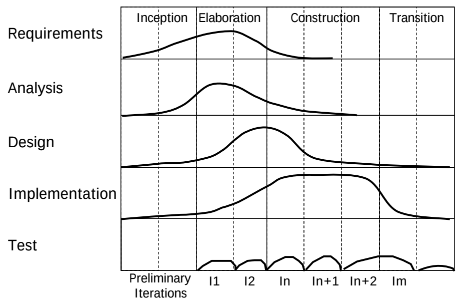
Software Configuration Management

Wolfgang Emmerich
Professor of Distributed Computing
University College London
http://sse.cs.ucl.ac.uk

Context



Learning Objectives

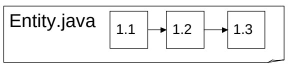
- To understand why SCM is of crucial importance in medium to large-scale software development projects
- To know the principles of version management and software configuration management
- To appreciate how SCM tools support coordination within a team of developers
- To be able to use an state-of-the-art SCM tool in your group project and beyond

Why do we need SCM?

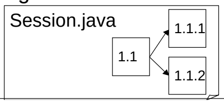
- Teamwork: multiple developers need a
  - Mechanism to share their artifacts
  - Update these artifacts in a controlled manner
- Maintenance: Teams need to
  - Deliver projects in several releases and
  - be able to re-establish earlier release, e.g. to provide a bug fix
  - Merge such changes into the current development baseline
- Safety net: Be able to revert to artifacts that were found to be of a certain quality level

Variants and Revisions

- Artifacts that exist in different versions are known as *configuration items*
- *Revisions* are versions of a configuration item that have emerged over time. They have revision numbers that are usually incremented from revision to revision

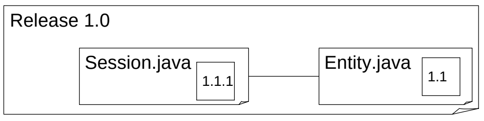


- *Variants* are versions of a configuration item that co-exist (at least for some time)



Configuration

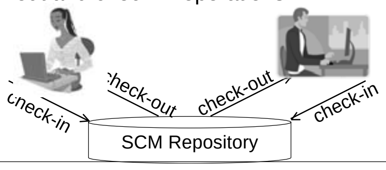
- A *configuration* consists of a number of configuration items. For each of these items one and only one version is selected to be part of the configuration.



- Version selection can be *implicit* (e.g. the last revision) or *explicit* (through tags/labels that mark a particular milestone or release)

Repository

- SCM *repositories* store CIs and their configurations
- Repositories are typically stored on a shared server that is accessible to all team members
- Developers have their own private workspace
- Transfers between repository / workspace through check-out and check-in operations



Concurrency Control in SCM

- Multiple developers may want to access the same CI
- Access needs to be synchronized
- Two different models:
  - Pessimistic: Use of locking and unlocking to prevent more than one developer to change a CI at the same time (used in VSS, for example)
  - Optimistic: Users modify private copies only and may do so concurrently. Private copies are merged together into a new version (This model is used in CVS and subversion)

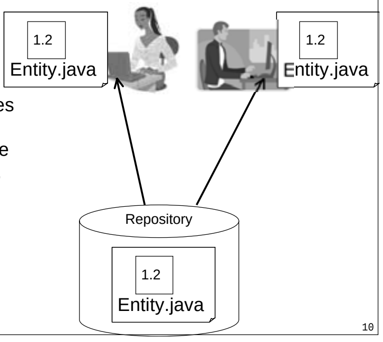
Problems with Locking

- Developers may forget to unlock a file after they have finished updating it
- It is possible that two developers want to edit disjoint sections of the same file and that is not permitted in the pessimistic model.
- Locking might give a false sense of security. Assume Alice locks Session and Bob locks Entity. Because Session calls Entity then Session might not compile after a new version of Entity is checked in. If the entire call graph is locked teamwork grinds to a halt.

How optimistic concurrency control works

Alice and Bob both check out file Entity.java.

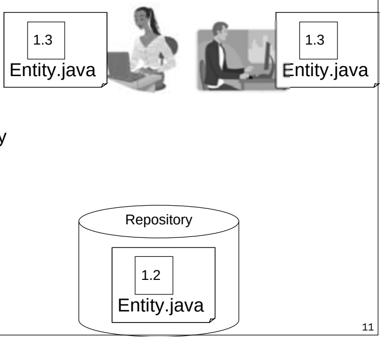
Here, check-out does not lock but only creates a copy in the private workspaces.



10

How optimistic concurrency control works

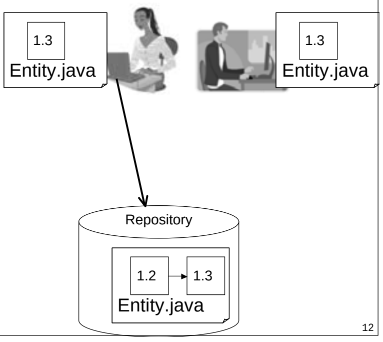
Now both Alice and Bob modify their copy of Entity.java in their private workspace



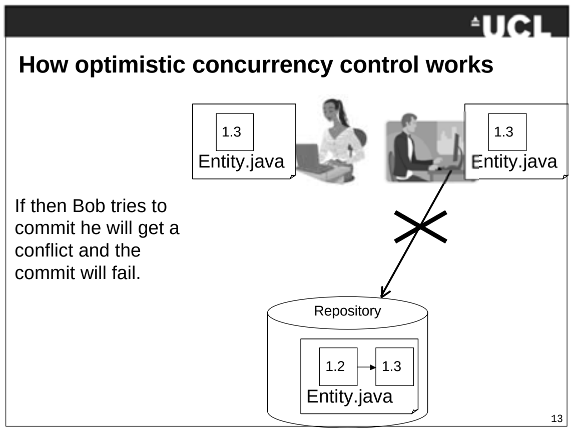
11

How optimistic concurrency control works

Alice commits her changes to the repository first

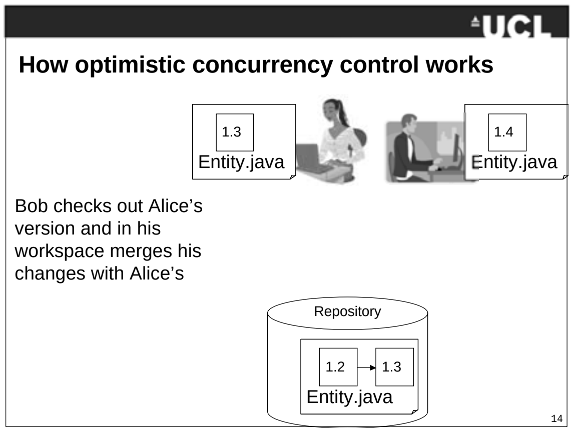


12



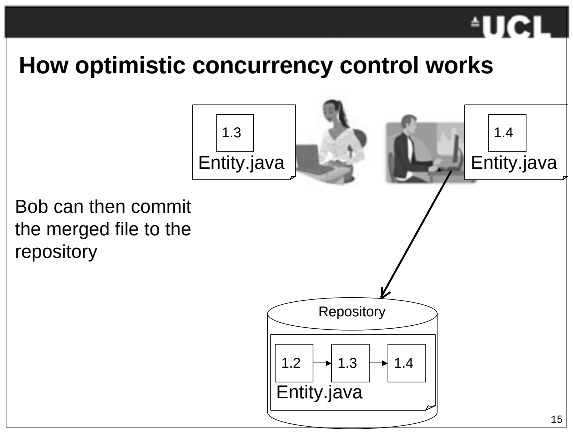
How optimistic concurrency control works

If then Bob tries to commit he will get a conflict and the commit will fail.



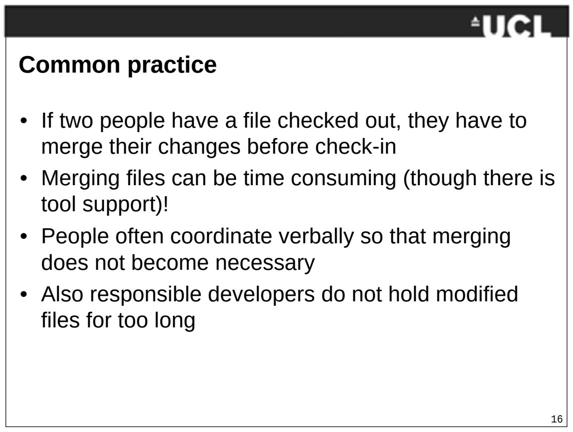
How optimistic concurrency control works

Bob checks out Alice's version and in his workspace merges his changes with Alice's



How optimistic concurrency control works

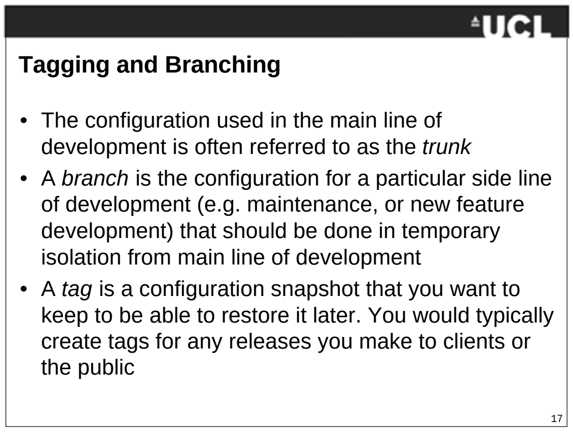
Bob can then commit the merged file to the repository



Common practice

- If two people have a file checked out, they have to merge their changes before check-in
- Merging files can be time consuming (though there is tool support)!
- People often coordinate verbally so that merging does not become necessary
- Also responsible developers do not hold modified files for too long

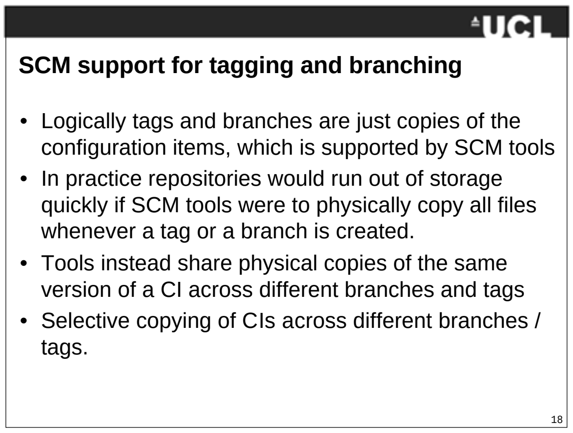
16



Tagging and Branching

- The configuration used in the main line of development is often referred to as the *trunk*
- A *branch* is the configuration for a particular side line of development (e.g. maintenance, or new feature development) that should be done in temporary isolation from main line of development
- A *tag* is a configuration snapshot that you want to keep to be able to restore it later. You would typically create tags for any releases you make to clients or the public

17



SCM support for tagging and branching

- Logically tags and branches are just copies of the configuration items, which is supported by SCM tools
- In practice repositories would run out of storage quickly if SCM tools were to physically copy all files whenever a tag or a branch is created.
- Tools instead share physical copies of the same version of a CI across different branches and tags
- Selective copying of CIs across different branches / tags.

18

Overview of current SCM tools

- Subversion (open source, we will use this in the labs)
- CVS (open source)
- RCS (can only handle versions, not configurations)
- Clearcase (IBM)
- P4 (Perforce Software)
- Source Safe (Microsoft)
- PVCS (Serena Software)

19

Key Points

- SCM tools are enablers of teamwork by providing
  - Sharing when needed
  - Isolation when required
- Provide safety net to restore previous releases
- Very good open source tools available for SCM

20

References

- J. Estublier et al.: Impact of software engineering research on the practice of software configuration management. ACM ToSEM 14(4):383-430.2005. DOI: 10.1145/280277.280280
- R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. ACM Computing Surveys 30(2):232-282. 1998. DOI: 10.1145/1101815.1101817
- B. Collins-Sussman et al. Version control with subversion. O’Reilly Media. 2002. http://svnbook.red-bean.com/

21