# *Version Control - HW2*

***Variants and bug fixes***

*Sometimes, code must be changed slightly when porting a program. If the earlier version of the program is to be maintained, both version are described as variants. When correcting errors in a variant, all further variants must also be considered. Can this be carried out automatically? (3 marks)*

Answer:

Automatically correcting all variants is not as easy as it seems. As defined in the problem statement, a variant is basically different versions of the same program that need to be maintained. However this does not imply that the code changes made to the versions would tend to be minimal.

If the changes are minimal, it could probably be carried out automatically but as the differences in the variants increase, the complexity of automatically handling it would also increase significantly. That would make it very impractical to automatically correct errors in the variant. Based on the changes made, different variants might have different issues that need to be resolved and this can't all be handled automatically as it would require keeping track of all the specificity required on a per variant basis.

***Variants in a VCS***

*Your company would like to bring out the system you have run up to now under UNIX in a new WINDOWS variant. Four alternatives are available for organizing the new variant:*
*Set up two separate SVN archives for both variants.*
*Set up a common SVN archive, where a branch is used for the management of the WINDOWS variant.*
*Merge both variants into a common file, controlled by SVN. One of these two variants is selected at compilation time via the C preprocessor (CPP) by "#ifdef … #endif".*
*Encapsulate all system-relevant parts into SVN-controlled subsystems selected for UNIX or WINDOWS when creating the program.*
*Consider the advantages and disadvantages of these approaches. To what extent can the schemes be extended by further variants? For example, think about the several UNIX variations or a variant for .NET as a further variant of WINDOWS. (10 marks)*

Answer:

- Setup two separate SVN archives for both variants:

From the looks of it, this looks like a good idea, because we would end up having different variants for individual environments, but it really is not. If we want to maintain a system which will run on both UNIX and WINDOWS, then the functionality they provide must be kept common too, albeit with the required changes to handle the different setup. If we setup separate SVN archives, then we end up splitting up the code base into specific versions attached to the environments.
This setup basically would increase the work because now we would have to ensure that the individual variants are kept in sync. The changes required to connect to different environments would probably not involve rewriting the entire code base. It might just entail adding some portion of environment specific code and this does not really justify setting up two separate SVN archives. In my opinion, this approach is appropriate only for scenarios where we have totally different code base with a huge amount of differences between them that exist currently or might be introduced later.

- Set up a common SVN archive, where a branch is used for the management of the WINDOWS variant:

In this approach, we at least negate the biggest disadvantage of the first approach, where we would have to totally split up the code into two environment specific SVN archives.  However in this approach we would have to maintain different branches and ensure that they are all patched correctly and kept in sync. Also there is the problem with merging the code at some point which is not really going to be that easy in case of multiple branches

In case we need .NET variant of the WINDOWS variant, then this would require another branch which means that we have extra branches to keep in sync. This approach would be convenient and appropriate if there was a predetermined value of the maximum number of branches that will be created and if that count is not really high

- Merge both variants into a common file, controlled by SVN. One of these two variants is selected at compilation time via the C preprocessor (CPP) by ”#ifdef … #endif”:

This approach sounds good too because we don’t even have to split up the code into different SVN archives, but when looked at from a maintainability perspective this also has major problems. One of the things is that we have to ensure that the ”#ifdef … #endif” commands are used correctly and tested out thoroughly with different environments. Also if a .NET Variant is needed, then this would imply that another set of ”#ifdef … #endif” would be needed to handle that scenario, ergo, the more number of variants we add, the more number of ”#ifdef … #endif” would be needed. It has to be kept in mind that this is probably not going to be a good idea as the size of the project keeps increasing because the impacts of fixes made for one environment would end up affecting other environments too.

- Encapsulate all system-relevant parts into SVN-controlled subsystems selected for UNIX or WINDOWS when creating the program:

Compared to the various shortcomings seen for the other options, this might actually be a best option. Since the system-relevant parts are maintained individually for UNIX or WINDOWS or even a .NET variants of WINDOWS, it is very easy for them to be handled separately to match their individual requirements. This approach also ensures that the common code is shared among all the variants while environment specific code is maintained in individual subsystems and ensure the custom behavior expected.

The company is better off following the approach in the order of preference mentioned above with encapsulation being the best fit for their requirements of supporting variants

***Linus on Git***

*Watch this Google TechTalk by Linus Torvalds on git and describe (in 2-4 sentences each)*
*Three things you learned*
*Three things heard that you agree with*
*Three things you heard disagree with*
*There are some wonderful Google TechTalks on YouTube, and I hope this experience encourages you to watch more such presentations, despite Linus’ somewhat abrasive personality. (3 + 3 + 3 marks)*

Answer:

Three things you learned:

1. Importance of Distribution - After listening to the talk, it became pretty obvious to me that the Distribution model is much more scalable and secure when it comes to versioning than a centralized model is. Also learning how the Distribution setup helps in the Collaboration effort and Release engineering definitely makes Git a good SCM

2. Advantage of using SHA1 Checksums: The fact that Git uses SHA1 checksums to get the strongest hash possible was another great thing I learnt from the talk.

3. Network of Trust: The explanation of this topic was very interesting and something I definitely learnt from. The fact that you pull from your Network of Trust and they pull from theirs makes the premise so much more appealing, since we can always narrow it down and also get more collaboration without hindering anyone.

Three things you agree with:

1. In trying to explain the advantage of Git over CVS, Linus Torvals mentions the fact that the way CVS is structured, people end up waiting to commit because they don't want to unless they are done. This happens all the time. When people finally get ready to commit, the code in the trunk or branch might have moved along and look totally different. The fact that Git allows users to work on their copy of the repository and pull in changes from other users for testing without breaking anything is definitely a really good feature to have.

2. Linus Torvals' also explained the CVS Commit Access problem wherein only a subset of people have actually access to committing code to avoid breaking the branch and how Git handles the same scenario. The fact that the design of distributed model ensures that anybody can have access anywhere and commit to their own copies of the repository is something I agree with

3. Lastly, I agree with Linus's statement that creating branches in CVS is pain and since it is visible to everybody anyways, then there is no significant advantage gained. Also the merging process is definitely harder in CVS and the fact that Git handles all these so efficiently using the Distributed model is a great feature to have

Three things you disagree with:

1. I completely disagree with Linus Trovals statement that Git is more easier to use than CVS. Even though Git might become easy once all the concepts associated with it are understood, CVS is still fundamentally much more easier to understand

2. The statement on Distribution said that it helps in Release Engineering by having Concurrent Development/Test/Release cycles. This is something I disagree with. The reason for having those cycles is because its the logical flow of work from Development to release. So even if it offered, I am not sure if it really helps, unless the company is just continuously pushing releases and this is something probably which might be more suited for startup environment rather than a company with a mature model.

3. Also while talking about the performance, he mentioned about doing several merges a day. Again, am not sure if this is a great thing to be doing instead of having a set schedule. A lot of merges going on in a single day might not really be the approach i would follow. So even though I disagree with the approach, I do agree wit the point mentioned that Performance is not secondary

***Equivalence of rm and add to mv***

*Explain why the git repo resulting from copying the following commands:*

```
cp foo /tmp
git rm foo

cp /tmp/foo bar
git add bar
git commit
```

*is identical to the repo results from the following commands*

```
git mv foo bar
git commit
```

Answer:

When the “git rm foo” command is run, it removes the files from the working tree and the index too. Executing “git add bar” adds the contents to the index and prepares for the content to be staged in the next commit. The “index” basically holds a snapshot of the content of the working tree and this is the snapshot that is taken as the contents of the next commit. “git commit’ serves to commit the record of the snapshot into the history

Now when we perform a “git mv” command it updates the index for both the old and new paths automatically. It is a convenience method and basically a shorthand for the commands “git add & git rm”. This is made possible because git uses the mechanism of content-addressable storage wherein the information is stored to be retrieved based on the content rather than the location of storage. When the “git mv” command is run, it’s content remains same except that the name is changed.

*Is the same true if the corresponding commands are run in SVN? (2 marks)*

Answer:

In case of SVN, if we delete “foo” and then add a new file “bar” (which is basically a copy of the deleted file), the commit would end up deleting the original file while committing the new file by creating it afresh. Since SVN is designed to track the files and does not do the hashing mechanism used by GIT, it is not going to make the connection that is made in Git. SVN will keep adding files to a repository even if they are copies of one file unlike Git.

In short, it is not true that the same thing would happen in SVN while performing the operations that were carried out using Git. There are variations in how Git and SVN handle the versioning processes

***Design patterns underlying Git***

*What design pattern does the Git Object store illustrate? Explain your answer. (2 marks)*

Answer:

The Design pattern followed by the Git Object store illustrates the Flyweight Pattern.

In this pattern, a flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects.

In a Git Obect store, the files are stored by its hash rather than the filename and this is referred to as “content addressable storage”. Since the filename is stored in “git” as a “blob” type, if another file with the exact same contents is added to the Git, it notices that the hash is identical and therefore does not create another copy of the file, thereby saving memory.

Thus by doing so, it is pretty clear that Git Object store follows the Flyweight design pattern

***Gitflow vs Forking workflow***

*Read this article comparing workflows based on git and use it to give one advantage of gitflow vs the forking workflow, and vice versa. (2 marks)*

Answer:

Advantage of GitFlow over Forking Workflow:

The GitFlow Workflow defines a strict branching model designed around project release which provides a robust framework for managing larger projects. So basically using a dedicated branch to prepare releases makes it possible to have well-defined phases of development. It allows teams to polish current release while another team continues working on features for next release etc., The GitFlow allows different branches for different defined purposes such as Master, Hotfix, Release, Develop, Feature etc, so this definitely has the huge advantage of keeping things on track for various development activities.

Advantage of Forking Workflow over GitFlow:

In a Forking Workflow, every developer is given a server side repository thereby allowing a user to have two Git Repositories, one private local one and a public server-side one. The advantage of this design is that contributions can be integrated without the need for everybody to push to a single central repository whereas in case of a GitFlow workflow, a Central repository is used as the communication hub for all developers. The Developers work locally and push changes to the central repo. So basically in the Forking workflow, when branches are shared, they get pulled into another developer’s local repository whereas in case of GitFlow Workflow, they get pushed to the official repository. The Forking workflow also has the advantage that it can lead to very thoughtful code merging, watched by a chosen person and pull requests, which lets developers perform code reviews with ease.

***Gitflow Shortcomings***

*What in your opinion would be the single biggest drawback of using gitflow in a startup environment. (Startup environments are characterized by the need to push features very often and very fast.) (2 marks)*

Answer:

Gitflow envisages using a strict branching model primarily designed around the project release and is more geared towards managing larger projects. For example, in Gitflow, there can be different branches meant for Master, Hotfix, Release, Develop, Feature etc., which implicity implies that the expected usage is for large and stable projects.

As pointed out in the question, startup environments need to push features very often so having such a rigid structure would actually work against the very philosophy of being a startup. A startup needs to be very flexible in their release process and the very idea of Gitflow would hold the release cycle for them and for that purpose, I believe Gitflow is a drawback for being used in a Startup environment

***Hooks***

*Version control systems such git and svn expose a number of integration points into its transaction lifecycle. These integration points are called hooks and they correspond to events in the repository such as committing a change, locking and unlocking files, and altering revision properties. When the VCS gets to each point in its transaction life cycle, it will check for and execute the appropriate hook script.*

*Hook scripts have access to the in-flight transaction as it is being processed and are passed different command-line arguments depending on which hook script is executing. For example, the pre-commit hook is told the repository path and the transaction ID for the currently executing commit. If a hook script returns a nonzero exit code, the VCS will abort the transaction and return the script's standard error output as a message to the user.*

*The following hook scripts are used most often:*

*pre-commit Executed before a change is committed to the repository. Often used to check log messages, to format files, and to perform custom security or policy checking.*

*post-commit Executed once the commit has completed. Often used to inform users about a completed commit, for example by sending an email to the team.*

1. *Give two examples of hook scripts that would be appropriate for pre-commit and post-commit. You do not need to write scripts, but be specific, e.g., "format files" is not specific enough. (2 marks)*

Answer:

Examples of Appropriate Pre-Commit Hooks:

i) Pre-commit hooks could be used for checking if a request contains the bug number or a request number before allowing to proceed with the commit. This would thereby ensure that all commits can be traced back to specific bugs or requests.
ii) The other way a pre-commit hook could be used is for sanitizing line breaks since they vary across multiple desktop environments. This way we could ensure that a accepted standard is applied on all the files before allowing to commit

Examples of Appropriate Post-Commit Hooks:

i) One of the examples of a Post-Commit hooks would probably be the sending of email on commit in case a Bug ID is detected in the Commit message. This would be helpful in keeping track of the commits in which particular bug fixes went in
ii) The second example of Post-commit hook can be the scenario where a backup of the repository is taken after each commit. This would ensure that sufficient backups of the repository are taken

*2. Should hook scripts be allowed to alter the content of the transaction? (1 mark)*

Answer:

In my opinion, it is ok for a hook script to alter the content of a transaction as long as it has to do with details like control characters, line breaks etc., thereby ensuring that a consistency is maintained across checkins from any environment.

Other than these and maybe other cosmetic changes, a hook script should not be allowed to alter anything else which is part of that transaction.

*Based on Mason's Pragmatic Guide to SVN: The book's example hook scripts are an excellent resource for learning what each script should do—they are well commented and explain a lot about what's going on when the script is executed. Also take a look at this article on git hooks.*