# Chapter 0.1 - The Tool Chain

**Walter F. Tichy**

# Change chaos:
# Who has not experienced it?

- Identification and tracking:
  - "This program worked yesterday!"
  - "I already corrected that mistake last week!"
  - "Where are my changes from last week?"
  - "That's an obvious improvement. Has it been tried yet?"
  - "Who is responsible for this change?"

# More change chaos

- Version selection
  - "Has everything been re-compiled? Re-tested?"
  - "How do I exclude this erroneous change again?"
  - "I can't reconstruct the customer‘s failure in this configuration!"
  - “Which changes are in this release?"
  - "Which customer bug reports have been resolved?"

# More chaos...

- Software delivery
  - "What configuration does this customer have?"
  - "Did we deliver a consistent configuration?"
  - "Did the customer modify anything?"
  - "The customer has not installed the last two releases. What happens when we send him the new one?"

- These problems sound familiar to anyone who has worked in software development.

# Origin of the configuration management

- U.S. space industry in 1950s and 1960s
  - During the development of rockets and satellites, numerous changes were made that were not documented.
  - Prototypes were destroyed during tests or shot into space, or sunk in the ocean, the original plans were obsolete, therefore reconstruction impossible.

→ Configuration management was to document all changes to spacecraft so that any development status could be restored.

# Configuration Management and Version Control

# Definition

***Software configuration management*** is the discipline of tracking and controlling the evolution of software.

Without effective configuration management, change chaos ensues.

# Configuration Management (CM)

- ISO 9001:
  "Configuration management provides a mechanism for identifying, directing, and tracking the versions of each software element."

# (Software) Configuration

- A (software) configuration is a uniquely named set of software elements, with their valid version identification, which are coordinated with each other in their operation and their interfaces at a specific point in time in the product life cycle, and which fulfill an intended task together.

# Software Element (SE)

- A software element is any identifiable component of a product or product line. A software element can be a single file, or a configuration.
- (a product line is a set of related products that have software elements in common).

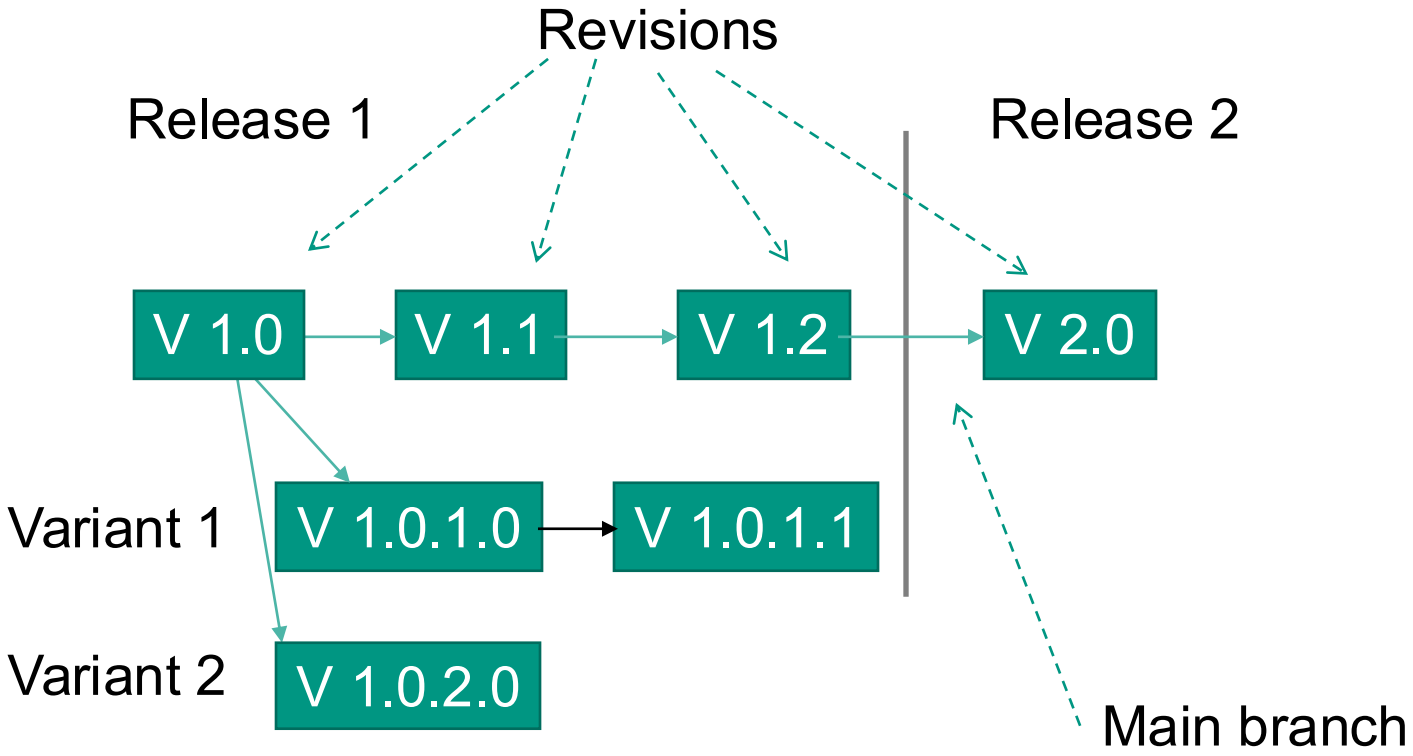
# Software Element

- Has system-wide unique identifier
- Change of an element creates a new version with a new, unique identifier to avoid misidentification
- Distinguish
  - Source element: created manually, e.g., with editor or drawing tool
    - needs to be stored in the repository indefinitely
  - Derived element: automatically generated, e.g., by translator, compiler
    - can be deleted, because it can be regenerated automatically

# Versions

- A **version** is the expression of a software element at a certain point in time.
- **Revisions** are successive versions (development states).
- **Variants** are alternative versions (customizations, or with alternative data structures/algorithms, for different platform, different GUI, etc.).

# Revisions and Variants of a Software Element



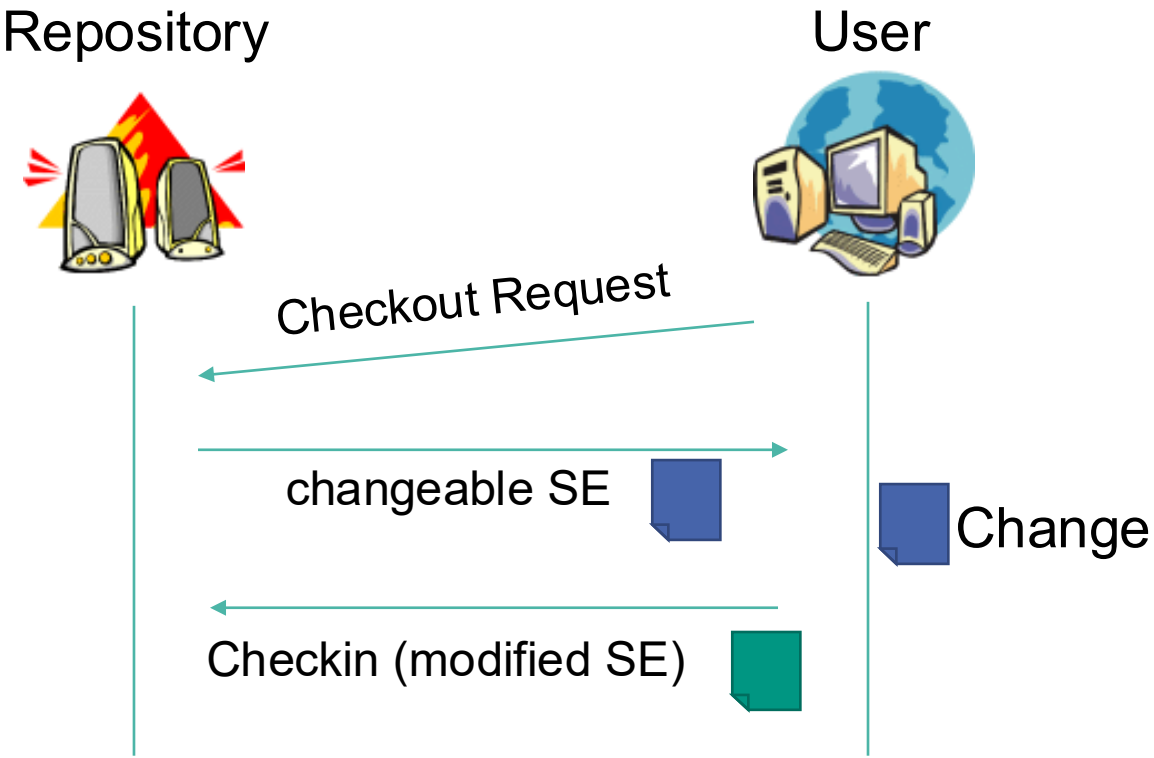
# Checkout/Modify/Checkin Cycle

- Software elements are collected in *repositories*
- **Checkout:**
  - Fetches copy from repository
  - Reserves copy for the person checking out, which means that only this person may file the next revision (**strict checkout**)
  - Copy may be changed, tested and checked in again

# Checkout/Modify/Checkin Cycle

- **Checkin** (also called **Commit**):
  - checks reservation
  - writes copy back to repository
  - deletes reservation
  - records meta data:
    - Author of the element/change
    - Checkin time
    - Logbook entry summarizing change
  - checked-in element can no longer be modified
  - only checking out again allows modifications (and produces a new version)

# Checkout/Modfiy/Checkin Cycle



# Checkout/Modify/Checkin Cycle

- Users are allowed to check out as many elements as they want.
- Read access for other team members is not prevented by change reservation (only modification). So, others can compile the entire system, for instance, even if there are change reservations.
- Distinguish
  - **Strict checkout**: with change reservation
  - **Optimistic** or **multiple checkout**:
    Without change reservation; may require merge operation later

# Strict Checkout

- Only one checkout with change reservation per version of an SE is allowed.
- Person with reservation has exclusive right to change (i.e., to deposit next revision)
- Advantage:
  - no merging effort when checking in, because only one developer at a time (the one who currently owns the reservation) is allowed to modify the version.
- Disadvantage:
  - only one person at a time can change the checked-out version.
  - This can lead to delays (in case of illness, absence, workload of person holding the reservation).

# Optimistic Checkout

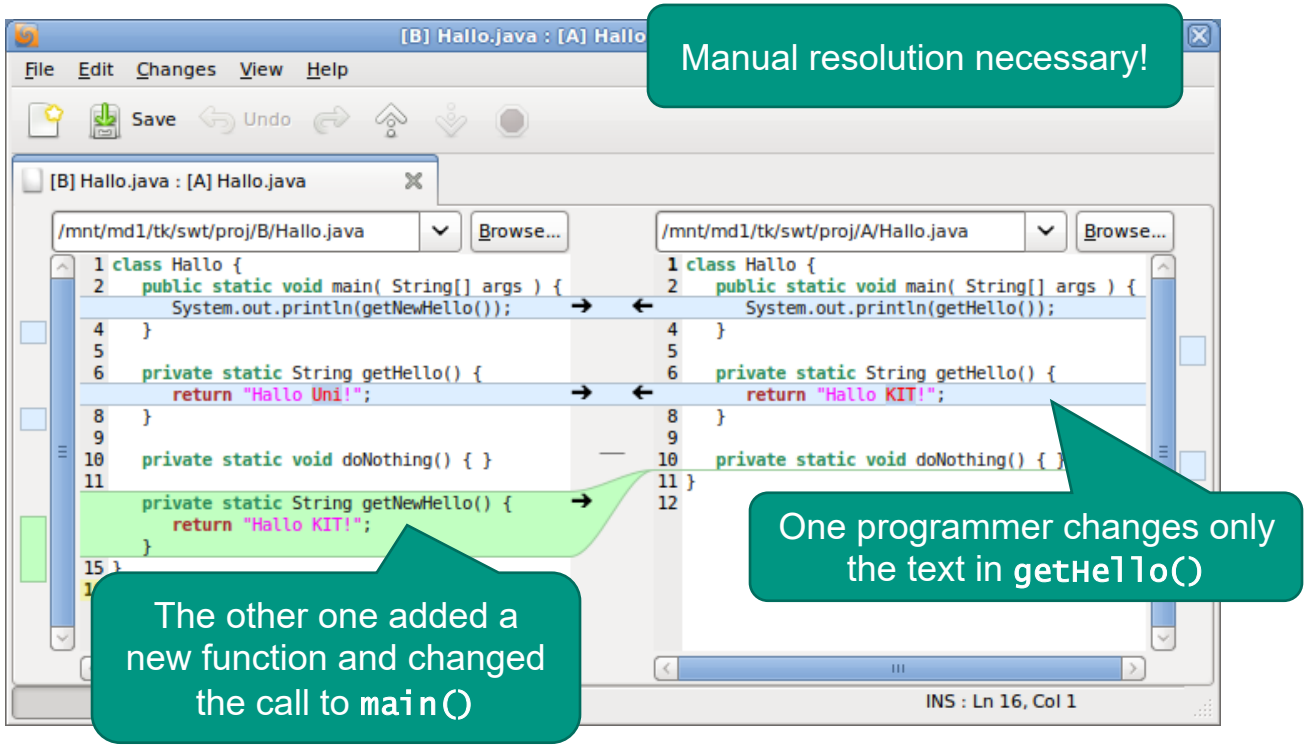
- No change reservation necessary. One can checkout and modify at any time.
- Advantage:
  - Multiple developers can work on the same element at the same time.
  - A single individual cannot stop the work.
- Disadvantage:
  - There may be simultaneous changes to the same version
  - This creates effort in merging the versions (the one that checks in first does not need to merge; but all later ones do).

# Optimistic Checkout

- Merging the versions
  - Status of the checked-out version may have changed in the repository
  - *Merging of* changes and resolution of conflicts may be necessary.
- Example (output version):

```
class Hello {
    public static void main( String[] args ) {
        System.out.println(getHello());
    }
    private static String getHello() {
        return "Hello Uni!";
    }
    private static void doNothing() {}
}
```

# Merge/Conflict Resolution: Example



# Components of a (software) configuration

- Source elements, e.g.
  - Program text
  - Libraries
  - Initialization data
  - Documentation
  - Configuration files for system to be built
- Derived elements (translated, formatted, linked,...)
- Tools used (translator, formatter, linker, packager,....)

# Configuration for intermediate steps of the SW development process

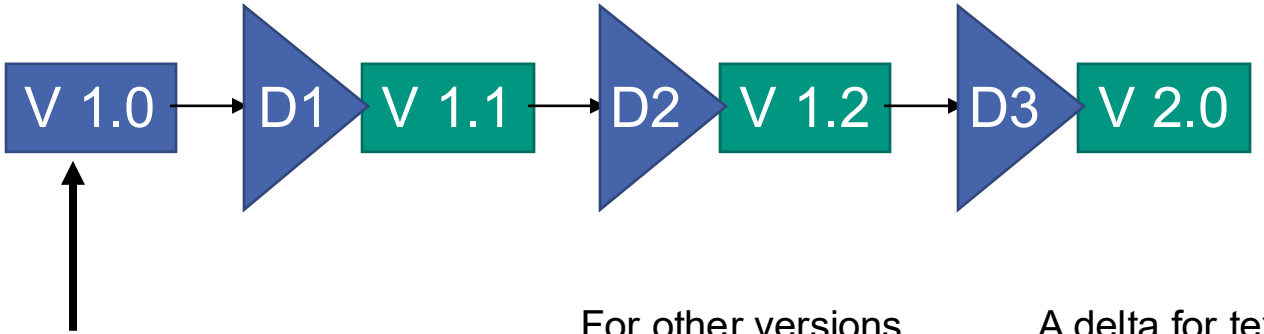
- Configurations for all results of all development phases, e.g.
  - Requirements documents
  - Models
  - Specification
  - Design
- tracking these enables
  - Reviewing of changes
  - tracing software to requirements and specifications (important for changes)
  - Reset to a working configuration

# How are versions managed?

- Complete saving of each version is space consuming.
- Alternative
  - **Forward deltas**: save basic version and changes made to it
  - **Backward deltas**: save current version and the reverse changes for previous versions
- A delta is the difference between two versions; a change script that transforms one version to the other. In software, a delta is usually about 1-2% of the size of a (full) version.

# Forward Deltas

Elements marked in blue are saved




Version V 1.0 saved in full

For other versions, changes are saved in deltas.
Regenerate V1.2 thus:
V 1.2 = D2(D1(V1.0))

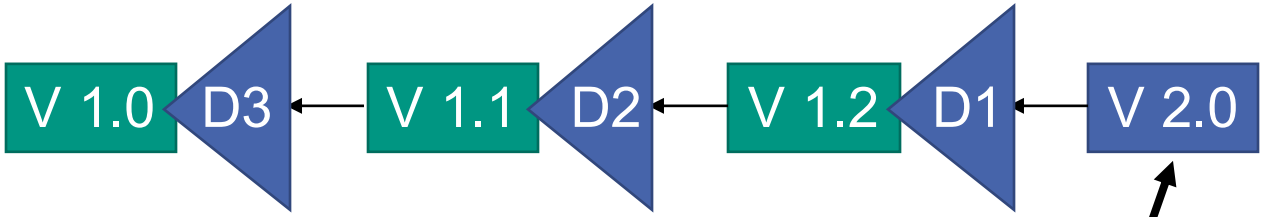
A delta for text, such as source code, is simply a series of commands of the form:
”delete lines x-y”
“insert the following lines after line z”

# Forward Deltas

- Advantages
  - Quick access to old versions
- Disadvantages
  - Slow access to more recent version
- Current version is needed most often

# Reverse Deltas

Only elements marked in blue are saved !




For other versions changes saved in in deltas. Regenerate V1.1 thus:
V 1.1 = D2(D1(V2.0))

Version V 2.0 saved in full

# VERSION CONTROL SYSTEMS

# Question

- Who developed the first version control system (**Revision Control System (RCS)**)?
  - Bill Gates (Founder Microsoft)

  - Larry Page (Founder Google)

  - Tim Berners-Lee (founder of the World Wide Web)

  - Walter F. Tichy

# Revision Control System (RCS, 1983)

- Manages multiple versions of a file
- Automated
  - Storage
  - Recovery
  - Logbook
  - Automatic identification of the software elements
  - Merging versions
- Versioning of directories not supported.
- No remote access (repository is local)
- One of the first open-source systems (1983)

# Revision Control System (RCS, 1983)

- Manages multiple versions of a file
- Automated
  - Storage
  - Recovery
  - Logbook
  - Automatic identification of the software elements
  - Merging versions
- Versioning of directories not supported.
- No remote access (repository is local)
- One of the first open-source systems (1983)

Excerpt from the man page to rcs:

```
IDENTIFICATION
       Author: Walter F. Tichy.
       Manual Page Revision: 5.9.4; Release Date: 2019-12-31.
       Copyright © 2010-2015 Thien-Thi Nguyen.
       Copyright © 1990, 1991, 1992, 1993, 1994, 1995 Paul Eggert.
       Copyright © 1982, 1988, 1989 Walter F. Tichy.
```

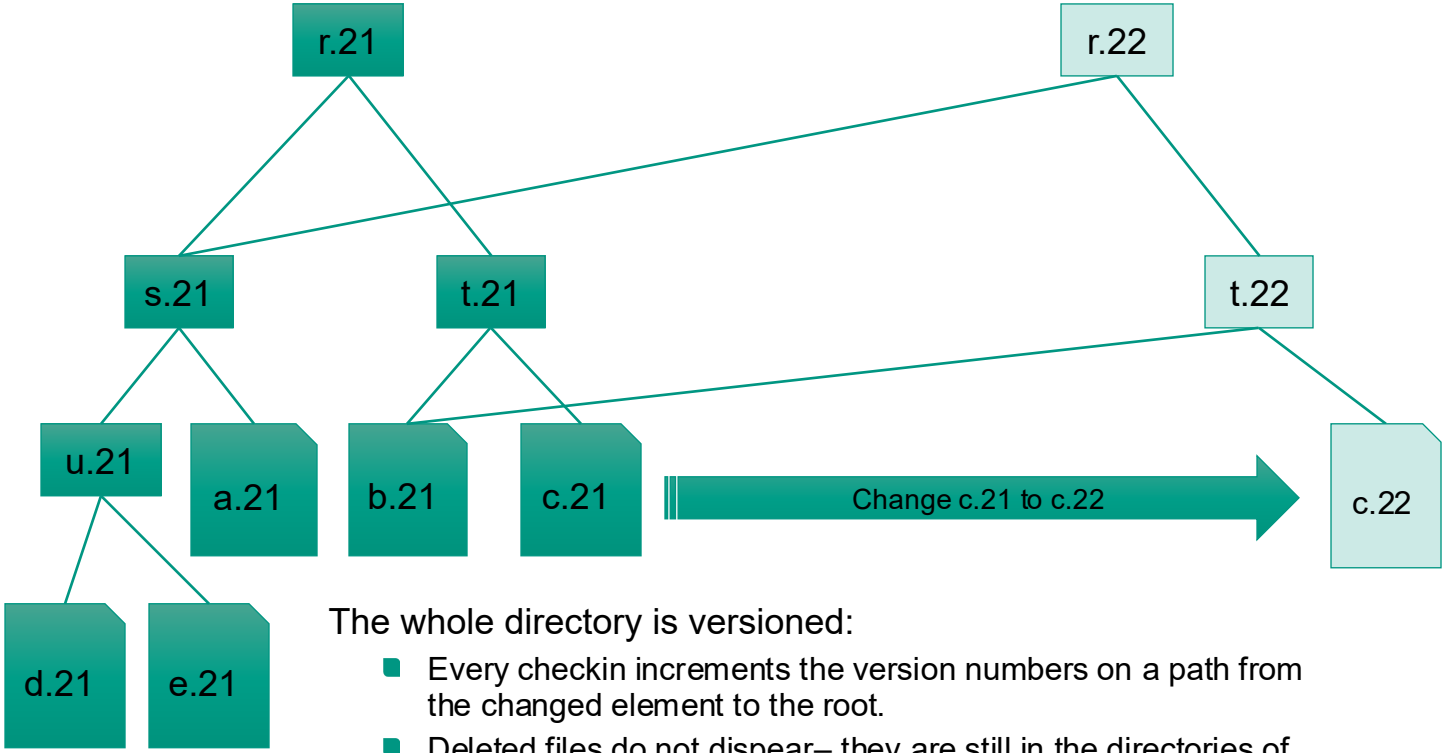
# Concurrent Versions System (CVS, 1990)

- Versions entire directory trees
- Internet enabled (checkout from, checkin to, remote repositories)
- Uses RCS internally

# Subversion (SVN, 2000)

- Further development of CVS
- Internet-enabled, i.e., the repository is stored on a server, developers access it via the Internet.
- SVN versions the entire project repository, including moving, renaming and copying directories and files.
- Checkin is **atomic**, i.e., a change is transferred to the project repository completely or not at all. There is no danger of an inconsistent state because of an aborted checkin.
- Optimistic checkout

# Versioning of Directories in SVN




The whole directory is versioned:

- Every checkin increments the version numbers on a path from the changed element to the root.
- Deleted files do not dispear– they are still in the directories of version before the „deletion“

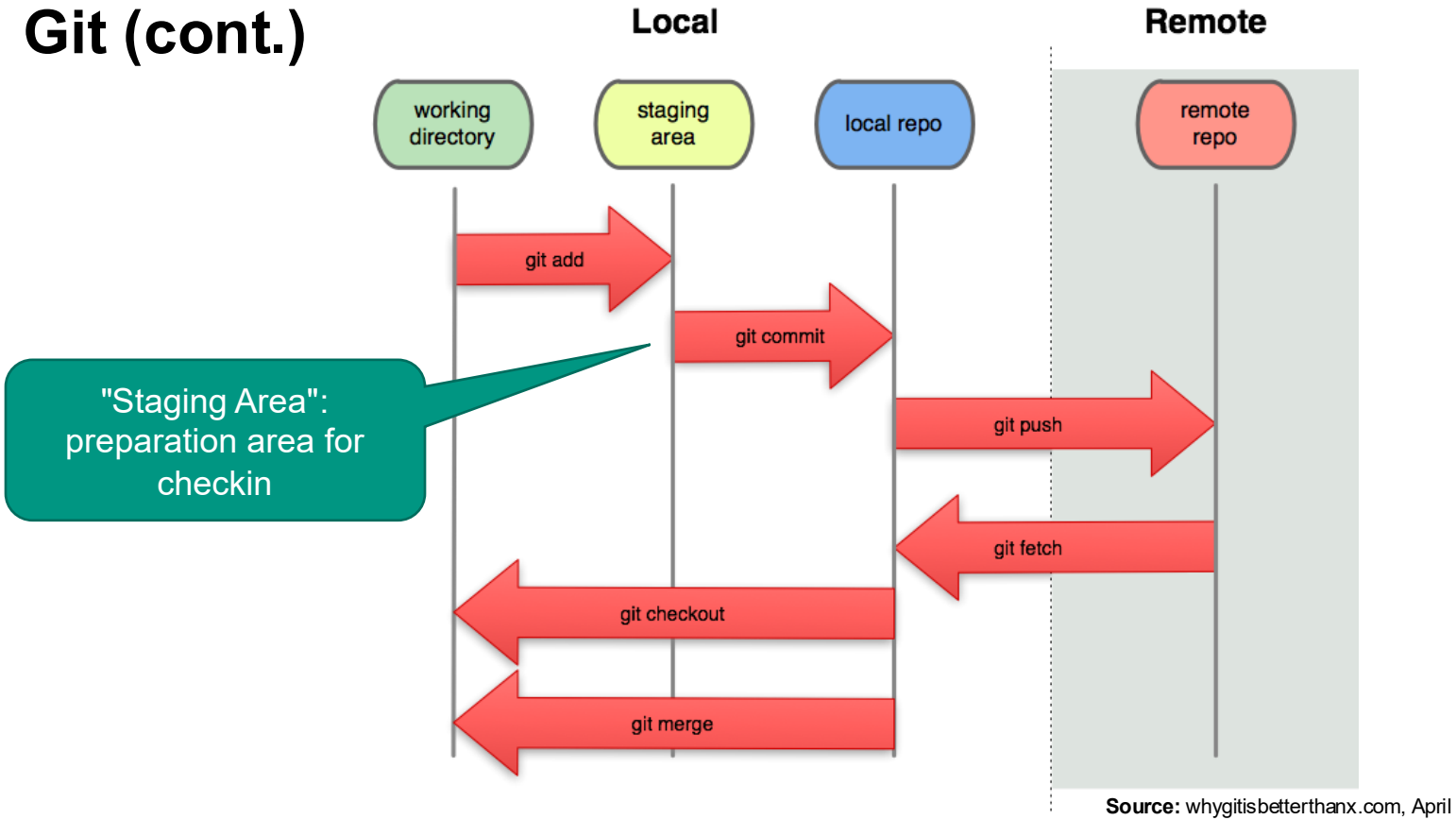
Configuration management with

# GIT

# Git (2005)

- abstraction of files; everything is an object: file contents, directories, configurations, variants
- Dealing with variants is an integral part
- Cryptographic security of the history
- Entire repository is a local copy for each user
- For this additional communication step: `push/pull/fetch` for transferring objects between repositories
- "command hierarchies" and workflows (for release of configurations)

# Git (cont.)




**Source:** whygitisbetterthanx.com, April 2013

# User interface

- `git <command>` , e.g.
  - `help` - list of commands
  - `init` - Create repository
    - Create empty project in repository or "transport" an existing project to repository
  - `clone` - load repository of a project
  - `add` - Add new file/change to be tracked by GIT; transfer file to the staging area.
  - `rm` – remove a file from tracking by Git
  - `commit` - Apply added changes to your own repository
  - `status` – status of your working directory, incl. files to commit
  - `push` - transfer commits to (remote) repository, now visible to all in project.
  - fetch - get changes from (remote) repository to local repository
  - `merge` - combine changes of one development branch with another (from local repo to working directory)
  - `pull` = `fetch` + `merge`

# Creating a Git repository

- The repository is located on your computer.

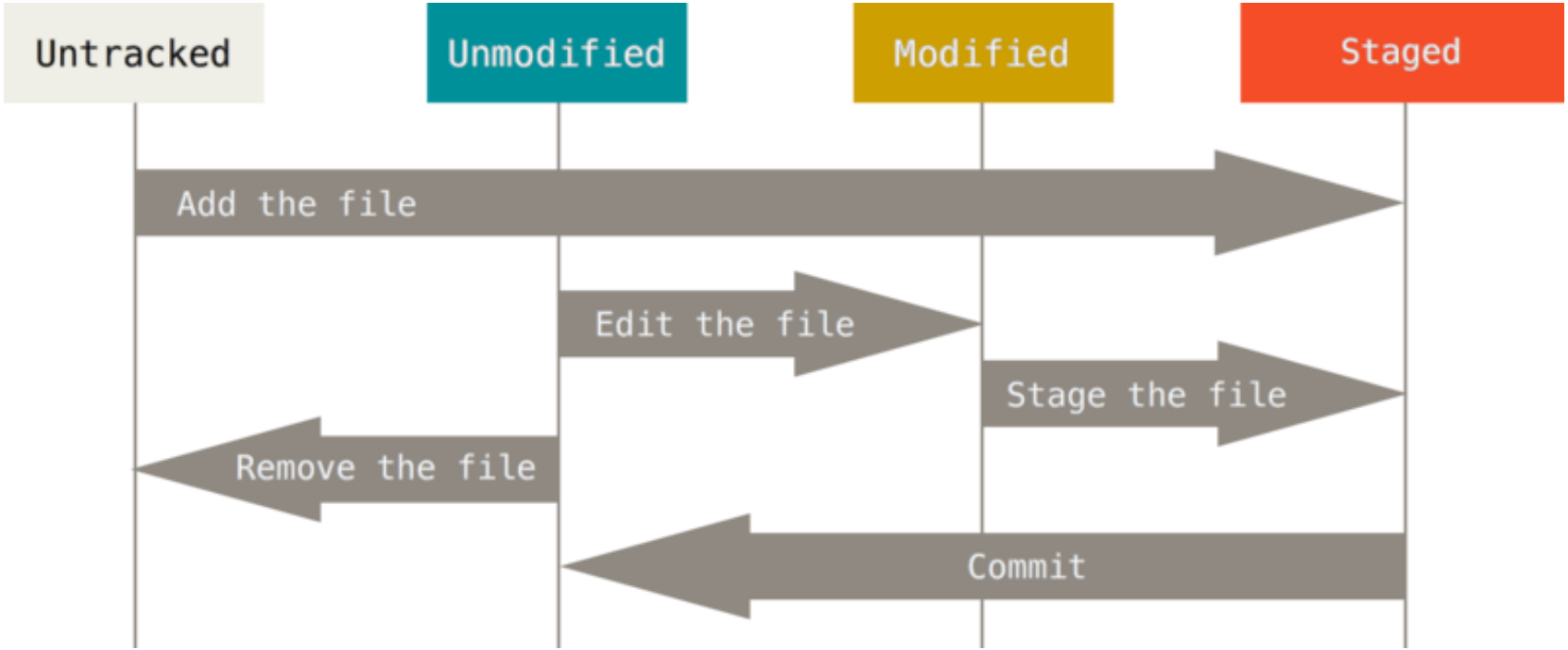
- `git init`
  - Creates new, empty repository in the current directory named .git
  - Example:
    ```
    c:\Test\> git init
    ```

- You're already done - no need to checkout for a working copy!
- Start populating the directory with files and directories.
  These are initially untracked by Git.

# Checkin

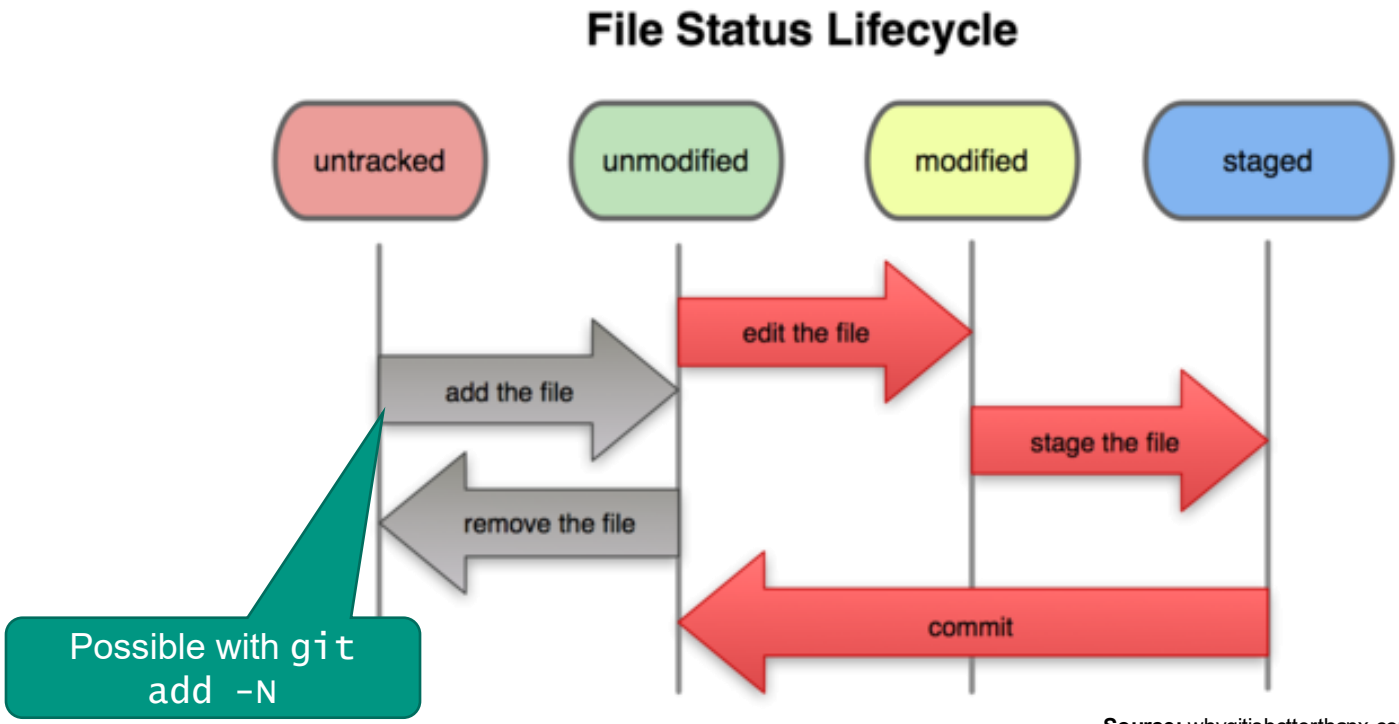
- `git add <file|directory>`
  - Adds `<file|directory>` to the staging area. Now the files are tracked.
  - This is how you collect semantically related changes and prepare for the `commit`
- `git commit –m 'initial project version'`
  - Writes back the files in the staging area to your own, local repository
  - No conflicts, since this is a local operation

# Life cycle of a file in Git




**Source:** Pro Git book, July 2020

# Life cycle of a file in Git




**Source:** whygitisbetterthanx.com, April 2011

# Git Example: Logbook

```
swt1@ipd:~/swt/$ git log Hello.java
commit 1c5845e2b25dbc76a70ee3fdb19edc77a3becab3 (HEAD, refs/heads/master)
Author: Tobias Hey <hey@kit.edu>
Date: Tue Feb 19 16:50:46 2018 +0100

    We welcome only the SWE

:100644 100644 22f3eec... 39423ae... M Hello.java

commit 818f1f71bb4979f19c49289a81947c00b112e4c4
Author: Tobias Hey <hey@kit.edu>
Date: Tue Feb 19 16:50:25 2018 +0100

    Hello world written

:000000 100644 0000000... 22f3eec... A Hello.java
```

Command: Get detailed logbook

Second checkin: checkin of a change.

First checkin: configuration number, user name, date and time, files logged in, log message.

# Git Example: Delta

```
swt1@ipd:~/swt/$ git diff 818f 1c58
diff --git a/Hello.java b/Hello.java
index 39423ae..22f3eec 100644
--- a/Hello.java
+++ b/Hello.java
@@ -1,5 +1,5 @@
 class Hello {
     public static String getHello() {
- return "Hello World!";
+ return "Hello SWE!";
     }
 }
\ No newline at end of file
```

Command: Get the delta between configuration 818f and 1c58.

Indication of the lines with changes: In both files in lines 1…5.

Read: "To go from configuration 1 to configuration 2, remove the line with 'Hello world!' and insert the line with 'Hello SWE!“

# Replicating a Git repository

- The repository can be made available via several protocols.
  - `file`, `ssh`: Access is directly via file system or via SSH connections to the remote server.
  - `http(s)`, `git`: Access via standard HTTPS ports with different HTTP authentication options (read-only unauthenticated access also possible).

- Replicate a repository from somewhere else with the command:
  `git clone <address>` e.g.,
  `git clone https://.../repos /tmp/projects`
- In contrast to the `svn checkout`, the entire repository **including its history is** transferred to the local computer!
- You can now modify the files, stage them with git add, and commit them.

http://git-scm.com/book/en/Git-on-the-Server-The-Protocols

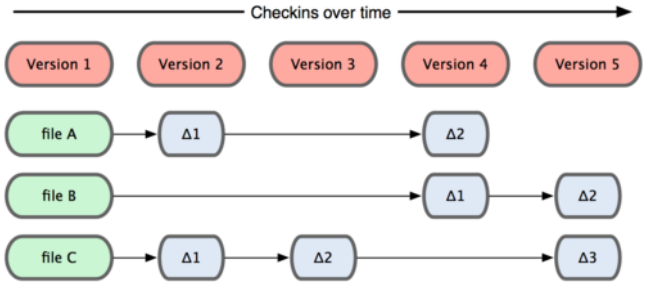
# How does all this work?

## READ:

- https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository
- https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

# Lines of Development in Git (1)

- Unlike SVN, Git does not store deltas, but snapshots.




This is how SVN & Co. think.

How Git thinks

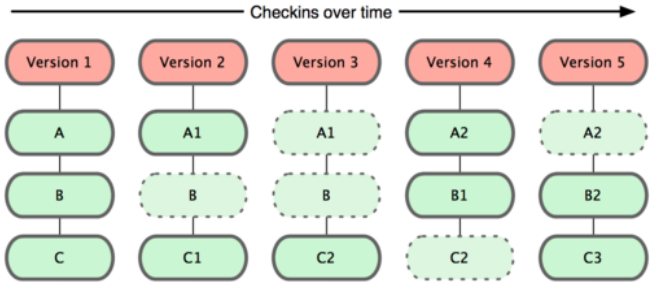



Image sources of the following slides:
http://git-scm.com/book/en/

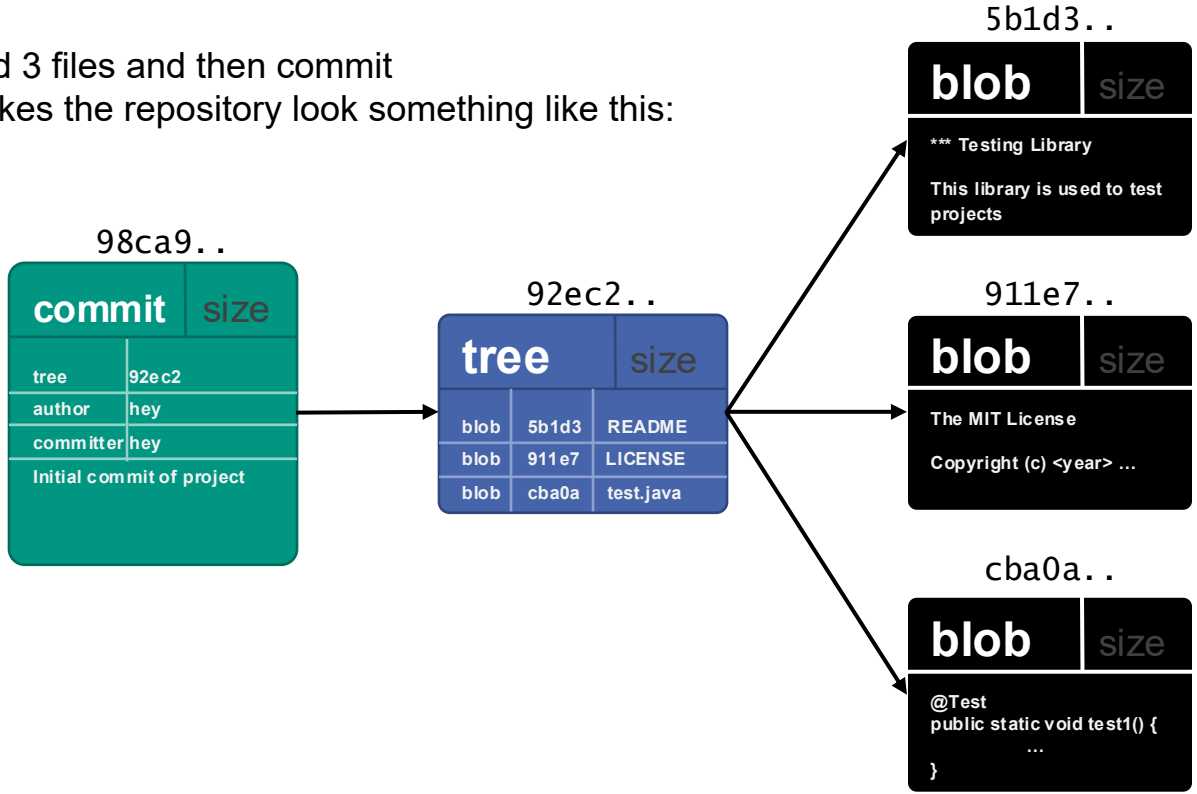
# Lines of development in Git (2)

- Unlike SVN, Git does not store deltas, but snapshots.

- When you add an object to a Git repository, Git also creates a commit object that stores the metadata about the commit:
  - A pointer to the snapshot
  - Author and logbook entries
  - Pointers to parents of the commit
- A parent of a commit is the predecessor in the history. If you merge, the commit has two parents!

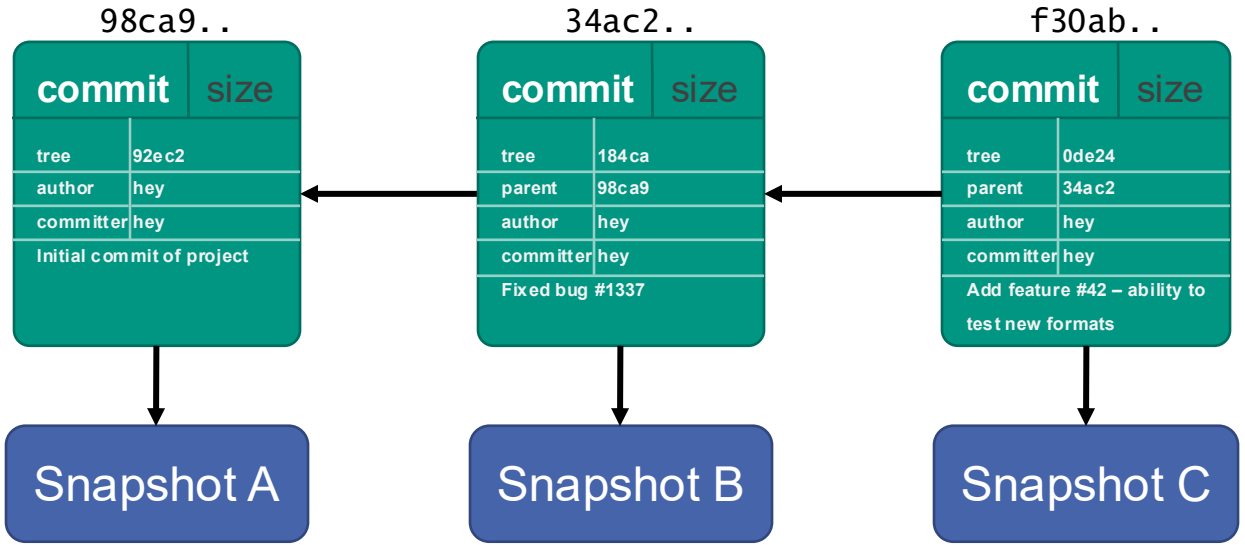
# Lines of development in Git (3)

Add 3 files and then commit
makes the repository look something like this:



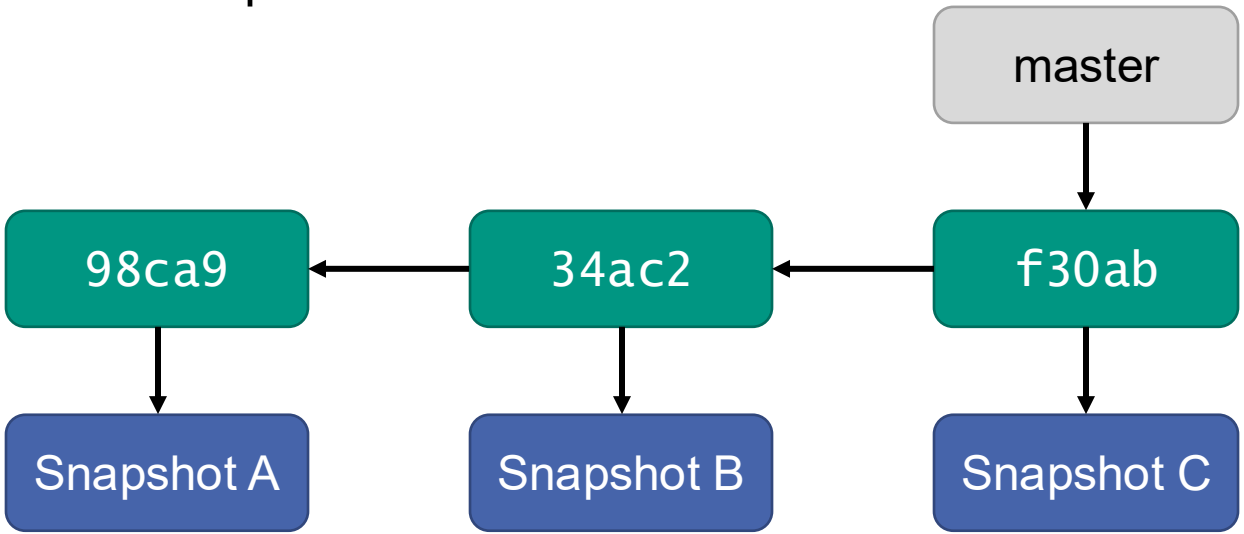
# Lines of development in Git (4)

- After three commits, it might look like this:



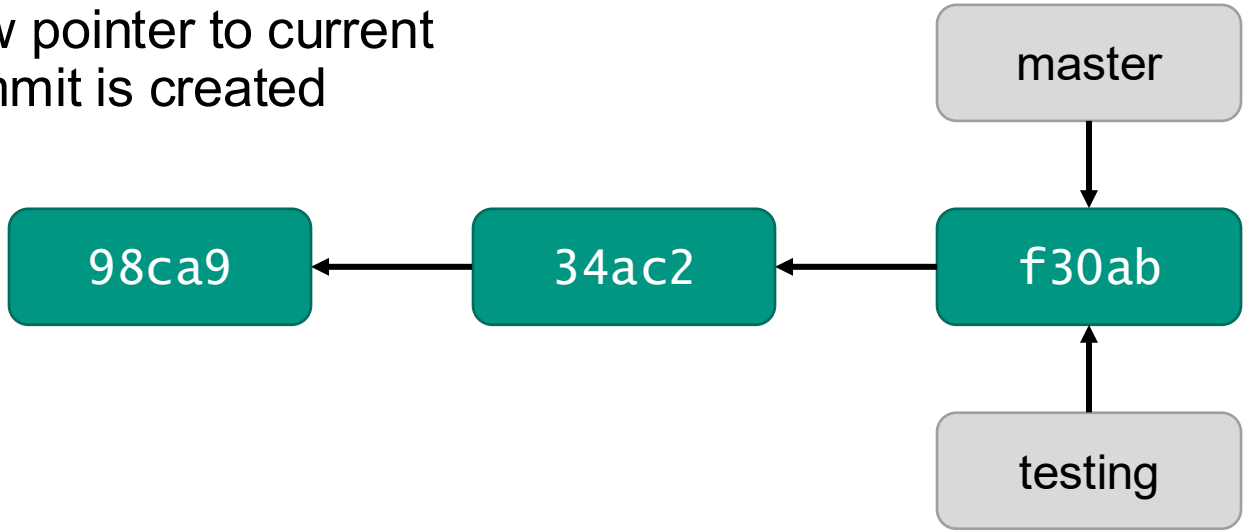
# Lines of development in Git (5)

- Implicit development branch: master



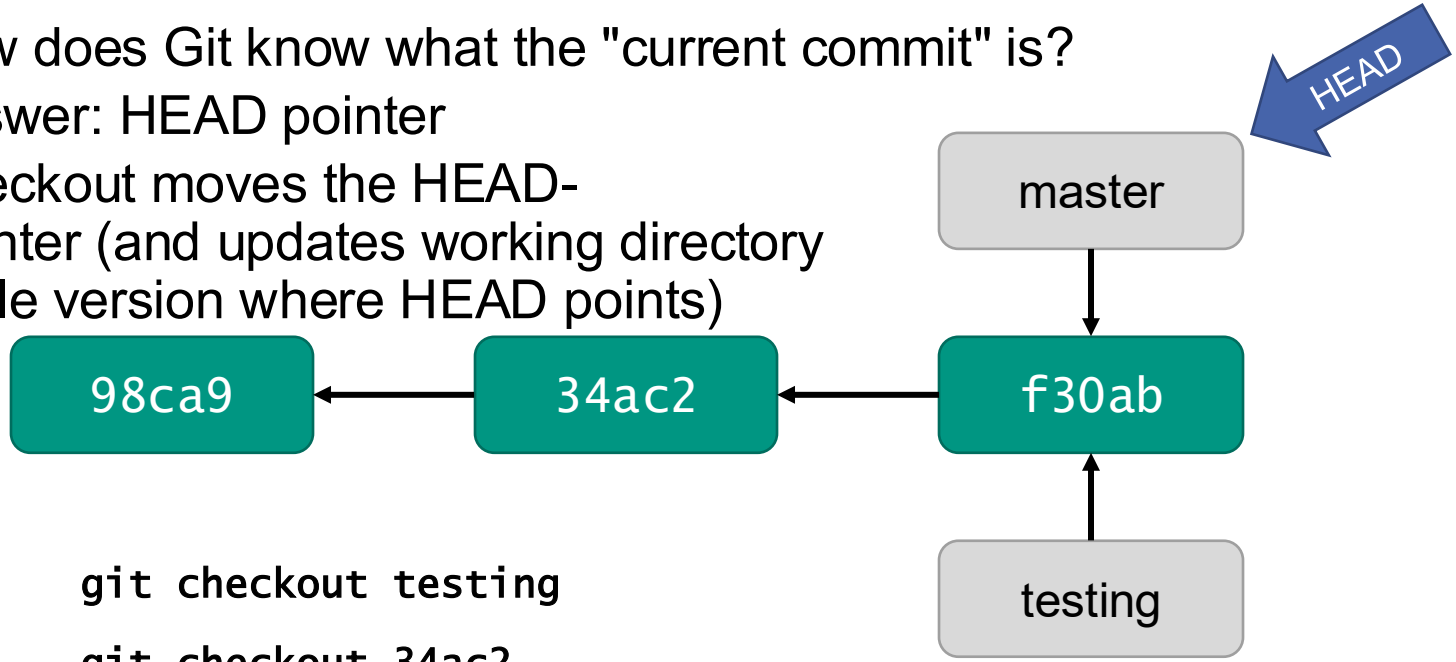
# Lines of development in Git (6)

- Creating a new branch "testing" with the command `git branch testing`
- New pointer to current commit is created



# Lines of development in Git (7)

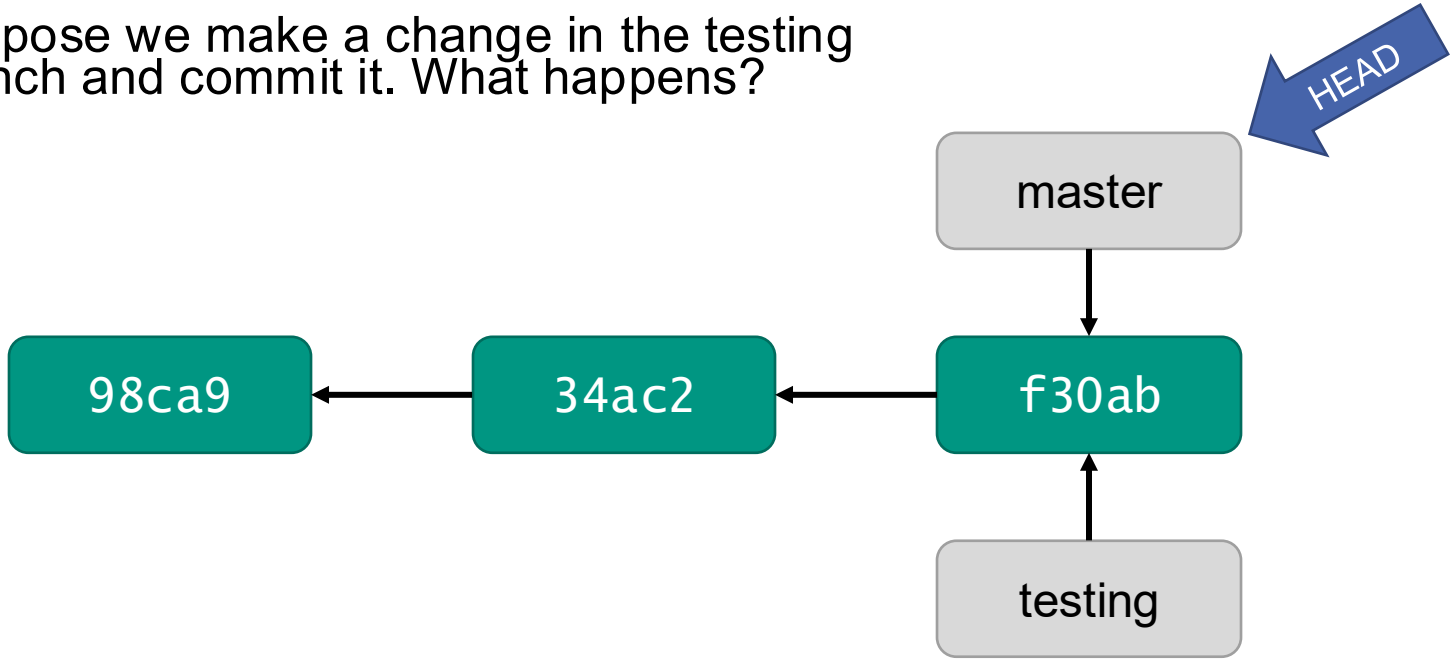
- How does Git know what the "current commit" is?
- Answer: HEAD pointer
- Checkout moves the HEAD-Pointer (and updates working directory to file version where HEAD points)




```
git checkout testing

git checkout 34ac2
```

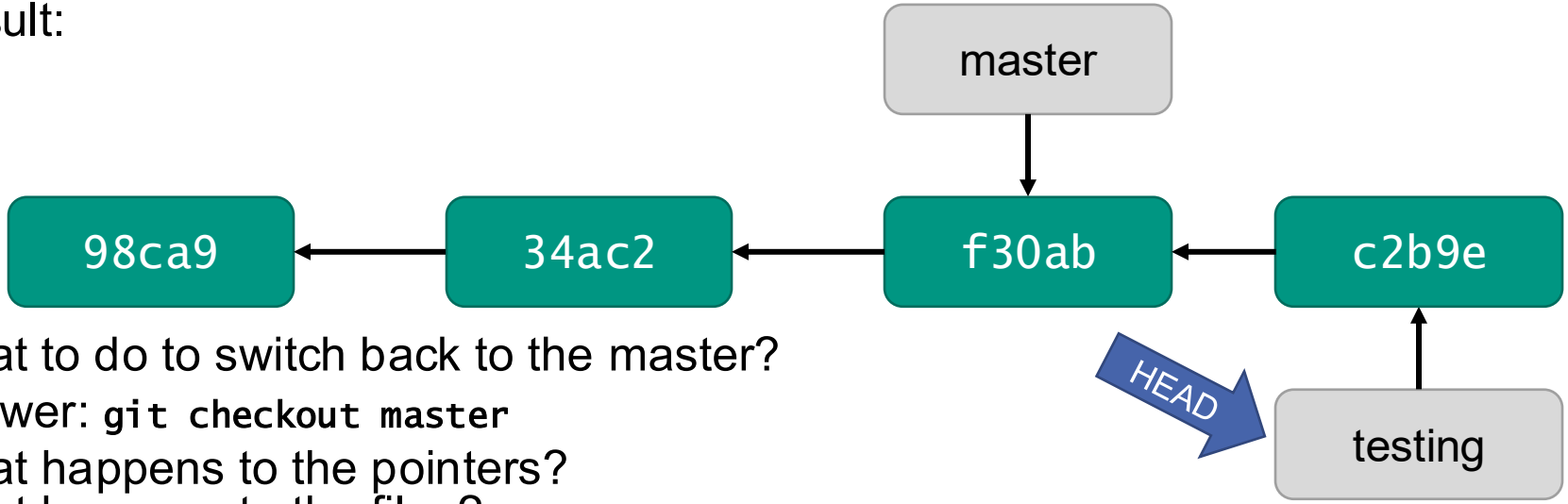
# Lines of development in Git (8)

- Suppose we make a change in the testing branch and commit it. What happens?



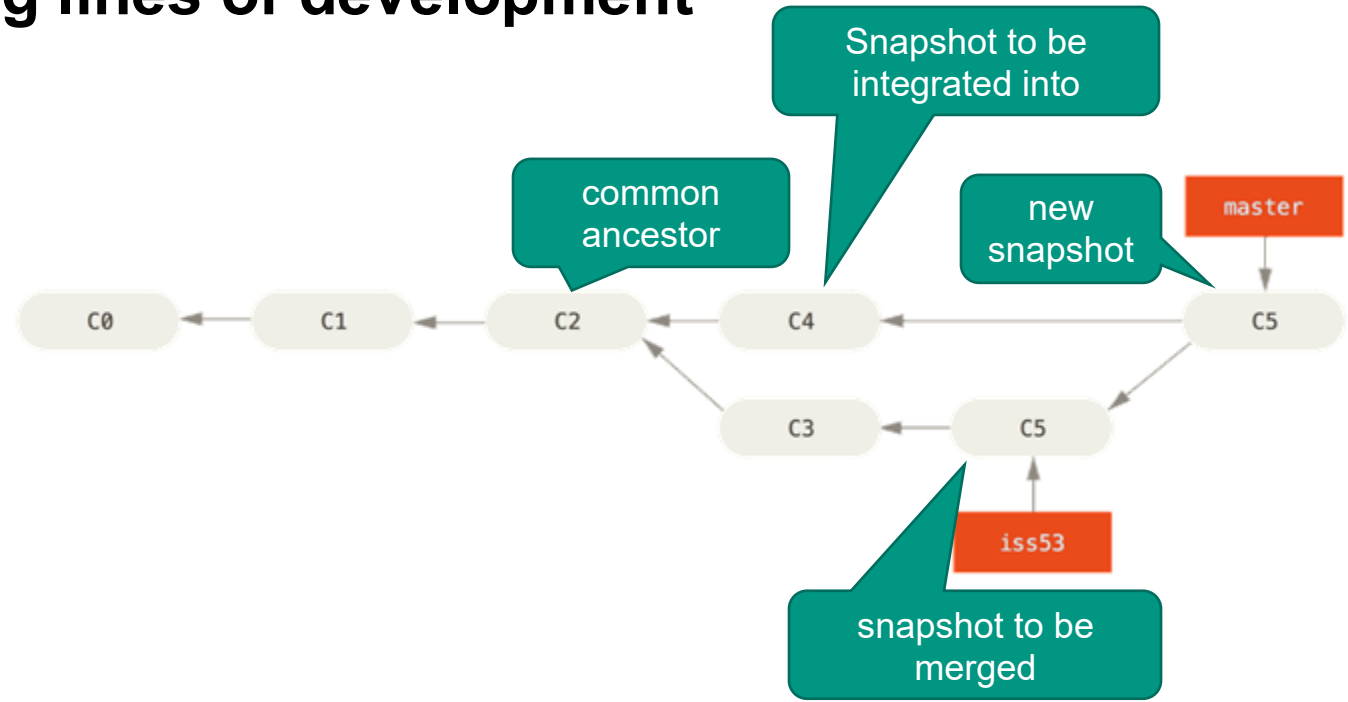
# Lines of development in Git (8)

- Suppose we make a change in the testing branch and commit it. What happens?
- Result:



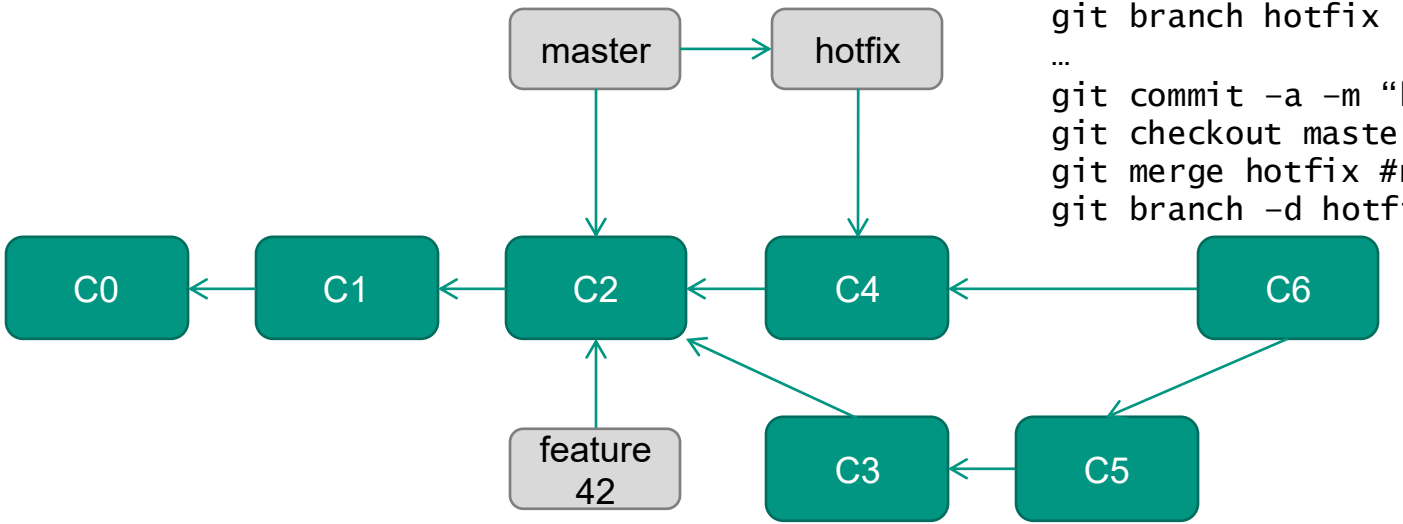

- What to do to switch back to the master?
- Answer: `git checkout master`
- What happens to the pointers?
  What happens to the files?

# Merging lines of development



# Merging lines of development

```
git branch feature42
git checkout feature42
…
git commit –a –m “f42 take 1”
git checkout master
git branch hotfix
…
git commit –a –m “hotfix done”
git checkout master
git merge hotfix #no merge
git branch –d hotfix #delete
```




```
git checkout feature42
….
git commit –a –m “f42 take 2”
git checkout master
git merge feature42 #result C6
```

**READ:**
**http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging**

# Synchronize local files with a remote repository

- `git pull <repository>`
  - (`fetch`) Fetches changes from the repository (origin)

```
        A---B---C master on origin
       /
D---E---F---G master (local)
```

  - (`merge`) Updates local files with new versions from the repository that have been committed since last checkout.

```
        A---B---C--        remotes/origin/master
       /           \__
D---E---F---G---H master
```

  - Possible conflicts are displayed in separate file
- Perform a `commit` before the `pull` / `merge`!
- **READ: https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches**

# Backup tip

- Create a private (!) repository on GitHub
- Use it as a remote repository
- Everything you push there will survive if your notebook should die!

# Literature

- Bruegge, B., Dutoit, A., *Object-Oriented Software Engineering: Using UML, Patterns and Java*, 2010, Pearson, Ch. 13.
- SVN:
  - Detailed manual: http://svnbook.red-bean.com
  - Project page: http://subversion.apache.org
  - Illustrated guide to Tortoise SVN: http://goo.gl/hNCfH
- Git:
  - Project page: http://git-scm.com
  - Git quick course for SVN users: https://git.wiki.kernel.org/index.php/GitSvnCrashCourse
  - Git internals: http://git-scm.com/book/en/Git-Internals

# TESTING

# Motivation

- Software artifacts always contain defects
- The later a defect is found, the more expensive it is to fix it
- The goal is to find defects  as early as possible

# Software amateur testing tools

- Testing with output statements in the program
- Testing with an interactive debugger
- Testing with test scripts

- Common to all
  - Manual check of the outputs

# Disadvantages of these methods

- Test cases must be re-run when program changes are made
  - Tedious and impractical and often impossible
    - Output instructions are often no longer present
    - if still present, the output instructions must be removed or disabled (and inserted or enabled later again)
    - when testing with debugger there is no information what variables should be inspected.

# Automatically running test cases

- Test case
  - Running a program under known conditions
  - Input data given
  - Result known
  - Goal: Detection of defects

# Automatically running test cases

- Automatically running test cases
  - Check result of their execution themself
  - Return: boolean value (successful or failed)
  - Optional: Raise an exception on failure
  - Can be collected into large test suits.
  - Can be executed any number of times (daily, hourly, at each change).
  - Can be used as a regression test.

# JUnit 5: A test framework for Java

- Allows fast and hierarchical construction of test suits
- returns the failed test cases
- by default, only textual display; graphical display is realized by the programming environment, e.g. Eclipse
- Authors: Erich Gamma, Kent Beck
- Where: https://junit.org/
- Similar tools exist for a variety of other programming languages

# Framework architecture for automatically running test cases

- Why a framework?
  - Simplifies the writing of the test cases
  - Combining test cases into test suits
  - Automatic running of entire test suits; only failures are reported.
  - Has a standardizing effect (everyone adheres to the conventions of the framework, summary of test sets thus possible).

# JUnit 5: Test class

- Contains (related) test cases in the form of methods.
- Usually holds references to the test objects to be tested
- The comparisons of target and actual values take place using assertions from the `org.junit.jupiter.api.Assertions` class.
- Labeling of the test methods is done with annotations.

# Excursus: Annotations as of Java 5

- Annotations are meta-information that can be added to the source code.
- They remain in the byte code, but do not directly affect the execution of the program.
- They can be read and processed at translation time by tools and at runtime by *reflection.*
- Annotations may occur in any declaration: Package, Class, Interface, Instance/Class Variable, Method, Parameter, Constructor, Local Variable.
- Example: mark method as deprecated

```
@Deprecated public void someMethod() { ... }
```

# JUnit 5: Example

```
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class LibraryTest {
  private library bib;

  @BeforeEach public void buildLib() {
    bib = new library();
  }

  @Test public void bookIsInLibrary() {
    boolean b = bib.testAvailability(”Harry Potter");
    assertTrue(b, "Harry Potter must be in the library.");
  }

  @AfterEach public void cleanup() {
    bib = null;
  }
}
```

Assertions are mapped via **`import static.`** Calling the methods is then possible without naming the **`Assertions class.`**

Methods with **`@BeforeEach`** are used to build test resources.

**`@Test`** marks a test case.

Methods with **`@AfterEach`** are used to release test resources.

# JUnit 5: Some assert methods

- `assertEquals(Object expected, Object actual)`
  Fails if the call to the `equals method of expected` with the parameter `actual` returns the value false.
- `assertSame(Object expected, Object actual)`
  Fails if the references point to different objects.
- `assertEquals(float expected, float actual, float delta)`
  Fails if `expected` and `actual` differ by more than `delta.`
- `assertTrue(boolean expression)` Fails of expression is false.
  `assertFalse(boolean expression)` Fails of expression is `true.`
- `assertNull(Object reference)` Fails if the reference is not null
  `assertNotNull(Object reference)` Fails if the reference is null
- `assertThrows(Class<T> exceptiontype, Executable subprogram)`
  Fails if an exception of type `exceptiontype` does not occur when executing `subprogramm.`
  - Example: `assertThrows(SQLException.class, () -> {...});`
- To all these methods there is a variant with a string for a message in case of error.

# JUnit 5: Annotations

- `@Test` - Marks a test case
- `@Timeout(42)` - Sets the maximum runtime in seconds
- `@Disabled` - Marks a test case that should not be executed
  - The reason is specified as a string:
    ```
    @Disabled("Functionality is still missing.")
    @Test public void borrowBook() { ... }
    ```
- `@BeforeEach` - Marks a method that is executed before each test case.
- `@AfterEach` - Marks a method that is executed after each test case.
- `@BeforeAll` - Marks a class method that is executed before the test class is instantiated.
- `@AfterAll` - Marks a class method that will be executed after all other methods mentioned so far.

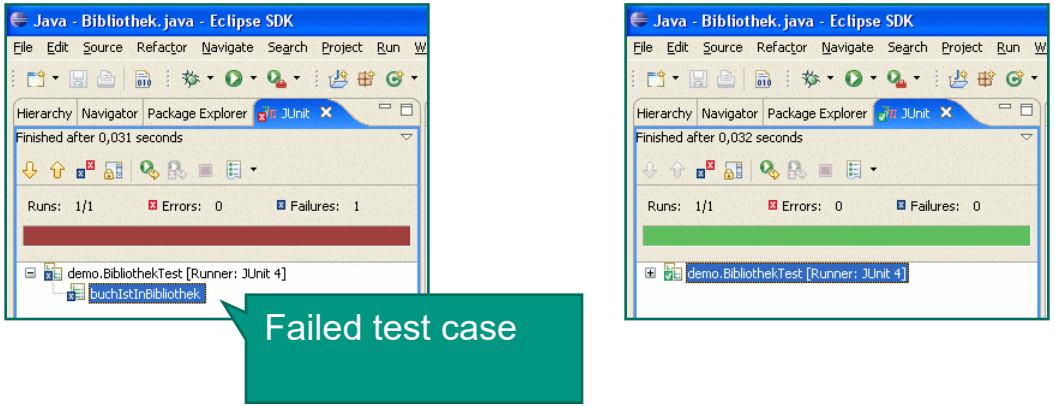
# JUnit 5:
# Execution order

```
[ ... ]

public class LibraryTest {
    [ ... ]

1   @BeforeAll public static void connectWithDB() { ... }

2 5 @BeforeEach public void buildLib() { ... }

3   @Test public void bookIsInLibrary() { ... }

6   @Test public void borrowBook() { ... }

4 7 @AfterEach public void cleanup() { ... }

8  @AfterAll public static void separateFromDB() { ... }

}
```

The order of the test cases is not guaranteed. Points 3 and 6 could also be swapped.

# Execution in Eclipse

- In Eclipse it is sufficient to select "Run As" → "JUnit Test" in the context menu of a test class; then the test methods are executed.
- Test classes can also be grouped into packages, and these in turn into larger packages; selecting with "Run As" → "JUnit Test" will execute all included test methods.
- The display takes place in a separate window.



# Summary

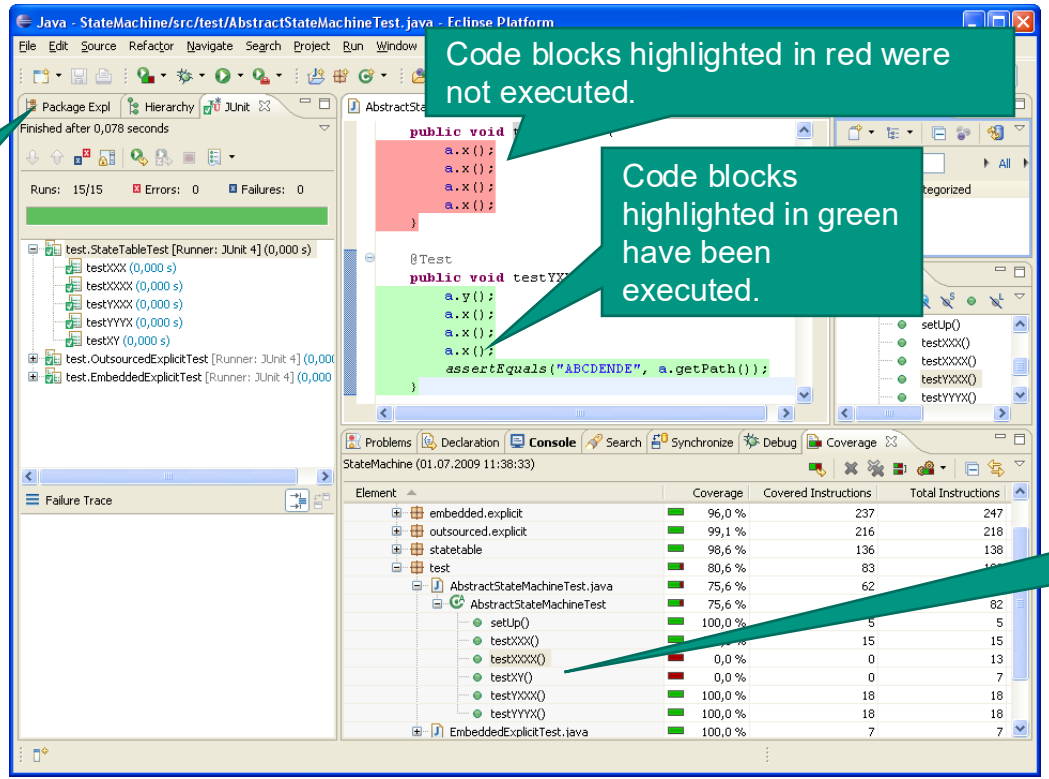
- Automatically executable test cases are particularly important for regression testing
- Regression testing helps to detect when old defects reappear
- JUnit and similar regression testing tools are extremely important in practice
- Test cases check themselves

# Test programs for IDEs - EMMA (1)

- EMMA determines the **statement coverage** of test cases for Java applications and generates statistics from them.
  - No Java code is needed for this (.class or .jar files are sufficient).
- The verifiable statement coverage ranges from entire packages, classes and methods to individual lines of code.
  - If a line of code is only partially executed, this is registered by EMMA.
- http://emma.sourceforge.net/
- http://www.eclemma.org/ (Eclipse plugin)
- https://www.jetbrains.com/help/idea/code-coverage.html (IntelliJ Integration)

# Test programs for Eclipse - EMMA (2)

It is possible to check the statement coverage of each code file of the project.




Method `testXY()` was not executed completely.

# What we need to do before a delivery

- Develop
  - Adhere to code conventions (Checkstyle)
  - Writing and executing tests (JUnit)
  - Check test coverage (EMMA)
- When all tests are green
  - Post new revision in version control (Git)
  - Send report(s)
  - Build deliverable JAR
  - If necessary: Gather current dependencies
- Deliver
  - Copy all JARs together
  - copy to USB stick, send by mail, install, publish...
  - Notify other developers of new revision or version

This does not all have to be done manually!

# AUTOMATION WITH MAVEN
# (NOT ON MIDTERM NOR FINAL EXAM)

# What is there to support in the development process?

- "Everything that is (potentially) done with the source code after programming."

- Data and operations in the development process:
  - Compile (and check dependencies for this)
  - Manage dependencies (and make them available for the translation process)
  - Testing
  - Check source code quality
  - Package (into a single file), install, publish
  - Generate documentation
  - Create reports (e.g. test results, source code statistics)

# What is Apache Maven?

- Tool for (partial) automation of tasks during the development process
- "Standardized" file format for specifying task steps, dependencies, and data in the artifact development process.
- Automated execution of development process tasks
  - Example: Execution of module tests depends on successful translation → Translation takes place before module tests
  - Example: A library is needed to run the artifact → Maven takes care of downloading and including it

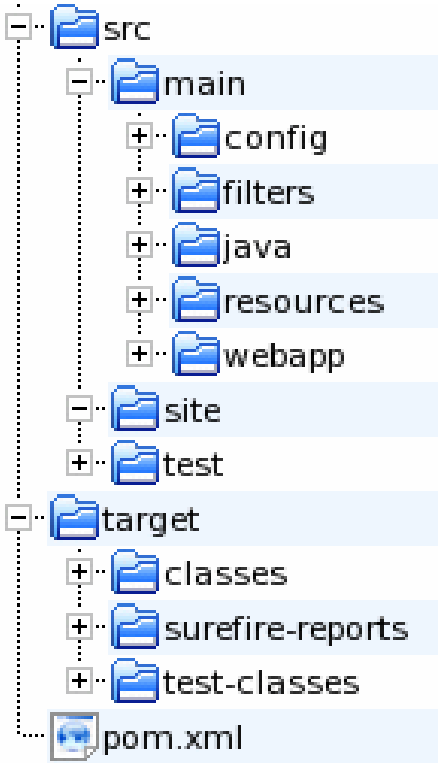
# How does Maven work?

- Convention (applies equally to each project)
  - "Reasonable" default values
  - Assumptions about the project and directory structure

- `pom.xml` ("Project Object Model")
  - Exactly one pom.xml per artifact
  - Contains the name and version
  - Describes deviations from the convention

# Convention for Directory Structure

- Source files
  - Program
  - Project information page
  - Module tests
  - Etc.
- Object files
  - Translated classes
  - Reports
  - Etc.
- `pom.xml`



# Vocabulary (1)

- Archetype
  - Programming project template, e.g. J2EE, sample project, Maven plugin
  - Create a new project: `mvn archetype:generate`
- Artifact ID
  - Artifact name
  - Multiple artifacts possible within one Maven project
- Group ID
  - Equivalent to the "package" line of the Java source files

# Vocabulary (2)

- `mvn < plugin>:< goal>`

- Plugin
  - Partial functionality of Maven for a specific task
  - Examples: Version control, compiling, testing, creating new (empty) project according to a template, ...

- Goal
  - Specific subtask of the development process
  - Defined from convention or `pom.xml`
  - A plugin provides a set of Goals

# How to use Maven? (1)

- Create empty artifact, e.g.:
  - `mvn archetype:generate -Darchetype=archetype-quick-start -DgroupId=edu.kit.ipd.swt1.HelloWorld -DartifactId=HelloWorld`
  - Result: subdirectory HelloWorld with `pom.xml` and two directory subhierarchies (for program and tests)
- Translate artifact:
  - `mvn compile`
  - Result: Java bytecode file(s) of the program source codes
- Test artifact:
  - `mvn test`
  - Result: Java bytecode file(s) of the module test source code, its execution and result report.

# Execution of tests in maven

- Run manually: `mvn test`

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running edu.kit.ipd.jmjrst.deduplicator.GreyHistogramComparatorTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.107 sec
Running edu.kit.ipd.jmjrst.deduplicator.GreyHistogramTest
Tests run: 5, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.02 sec <<< FAILURE!
Results :
Tests in error:
    testGetHistogram(edu.kit.ipd.jmjrst.deduplicator.GreyHistogramTest):
        javax.imageio.IIOException: Can't read input file!

Tests run: 9, Failures: 0, Errors: 1, Skipped: 0
```

# How to use Maven? (2)

- Pack artifact:
  - **`mvn package`**
  - Result: An executable/reusable artifact, i.e. a JAR file containing the module; e.g. "HelloWorld-1.0-SNAPSHOT.jar".

Note: `mvn package` also runs tests...

- Goals can depend on each other
  - Example: Maven packages (by default) only tested artifacts
  - If one calls **`mvn package`**, the tests are executed automatically. Only if the test run is error-free, then the artifact is packaged (e.g., into a JAR).

  - Example: For testing you need to have translated the source code
  - If testing is to be performed, all sources are automatically compiled

# Maven artifact management (local)

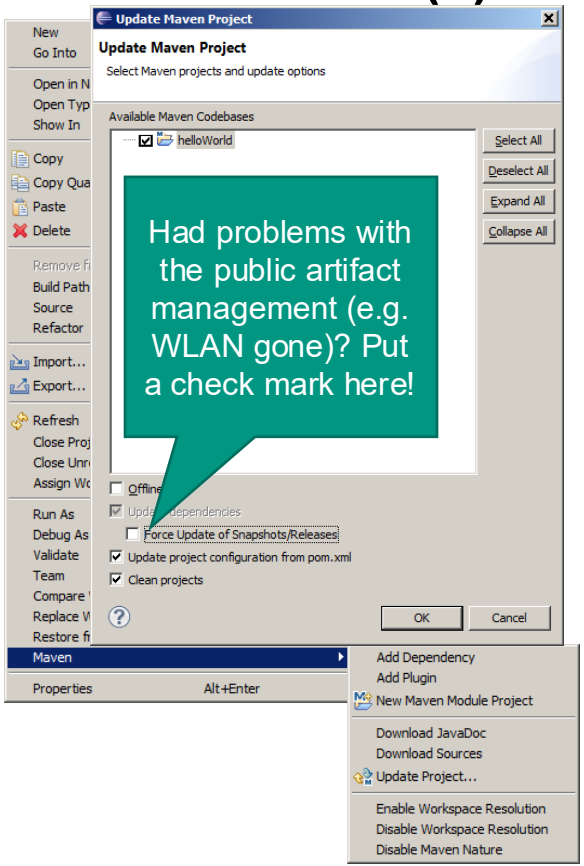
- The artifact management contains artifacts (possibly in different versions).
- Maven ...
  - ... maintains the artifacts across projects.
  - ... downloads needed artifacts.
  - ... manages the Java class path.
- An `mvn install` packages an artifact (uses `mvn package`) and "installs" it in the local artifact manager
- An `mvn deploy` sends the packaged artifact to a public artifact manager
- (If you have problems with Maven, sometimes emptying the local artifact management helps).
  - `-U` option forces update of locally stored images

# Maven in an IDE
# General

- **Note:** IDEs provide their own mechanisms for some development process tasks that overlap with Maven.
  - Do not define project/artifact dependencies in the IDE, but **only** in `pom.xml`.
  - Only use Maven in the IDE for tasks from the development process.

- Maven plugins for IntelliJ and Eclipse (for Java developers) are pre-installed in the current versions

# Maven in Eclipse (2)
# Project context menu (1)



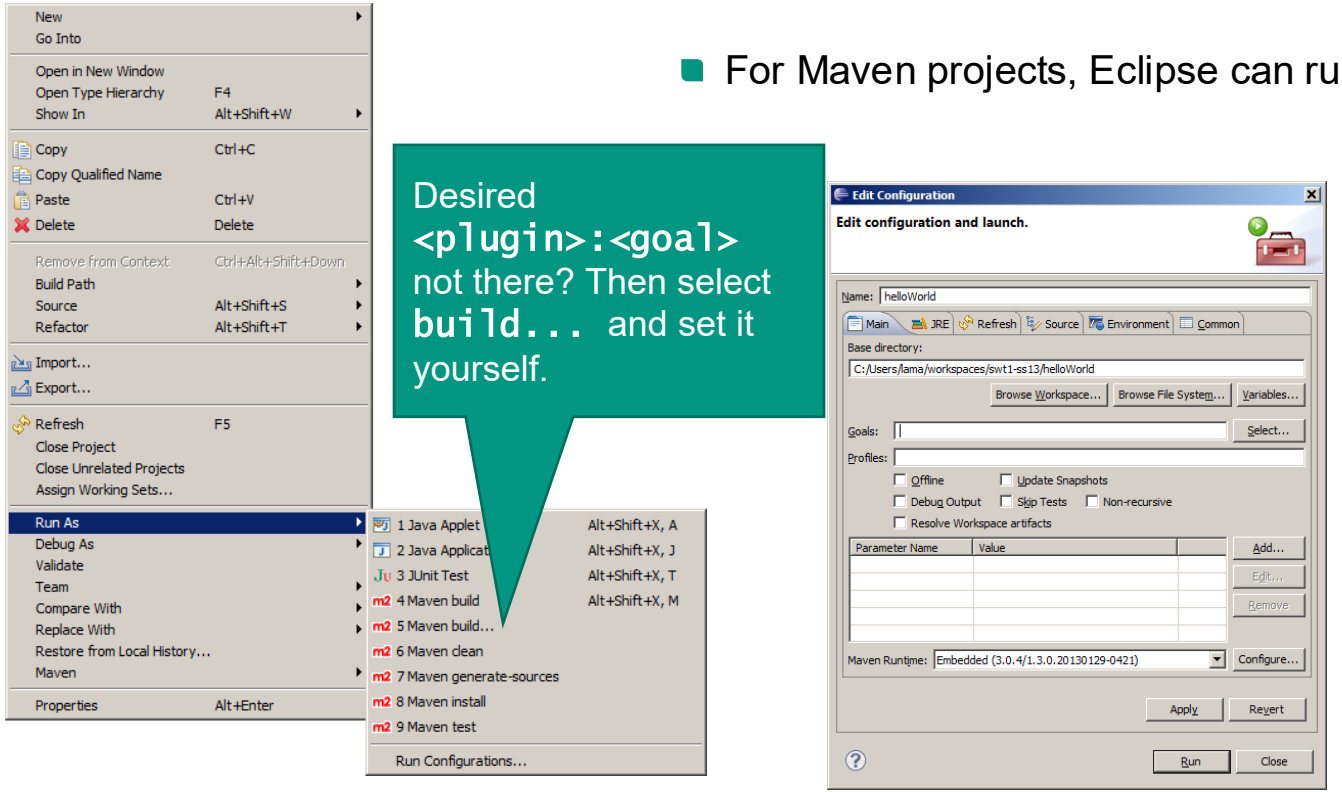

- Management aids for **`pom.xml`**

Important points:

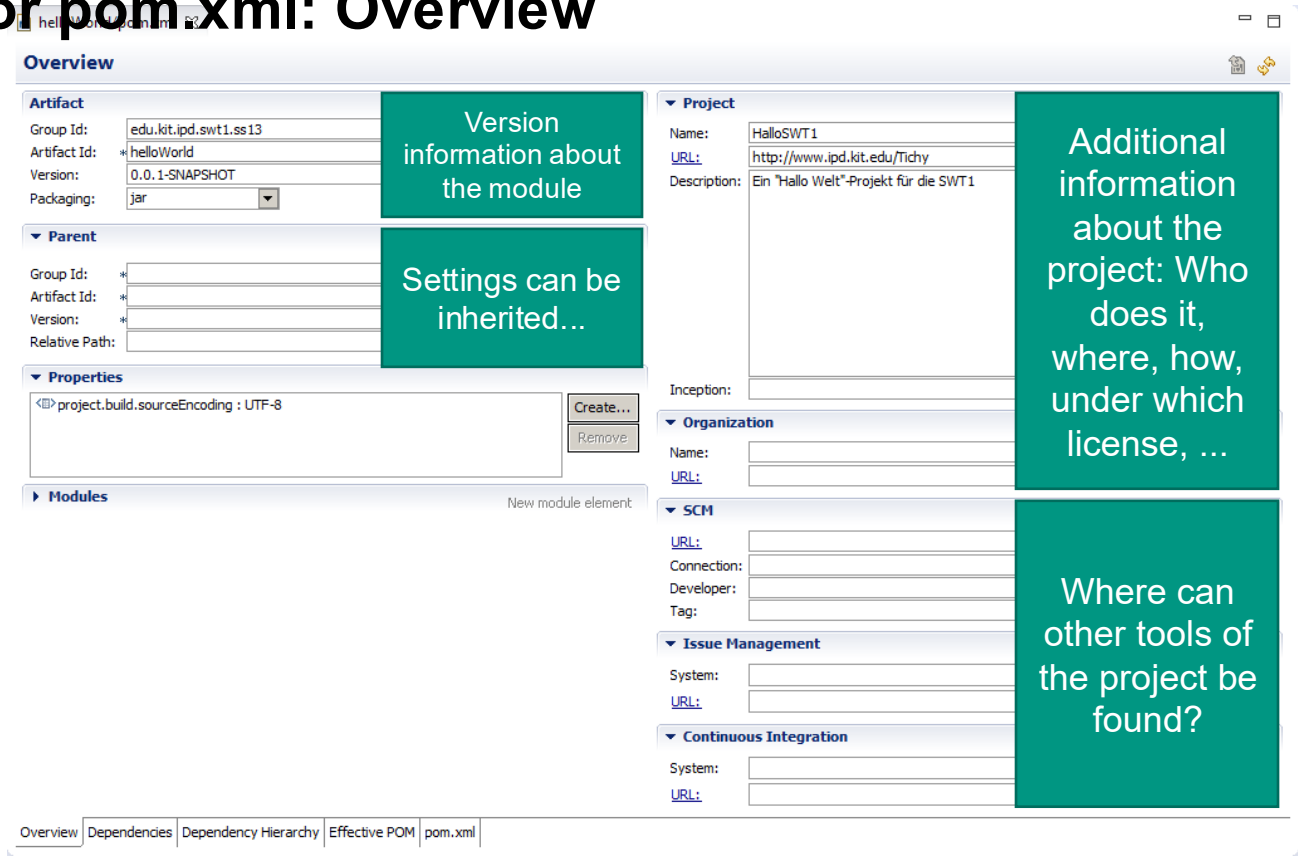
- Add Dependency
  - Adding a dependency
- Download JavaDoc/Sources
  - Documentation and source code download
- Update Project
  - Apply changes to **`pom.xml in the`** Eclipse configuration; e.g., include new libraries in Eclipse class path

# Maven in Eclipse (2)
# Project context menu (2)

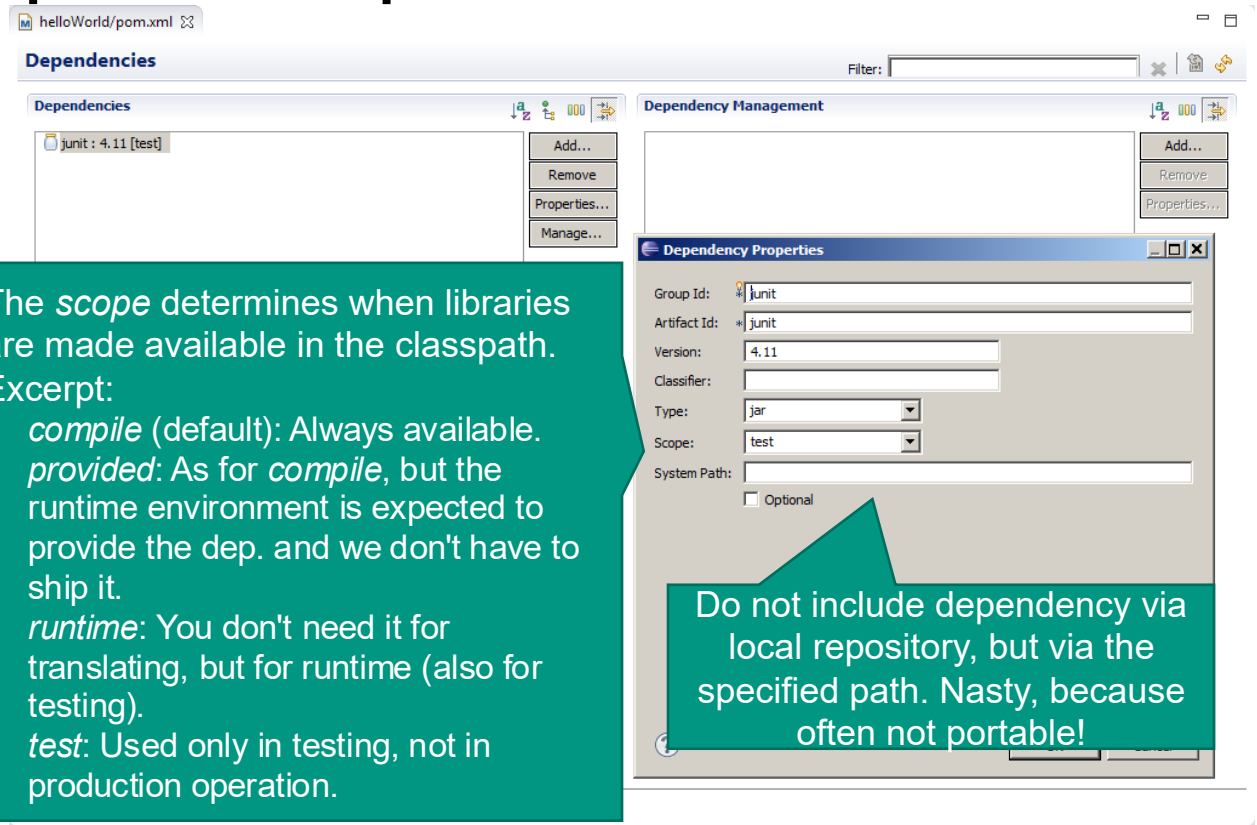
- For Maven projects, Eclipse can run Maven



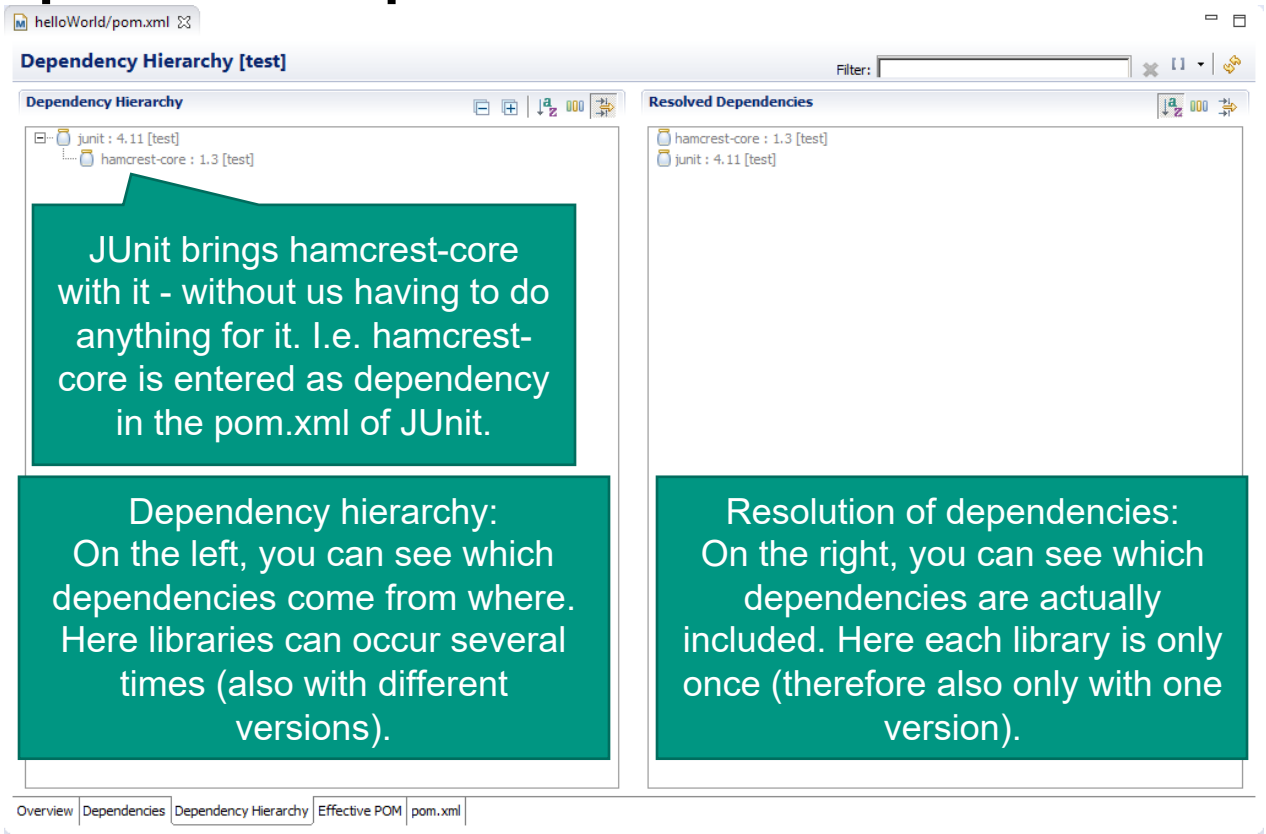
# Maven in Eclipse (3)
# Editor for pom.xml: Overview



# Maven in Eclipse (3)
# Editor for pom.xml: Dependencies



# Maven in Eclipse (3)
# Editor for pom.xml: dependencies resolved



# Literature: Self-study - read on!

Apache Maven: http://maven.apache.org/

→ http://maven.apache.org/guides/getting-started/