The University of Auckland | New Zealand

SOFTENG 254

21/09/2012

SE Software Engineering

The University of Auckland

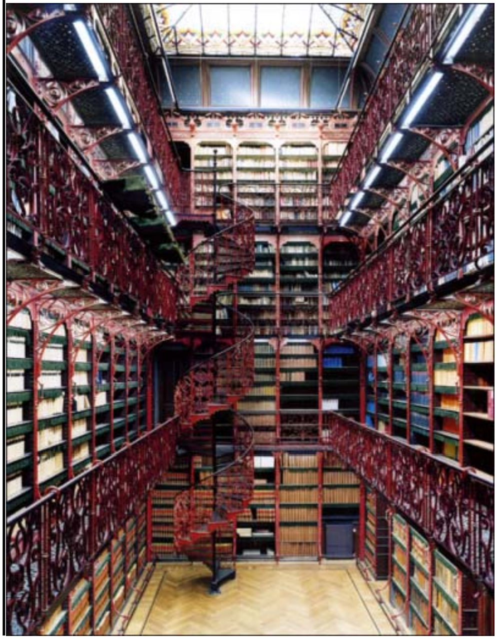
Quality Assurance
Version Control 1

Part II - Lecture 6

Today's Outline

- Version Control
- Managing Concurrency

Version Control



← *Have always used version control*

Has never used version control →

Version Control

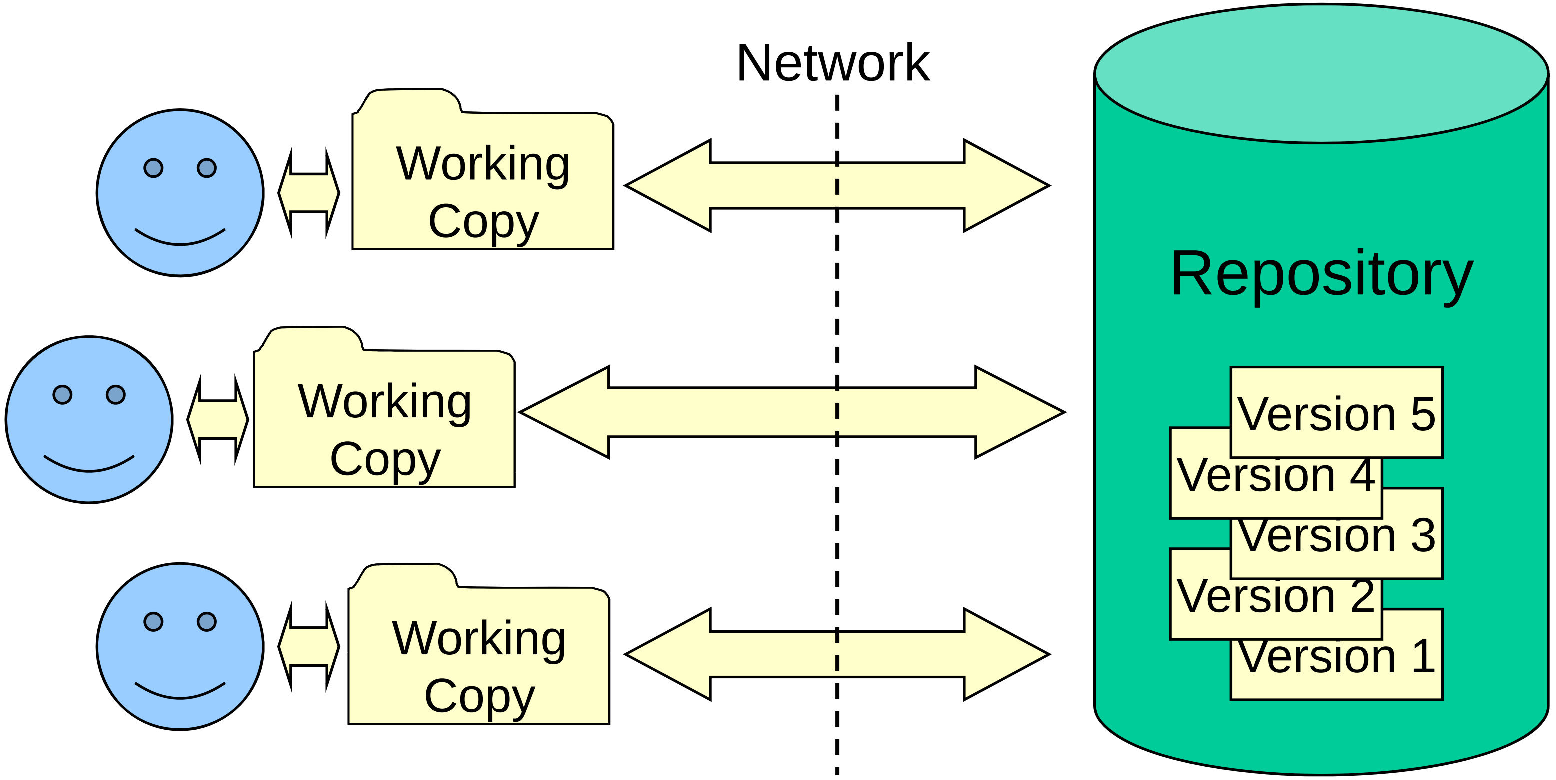
Common problems in a software project:

- A change needs to be **undone**
- Old code that was **overwritten** would be useful again
- Several developers work on the same program part **simultaneously**
- How do I get the **latest version** of the code?

The solution: a Version Control System (VCS)

- Manages a **common repository** for all artefacts
- Controls **concurrent access**
- Creates new **version** for each change (redo/undo possible)
- Helps to **merge** several contributions to same part

Version Control System



- Developers work on their **local working copies**
- Developers **synchronize** their working copy with the repository
- Repository usually uses **delta encoding** for the versions
- Two ways to **avoid conflicts**: locking and merging

Product Space and Version Space

Product space: What is versioned? How is the data organized?

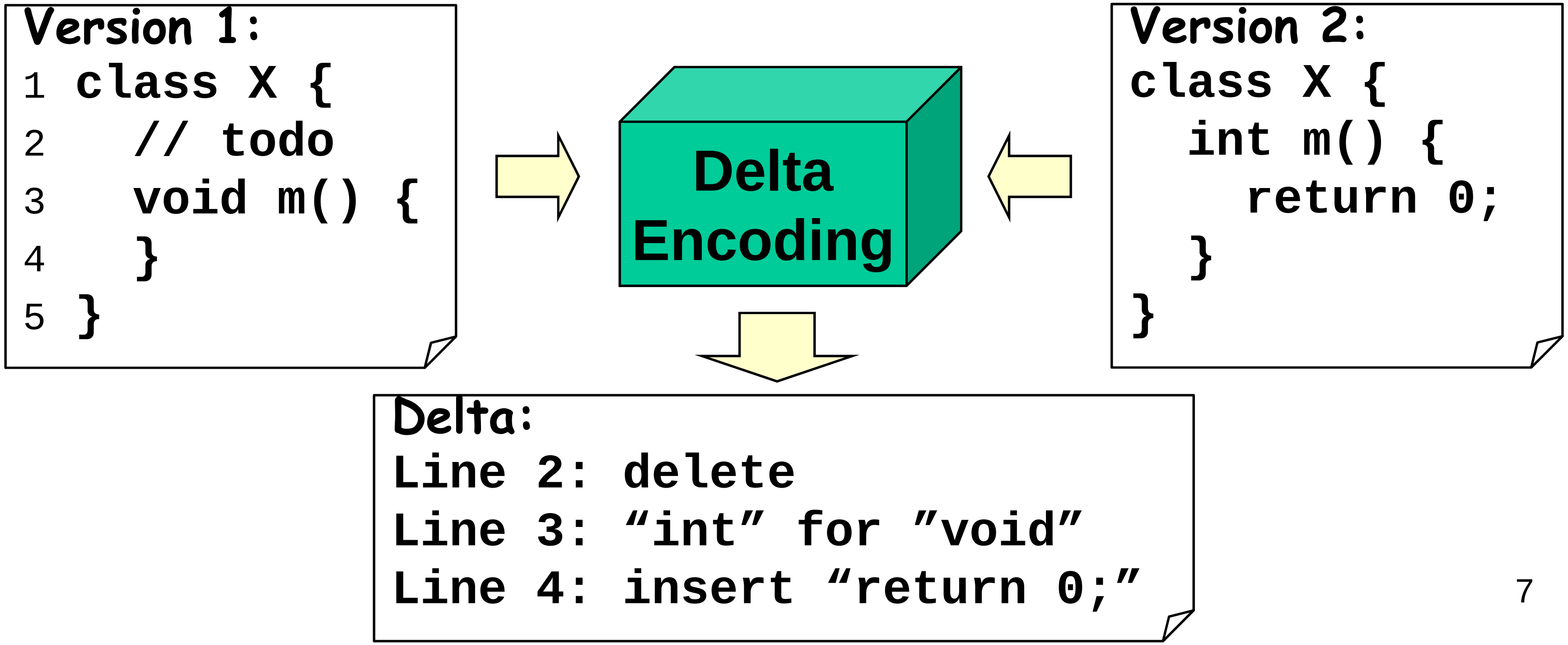
- **Just files**: each file has a version number which is increased when the file is changed (e.g. CVS)
- **Files and folders**: the whole file-folder structure has a single version number which is increased for any change done to any file/folder (e.g. SVN)
- **Other data models**, e.g. PD model in PDStore (instances, links)

Version Space: How is it versioned? How are versions organized?

- **Version identifiers**:
  e.g. serial numbers (1, 2, 3, …), dates (e.g. 20060901), …
- **Version history**:
  - How are versions ordered? Parent-version / child-version
  - Versions with several parents? -> Merging
  - Versions with several children? -> Branching

Delta Encoding

- Storing every version of a file takes up a lot space
- Idea: just store differences between versions
- Differences ("deltas" / "diffs") can be calculated automatically with various algorithms
- Deltas can be recorded in a separate file and used to update files (e.g. for "patches")



Version 1:

```
1 class X {
2   // todo
3   void m() {
4   }
5 }
```

Delta Encoding

Version 2:

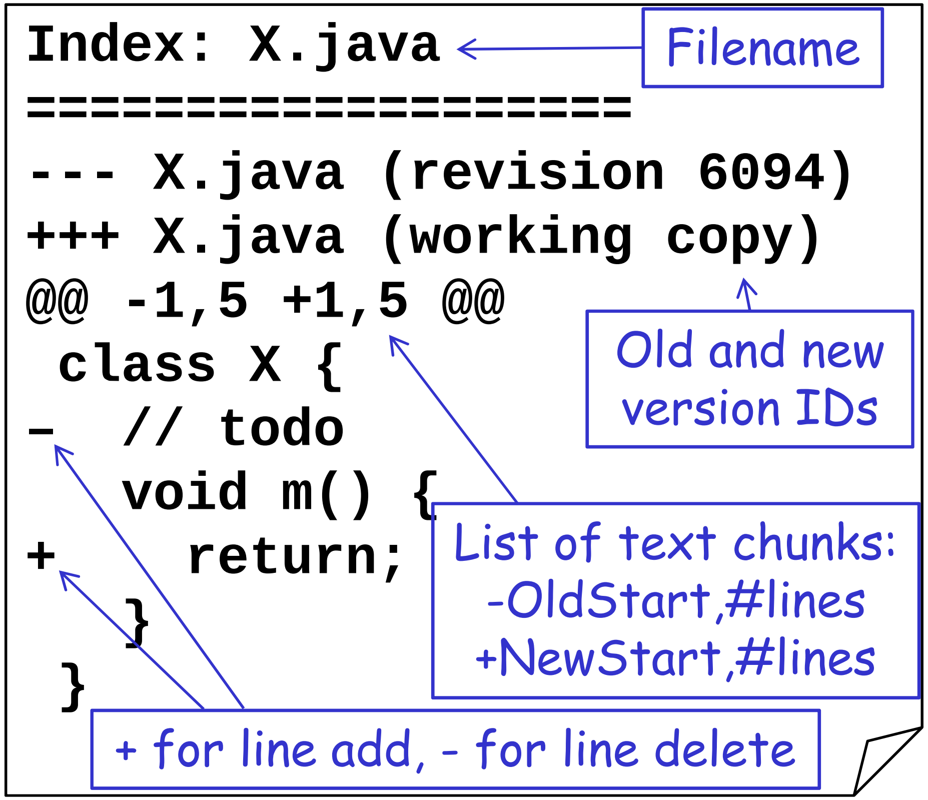
```
class X {
  int m() {
    return 0;
  }
}
```

Delta:

```
Line 2: delete
Line 3: "int" for "void"
Line 4: insert "return 0;"
```

The Unified Diff Format

- Standard format for exchanging patches understood by most VCS
- Example:
  remove comment and insert "return;"
- **Line-oriented**:
  only full line insertions and full line deletions
- No line parts or moving
- Some leading and trailing lines for each chunk for "fuzzy" patching (applying patch to version where it does not fit exactly)



```
Index: X.java
===================
--- X.java (revision 6094)
+++ X.java (working copy)
@@ -1,5 +1,5 @@
 class X {
-  // todo
   void m() {
+    return;
   }
 }
```

Filename

Old and new version IDs

List of text chunks:
-OldStart,#lines
+NewStart,#lines

+ for line add, - for line delete

Branches & Tags

Branches: different copies of a project which are developed simultaneously; “self-maintained lines of development” (**`/branches`**)

- One main branch (**`/trunk`**)
- Maintenance branches: used for maintaining old versions which are still widely used (e.g. commercial OS)
- Experimental branches: used for trying out new features before merging them into the trunk
- Personal developer branches: for people trying out their own ideas

Tags: particular marked versions of the project (**`/tags`**)

- Can be used to refer to and recreate an old version
- Actually also like a copy of the project at a particluar point in time
- Difference to branches: usually not changed any more

Version Control Best Practices

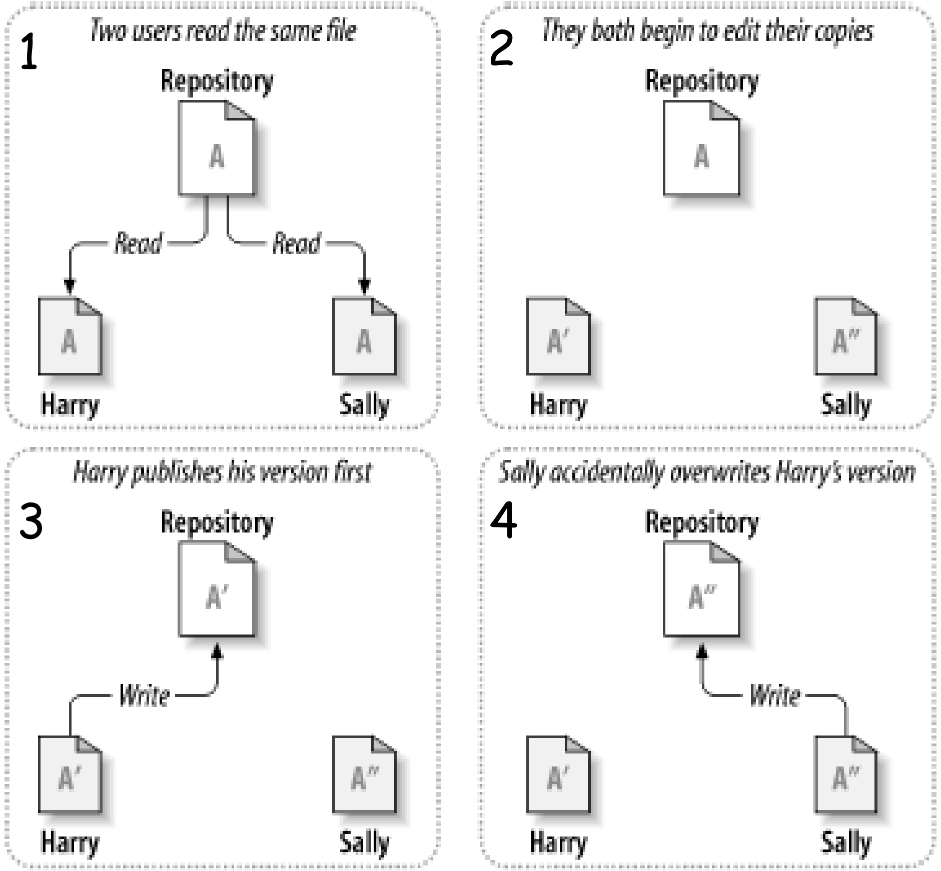
1. Complete **one change** at a time and commit it
   - If you committing several changes together you cannot undo/redo them individually
   - If you don't commit and your hard disk crashes…
   - Continuous integration (see XP)
2. Only commit changes that preserve system **integrity**
   - No "breaking changes" that make compilation or tests fail
3. Commit only **source files** (e.g. not **`.class`** files)
4. Write a **log entry** for each change
   - What has been changed and why
5. **Communicate** with the other developers
   - See who else is working on a part before changing it
   - Discuss and agree on a design
   - Follow the project guidelines & specifications

Software Engineering
The University of Auckland

Managing Concurrency

Concurrent File Access: "Lost Update" Problem

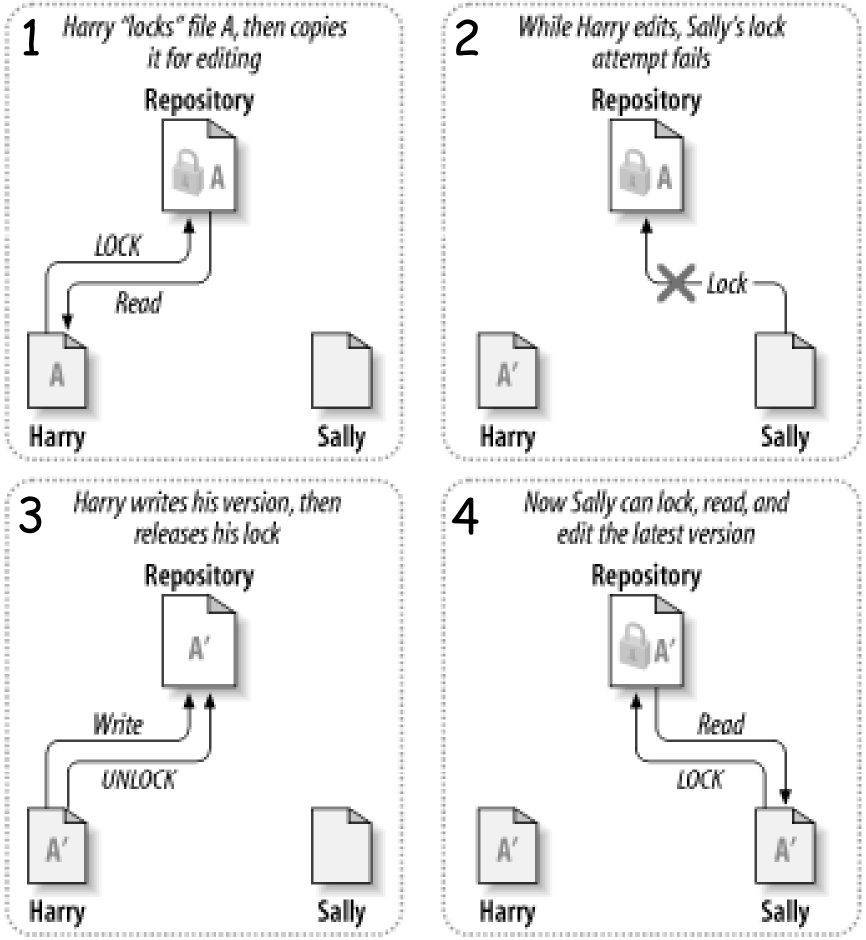
- When sharing files developers can accidentally **overwrite** each others changes
- Consider two developers working on the same file
- Two approaches for solving this:
  - Reserved checkouts ("locking")
  - Unreserverd checkouts ("merging")
- Many old version control systems support only locking (e.g. RCS, SCCS)
- Newer systems offer merging
- Both approaches have disadvantages


1
Two users read the same file
Repository
A
Read
Read
A
Harry
A
Sally
2
They both begin to edit their copies
Repository
A
A′
Harry
A″
Sally
3
Harry publishes his version first
Repository
A′
Write
A′
Harry
A″
Sally
4
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
Harry
A″
Sally

Images taken from the SVN Book (see resources page)

Locking (Reserved Checkouts)

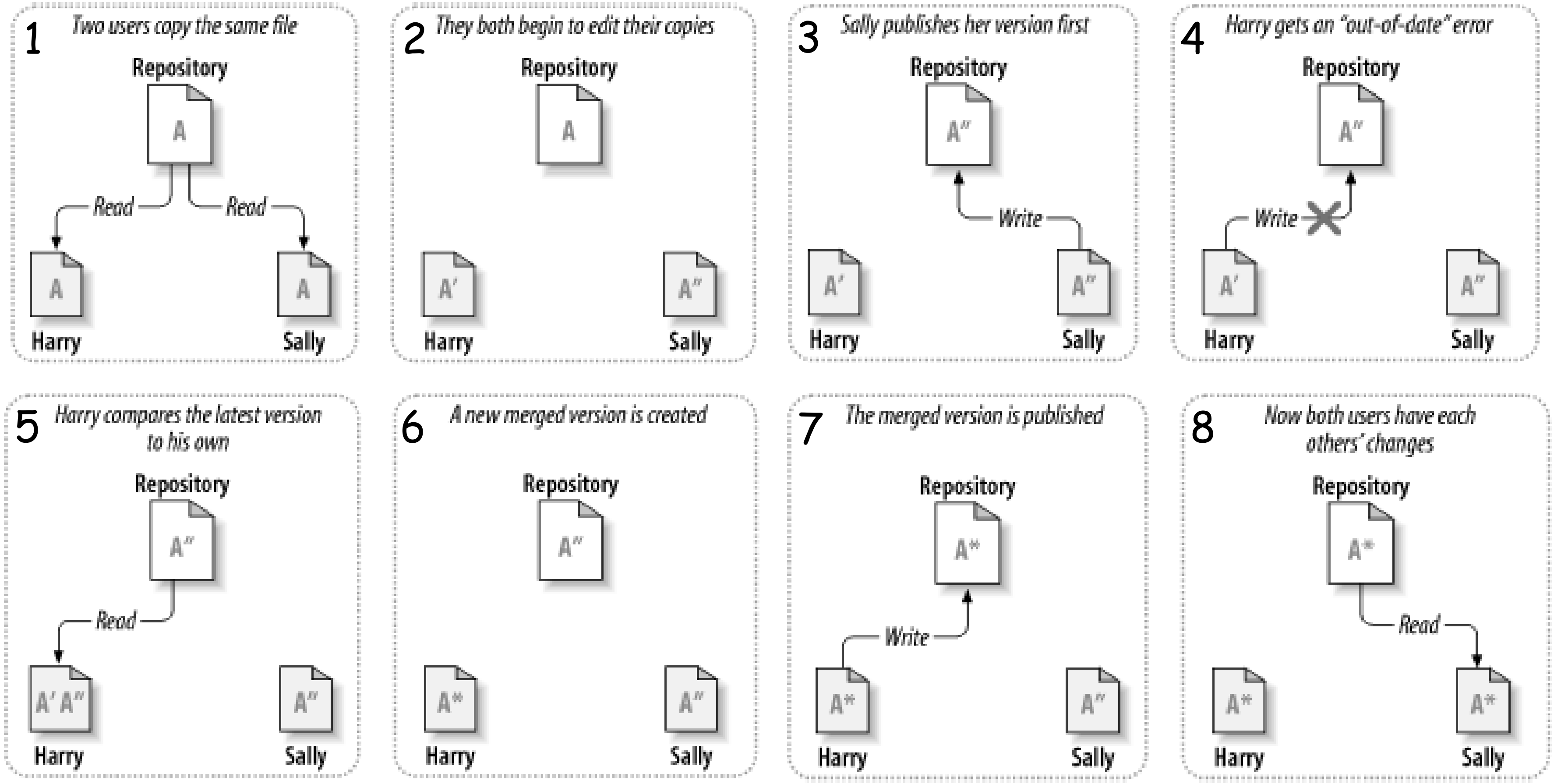
- Only one person can edit a file at a time
- Before getting write access developer has to **acquire the lock** of the file
- Attempts to get lock while someone else has it fail
- Sally has to wait for Harry to release the lock
- Access to files is **serialized**
- Workflow: lock-modify-unlock


1
Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
2
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A'
Harry
Sally
3
Harry writes his version, then releases his lock
Repository
A'
Write
UNLOCK
A'
Harry
Sally
4
Now Sally can lock, read, and edit the latest version
Repository
A'
Read
LOCK
A'
Harry
A'
Sally

Merging (Unreserved Checkouts)

SE Software Engineering
The University of Auckland

- Everybody can modify their working copy whenever they want
- But own changes have to be merged with changes of others before they can be written to repository (copy-modify-merge)



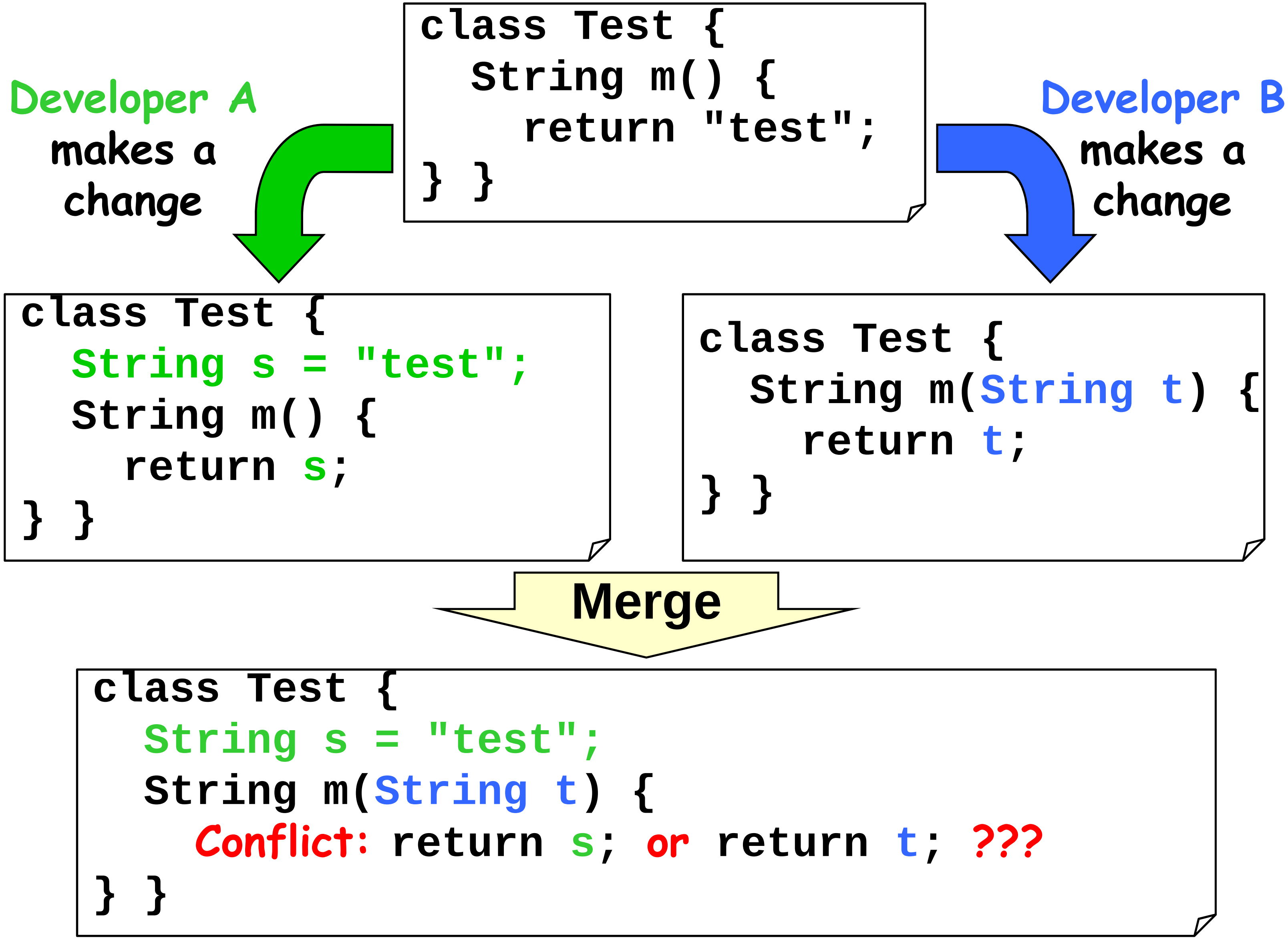
1 Two users copy the same file
Repository
A
Read
Read
A
A
Harry
Sally
2 They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
3 Sally publishes her version first
Repository
A″
Write
A′
A″
Harry
Sally
4 Harry gets an "out-of-date" error
Repository
A″
Write
A′
A″
Harry
Sally
5 Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
6 A new merged version is created
Repository
A″
A*
A″
Harry
Sally
7 The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
8 Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

21/09/2012

SOFTENG 254

The University of Auckland | New Zealand

Merging Example



```
class Test {
  String m() {
    return "test";
} }
```

Developer A makes a change

Developer B makes a change

```
class Test {
  String s = "test";
  String m() {
    return s;
} }
```

```
class Test {
  String m(String t) {
    return t;
} }
```

Merge

```
class Test {
  String s = "test";
  String m(String t) {
    Conflict: return s; or return t; ???
} }
```

Merging: Textual and Semantic Conflicts

- Textual conflicts
  - Changes of different developers are very close or **overlapping** each other (“overlap”)
  - Merge tool cannot automatically combine them
  - Merge tool **detects** such conflicts & reports them to the user
  - Version control system will refuse to write a file with unresolved textual conflicts to the repository
- Semantic conflicts (logical conflicts)
  - Changes are semantically incompatible, but may not be overlapping (e.g. in different files)
  - E.g. developer A changes method signature of method **m**, developer B inserts method calls to **m** using the old signature
  - Non-overlapping semantic conflicts are **not detected** by a generic merge algorithm!!!
  - Can be avoided by following specifications and communicating with others
- Both textual and semantic conflicts have to be resolved by the user

Locking vs. Merging

Arguments against locking and for merging

1. Administrative problems: people forget releasing their locks; frequently administrators have to do it
2. Unnecessary serialization: very counter-productive
   - Locking prevents people from editing different parts of the same file
   - In reality conflicts occur rarely and can be resolved without problems
   - Conflicts usually indicate lack of communication
     - Developers have not agreed on a proper design
     - With mutual agreement on design conflicts are usually straightforward to merge
3. False sense of security: locking does not prevent semantic conflicts of distributed changes (i.e. in different files)

Locking vs. Merging

Arguments for locking and against merging

1. "Unmergeable" files: a generic merging tool does not work for all file types
   - For some formats (e.g. for Java class files) generic merging leads to many conflicts
   - Conflicts can be very hard to resolve (e.g. for binary formats)
   - One of two conflicting changes get lost (because they cannot be merged)
2. Tradition: an organization might have always used a locking VCS

Today's Summary

- A **Version Control System** manages the different versions of all artefacts in a project
  - Many local working copies and one shared repository
  - Compressed with delta encoding
- **Prevents lost updates** through locking or merging
  - Supports automatic merging and detects textual conflicts
  - Cannot detect non-textual sematic conflicts
  - Conflicts always have to be resolved manually

Software Engineering
The University of Auckland

Quiz

1. What is delta encoding? Give an example.
2. What is the difference between locking and merging? When should each of it be used?
3. What is a semantic conflict? Why can it be a problem?