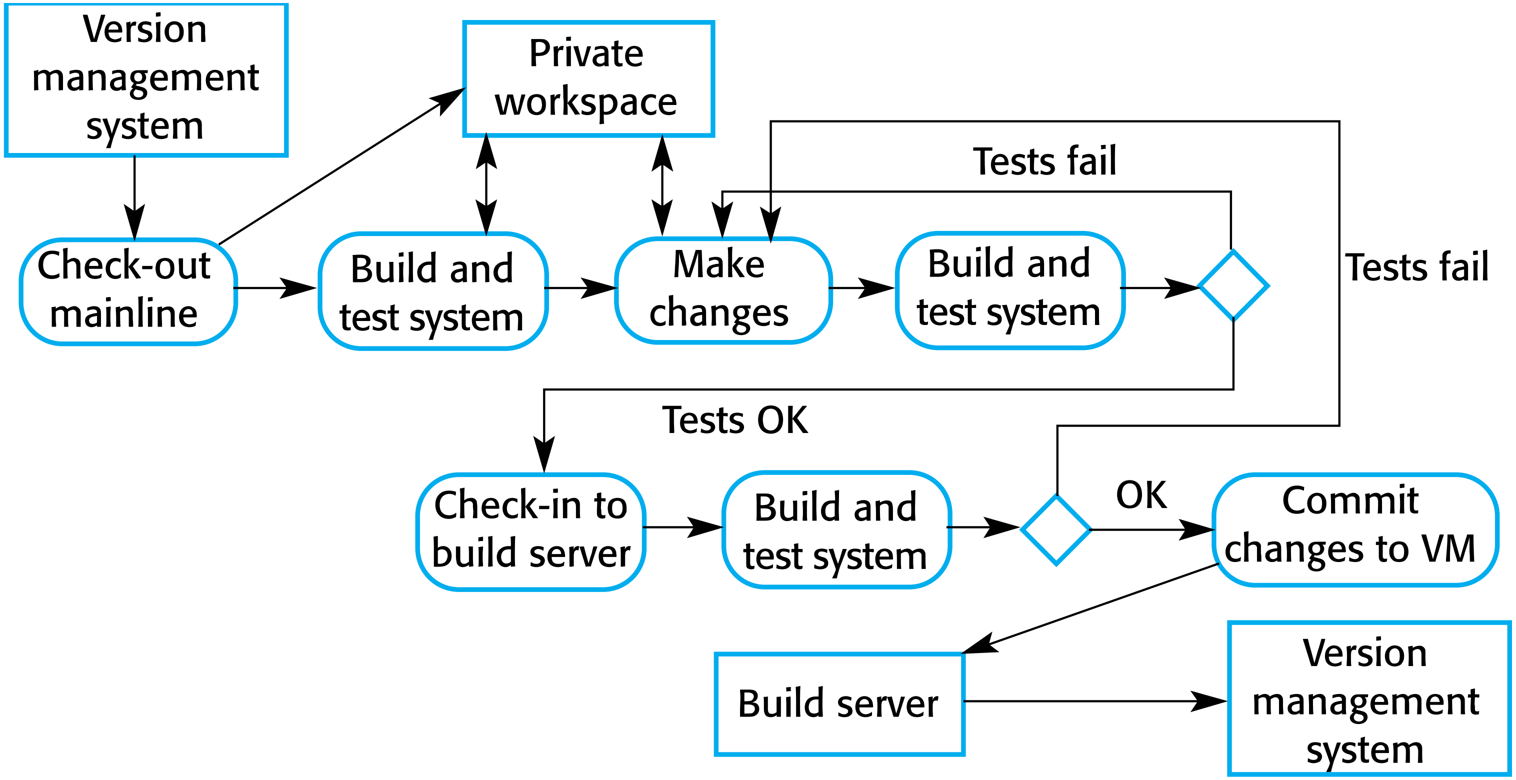
Agile building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.

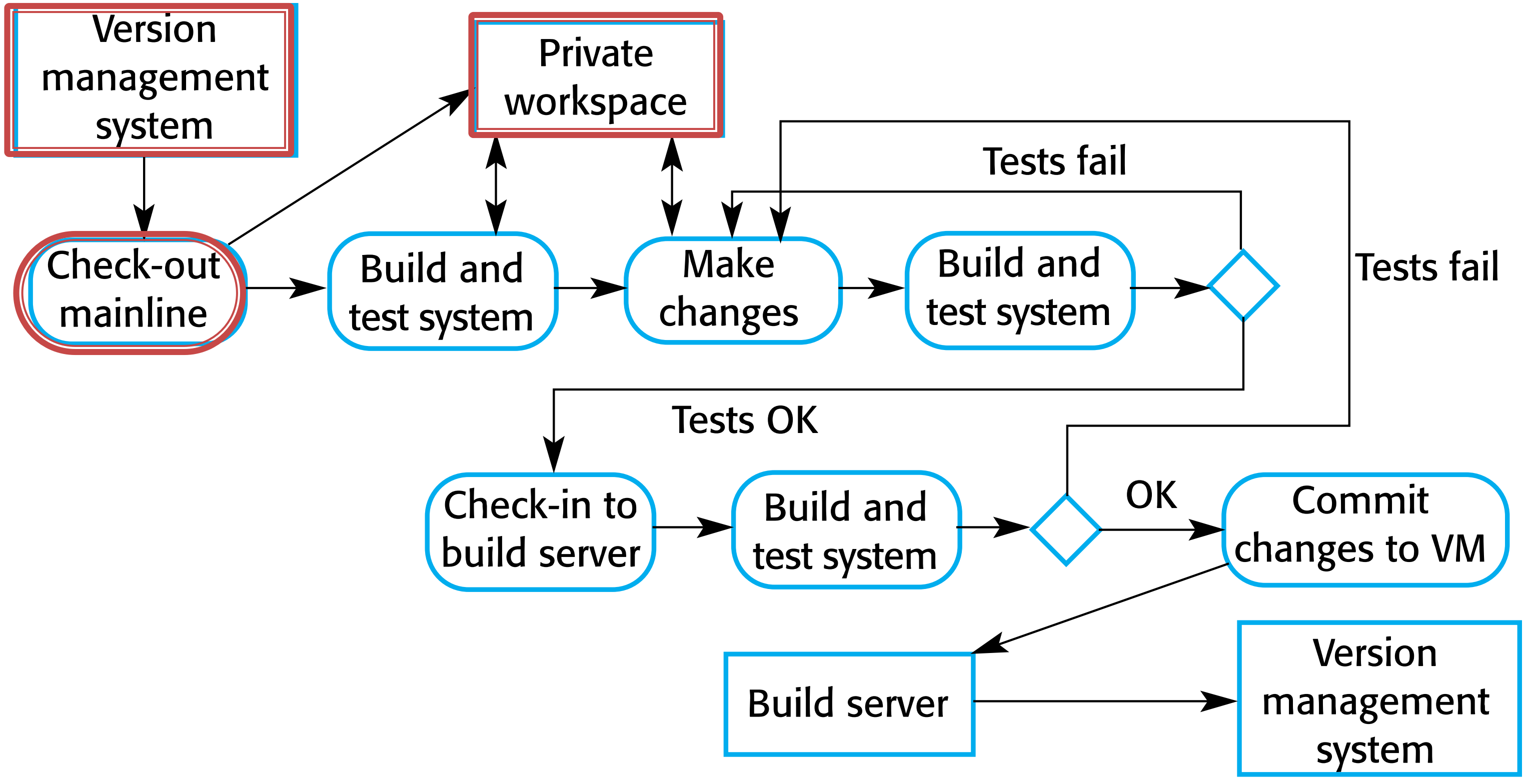
- Build the system on the build server and run the tests.
  - You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.

- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

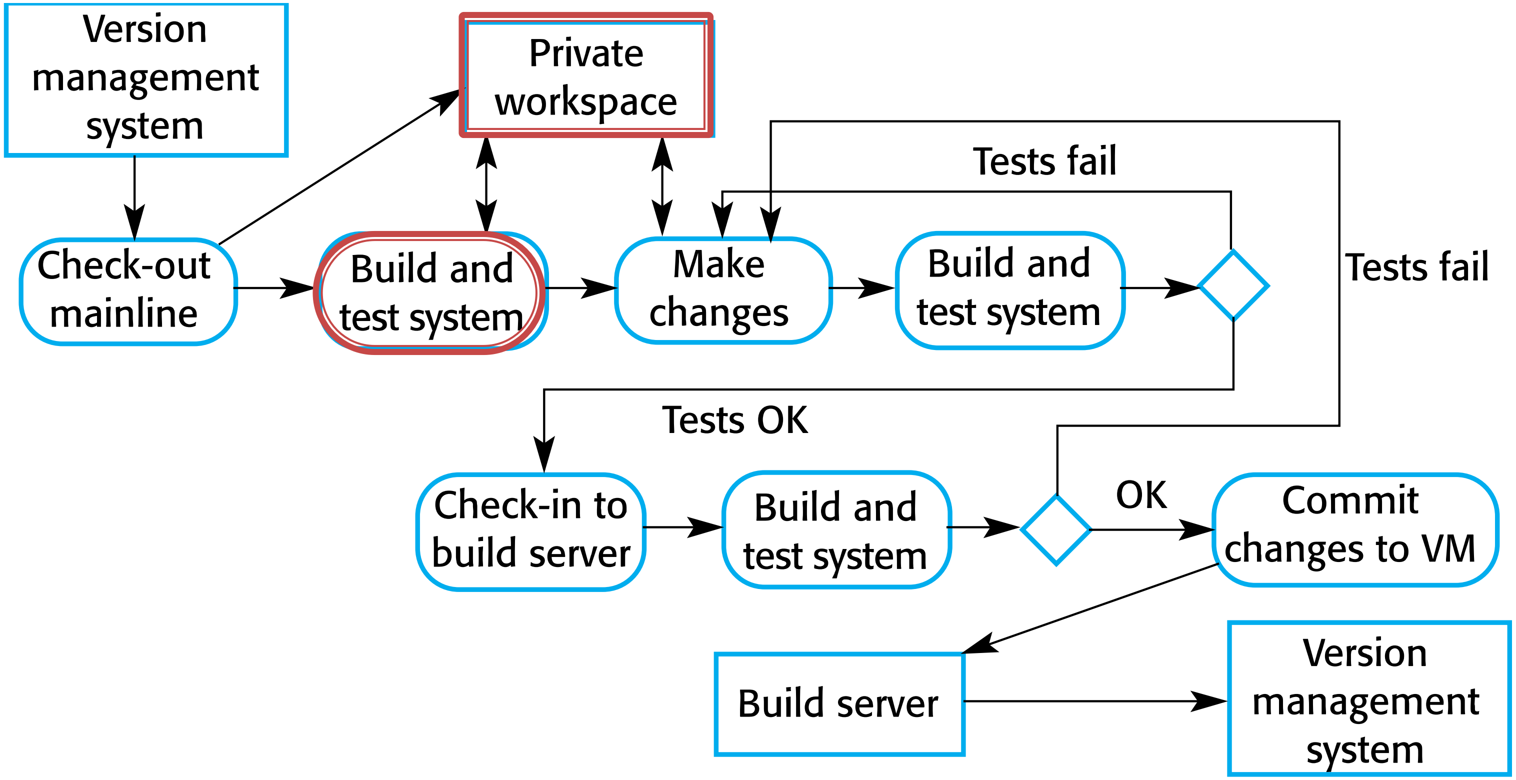
Continuous integration



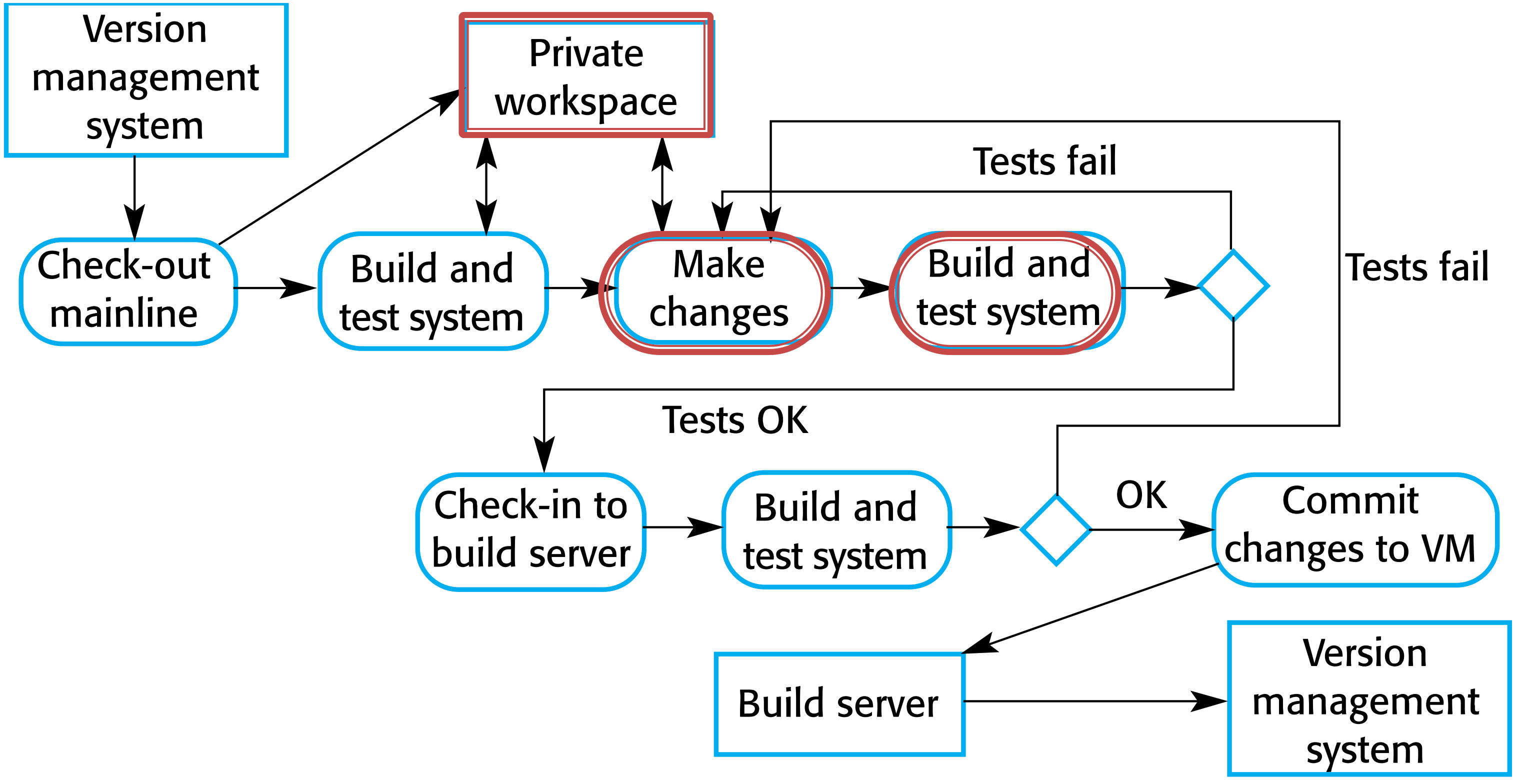
Continuous integration



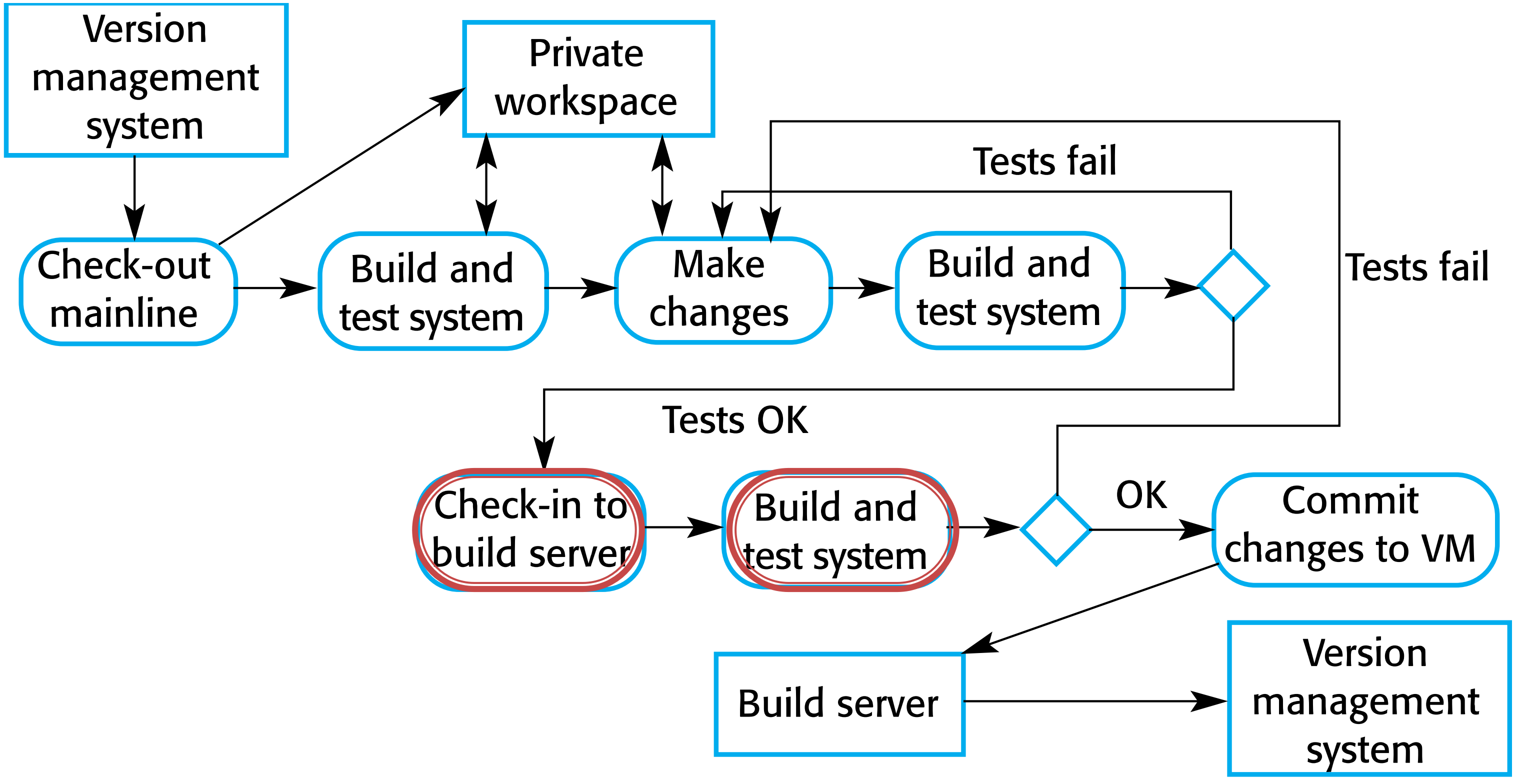
Continuous integration



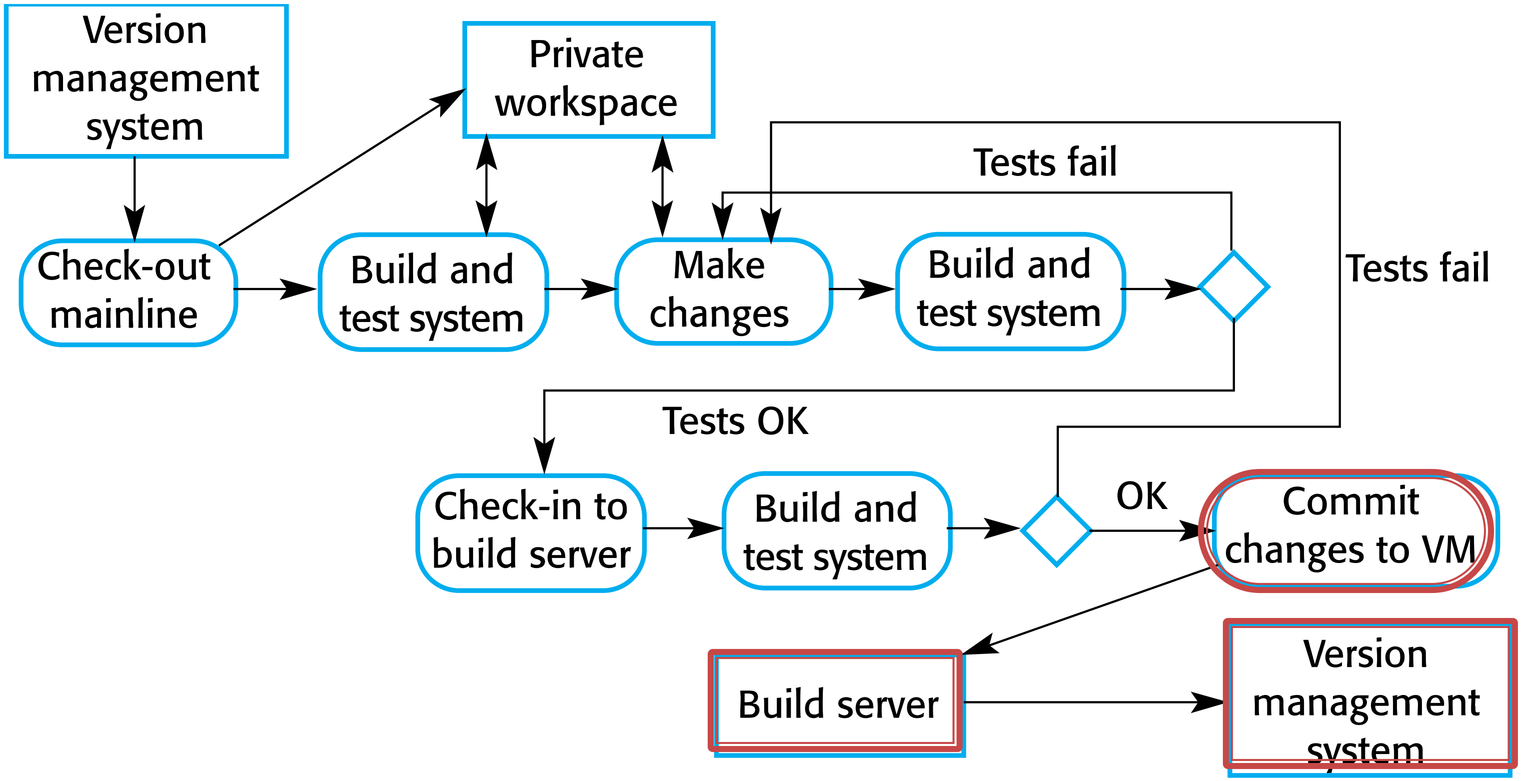
Continuous integration



Continuous integration



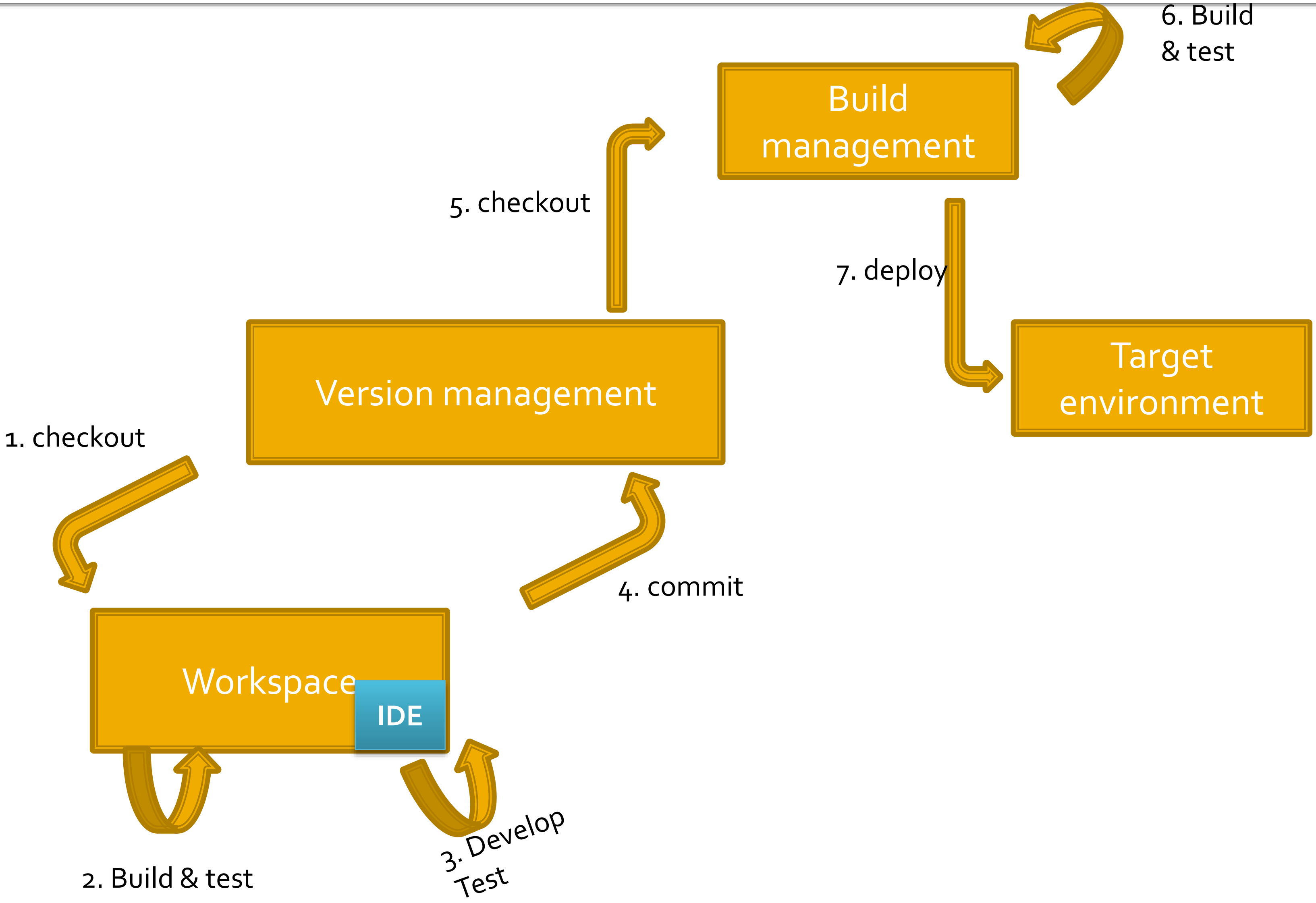
Continuous integration



Daily building

- The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

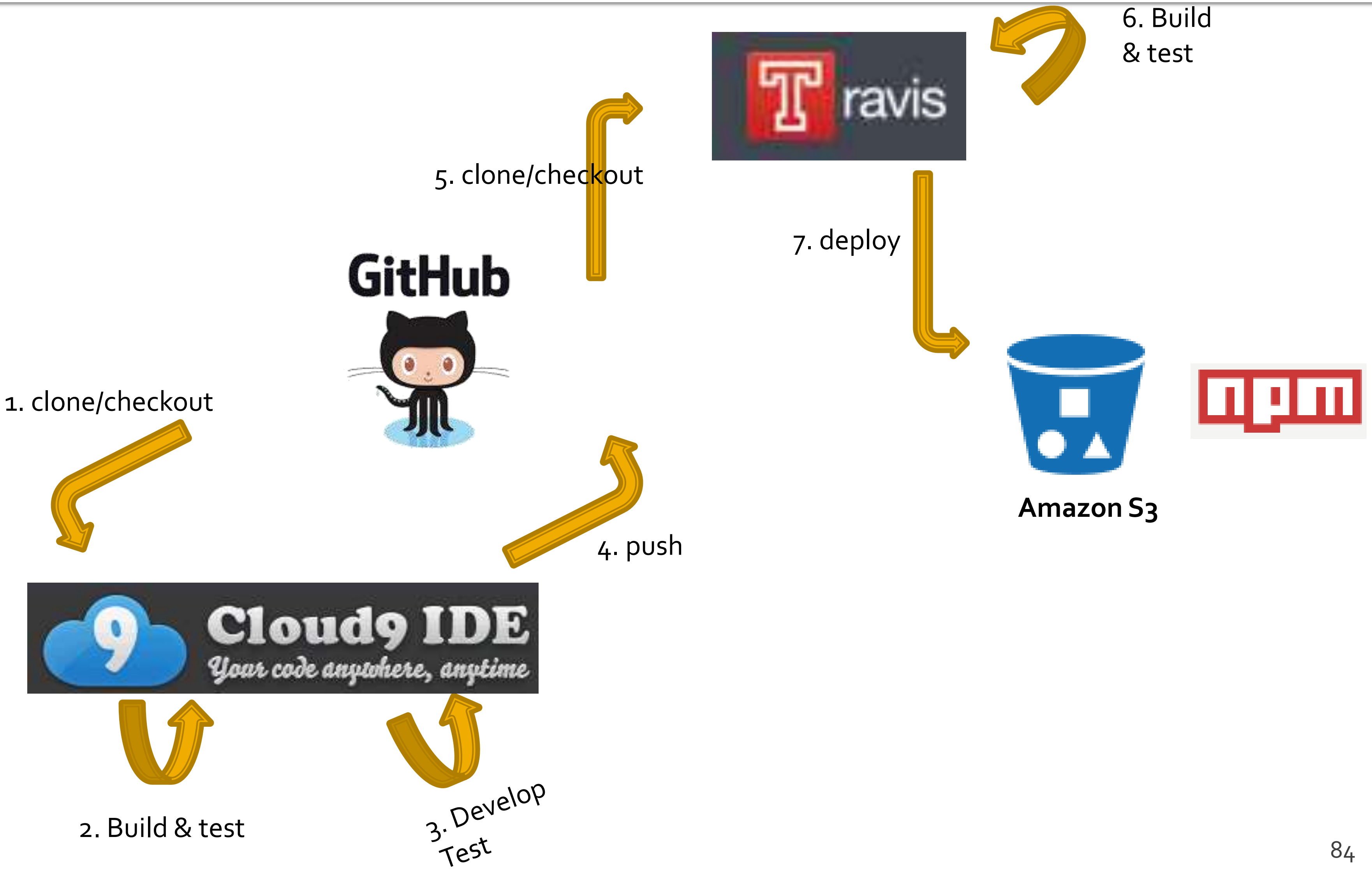
Continuous integration



Tools supporting the continuous integration model

- Extensive integration with other tools
  - version management systems, building automation systems, IDE
- Examples
  - Jenkins
  - AnthillPro
  - Cruise Control
  - Apache Continuum
  - TeamCity
- CI as a service (in Cloud)
  - TravisCI, Cloudbees (Jenkins), codebetter (TeamCity)

CI in the Cloud Computing Environment



1. clone/checkout

2. Build & test

3. Develop Test

4. push

5. clone/checkout

6. Build & test

7. deploy

Release management

Configuration management

Release tracking

- In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

Release reproduction

- Release documentation record:
  - specific versions of the source code components that were used to create the executable code.
- copies of the source code files, corresponding executables and all data and configuration files.
- versions of the operating system, libraries, compilers and other tools used to build the software.

Release planning

- As well as the technical work involved in creating a release distribution,
  - advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

Release components

- As well as the the executable code of the system, a release may also include:
  - configuration files defining how the release should be configured for particular installations;
  - data files, such as files of error messages, that are needed for successful system operation;
  - an installation program that is used to help install the system on target hardware;
  - electronic and paper documentation describing the system;
  - packaging and associated publicity that have been designed for that release.

Factors influencing system release planning

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law	This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance.

Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Customer change proposals	For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.

Software release lifecycle

- the sum of the phases of development and maturity of software.
  - Includes development, testing:
    - Pre-alpha
    - Alpha (IBM terminology – M. Belsky)
    - Beta (perpetual beta problem)
    - Release candidate
  - and support stages:
    - Release to manufacturing (RTM) / „going gold"
    - General availability (GA)
    - Service release
    - Service pack

Software release versioning

- the process of assigning either unique **version names** or unique **version numbers** to unique states of software.
  - meaningful numbers over convention
- Various versioning schemes
  - e.g. Sequence-based identifiers
    - Each software release is assigned an identifier consisting of several sequences of alphanumeric characters or numbers.
    - 1.0, 2.3.34.1546, 9.3b, 3.4.1
    - There is no standard "meaning" of such sequence

Versioning and changes

- In some versioning schemes, the meaning of the change is reflected by the sequence level, e.g.:
  - major.minor[.build[.revision]]
- The versioning scheme may also carry information about the scope of actual tests performed:
  - 1.1b1, 1.1rc2, 1.1
  - 0.1, 0.3

Other versioning schemas

- Based on:
  - date (np. Ubuntu – 8.04, 9.10)
  - year of release (np. Office 2003)
  - Alphanumeric codes (np. Adobe Photoshop CS2)
  - excentric (e.g.. tex)
  - undecided (e.g. MS Windows)
  - hybrid (i.e. marketing) (e.g. Java 1.5 = Java 5)

Semantic Versioning
http://semver.org/

- X.Y.Z
  - X – major, Y – minor, Z – patch
- Any change in the version published in a given version requires revision.
- X = zero - project initiation - everything can be changed
- X++ - incompatible changes implemented
- Y++ - backward compatible changes
- Z++ - a correction has been introduced (without changes in functionality)
  - 1.0.0-alpha, 1.0.0-beta+exp.sha.5114f85

Did I catch your attention?

- What is delta version?
- Codeline vs. baseline
- Checking in vs. checking out
- Alpha vs. Beta versions