VERSION CONTROL USING SUBVERSION

Introduction and Practical Examples

Dipl.-Inform. Oliver Pajonk
Institute of Scientific Computing
Technische Universität Braunschweig

What is a Version?

- **Definition:**
  - A version of a *unit of information* references its *state* (or a *version relevant part* of it) at a specific point in time.

- Possible **units of information** are:
  - File, set of files, piece of software, …
  - Book, law, any piece of written text,…
  - Build instructions for an industrial product (VW Golf I, II, III,…)
  - Many more

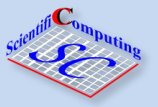
- Common way to **denote a version**:
  - We assign a number to the initial state (often simply “1”) and increment it with each change:

  - Different ways are possible, of course.

What is Version Control?

- **Definition:**
  - Version control is the process of *tracking* and *managing* multiple versions of a unit of information.
- **Purpose:**
  - Information is of paramount importance in the information age.
    - Outdated units of information are still units of information!
    - Changes between versions represent the history of a unit of information → additional information!
  - Version control means having access to this history,…
    - …which allows us to perform changes more confidently (we cannot loose anything!).
    - …which allows us to perform multiple changes in parallel and later combine them.

Reasons for Version Control



- **Single person projects:**
  - File, directory and release version management
  - Change history with log messages (answers “Why did I do that?”)
  - Consistent, repeatable builds as every version can be “resurrected”
  - Synchronization between different development locations
  - Provides confidence for aggressive refactoring
  - Continuous backup for free
- **Team projects:**
  - Parallel development
  - Communication and teamwork
- Not only for projects:
  - Some people use version control for their ***entire*** home directory including configuration files etc. (search for “Keeping Your Life in Subversion”)

Beyond Version Control: Configuration Management

- Configuration Management in SD = Version Control System + Version Management
  - You can do CM without a dedicated VCS, but it is naturally more difficult and error-prone!

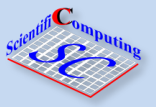
- The first part is presented here in this exercise by the example of „Subversion“
- The second part describes how the **process of versioning** is organized:
  - How is the versioned data organized?
  - Who is allowed to do what with versioned items? Who is responsible?
  - When are „production“ versions released? What are the preconditions? (internal tests, beta-tests,…)
  - How are production versions named?
  - How are bugs/errors handled within the project?
- This is tightly coupled to project management, roles, development model…

➢ Today proper version management in SD relies on version control systems

Introduction to Subversion

Institute of Scientific Computing
Technical University Braunschweig

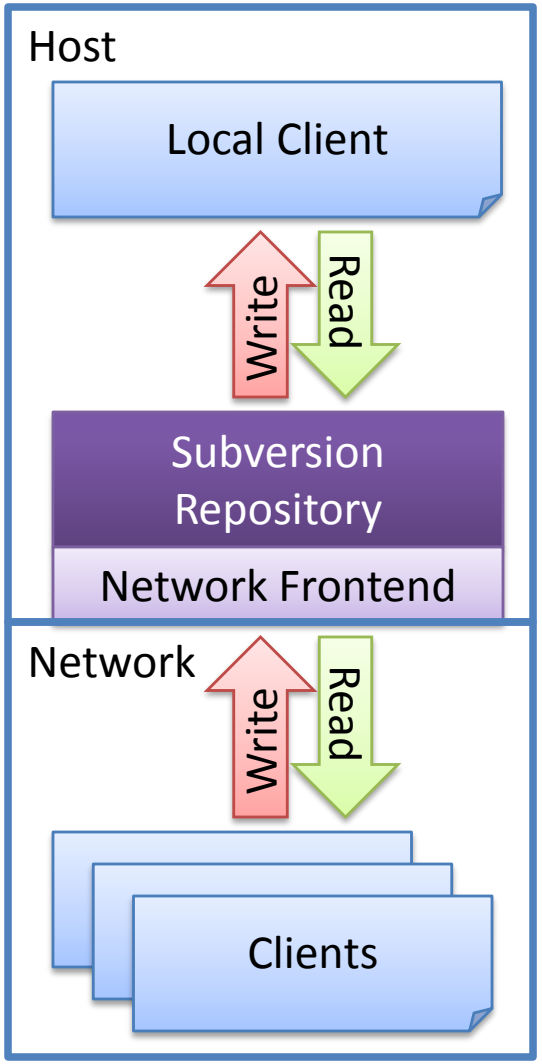
History and Features of Subversion



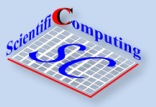
- In early 2000 the Subversion project started, in late 2001 it became self-hosting and version 1.0 was released in early 2004 → four years for first stable version
- Subversion aims to be a compelling replacement for CVS so it has similar features and syntax
- Important features of subversion (taken from http://subversion.tigris.org):
  - Directories, renames, and file meta-data are versioned
  - Commits are *truly atomic*
  - Client/server protocol sends diffs in both directions
  - Costs are proportional to change size, not data size
  - Efficient handling of binary files (uses binary delta algorithm)
  - Versioning of symbolic links
  - Local access, standalone server or Apache server
- Subversion is the sole leader in standalone SCM systems according to Forrester research (2007)

Concepts: The Subversion Repository

- The central store of data
- Essentially works like a normal file server (Samba, NFS)…
  - …but remembers every change ever made to it
  - …but stores metadata for files and directories
- Methods to access the repository:
  - Local file system
  - Network frontend
    - Apache (WebDAV)
    - SVNServe (proprietary protocol)
    - SSH tunnel (like CVS)
- All access methods are concurrently possible!



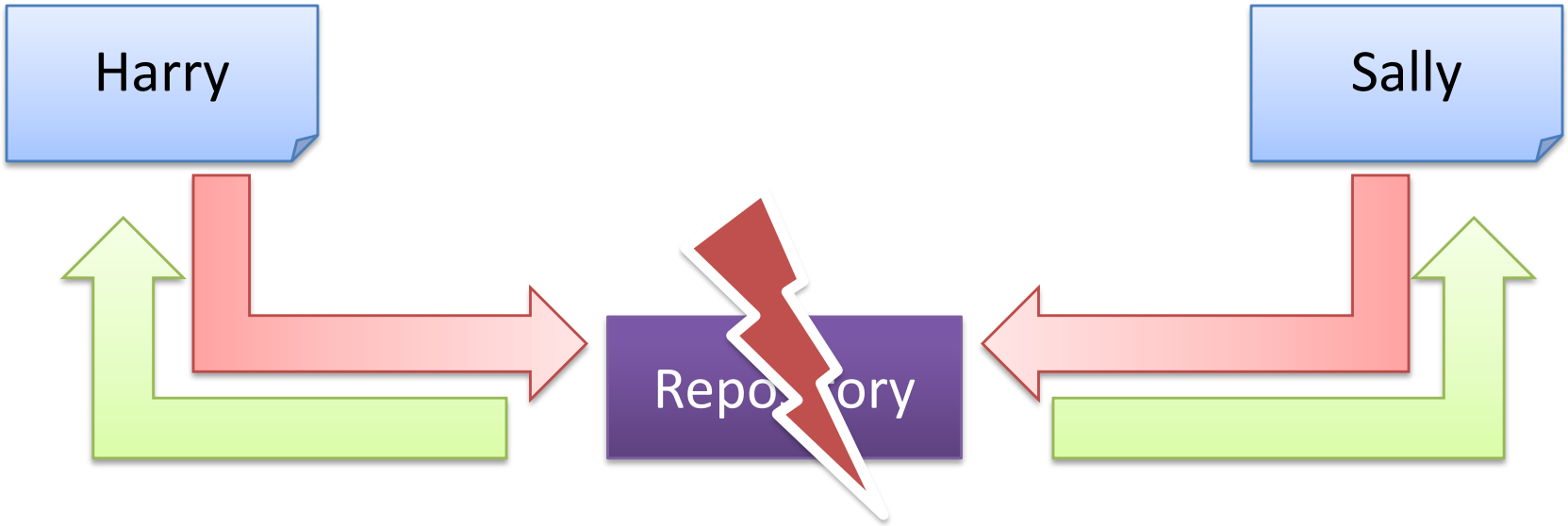
Concepts: The Working Copy



- The working copy is a **normal directory and file structure** that you can work with using your normal development tools
- It contains a **local copy** of the files and directories under version control
- It contains additional ".svn" directories in each directory with metadata for Subversion

- **Important difference:** file system operations like deletion or moving of files and directories have to be performed with the Subversion client!

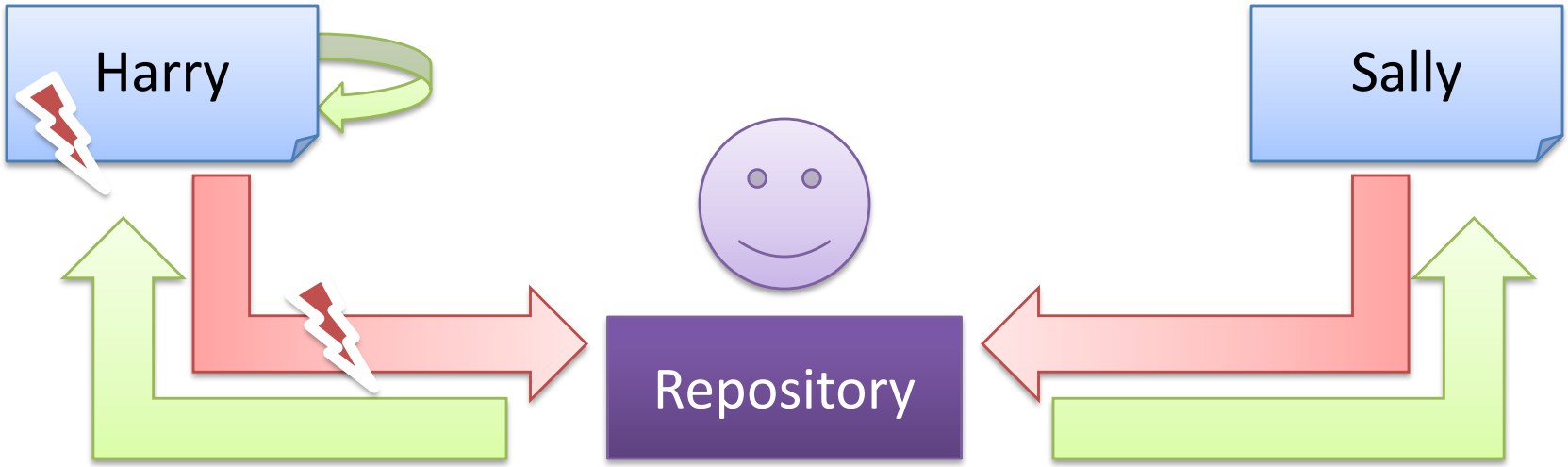
Teamwork: The Problem to Avoid



1. Harry and Sally get their working copy from the repository
2. Both edit ***the same part of the same file***
3. Sally finishes first and writes her changes to the repository
4. Harry accidentally overwrites Sally's changes

Question: How can we avoid this problem
while still allowing parallel changes?

Subversions Solution: The Copy-Modify-Merge Model



1. Harry and Sally get their working copy from the repository
2. Both edit ***the same part of the same file***
3. Sally finishes first and writes her changes to the repository
4. Harry tries to write his changes to the repository but gets an “out of date” error
5. Harry updates his working copy and gets a ***conflict*** for the changes that overlap with the ones from Sally (non-conflicting changes are merged automatically)
6. Harry manually merges the conflicting changes (probably with the help of Sally) and marks the conflict as ***resolved***
7. Harry stores the merged file in the repository
8. Both changes are in the most recent version in repository

Practical Example

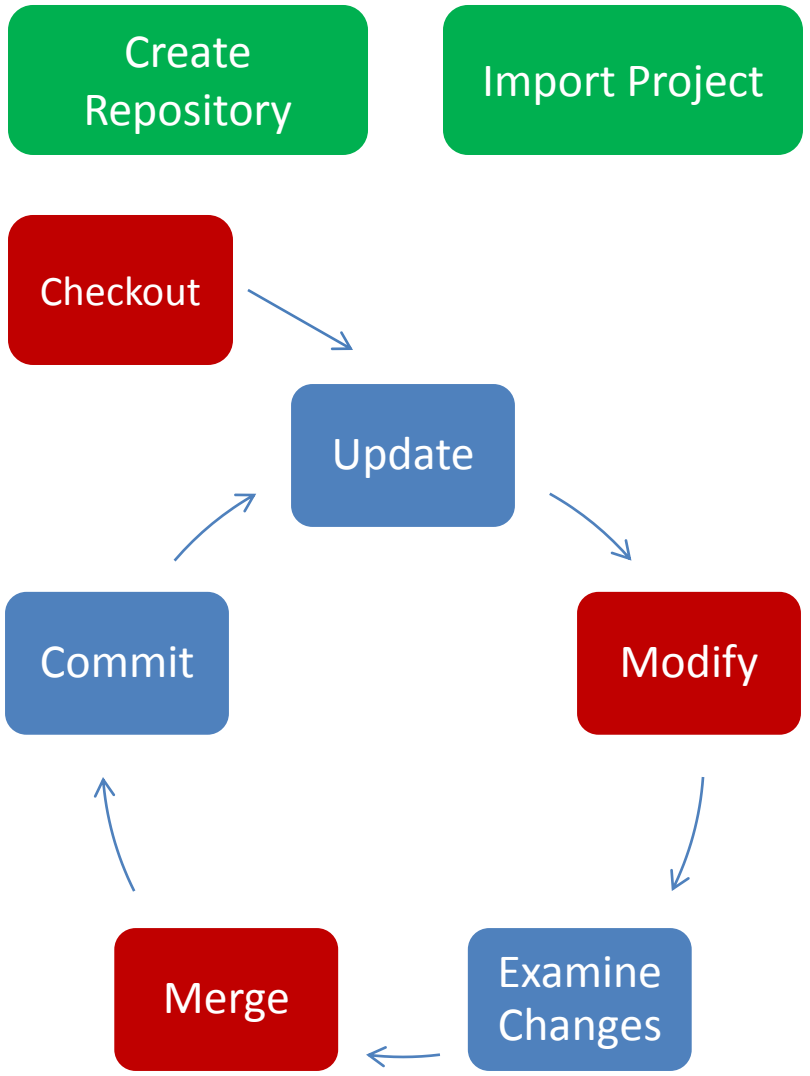
Using Subversion

Institute of Scientific Computing
Technical University Braunschweig

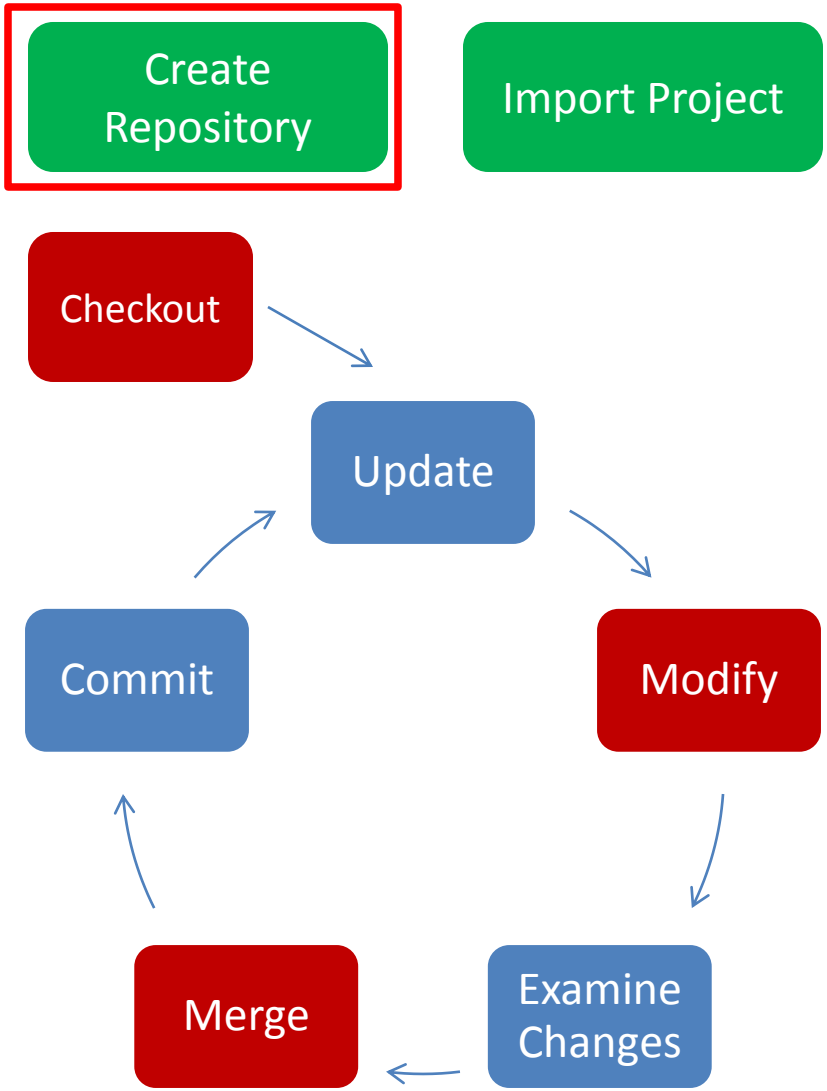
The Basic Work Cycle

1. **Create & Import:** optionally create a repository and import your project
2. **Checkout:** create a *working copy* of the repository (or a part of it) in your development environment
3. **Update:** fetch the latest changes from the repository
4. **Modify:** make your own modifications to the working copy
5. **Examine changes:** show differences to previous versions, revert changes, …
6. **Merge:** update your working copy again and merge changes from other persons (automatically or, in case of a conflicting change, manually)
7. **Commit:** send your changes and results of merges back to the repository

Best practice: one cycle for one logical change *but at least once a day*



The Basic Work Cycle: Overview



Creating a Subversion Repository

- The tool you need for this is called ***„svnadmin“***
- To create a new repository simply type:

```
$~ svnadmin␣create␣$HOME/svn ⏎
```

- Then it makes sense to create the recommended repository layout (now you need ***“svn”***):

```
$~ svn␣mkdir␣file://$HOME/svn/trunk␣-m␣“mkdir” ⏎
$~ svn␣mkdir␣file://$HOME/svn/branches␣-m␣“mkdir” ⏎
$~ svn␣mkdir␣file://$HOME/svn/tags␣-m␣“mkdir” ⏎
```

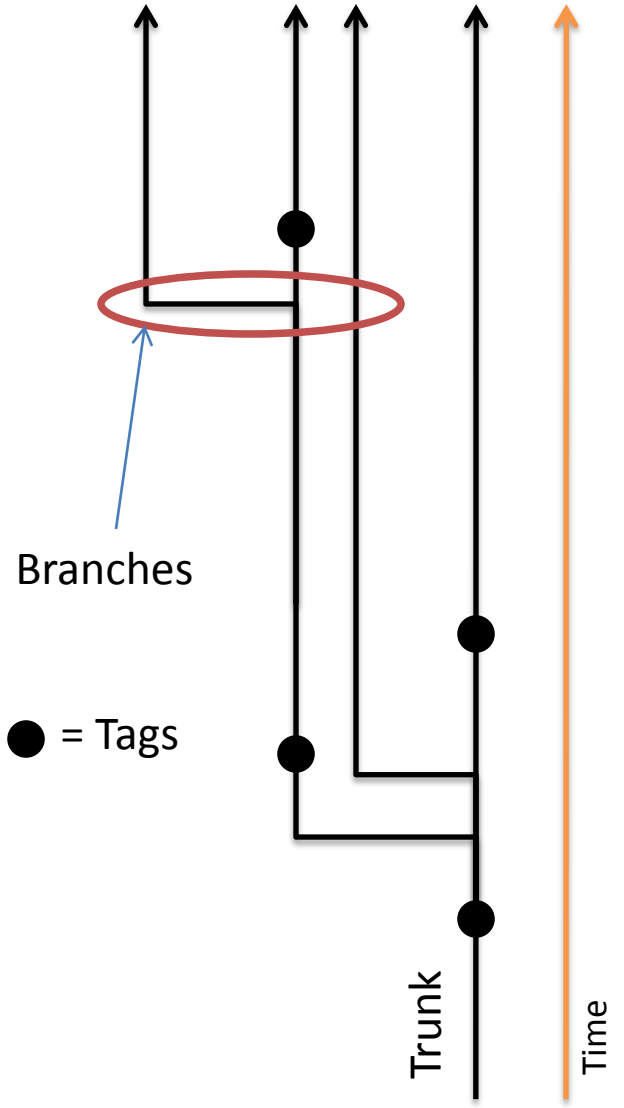
- To see what you’ve just done use:

```
$~ svn␣list␣file://$HOME/svn ⏎
```

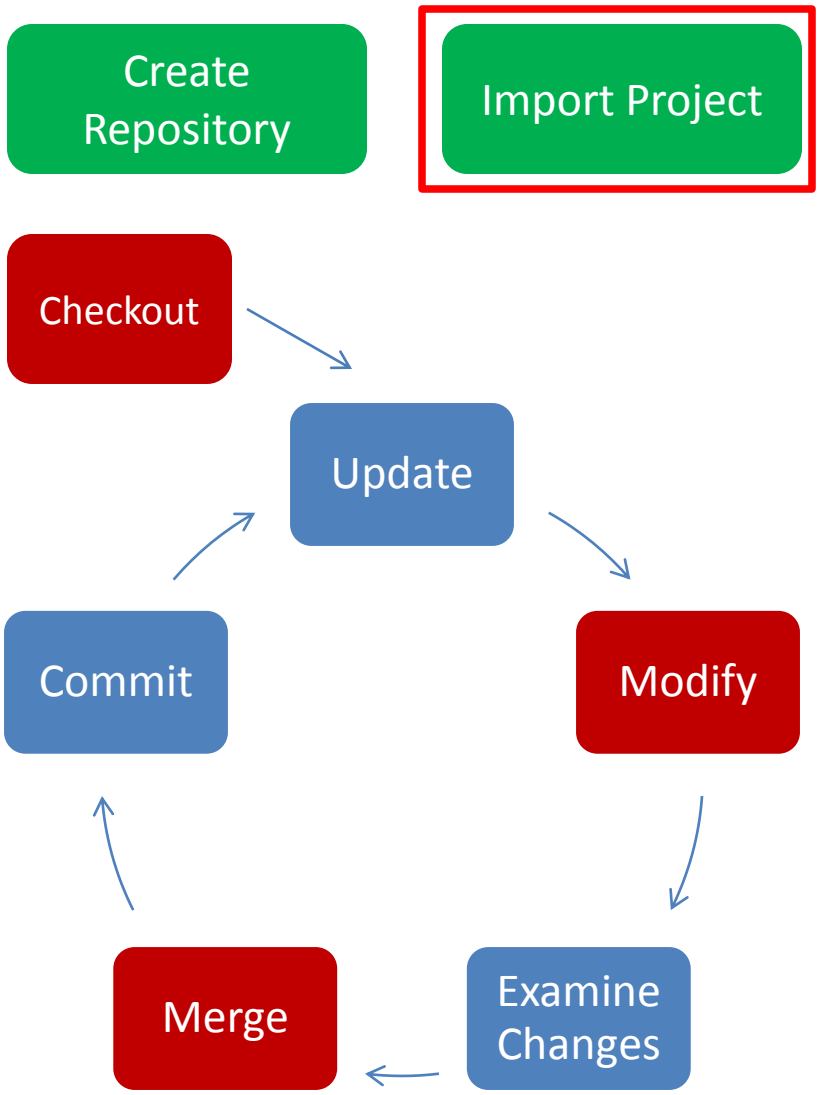
- This will show you the list of files and directories in your repository root just like *“ls”* or *“dir”* does with normal directories

CM Best Practice: Repository Layout

- **Trunk:** main line of development – the most current codebase
- **Branches:**
  - release branches like “1.x.x”, “1.2.x” etc. – for fixing bugs etc.
  - development branches for large changes that should be inserted into the trunk later (feature branches)
- **Tags:**
  - give a name to a single revision
  - are never changed (**never** commit to a tag!)
  - names like “1.1.2”
- One project per repository: put *“branches”*, *“tags”* and *“trunk”* directly in the repository root (as we’ve just done)
- Several projects per repository: put them in each project directory
- These are no laws, only suggestions: tailor them to your project!



The Basic Work Cycle: Overview



Preparation: Creating a Sample Project

- We first need an example project for the next steps
- I have created a small project that we will use for this purpose
- Use the following commands to download and extract the data:

```
$~ cd ↵
$~ wget␣http://www.wire.tu-bs.de/
       mitarbeiter/opajonk/files/sample-project.tgz ↵
$~ tar␣xvfz␣$HOME/sample-project.tgz ↵
```

- The directory *"sample-project"* now contains one file named *"main.cpp"*, which is a typical *"Hello World"* program written in C++

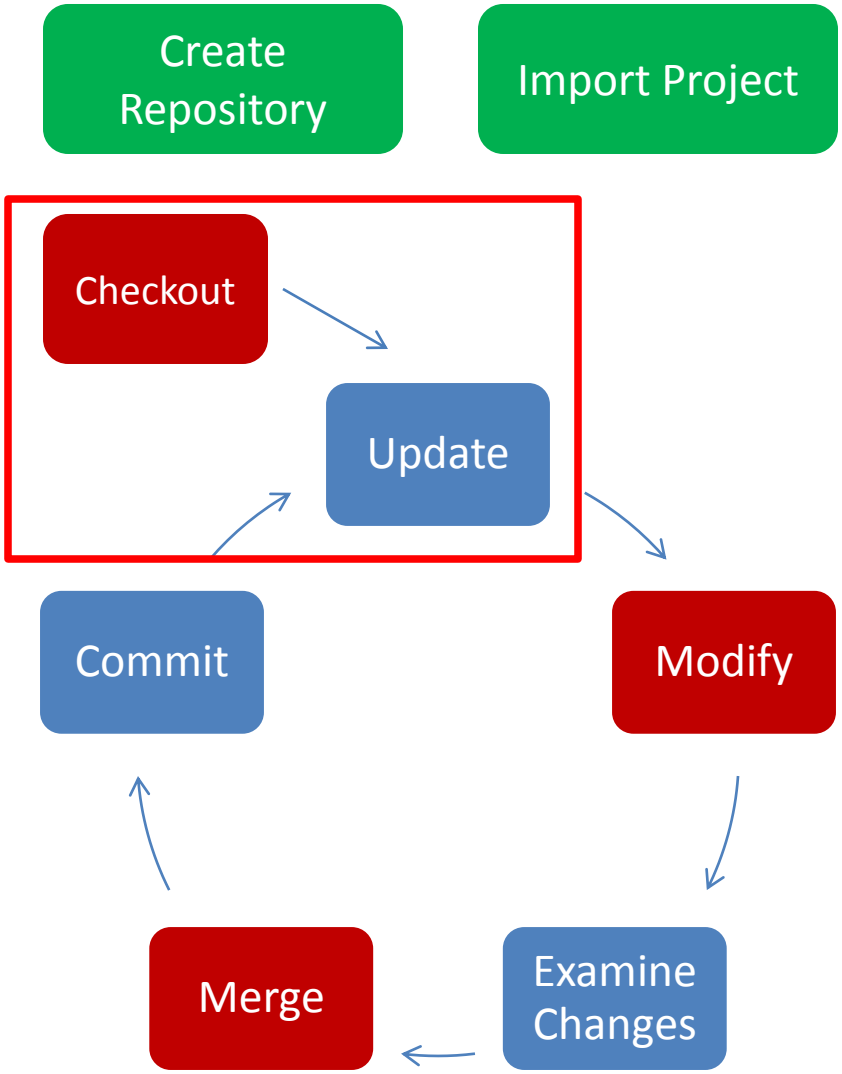
Importing Data Into Your Subversion Repository

- **Important:** Always make sure that you have a **clean copy** of the project you want to import
  - No temporary files
  - No configuration files: store templates for them!
  - No generated files like “*.o” or “*.so”
- Then use the following command to import it into the trunk of your repository:

```
$~ svn␣import␣sample-project␣file://$HOME/svn/trunk␣-m␣
       "initial␣import"↵
```

- Your local copy *“sample-project”* now is not altered at all by the import
- **Important:** Do not mess with the files in $HOME/svn unless you know exactly what you’re doing! The repository itself cannot be modified without using Subversion!

The Basic Work Cycle: Overview



Creating the Working Copy and Updating It

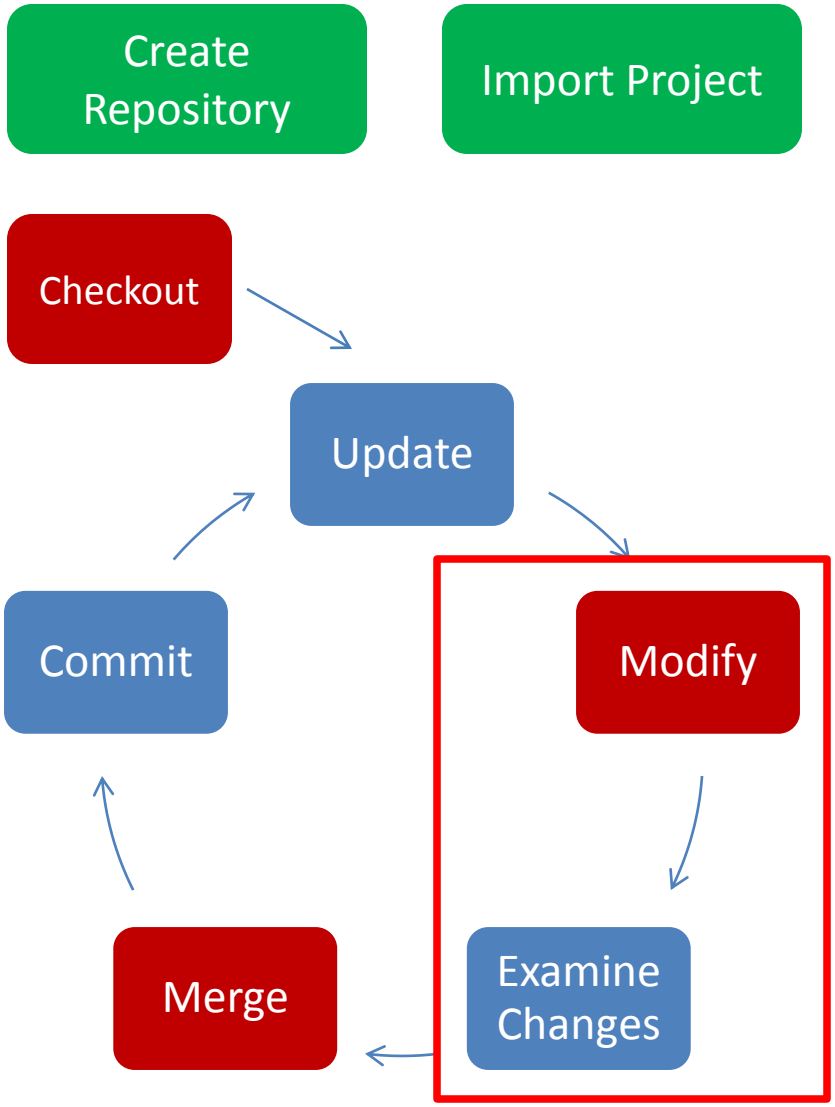
- The process of creating a working copy is called “checkout”
- You have to do this on each computer where you want to work on your project
- We create two working copies now because we need both of them later. Use the following commands:

```
$~ svn checkout file://$HOME/svn/trunk wc1 ↵
$~ svn checkout file://$HOME/svn/trunk wc2 ↵
```

- The directories *“wc1”* and *“wc2”* now contain your working copies of the *“trunk”* of your project
- To update the first working copy use:

```
$~ cd wc1 ↵
$~/wc1 svn update ↵
```

The Basic Work Cycle: Overview



Making Modifications and Examining Them

- **Remember:** file system operations have to be performed using the Subversion client!
- Use *“svn add”*, *“svn move”*, *“svn delete”* or *“svn copy”* and your development tools to make changes to the working copy. Example:

```
$~/wc1 svn copy main.cpp hello.cpp ↵
$~/wc1 kwrite main.cpp ↵
```

- Change “Hello Moon” to “Hello Sun” and save your work
- You can examine your changes with the commands *“svn status”*, *“svn diff”* and *“svn revert”*. Example:

```
$~/wc1 svn status ↵
$~/wc1 svn revert hello.cpp ↵
```

Advanced Topic: Ignoring Unversioned Items

- Often you do not want certain files and directories to be under version control:
  - Temporary files
  - Backup files automatically created by some editors
  - Build artifacts like object files and shared libraries
  - System dependent configuration files
- Subversion supports this: you can set the property **“svn:ignore”** on a directory for example to **“*.so”** to ignore all shared object files
- Multiple patterns are separated by newlines
- Now *“svn status”* and others does not show the garbage output which you do not want to see anyway
- Some will know this concept from CVS

Preparation: Creating Conflicting External Changes

- We will now simulate the creation of some external changes
- Use the following commands:

```
$~/wc1 cd .. ↵
$~ cd wc2 ↵
$~/wc2 kwrite main.cpp ↵
```

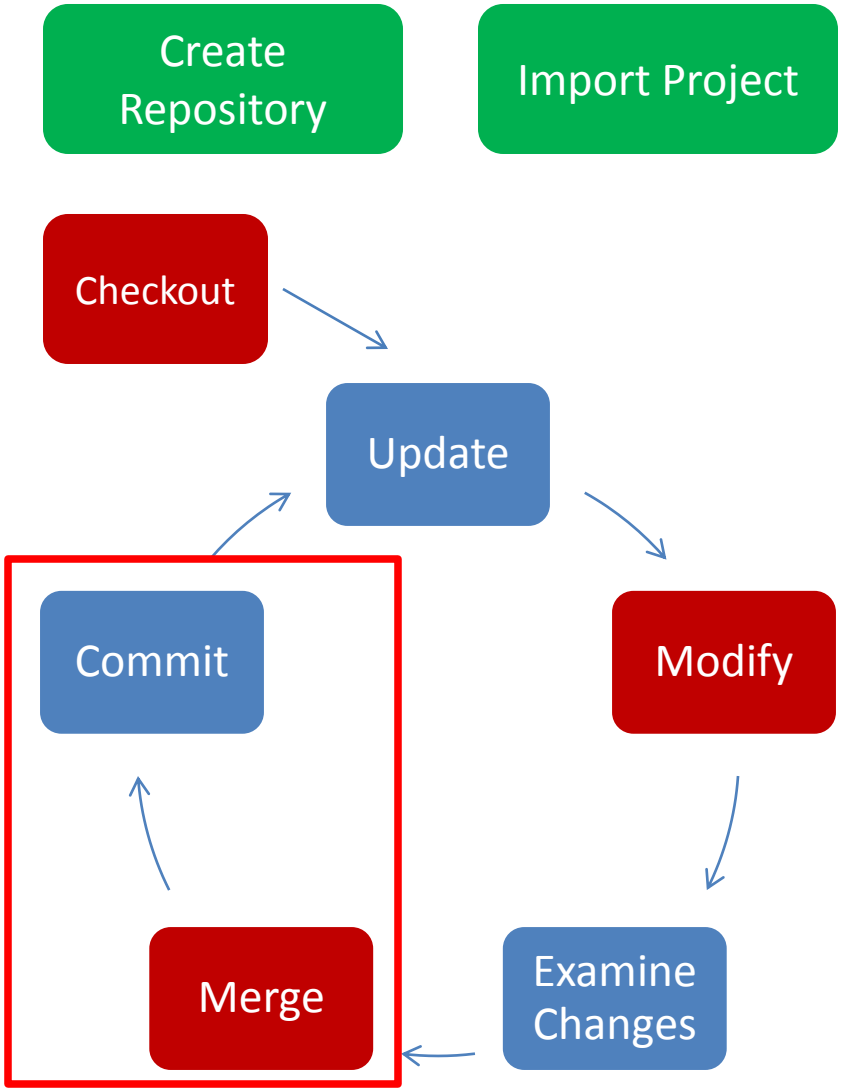
- Change “Hello Moon” to “Hello Mars” and save your changes
- Create another file called “README” and add some content
- Then send your changes to the repository using:

```
$~/wc2 svn add README ↵
$~/wc2 svn commit –m “I live on Mars, added readme” ↵
```

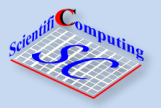
- Now you can go back to the first working copy using:

```
$~wc2 cd .. ↵
$~ cd wc1 ↵
```

The Basic Work Cycle: Overview



Merging Changes from Others (1/3)



- During your own work others may have sent their changes to the repository (and we know they have!)
- You first have to merge these changes into your working copy using *"svn update"*
- Most of the time this can be done by the Subversion client automatically (for non-conflicting changes)
- Sometimes you and some other person have changed the same position in the same file
- Then you have a **conflict** that you have to resolve manually.
- Use the following command and note the small *"C"* next to *"main.cpp"*:

```
$~/wc1 svn update ↵
C main.cpp
A README
```

Merging Changes from Others (2/3)

- For each conflict Subversion creates three extra files:
  - *filename.mine* – your local version
  - *filename.rOLD* – the version before you changed it
  - *filename.rNEW* – the version from the repository
- If line-based merging is possible Subversion places conflict markers in the original file:

```
$~/wc2 kwrite main.cpp
```

```
#include <iostream>
int main(){
<<<<<<< .mine
        std::cout << "Hello Sun!" << std::endl;
=======
        std::cout << "Hello Mars!" << std::endl;
>>>>>>> .r5
        return 0;
}
```

Merging Changes from Others (3/3)

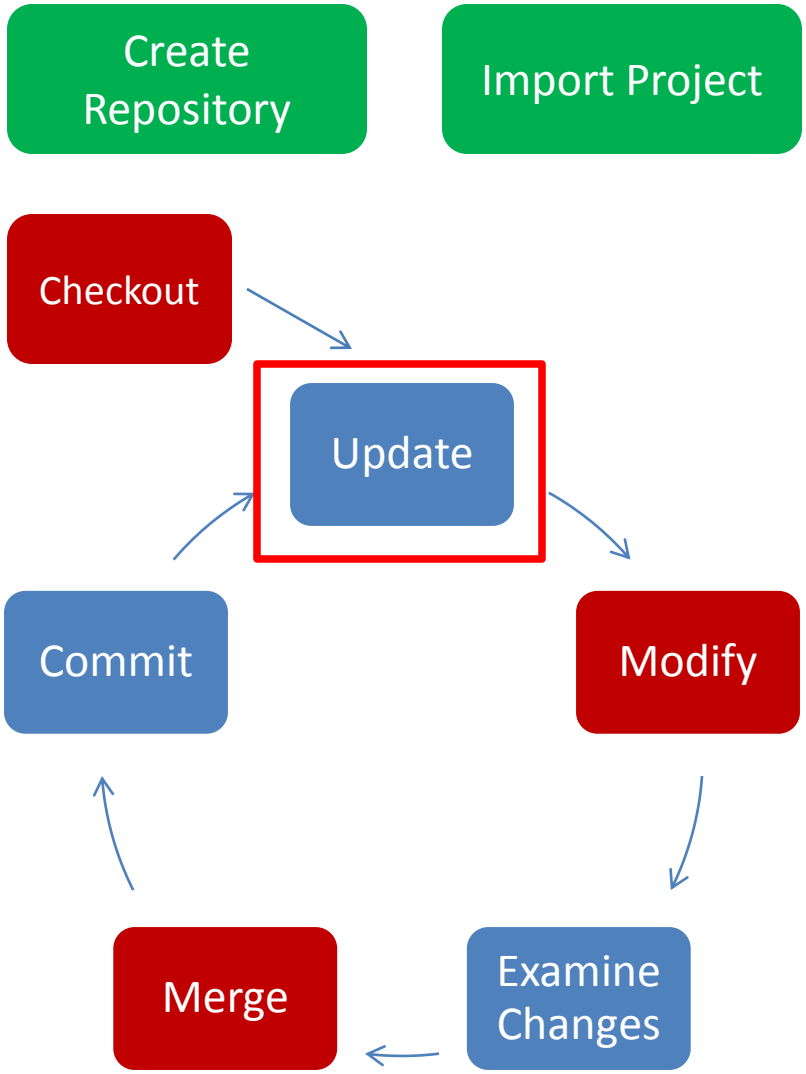
- Now you can handle the conflict by
  - using the revision from the repository
  - using your local version
  - manually merge the changes
- Resolve the conflict using “Hello World”! Then tell Subversion that the conflict has been resolved:

```
$~/wc2 svn␣resolved␣main.cpp
```

- This will delete the temporary files and enable the commit
- Now you are ready to send your changes to the repository:

```
$~/wc2 svn␣commit␣-m␣“I live on earth”
Sending main.cpp
Transmitting file data .
Committed revision 6.
```

The Basic Work Cycle: Overview



Update: Closing the Loop

- Now the process continues with another update before you start making the next changes.

- A short summary of the basic work cycle:
  - Create a repository (if you don't have one)
  - Import your data (if it's not already imported)
  - Create your working copy
  - First **update**, then **modify**, then **merge** and **commit**

- If you stick to these short guidelines you will get the maximum out of version control using Subversion with a minimal effort

- There is a freely available, very good book from the authors of Subversion (the O'Reilly book):

  http://svnbook.red-bean.com/

- Of course there are graphical Subversion clients for a wide variety of operating systems. A small sample:
  - **Tortoise SVN:** Windows Explorer integrated client, probably the best GUI available, open source
  - **Subversive:** Eclipse Team Provider Plugin, open source
  - **SmartSVN:** Java GUI, commercial
  - **RapidSVN:** Cross-platform GUI written in C++, open source
  - **KDESVN:** KDE integrated Subversion client, open source
  - **AnkhSVN**: Visual Studio team provider addin, open source
  - **Versions:** MacOS GUI, currently pre-beta, commercial
  - **SyncroSVN:** Java GUI, commercial
  - And many, many others…

End

Advanced Subversion Topics
- Client Side -

Institute of Scientific Computing
Technical University Braunschweig

Advanced Topic: Versioned Metadata

- You can add metadata to files and directories
- It follows the concept of *property maps*: they map from a human readable name to any binary data
- Metadata is versioned just like files and directories
- **Example:**

```
$~/wc1 svn propedit copyright main.cpp
```

- This command launches your favorite editor and allows you to directly edit the property called *"copyright"*
- There are special properties which start with **"svn:"**. The most important ones are:
  - **"svn:log"** contains the commit log message
  - **"svn:ignore"** stores which patterns are ignored by the client
  - **"svn:keywords"** makes the client replace certain keywords within a file with version information (similar to CVS)
  - **"svn:externals"** allows you to create "soft links" between repositories

Advanced Topic: Keyword Substitution

- Sometimes it is helpful to have certain information on the version of a file directly available inside it
- Subversion supports the **“svn:keywords”** property which enables keyword substitution similar to CVS:
  - The keywords are called: **Date**, **Revision**, **Author**, **HeadURL** and **Id**
  - Place them into your text like *“Id”* and set the property of the file to *“Id”*
  - Subversion will expand the keyword in your working copy then, for example to

    $Id: main.cpp 1234 2007-08-03 22:31:41Z opajonk $
  - The unexpanded keyword is versioned and thus stored in the repository, NOT the expanded one

Advanced Topic: Externals Definitions

- A problem you may know: you want to use some versioned files across several projects, but only want to version them once
  - BibTeX databases
  - Common LaTeX files
  - Libraries you have written

- Subversion supports this with so-called *externals*:
  - Set the property **"svn:externals"** on a parent directory in the following way:

```
dir-name
https://www.example.com/svn/project
```

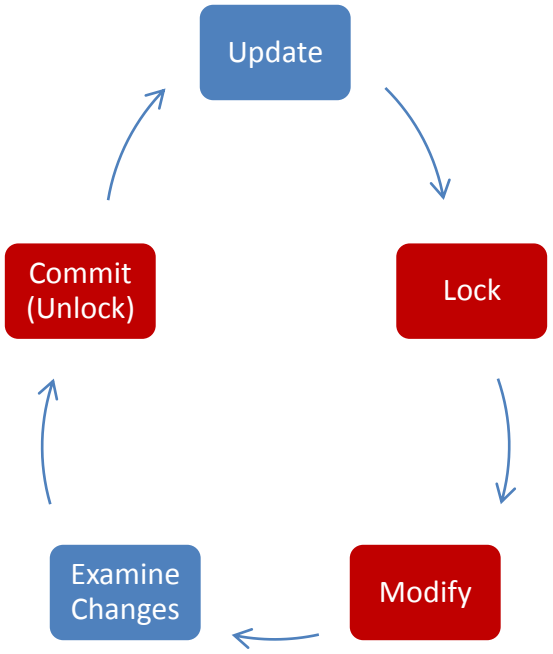
  - At the next update a directory called **`dir-name`** will appear which is a working copy of the project
  - New working copies of your project automatically contain the external working copy
  - All operations on the parent working copy are also performed on the child working copies (if you have the right to do so!)

Advanced Topic: Lock-Modify-Unlock Model Support in Subversion (1/2)

- Subversion also supports the ***"lock-modify-unlock"-model*** of version control (or **"reserved checkout"**)

```
$~/wc1 svn␣lock␣main.cpp␣-m␣"refactoring"↵
```

- The file is now locked in the repository and can only be modified using the present working copy
- After the next commit the lock is gone and anyone can modify the file again
- Locks are useful for
  - Binary files which cannot be automatically merged
  - Files that will undergo heavy modifications



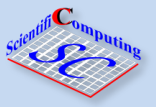
Advanced Topic: Lock-Modify-Unlock Model Support in Subversion (2/2)

- You can *encourage* that a file can only be modified if one holds a lock:
  - Set the **"svn:needs-lock"** property on a file (its value is irrelevant)
  - Subversion clients will make the file read-only in any working-copy unless you hold a lock for it
- To help team communication use lock comments describing things like
  - Why did you take a lock
  - How long you are planning to hold it

Advanced Subversion Topics: Server Side

Institute of Scientific Computing
Technical University Braunschweig

Advanced Topic: Hook Scripts



- Subversion repositories allow the administrator to place scripts that are executed on several occasions:
  - Before a commit takes
  - When a commit starts
  - After a commit has taken place
  - Before / after taking a lock
  - Before / after releasing a lock
  - Before / after doing a revision property change
- These are normal shell scripts or batch files
- Example usage:
  - Sending e-mails after a commit
  - Enforcing log messages with a special format

Advanced Topic: CVS to SVN

- The huge amount of advantages that Subversion has over CVS imply that one should completely move to Subversion
- Fortunately: there is a tool to convert existing CVS repositories to Subversion ones: **CVS2SVN**
- It is a Python script for one-time conversions, *not* for repeated synchronizations!
- It is able to convert braches and tags
- See http://cvs2svn.tigris.org/ for further information

End

Subversion Setup at the Institute of Scientific Computing

Institute of Scientific Computing
Technical University Braunschweig

The Subversion Setup at the Institute of Scientific Computing (1/2)

- A fully featured **Subversion server** is available at

  https://valkyrie.sc.cs.tu-bs.de/svn/{repository}

  - Replace {repository} with the repository name, for example *ctl*
  - As it uses the WebDAV protocol it works across almost any firewall and any proxy server
- We have an **administration frontend** at

  https://valkyrie.sc.cs.tu-bs.de/svnmanager

  - You can create, modify and delete repositories yourself (once you have an account!)
  - You can add and remove access rights to the repositories owned by you

The Subversion Setup at the Institute of Scientific Computing (2/2)

- If you or any student wants to have access to the Subversion setup contact me, Dominik Jürgens, Martin Krosche or Elmar Zander. They can add new users.
- To save resources I encourage you to create dumps of repositories that are no longer needed and delete them from the server (→Webfrontend)
- If they are needed online again the dump can be reloaded at any time (→Webfrontend)

Thank you for your patience!

Any questions?