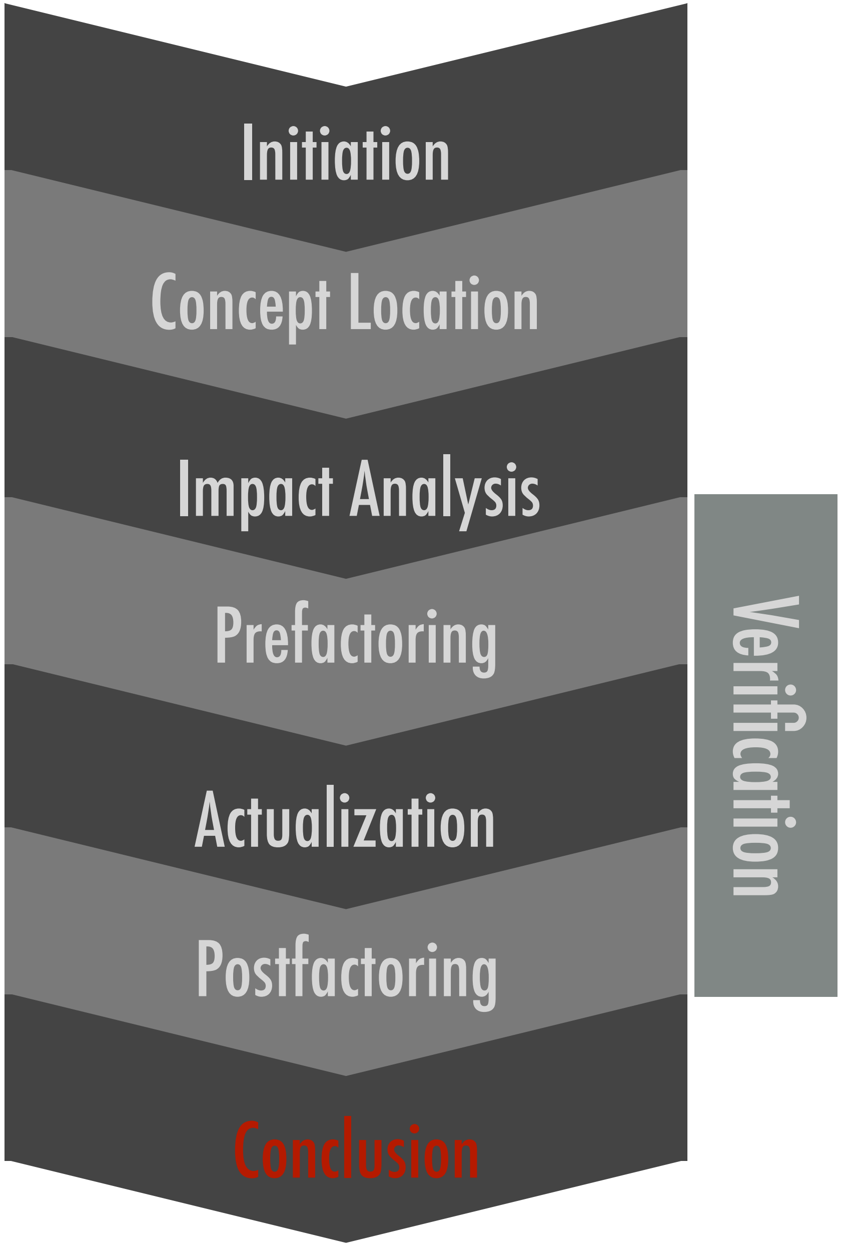
Conclusion of Software Change

Prof. Romain Robbes

Conclusion updates the project



The programmer commits the changes, creates a new baseline, updates the documentation.

Outline

Principles of SCM and version control

Making and testing builds

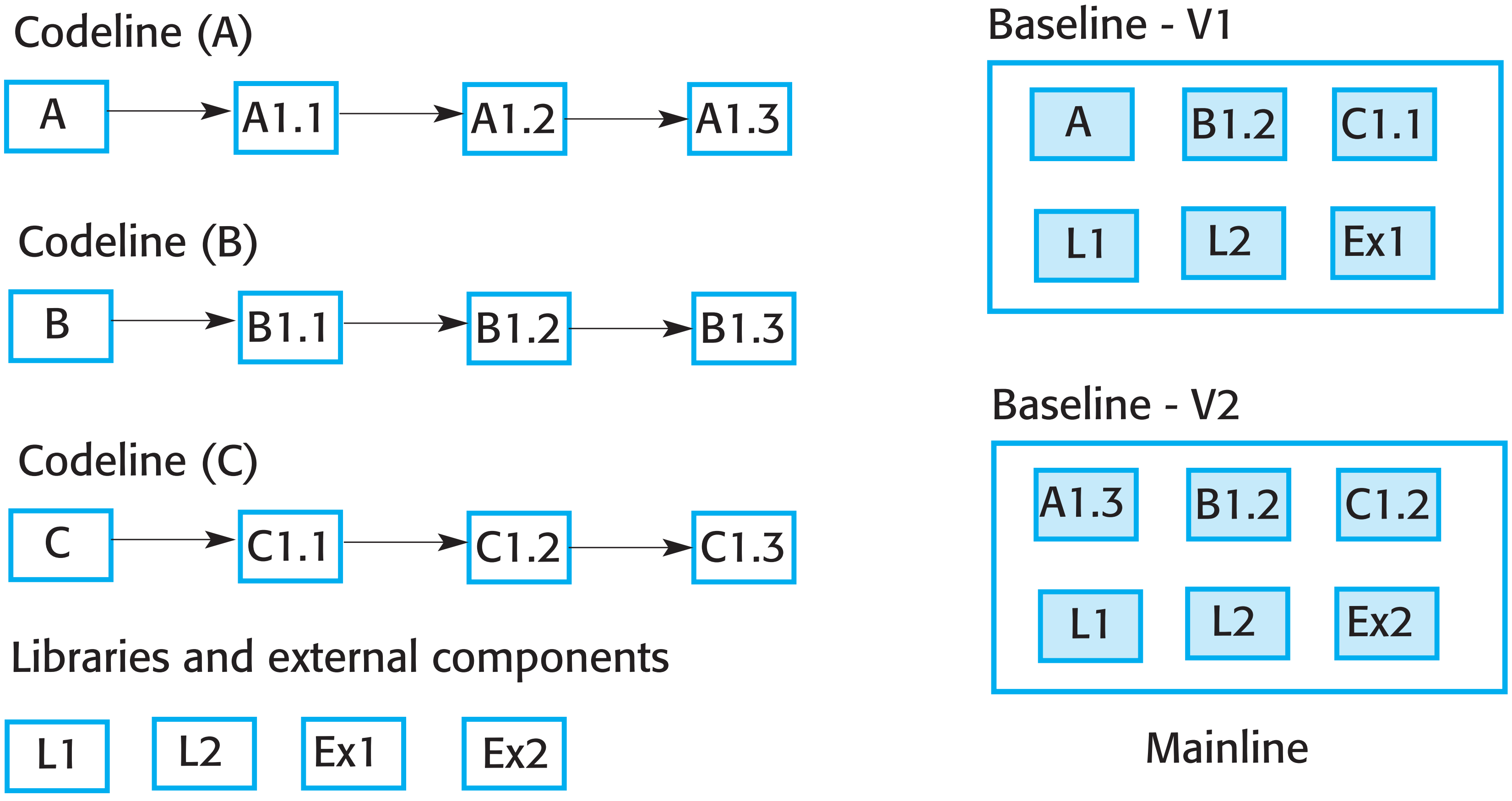
The conclusion phase

Principles of SCM

SCM terminology

Term	**Explanation**
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed properly.
Software configuration item	Any item (design, code, test data, document, etc.) under **configuration control**.
Version	An instance of a configuration **item** that differs from other instances of that item. Versions have a unique identifier (e.g. name + version number).
Baseline	A collection of component **versions** that make up a system. The versions of the components of the baseline cannot be changed (for reproducibility).
Codeline	A set of **versions** of a software component.

Version control keeps track of the changes to system components



The version control system manages codelines and baselines

Git and Subversion supports tagging to define baselines

```
git tag -a v1.2 “commit id”
```

see: http://git-scm.com/book/en/v2/Git-Basics-Tagging

```
svn copy http://svn.example.com/repos/calc/trunk
         http://svn.example.com/repos/calc/tags/release-1.0
         -m "Tagging the 1.0 release of the 'calc' project.”
```

See SVN documentation for details

SCM terminology

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.
Workspace	A private work area where software can be modified without affecting other developers.
Branching	The creation of a new **codeline** from a **version** in an existing
Merging	The creation of a new **version** of a software component by merging separate **versions** in different **codelines**.
System building	The creation of an **executable system version** by compiling and linking the appropriate **versions** of its components and libraries.

Other version management features

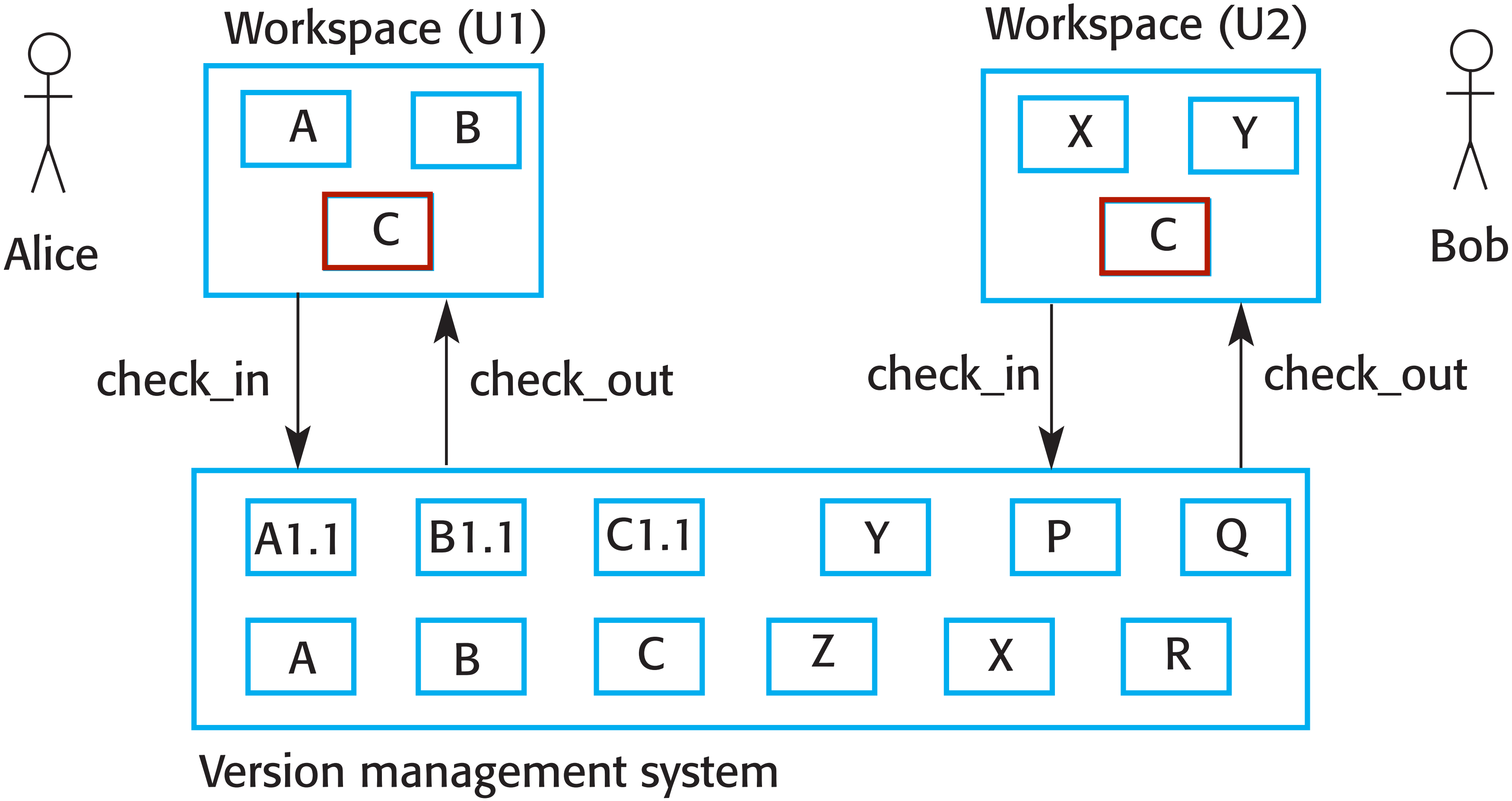
Assigning identifiers to versions and releases

Storage with deltas (saving space)

Change history recording (no changes are lost, “blaming”)

Handling concurrent development
(locking, conflict detection, branching, merging)

Concurrent changes



There are two strategies to deal with concurrent changes

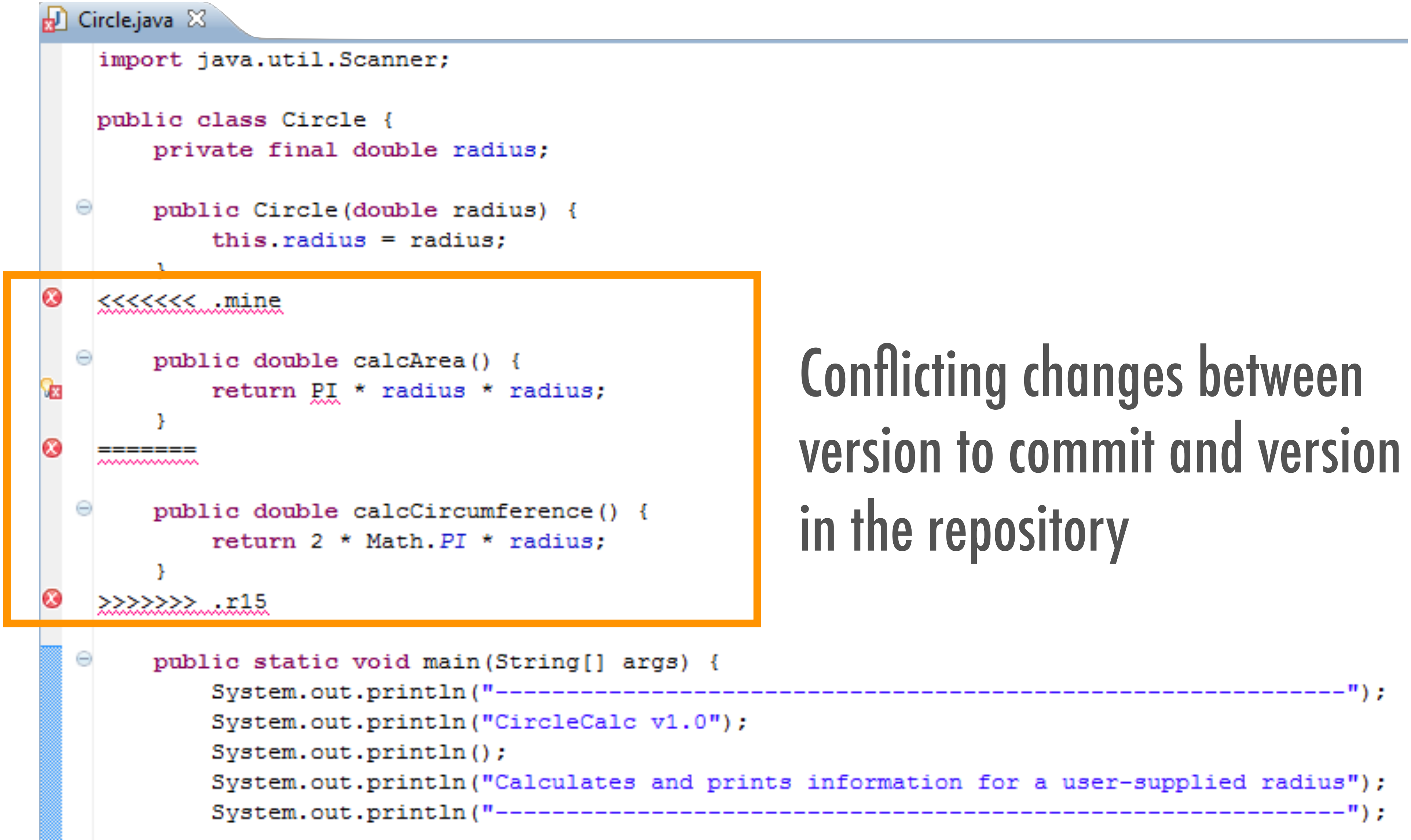
pessimistic

forbid concurrent changes:
files are locked when checked out
common in “old” SCM systems

optimistic

authorize concurrent changes:
potential conflicts will be resolved
Git and Subversion use this

Conflicts can be resolved automatically, or manually



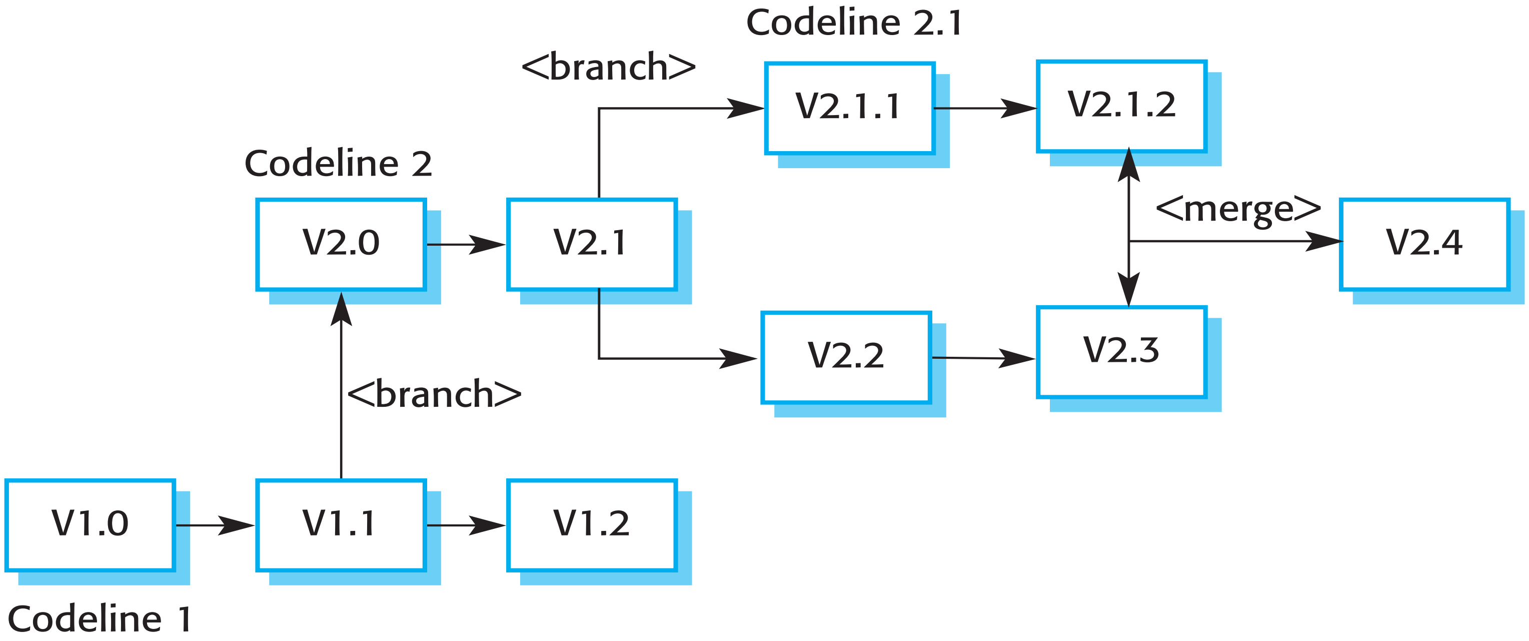
```
import java.util.Scanner;

public class Circle {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
<<<<<<< .mine

    public double calcArea() {
        return PI * radius * radius;
    }
=======

    public double calcCircumference() {
        return 2 * Math.PI * radius;
    }
>>>>>>> .r15

    public static void main(String[] args) {
        System.out.println("------------------------------------------------------------");
        System.out.println("CircleCalc v1.0");
        System.out.println();
        System.out.println("Calculates and prints information for a user-supplied radius");
        System.out.println("------------------------------------------------------------");
```

Conflicting changes between version to commit and version in the repository

Branching and merging

Often it makes sense to work independently for a while.
Howver, conflicts are not visible until the branches are merged.

Different versions of the system need to be maintained. Changes from one branch may be brought back in another (merging). This may be time consuming!



Impact of branching and merging

Microsoft has studied the impact of branches in their software development process:
Assessing the Value of Branches with What-if Analysis
http://research.microsoft.com/apps/pubs/default.aspx?id=172572

Too many levels of branches may slow you down; but the simple branching you use in your projects should be fine.

More on Git and Github

Git command-line basics:
https://try.github.io/levels/1/challenges/1

Git bootcamp
https://help.github.com/categories/bootcamp/

Merging and code reviews with pull requests:
https://help.github.com/articles/using-pull-requests/

Continuous integration allows you to react rapidly to conflicts

The more time passes, the harder to fix the conflicts:

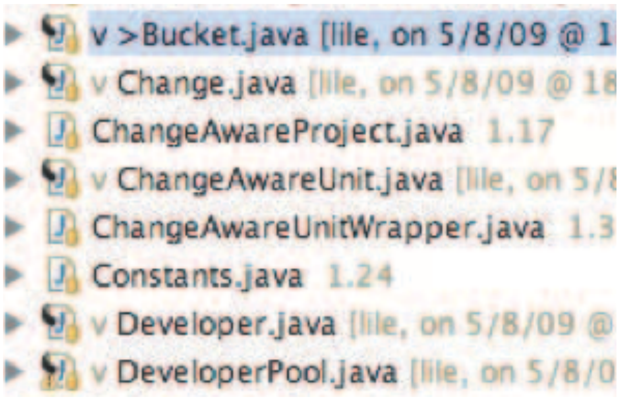
- Merge conflicts
- Compile conflicts
- Test conflicts

See also: https://travis-ci.org

Best way to resolve conflicts?
Prevent them!

Schedule changes so that you work on different things.

There is research in change awareness tools
(Palantír, Syde)



```
v >Bucket.java [lile, on 5/8/09 @ 1
v Change.java [lile, on 5/8/09 @ 18
ChangeAwareProject.java  1.17
v ChangeAwareUnit.java [lile, on 5/8
ChangeAwareUnitWrapper.java  1.3
Constants.java  1.24
v Developer.java [lile, on 5/8/09 @
v DeveloperPool.java [lile, on 5/8/0
```

Making and testing builds

What’s wrong with `javac *.java`?

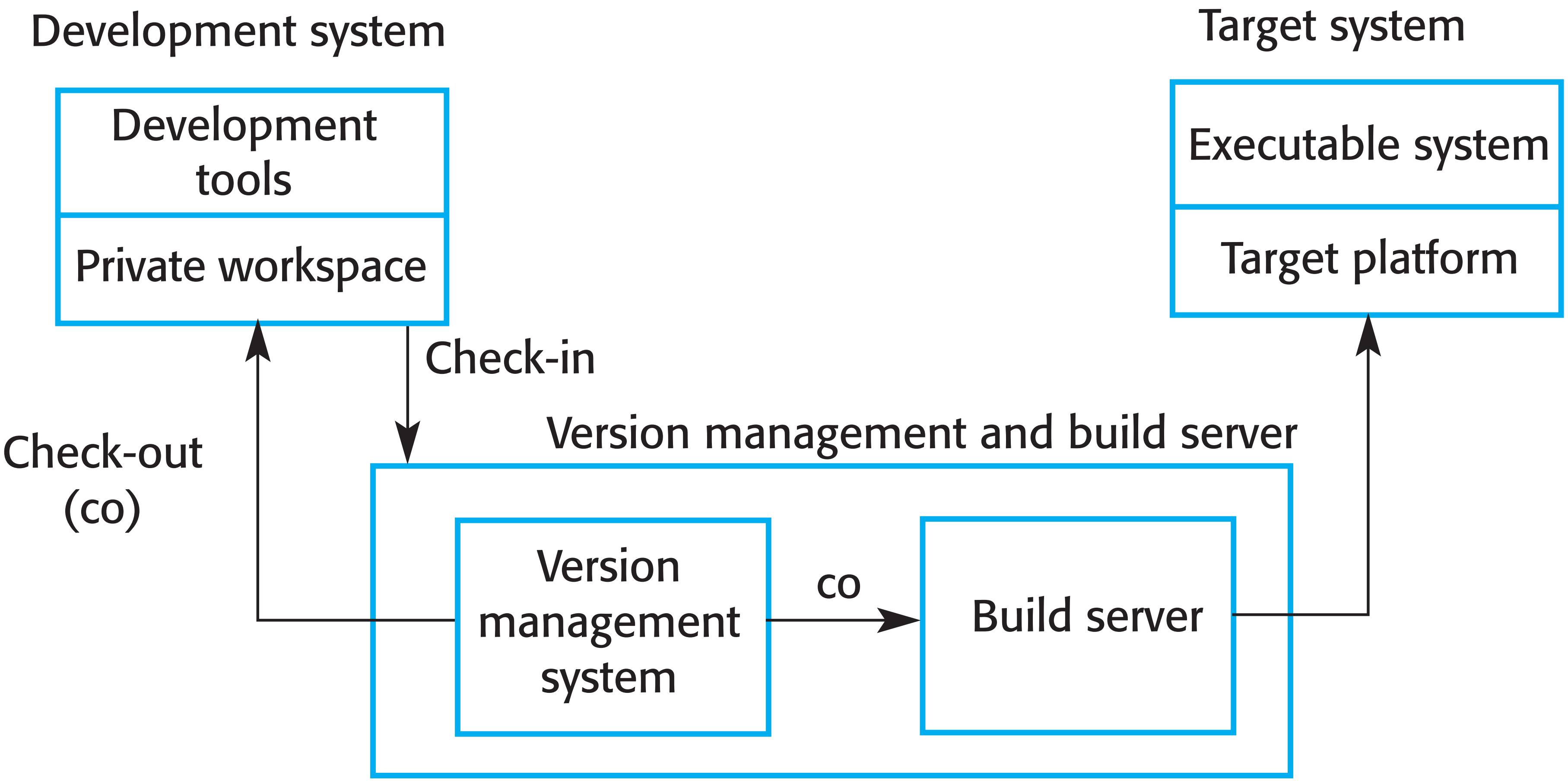
There is more to system building than that!

We must first choose a version of the system to build

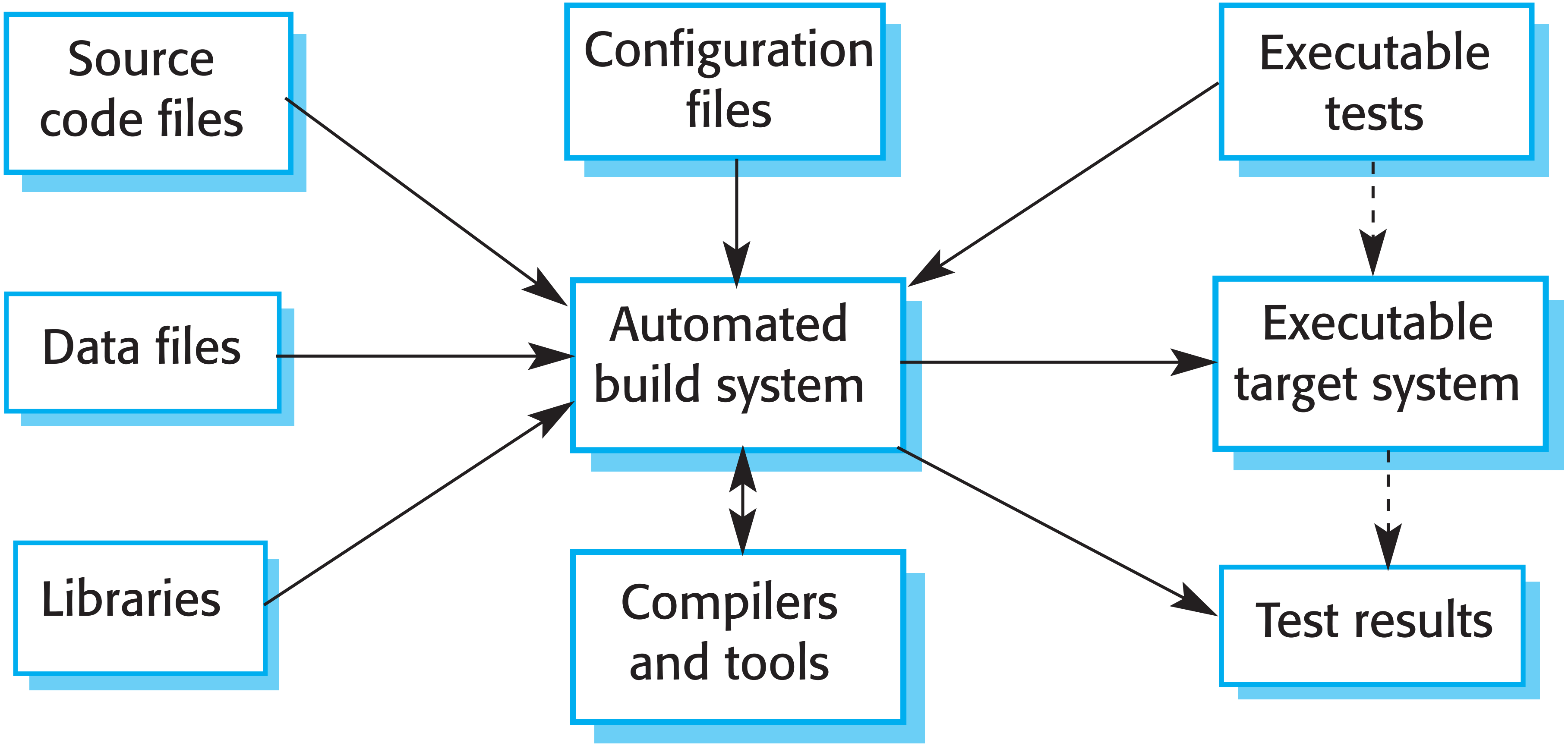
Compiling everything can be very long (hours!)

There are other actions to perform, such as running tests, etc

Development, build, and target platforms



Building the system



Features of build systems

Integration with version management

Integration with continuous integration system

Minimizing recompilation

Building the system (duh!)

Running the tests

Creating reports

Generating documentation

Example: Apache Ant

```
<project>

    <target name="clean">
        <delete dir="build"/>
    </target>

    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>

    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="oata.HelloWorld"/>
            </manifest>
        </jar>
    </target>

    <target name="run">
        <java jar="build/jar/HelloWorld.jar" fork="true"/>
    </target>

</project>
```

Refactoring, and specifying dependencies

```
<project name="HelloWorld" basedir="." default="main">

    <property name="src.dir"     value="src"/>

    <property name="build.dir"   value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property name="jar.dir"     value="${build.dir}/jar"/>

    <property name="main-class"  value="oata.HelloWorld"/>


    • • •


    <target name="run" depends="jar">
        <java jar="${jar.dir}/${ant.project.name}.jar" fork="true"/>
    </target>

    <target name="clean-build" depends="clean,jar"/>

    <target name="main" depends="clean,run"/>

</project>
```

Integrating automated testing

```
<target name="junit" depends="jar">
    <junit printsummary="yes">
        <classpath>
            <path refid="classpath"/>
            <path refid="application"/>
        </classpath>

        <batchtest fork="yes">
            <fileset dir="${src.dir}" includes="*Test.java"/>
        </batchtest>
    </junit>
</target>
```

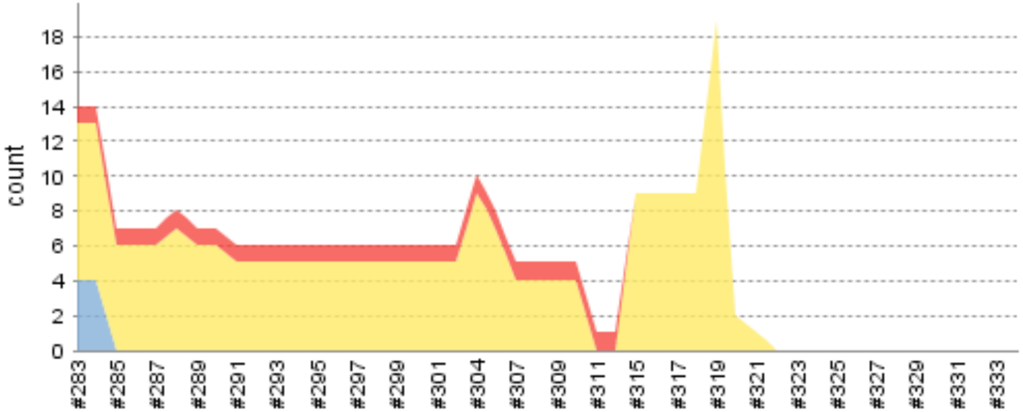
```
<target name="junitreport">
      <junitreport todir="${report.dir}">
          <fileset dir="${report.dir}" includes="TEST-*.xml"/>
          <report todir="${report.dir}"/>
      </junitreport>
  </target>
```

Build system and continous integration

Jenkins can run <APACHE ANT> after each commit.

Also other plugins (Junit, checkstyle, coverage, findbugs, fitnesse, http://www.youtube.com/watch?v=1EGk2rvZe8A)...

It also increases the visibility of the projects (historical trends, etc)



However you should still verify the build before committing!

Other build tools

`Make`: the ancestor, still widely used (Makefile)

Apache Maven Project
http://maven.apache.org/ more advanced

manages dependencies to libraries, etc ...

The conclusion phase

Activities in the conclusion phase

Committing changes to the baseline
Updating product backlog
Building the product
Preparing the product for future changes

Committing changes & update the backlog

The version control system allows to document the changes; do it!

Resolve conflicts due to parallel changes.

Update the status of the change request.

Large product have a deadline for the builds

If you miss it, you have to wait for the next one!

You may have more conflicts with other people, so ... be on time!

Frequent builds are recommended (daily, or multiple daily); the more changes, the harder it is to integrate them

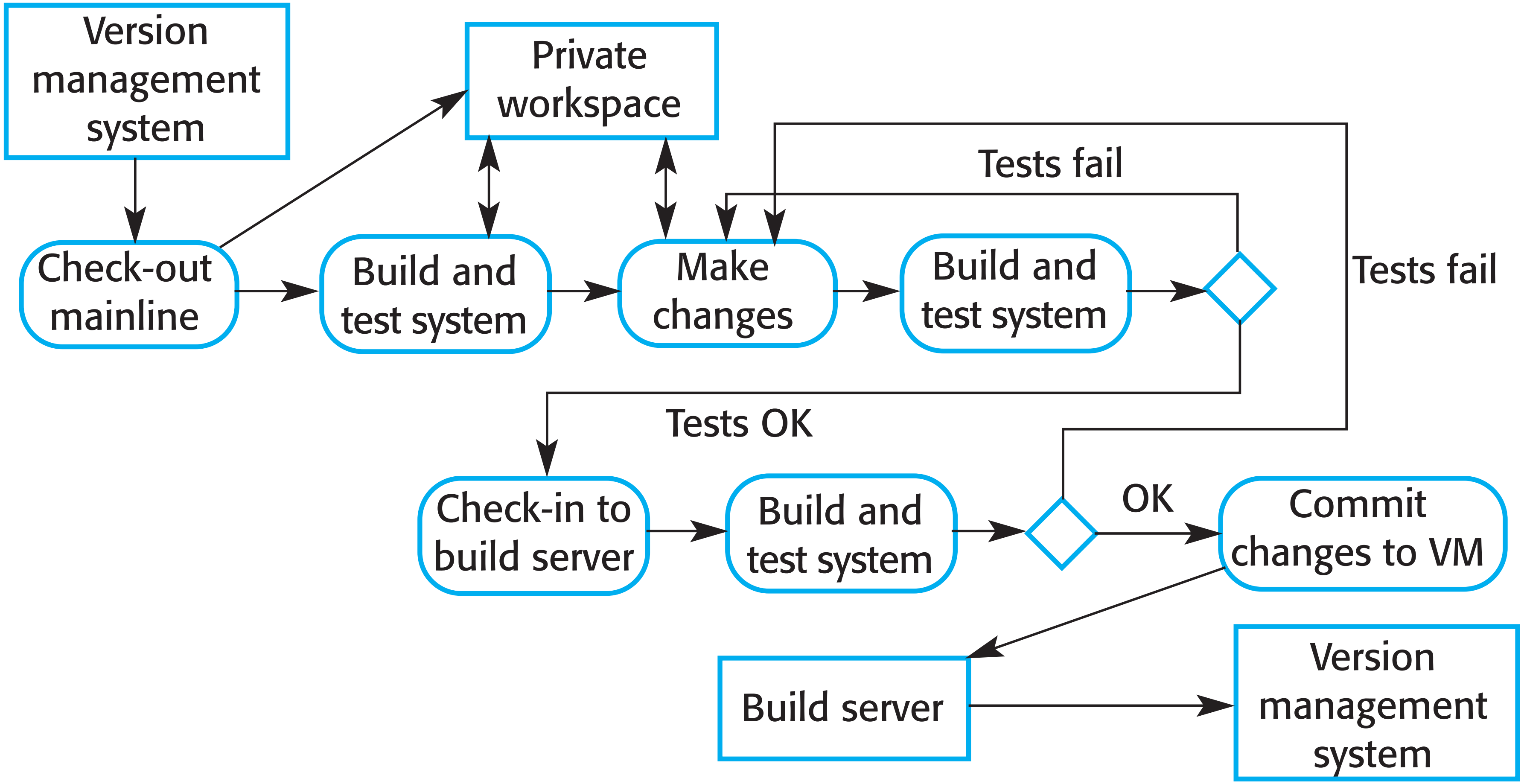
The build is extensively tested

If the tests pass, a new baseline is created.

New defects can be pointed out, and fixed.

In some case, the build can be “broken”;
no baseline is created (which is expensive!).

An example testing and building process



(svn: tag as part of baseline)

(missing review of commits by other devs)

Building Windows NT

Size: 5.6 MLOC
Build time: 19 hours
Daily builds were used

Used a first test step (“Smoke test”), before doing the full tests, to speed up broken build discovery.

Programmers had beepers and would be waken up if they broke the build!

Programmers also prepare for future changes

- Updating the change requests
- Extensive postfactorings of the entire code base
- Reevaluating the priority of change requests
- Updating the documentation

The “truck factor”

What happens if a programmer leaves?

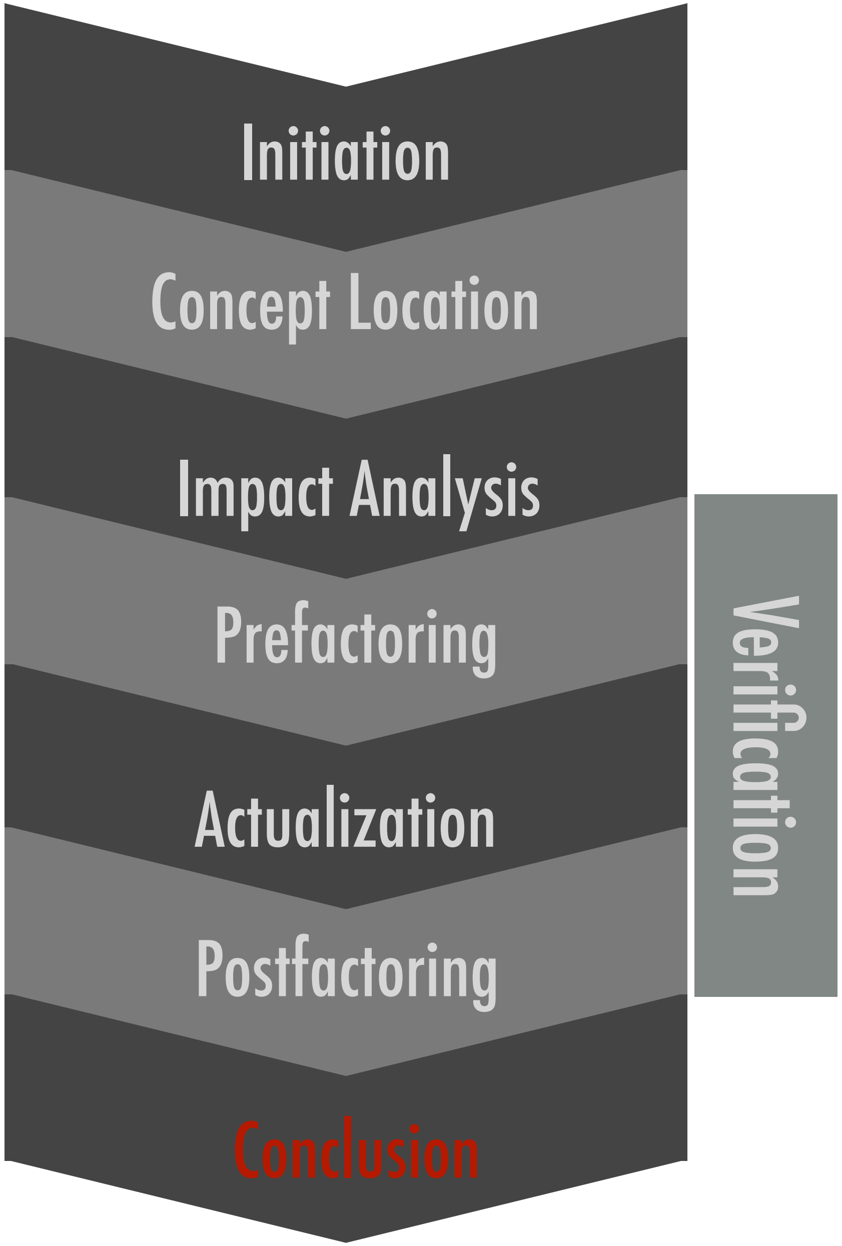
All his knowledge of the code leaves with him/her!

Every part of the code should be known by several people
- pair programming can help for that
- reviews and inspections as well
- cf review at checking at google, linux

The documentation should be updated frequently!

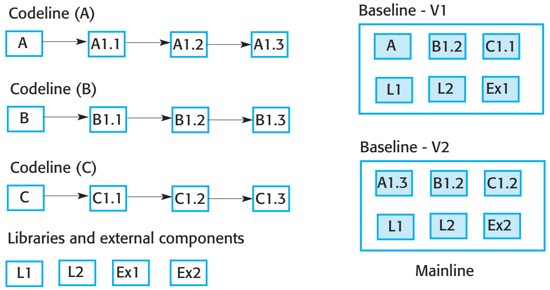
Conclusions

Conclusion updates the project



The programmer commits the changes, creates a new baseline, updates the documentation.

Programmers use a variety of tools to manage changes, versions, and builds



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

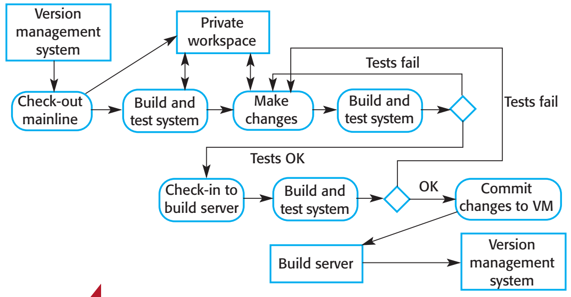
Libraries and external components
L1 L2 Ex1 Ex2

Baseline - V1
A B1.2 C1.1
L1 L2 Ex1

Baseline - V2
A1.3 B1.2 C1.2
L1 L2 Ex2

Mainline

Change request tracking system
Version control system (SVN, Git)
Build system (make, ant, maven)
Continuous integration (jenkins, hudson)



Version management system

Private workspace

Check-out mainline

Build and test system

Make changes

Build and test system

Tests fail

Tests fail

Tests OK

Check-in to build server

Build and test system

OK

Commit changes to VM

Build server

Version management system

Builds and releases should be performed often
(according to the principle of ... incrementality!)