Versioning Control System for Graphic Design – A layered approach

Master's Thesis
submitted to the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Adrian ISBICEANU

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Dr. Kai Kasugai

Registration date: 22 December 2015
Submission date: 1 August 2016

Eidesstattliche Versicherung

____________________________	____________________________
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

____________________________	____________________________
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

____________________________	____________________________
Ort, Datum	Unterschrift

Contents

List of Figures

List of Tables

Abstract

Revisionary control systems are common in the world of software development; sites like GitHub and BitBucket have become the go-to place for any developer to learn, to explore and to contribute. This is not the case when it comes to design; normally these kinds of files are harder to manage than text files.

We initiated the study with an analysis of designers' completing of routine tasks which led to the observation that, when using Photoshop, they had a strong tendency of organizing their work into groups of layers, and only working on small parts of the file. We have thus proposed a new Version Control solution, which tracks the history changes on a layer based system, instead of using the whole document history.

With this approach we showcase an innovative model of navigating the history. We propose and explain filters that increase the speed and enable finding a desired version. We present a viable option for live collaboration between designers through the use of branches and how layer-based merges can also be accomplished. In conclusion we propose a solution by integrating the traditional 'undo' system into the Version Control System, thus removing the limitations that come with the linear 'undo' System.

Acknowledgements

I would like to thank my supervisor Jan-Peter Krämer for all guidance, feedback and reminding me to stay focused. I have to thank Moritz Wittenhagen for the initial introduction in the field.

I also wish to thank Mark, Adonis, Simi and Cosmin for all their time spent testing the different versions of the tool.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in American English.

Download links are set off in blue color.

File: myFile[a]

[a]http://hci.rwth-aachen.de/public/folder/file_number.file

Chapter 1

Introduction

Revision control systems are common in the world of software development. Sites like " GitHub" and " BitBucket" have become the go-to for any developer. This has paved the way for popular open source projects like node.js and angular for any programmer to use, contribute or test. However, something similar does not exist in the world of design. There are sites like "behance.net" or "dribble.com", however, they are used to showcase a portfolio and not to facilitate learning and collaboration. From our study 3.1 we found that 90.9% of designers save old versions but only 21% use at least a cloud solution to keep these versions in sync. Why is it that designers don't use git to manage their work?

Revision control systems are common in the world of software development but don't have the same use in the design workflow.

Current revision control systems are designed to work on text files, as they are much easier to work with and are more common in the workplace. Line differential algorithms have been developed to compare text files and efficient storage solutions have been proposed. Unfortunately, this was not the case for design tools. Traditionally these tools used a binary format that makes it difficult to produce any universal change tracking algorithm. Binary deltas can be obtained when comparing two files but offer little information compared to the line approach.

Current projects are significantly bigger and no more is this the case than in the advertising and app industries [smartinsights]. In recent years new technologies have emerged, from smartphones to wearables. Users expect that same great experience regardless of platform. This has increased the demand for custom design for each platform, while at the same time keeping the same look and, if possible, the function has become a much harder task. Managing the resulting files and changes has become an increasingly difficult job. How often have we seen the following situation?

Current design projects are much bigger and target multiple platforms.

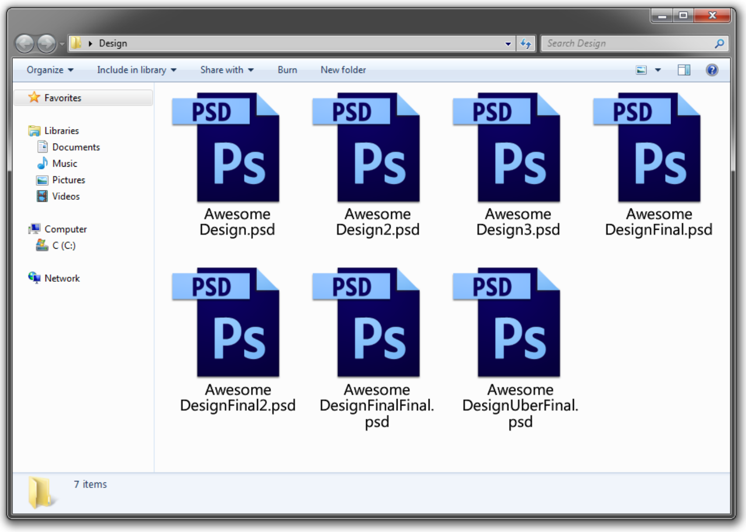


Figure 1.1: Pitfall of using different files to manage the change history.[Thong]

From our interviews we found the lack of a general versioning control system when it comes to the design files, designers and manager having to improvise one of the following solutions to secure their work:

- Create separate layers to keep track of different versions, you can use the same workaround for states if you are designing an application.
- Assign different file names for each version and use a

storage tool like "Dropbox" to save them in the cloud for backup and collaboration.

- Try to use "Git" or any other text versioning control system and save the contents as binary.
- Try to learn/utilize a new design tool that offers some version control.

None of these options result in an easy to follow or useful workflow for designers. Progress into fixing these issues have started from the beginning of 2010. We now have tools like "pixelapse.com", which takes the saves from "Dropbox" and tries to present them in an easier way to work with. While this is a step in the right direction, an integrated tool will make the whole process much easier to follow. Ben Shneiderman has presented this idea in his paper [Shneiderman], future tools need to "Provide rich history-keeping", so users have a record of what they have tried, compare alternatives and find new inspiration.

Following this idea, systems were proposed that integrated the versioning control system in the designing software. They shown the power of having a file history for learning [Fitzmaurice], working [Chang] and collaboration. These systems tried to applied the knowledge from the development versioning control tools to the world of design. They simplified some of the vocabulary and adapted operations to better fit the new environment. We believe future simplification can be applied to lower the learning curve, and increase adoption.

From our analysis we haven't found any tool that uses the fact that users work on a small part of the file and usually organized they work into layers or groups. All the tools have treated the history as a change-set of the whole document. This idea has been tried for building the undo system [W. Keith Edwards, Takeo Igarashi] with great results, but was not applied for the whole versioning control system.

This thesis explores the idea of using layers to track changes in a design document. Layers and groups are the traditional way that designers organize their work so it makes for a natural mapping of grouping changes.

The most prominent example of a revision control for images is the one from [Chang]. This system built on GIMP uses RevG to display the history. While this is a step in the right direction it doesn't offer support for layers, which are critical when doing any mobile application or website work. Their example focuses more on painting applications, and offers a simple to use merging system. It also noted that their system used a manual save approach for users to create versions or branches. All revisions are presented in a liniar faction, the user can click an version to explorer the RevG structure. While an adequate solution for a small number of changes, our study has shown that designers do on average at least five iterations per file in timespan of a few weeks. Such navigation will not handler such work without cutting down on the number of revisions, eliminating one of the advantages of having a complete history. The RevG will help in locating all of the changes, but because of the small number of revisions it will contain a large number of nodes, again making the selective undo process difficult.

Navigating the change history has become a big problem.

Another interesting paper was [W. Keith Edwards, Takeo Igarashi]. The authors explore the concept of a multi-level timeline and how it can be implemented. Their system, "Flatland", had an individual history per item but also offered a global history. Our system implements a similar idea but tracks the individual layers instead of items, with the same challenges as in the paper. For information on the presented systems please check chapter 2.

Proposed versioning control system don't use layers to organize the change history.

Through user interviews and further experimentation, we have concluded that this layers approach provides for a familiar and powerful versioning control solution. An automatic save model and an easy to follow branching system means that designers can experiment without the worry of losing any work. Integrating the undo system into revision control system has eliminated some of the problems associated with a linear undo system, without adding the complexity of a selective undo system.

In the following chapter we will go over what a revision control is and what creative tools need to provide. We continue with the recent research in the field of versioning control systems that assists graphics designers. Chapter 3 presents our user study results from our questionaire, interviews and observations. The following chapter contains our third iteration on our versioning control system, with assumption, implementation and evaluation. We end with a summary of our work and presenting future research topics.

Chapter 2

Background and Related Work

We start by discussing the basics of a versioning control, how it works, how it can be implemented and what is the common vocabulary been used. We follow with what are the requirements of a creative tool, and the need of document history. We examine the different ways document history can be represented and how it affects user interactions.

We continue with the latest research in the world of versioning control for graphics design, exploring tools bring concepts from the development tools like git to the world of design. We finalize by showcasing a new way of thinking about computing, organizing all information around time and giving the user his very own "time machine".

2.1 Fundamentals of a Version Control System

Version Control, also known as source control, is a computer software that is used to record, restore and manage document changes. Version control has been in development since the early days of computing, but the origins can be traced back to the printing industry with the different revisions and editions of books. Initially, developers used to save different versions of the files in different folders. As you can imagine this required great discipline and was prone to error. The very first version control system was developed to save these changes automatically for the user. The program will keep the differences between each version in a database. Each version could be recreated by starting from an empty file and iterating through the database, building the change history step-wise. This was sufficient for simple projects and can still be found in many modern operating systems. For instance, Mac OS X still includes RCS, a popular tool that implemented this idea in its developer package. [Straub]

Version control is not a new idea.

We call revision each new version of a set of files. They are usually identified by a number or a letter code. Example: "revision1" is the initial state of the file. When we perform a change, the resulting files are labeled "revision2". Using a number or a letter code is sufficient for a centralized version control, however, as we will later see, for a distributed version control we need to use a hash of the revision as an identifier, in order to avoid any name collisions. This hash naming can be used in a centralized version control, however, it is not required.

If we try to graphically represent the revisions, we obtain a graph or more precisely a directed acyclic graph. The simplest revision history has a series of changes with no branches or undo. If we draw this history we obtain a direct linear graph, with each revision as a node in a straight line. Arrows connect each revision with the convention that the arrows points from the older version to the newer version. This graph can be also thought about as a tree with the root the oldest version and each node having exactly one child with the exception of the newest version which we call the HEAD node.

While a linear history works for simple tasks sometimes we would need the option to experiment without affecting the history of the working copy. We want to clone the current

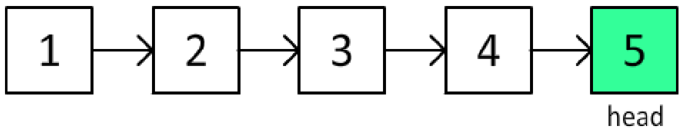


Figure 2.1: Linear Graph History

version, perform our changes and record the linear history of this clone. We want to make a branch.

If we started adding branches to our linear graph it becomes a tree or a directed graph. In this graph a revision can have more than just one child. The HEAD problem arises; how do we decide what revision is the head revision. We may tend to make the head the latest revision, but this will make the head move from one branch to another. In practice, when a new node is created it is either the new head node or the new branch. The resulting path from the root to the HEAD is called the main branch (also known as master branch or trunk).

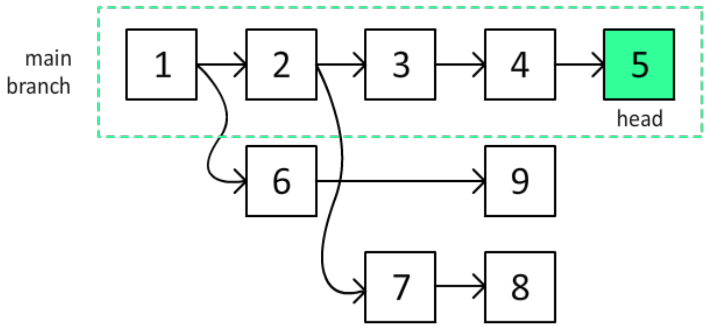


Figure 2.2: Branches in a Version Control System

Branches allow us to experiment and once we are pleased with the results we want to integrate them into our working copy, i.e. “merge” our changes back to the trunk. Adding merge to our tree, results in having revisions with more than one parent. The resulting graph is no longer a tree but a rooted directed acyclic graph. Rooted because there is always an oldest revision, directed as each change is ordered by the time it was created and acyclic as the graph does not contain any cycles. To simplify our work, we can consider the merges as external to our main branch, creating a new version without referencing the initial branch from were that change has come from. The resulting structure is a tree with merge.

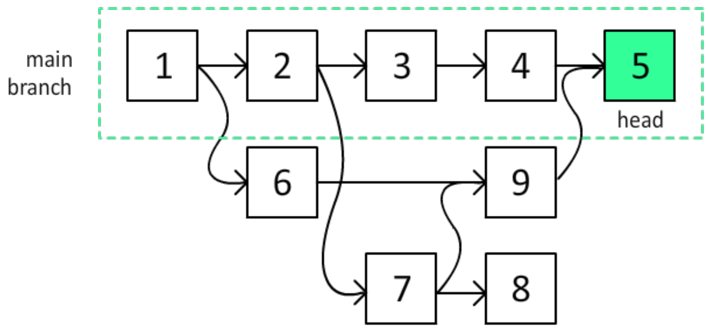


Figure 2.3: Rooted Directed Acyclic Graph

Revision history can be represent by a rooted directed acyclic graph

2.2 Constructing a Version Control System

Let us switch focus and analyze how a version control system can handle changes. We first need to define what an atomic operation is. For an operation to be atomic, the system has to remain in a consistent state even if the operation is interrupted. We want all of our operations that save changes to the system as well as the create revision to be atomic, based on the principle take all or leave all.

When it comes to the architecture of a version control system we have one of the following two patterns: a centralized model or a distributed model.

In a centralized model we have a server that is hosting the repository. The user connects through a client application to this server and requests for changes to be saved and downloads any available changes. When it comes to saving those actual changes the system has 2 options: it can either send over the whole file or just send over the differences that need to be applied to the previous version to obtain the new file. As you can imagine this is easier for a text field where we send over the changed lines but gets significantly difficult if we are talking about binary files. As previously mentioned, all of these operations have to be atomic to protect the system. The centralized model server does not necessary have to be on another machine, it can run locally as in the case of RCS that we previously mentioned.

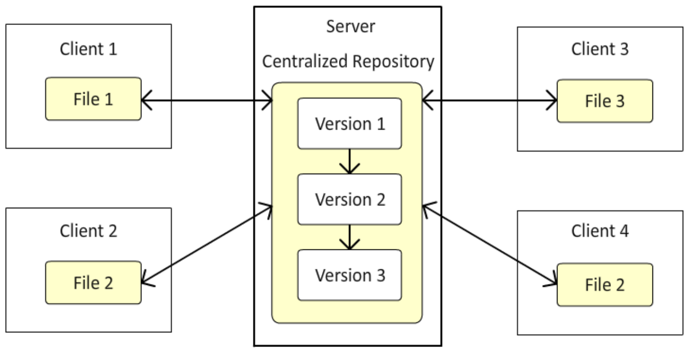


Figure 2.4: Centralized version control architecture [Straub]

A centralized architecture is easy to understand.

The distributed model takes a peer-to-peer approach. Here, all users have a local centralized repository and exchange patches between them. In this model, it is also possible for the repository to have more than just a single root. Users agree which is the original point of the project. While this model is more complex and harder to understand than the centralized model it has its advantages.

A distributed architecture provides flexibility but at the cost of being harder to understand.

As software started to grow, programs started being devel-

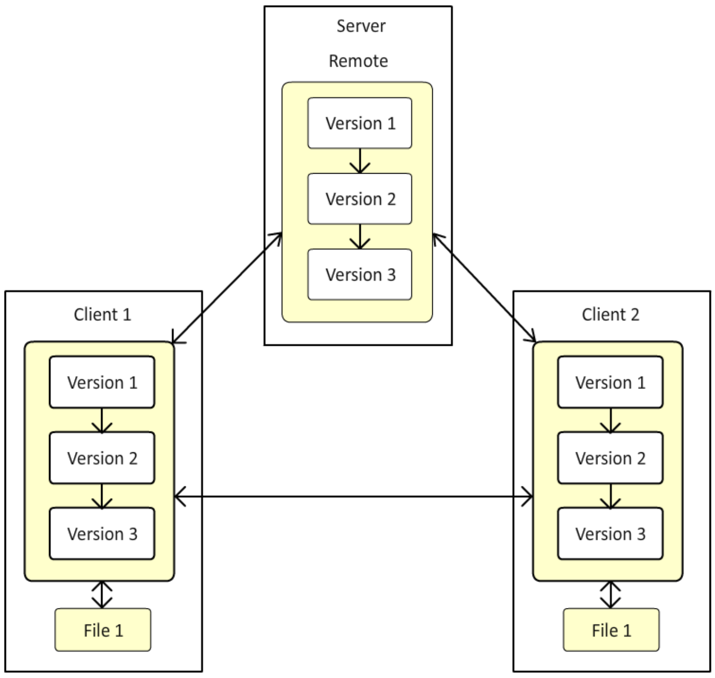


Figure 2.5: Distributed version control architecture [Straub]

oped by teams and not by a single programmer. This was not fully supported by the tools of the time, as each user had a different version of program on their computer. They started splitting the program into multiple pieces with each programmer working on just one piece, later compiling the pieces into the complete program. While great on paper, this presented many issues such as the lack of system testing until all pieces were developed, one piece could cause problems in other pieces and let us not forget deciding how to split the program in the first place. Developers needed a way to work with a whole program but at the same time track the history of changes.

With a centralized approach, developers work on a shared sever; check a file, making the changes and sending back the updated file. File locks were implemented to allow exclusive access to one user at a time. Users will lock the file, will perform their changes on it, complete the tasks, make sure that the changes will not break the system and unlock the file. While the user works all the other users could only read that file. It was the job of the user making the change to ensure that the system worked at the end and that the file was released. [Pilato]

As software continued to develop, simple file locking was not enough, users needed a way to work concurrently on the same file. In this version, each user will work with a copy of the file from the centralized repository, making the change and send it to server. The next user will first have to download the other user's change, merge it with his version, check that it was error free and finally resolve conflict on the centralized repository. This conflict will occur each time more than one user will work concurrently on the same file.

In the distributed manner, one of the users could set the initial repository and the other users will just clone that repository. Each user will work on their local copy and once they have finished they can invite the other users to download their changes. Conflicts are resolved in a similar manner, once you copy over the changes of another user the system will try to do an automatic merge based on the line of texts. If this fails, the user is asked to decide what lines should be included in the merged version to resolve the conflict. This resolution can be pushed to the other users so only one user has to resolve the conflict. Unlike the centralized approached there is no main repository, just a collection of repositories that contain the full code and history. In practice, the users set up a remote master where they put the working copy of the program.

Now that we see both approaches let us do a short comparison of the pros and cons of each method. With the centralized version it is easier to see where the code is, it is easier to learn and understand. But the down side is that it is hard to adapt for complex projects or big teams as the architecture of the software needs to minimize conflicts. Changes can disrupt the whole system and can stop the entire development process until a fix is issued. The centralized one is also slow by nature, because all users have to share the same sever. Connection issues may happen over the network, so any operation, like merge, is slow. In the case of a sever failure the whole project and history could be lost. So a mirror system is required. This again adds to the performance penalty.

The distributed approach is faster, in general, as each user works with their local repository. Branches and merges are done locally before they are pushed to other peers. In the case of text files, merges are usually easier to perform as the users work on a line level and not a file level. In the case of a disk failure only the changes not pushed are lost. The code and the history can be recovered from any of the peers. The distributed nature also allows for greater experimentation as users do not have to worry that their incomplete changes can harm the program of another user. On the con side, the distributed system is harder to learn and understand. The current and most popular distributed versioning control tool, "Git", has a lot of confusing command names that make learning, for first time developers, hard. To bring some clarity in the next subchapter we will go over the common vocabulary used in a version control system.

2.3 Glossary of a Version Control System

Initialization creates a new empty repository
Clone creates a new repository from another repository, if the other repository is empty the operation is similar to initialization.
Repository the place where the files are stored together with their change history, usually this is on a server
Version, or revision is a set of changes on the files currently in the repository; we can also imagine the revision as a snapshot of the state of the set of files we are tracking changes on.
Working copy this refers to the local copy of the files from a repository, any changes we wish to make have to be made first in the local copy. This sandbox is introduced to protect the repository from incomplete work and let the user experiment.
Checkout copying over the files from the remote repository to the working copy region. The user can select the latest version or a preceding version.

Export similar with to checkout it copies the file over from the repository to the working copy region, but it first creates a clean folder followed by copying the files without any tracking metadata. It is used to publish the project.
Import copies over a folder tree to the repository, this folder is not present in the working copy directory.
Commit or check in saving the changes from the working copy and merging them into the repository.
Change or diff is the delta between two versions of a file, this change can be lines of texts in some version control system
Conflict happens when two or more users edit the same file in such a way that the system cannot decide how to merge the resulting file. The user has to manually decide what goes in the combined file choosing from the list of changes. Even if the system can automatically merge it is recommended that users review the merge to make sure that no bugs were added to the program.
Resolve fixing a conflict (usually manually combining changes from the conflicting files).
Merge sometimes called integration, as the name suggests, adding multiple changes to the same file. Merges can happen when two user work on the same file, when we want to check in a file or simply transfer over changes from one branch to another.
Tag or label some changes are more important than others, we like to give them a name in other to easily refer to them later; this action is called labeling.
Change list is a set of changes that have been made in a single commit, a commit can have multiple changes on different files.
Branch sometimes we want to develop a file in an independent way, we can do this to experiment with a new feature, develop a new product or simply to test the current version. Branch allow us to do without impacting the main program, users can still work on the main product, and at the future time the branches can be merge to transfer over new features or bug fixes. Branches can be used also to showcase different versions of the same product.
Trunk or main branch, refers to the main line of development without any branches. As an easy way to visualize one can imagine a line from the initial commit to the head.
Head refers to the latest commit, while each branch can

have a latest commit, head is used to refer to the lasts commit of the trunk or the main branch.
Share sometimes we want to have the same file across multiple branches and update it once and the update be reflected to all branches, that file or folder can be shared across multiple branches.
Remote are versions of your repository hosted on the internet or on the network. Remotes are required if you want to collaborated with other users. Permissions can be set on the remotes so only approved users can changed the remote. Remotes can be also used for backups, review or testing areas.
Pull copying changes from one repository to another. In the case of pull we copy changes from a remote to our local repository. The changes are then merged with our working copy.
Push copying changes from one repository to another. In the case of push, we copy changes from a local repository to a remote.
Update merges changes from our local repository to our working copy.
Fetch copying changes from one repository to another, similar to pull but doesn't merge the changes to our working copy. We could say pull is a fetch followed by an update. We use fetch if we want to put our working copy changes somewhere else, and not merge them directly with the changes from a remote repository.
Pull request popularized by GitHub, refers to notifying someone contributing to the project that your changes have been pushed to the remote repository and they should review them, discuss any changes with the committer and finally integrate them in their local repository.[Wingerd]

1

[1]The current order was chosen instead of the alphabetical approach as it is more natural. The terms appear in the order that the user will/should encounter them while working with a version control system.

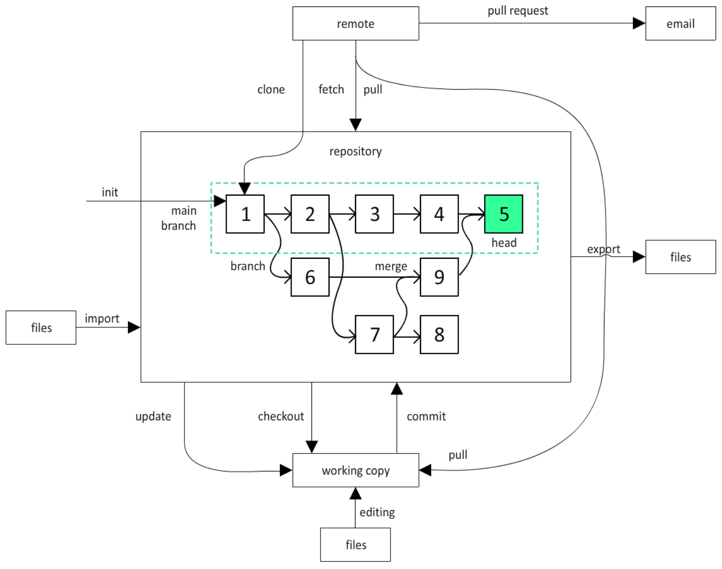


Figure 2.6: Common Glossary

2.4 Creative Tools

We began our journey into existing literature by exploring what a design tool needs to facilitate creation. According to Ben Shneiderman they use the following principles:

- Support exploratory search, in order to innovate and create something new you need to be aware what is already up there. While a search engine is good for text query they struggle when it comes to design work. Sure Google can search for an image and give you similar results but ask him for print design from the 50s for consumer goods and you will get a list of ads and magazines with some from the 50s. It is hard to filter by a company or designer. Sites like "awwwards.com" and "cssdesignaward.com" can be used to look up modern trends. If you want to look up old designs your best bet are still art albums and advertising manuals.
- Enable collaboration, innovation requires collaboration. Special care needs to be taken when it comes to

judging one's work, as creators may be scared of possible rejections.

- Provide rich history-keeping, while many people believe the creation process is unstructured, semi-rigid approaches have showcased great results. We need to have a general idea where we are and where we want to be. Keeping history is important as we can use to reflect on pass mistakes, try new ideas on old works or brings olds to works to modern time. History ties in with the previous 2 principles, peers can review one's history, examine his methods and understand how he devises his invention. Besides the review process, training and learning also benefit from history

  > "Put simply, copying is how we learn. We can't introduce anything new until we're fluent in the language of our domain, and we do that through emulation."

  [Ferguson]

- Design with low thresholds, high ceiling and wide walls, tools need to be design to be easy to grab but grow as the user skills depend. Some of the most popular creative tools like Adobe Creative Cloud follow that principle to heart. Novice users can begin using After Effects as a simple 2D animation tools, moving shapes with simple tweens. As the users gets comfortable with tools he discovers new options that let him move past the simple 2D layers. Effects, cameras, filters and many more are introduced that move the capabilities from just 2D animations to a full motion graphics suite.[Shneiderman]

2.5 History Models

Applications need to provide methods for undoing user actions and/or mistakes or warn them when such actions are unavailable. This has been one of the critical design guidance followed by every interface designer. One way this can be accomplish is with the use of the Command design pattern. To take an example let us consider a basic 2D drawing application that allows the user to draw and transform (move, resize, and change color) circles on a canvas. We can design a Circle class with the following properties: x and y holding the position properties of the center of the circle, color representing the color of the circle and r, the radius of the circle. We can have a method setColor that changes the color of the circle, a setRadius that changes the size of the circle and of course a move method to position the circle. We also need a draw method that accepts a canvas objects that draws our circle.

A simple application can create a new circle object when the user creates a new circle and adds it to the display list followed by a clearing of the canvas and redrawing of the display list. Any changes that are performed on the circle object are followed by a clearing of the canvas and a redrawing of the display list. While stratifying our initial requirements this application doesn't provide an undo mechanism. A naive approach will be to save the state of each display list state and reuse that when undoing. While it will work, it doesn't scale up plus it ignores the decencies of the states, each state is derived from a previous state plus a change by the user. This is where the Command pattern comes in.

Instead of calling directly transformations on our circle objects we encapsulate them into objects that we call commands. Using an object-oriented approach, we can create a Command interface with a method execute having an actor parameter and transform object parameter. From this interface we can write the following classes: Move (that takes a circle as a parameter and a transform object having the x and y values for our new location), Color (takes a circle as parameter and transform object with the color code), Resize (takes a circle as parameter and a transform object having a positive number representing the new radius of the circle). Whit this transform classes we can represent all of our circle transformation actions. Each command objects maps a user action, thus forming the editing history. These objects are stored in a list where each new user action adds a new object to the list. The current state of the circle can be recreated starting with an initial circle and iterating through the list.

Undoing can be accomplished by removing the last element from the list and reiterating over the list. An optimization can be added, if we extend the Command interface to support an undo method. Then instead of reiterating over the whole list we can just run the undo method on the last object from the list. A current pointer can be added to facilitate undo and redo actions. If we graphically represent the history list, we obtain a simple line.[Nystrom]

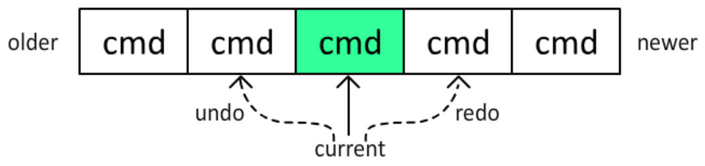


Figure 2.7: Command Pattern

While sufficient for most application this model presents some serious drawbacks that come into light when working with designing applications. One such limitation is the lack of selective undo, let us assume we don't want to undo the last command but just its parent, the next-to-last command. With the current linear model, we cannot. And once we undo a command and start adding new commands we lose the possibility to the redo losing the initial command. This problem hurts experimentation, as users know they can lose commands. It doesn't help that most program store history in memory and once a file is written and closed the history is lost. Improvements have been made over the years, some programs showcase the history in a panel, where users can deselect action they don't want to be executed. While this model is more suitable for vector graphics applications, bitmap implementations have been suggested, see Aquamarime [Le].

A non-linear model can solve these issues, by allowing commands to have one or more children, the history can be branched, edited and merged without losing any of the commands. While powerful, this model is much harder to understand by the user and also presents problems when it comes to an efficient implementation.

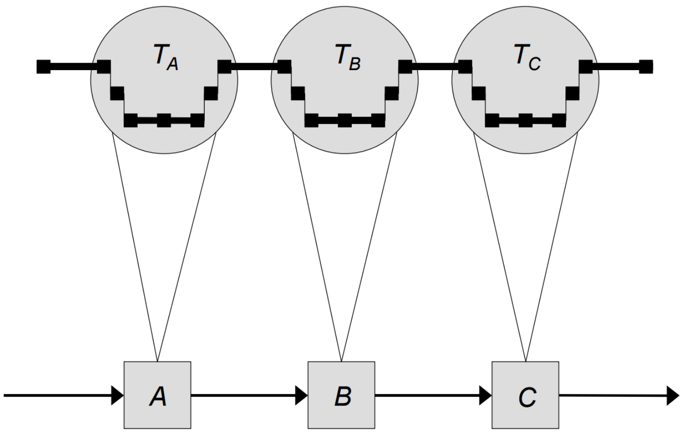


Figure 2.8: Nested extension to the linear history [W. Keith Edwards, Takeo Igarashi]

As an example let us consider Flatland application from [W. Keith Edwards, Takeo Igarashi] paper. Flatland is a whiteboard application where user can draw segments and select a behavior. Segments can be made to act like an unordered list, a map etc. Like any interactive application it needs a way to let the user undo or redo an operation. But what makes Flatland special is that action one stroke can affect another stroke. So a simple linear history will not work as it will allow the user to scroll to invalid states. The first solution that authors proposed is a nested timeline that puts on stack operations that have a causality, thus eliminating any invalid state, as the timeline will undo the whole stack to get a new state. Next they wanted to support a local timeline for each segment so that the users can perform a local undo. A first solution was to have a have a list of the complete history, each one for one segment. The global history will be built by sticking tougher all the local histories. This works fine as long as one segment change doesn't affect another segment. However, in Flatland this is not the case. The solution follows the previous idea, in the global timeline we need to nest actions that started with the same cause, doing so avoids any invalid states. The local timeline is preserved and all actions can be undone in a predictable fashion.

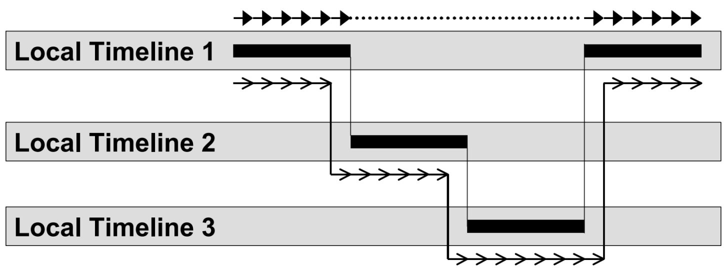


Figure 2.9: Global history is made up of multiple local histories [W. Keith Edwards, Takeo Igarashi]

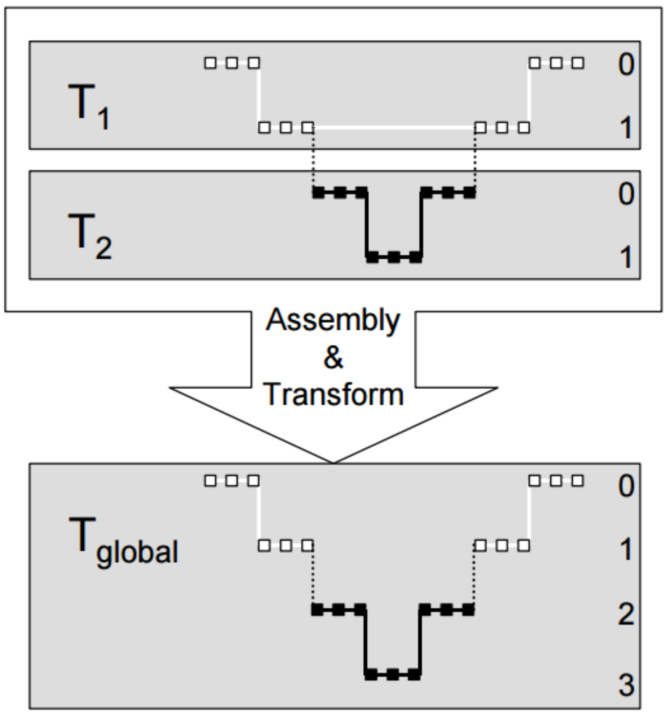


Figure 2.10: Push-transform moves top Level Location and transitions it to a new global nesting [W. Keith Edwards, Takeo Igarashi]

2.6 Version control for Graphic Design

We continue presenting some of the leading research in the field. First is the system of [Chang] built on GIMP it offers an efficient versioning control system for painting and sketching applications. The system uses a DAG internally to represent the editing actions, each node mapping an editing operation. This graph is then processed to display a RevG graph so the user can easily navigate with it. The RevG graph is generated from DAG graph generating a preview of the set action. The system implements the traditional version control vocabulary, users manually commit changes, make branches etc. The system comes with a simple use merging system that allows users to select what should be merged. A zoom-in exploration highlight the underlining changes, allows the user to undo actions selectively. Playback and compare actions are added to complete the tool. The evaluation study found that users can perform the basic revision control action with little training. Problems arose while navigating the RevG graph, while users like the final detail view, they sometimes got lots lost in the details.

Problems also arose of when actions are needed to be displayed, to support selective undo the system doesn't display actions in a temporal manner. The replay function was found to be very useful especially for stroke playback, as it is difficult from a flat image to deduce how it was drawn. Performance of the app is also impressive, the DAG is built on the fly and commits only save the different changes notes, a more efficient storing option than the binary deltas. From our user observation (See more in chapter 3) we found that most users when working on designing applications or web sites use layers to organize information. Work chunks are usually performed on a single layer or layered group. The previous system completely ignores this fact treating the whole document as a huge canvas. Using this information, we can more efficiently organize the document edit history. Another limitation of the system is the lack of powerful search options; this becomes a serious problem when working with large files that present a huge change history making the current navigation unsuitable. The system in the current form doesn't support multiple users but this can be added as the underling framework supports it.

The second system we are going to have a look at is Chron-

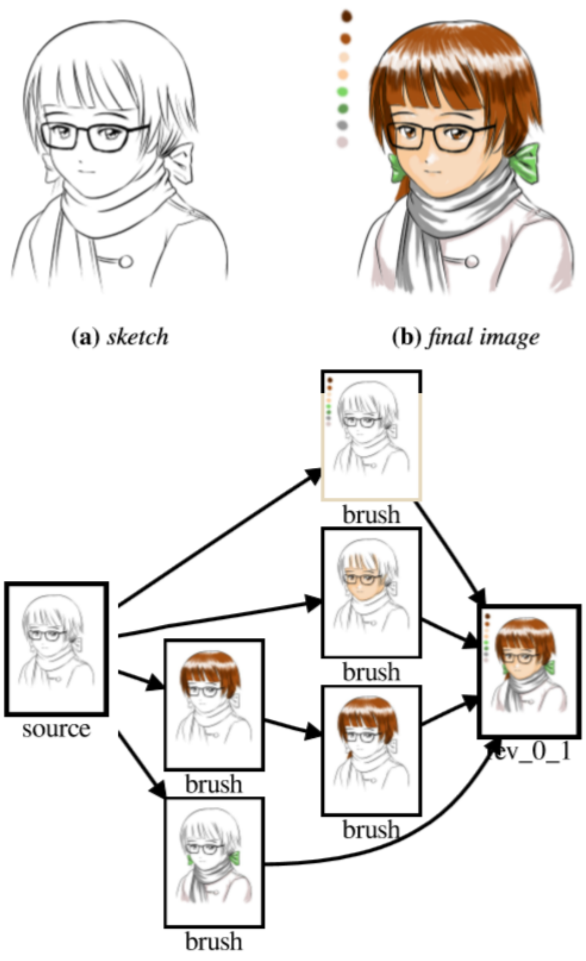


Figure 2.11: RevG fro a digital sketch
[Chang]

icle. Chronicle records a video of the hole edit history and provides information to navigate this history. The main focus here is learning. Users can filter a particular area of interest and play the history in order to see the workflow. Settings can be reviewed and copied over. Chronicle offers powerful filtering options, users can filter changes that include an area, a tool and like our application a layer. Multi-user support is also enabled, users can see changes of a particular author. Annotations offer even more information; users can mark specific parts of the history to give hints. While not a version control tool, Chronicle show-

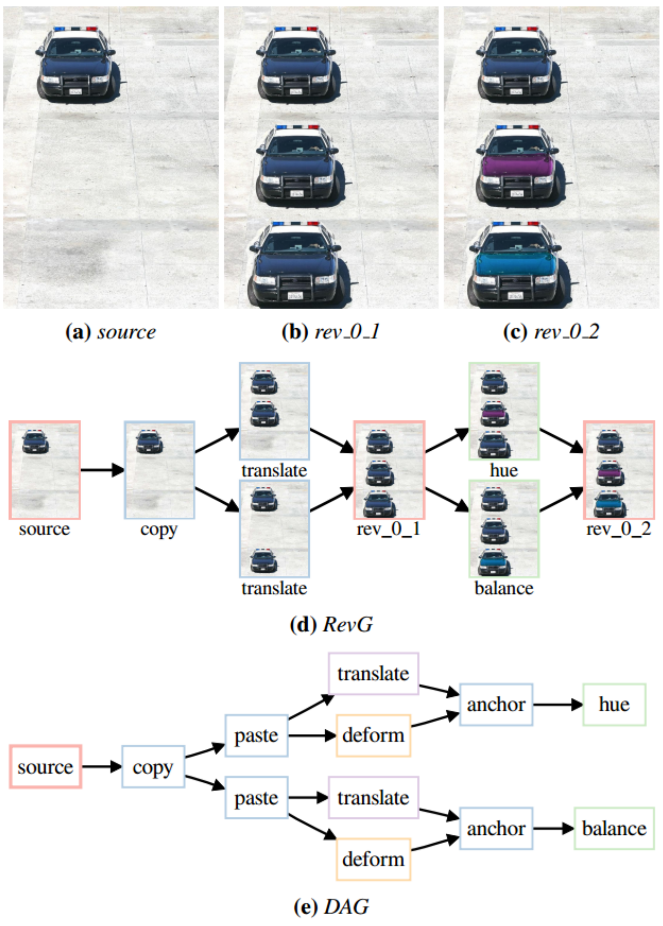


Figure 2.12: Construction of the DAG graph and the RevG from 2 revisions [Chang]

cases one of the most important benefits of edit history; training. New users can be trained in the tool with minimal effort by the original authors. Furthermore, this tool facilitates the integration of new team members in a project. The filter mechanism highlights the importance of navigation, without such a mechanism history loses its benefits, as it too hard to use. Unfortunately, the history is only recorded as a video and a series of events and doesn't provide any of the features a version control system like branching. This is understood as it can be more easily implemented on existing tools with little to no changes of the architecture.[Fitzmaurice]

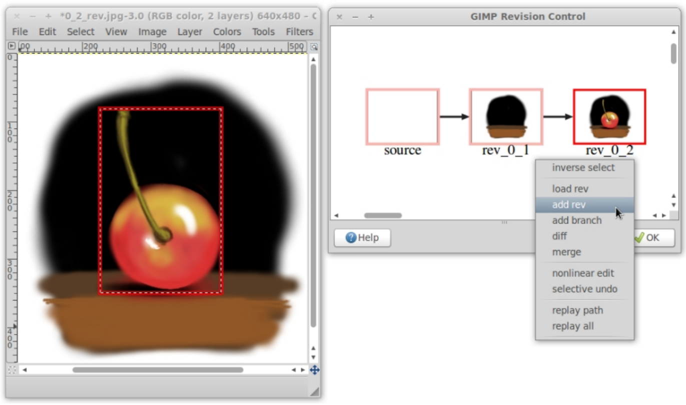


Figure 2.13: Revision Control System user interface [Chang]

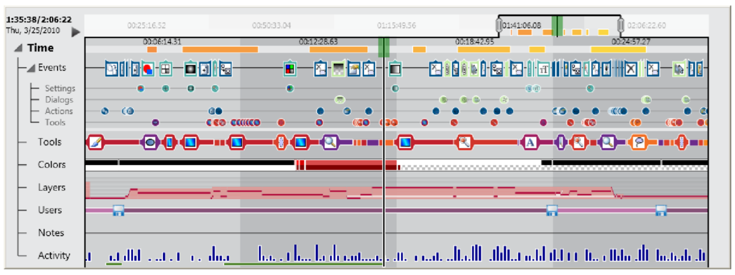


Figure 2.14: Chronicle Timeline user interface [Fitzmaurice]

2.7 Time-Machine computing

The final system that we are going to analyze is the one described in the Time-Machine computing. On this computer all machine states are archived, users can travel to any previous machine as if having a "time machine". This interesting organization method limits the need for a folder to organize information. It is important to note that actions are permanently archived, like create a file send or receive an email, etc. This archived function is implemented at the operating system level. This time metaphor allows the user to organize the desktop like a real-life desktop. You create your file work on it and once you are done you simply delete it from your desktop. The file is saved, you can simply travel in time and see that file on your desktop.

Navigation in this system presents some unique challenges, the users cannot always remember the exact time. One option is to provide navigation by time based on some events, like when I started project A, or when I received an email from Bob. A playback option is also supported to allow for browsing of the history of the machine. Probably the most interesting way to navigate in the past is through the use of post-it notes that allow the user to mark specific events. As a final display option the system also provides a calendar view. This is useful if the user wants to have a familiar overview.

The initial version of this system had the history read only. This was quickly challenge by the user who wanted to go back and change past event, this again highlights need for a branching version control system.[Rekimoto]

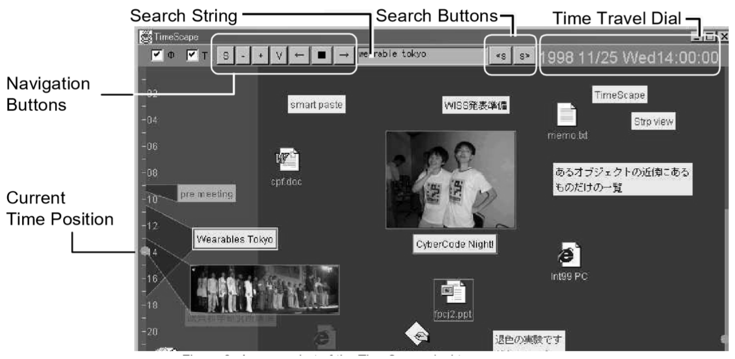


Figure 2.15: TimeScape desktop user interface [Rekimoto]

Chapter 3

Pilot study

In the last chapter we have explored what a version control is and what researchers have done to bring it to the graphic designers. In this chapter we will look at what a designer expects from a version control system and why they have not adapted the traditional solutions from the development work to their workflow.

3.1 Online questionnaire

We began our mission by creating an online questioner and distributing it on designer forums and designer groups on Facebook. We have chosen this option instead of going directly to their workplace in order to get a more casual answer regarding their experience. We asked them 11 questions and received a total of 33 answers.

Below we have a summary of their answers:

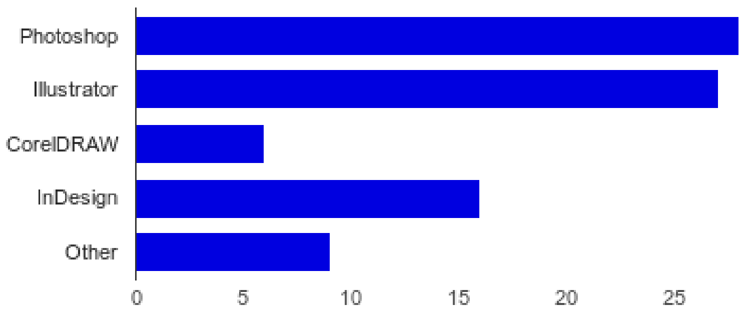
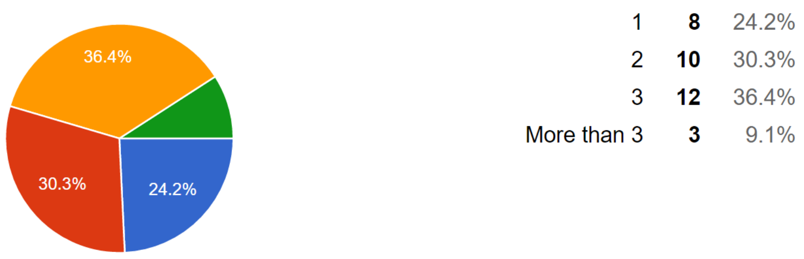
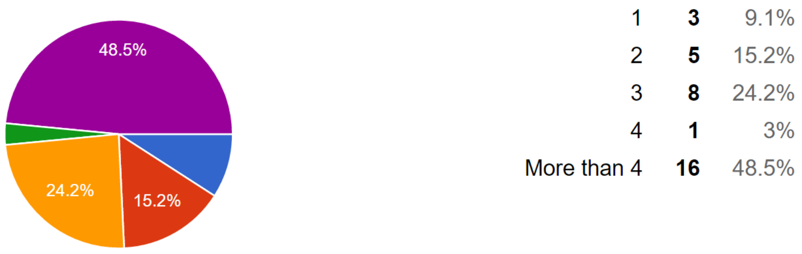


Figure 3.1: What tools do you use?



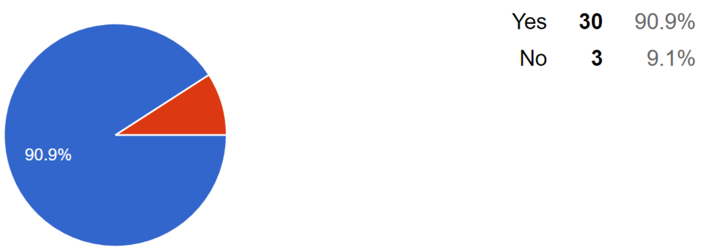
1	**8**	24.2%
2	**10**	30.3%
3	**12**	36.4%
More than 3	**3**	9.1%

Figure 3.2: On how many projects do you work on a given day?



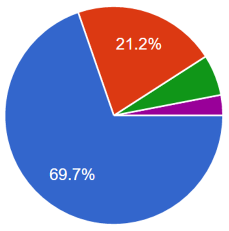
1	**3**	9.1%
2	**5**	15.2%
3	**8**	24.2%
4	**1**	3%
More than 4	**16**	48.5%

Figure 3.3: How many files do you have to open on a given project?



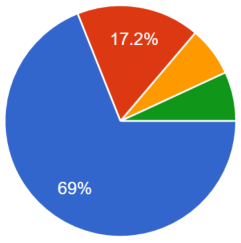
Yes	**30**	90.9%
No	**3**	9.1%

Figure 3.4: Do you save old versions of the files?



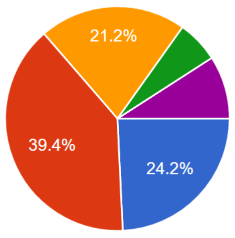
I save a local copy with a different name, like version1, version2 etc	**23**	69.7%
I save my files on a file hosting service like dropbox, google drive, creative cloud etc	**7**	21.2%
I use git / cvs / team foundation server	**0**	0%
I don't save old versions	**2**	6.1%
Other	**1**	3%

Figure 3.5: How do you manage old versions?



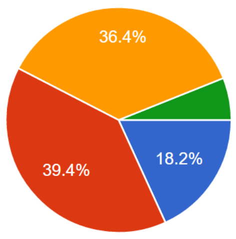
No	**20**	60.6%
Yes, one more person	**5**	15.2%
Yes, two people	**2**	6.1%
Yes, more than two people	**2**	6.1%

Figure 3.6: Are there multiple people working on the same file?



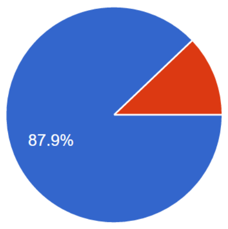
Less than a week	**8**	24.2%
A few weeks	**13**	39.4%
A few months	**7**	21.2%
More than a year	**2**	6.1%
Other	**3**	9.1%

Figure 3.7: On average how much time do you spend on a project?



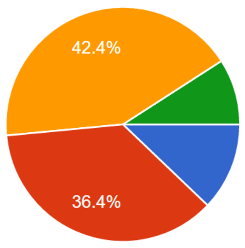
Yes, often (more than 3 times per file)	**6**	18.2%
Yes, sometimes (at least once per project)	**13**	39.4%
Not really (once on some projects)	**12**	36.4%
No	**2**	6.1%

Figure 3.8: Have you been forced to revert to a previous version?



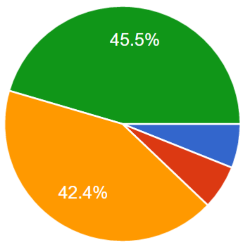
Yes	**29**	87.9%
No	**4**	12.1%

Figure 3.9: Have you been forced to revert some parts of the file to a previous version?



Locating the file	**4**	12.1%
Merging the files (if you are restoring parts of it)	**12**	36.4%
Keeping track of the new file resulting from the restoring	**14**	42.4%
Other	**3**	9.1%

Figure 3.10: What's the most tedious thing when reverting a file or a part of the file?



Just one	**2**	6.1%
2	**2**	6.1%
2-5	**14**	42.4%
The design is never done :)	**15**	45.5%

Figure 3.11: On average how many iterations do you need, to come up with the final design?

Bellow we draw some conclusions from the questioner:

- Any tool that we want, design should work in a similar manner to the tools the designers are already using, to make learning as easy as possible, ideally it should be an extension to their tool. In the case of Photoshop, we can image it as a separated panel.
- An interesting result we have is for the second answer. While it is clear that some designers work only on one project a day some of them work on multiple projects. We assume that the ones that work on multiple projects must work in a design studio, where more projects are being developed simultaneously.
- It is clear that designers work on multiple files at the same time, so extra attention has to be given to navigation and management of files. The system will need to make sure that the designers do not get lost in the multitude of files.
- Saving earlier versions of the design is a common practice, hinting at the importance of the history.

- Designers do not use the traditional versioning control tools. This is again will be explorer in our interview section.
- Simultaneous editing does not seem to be such a needed feature. Most designers are the sole author of the project. We suspect this as a side effect of the lack of tooling support or because designers want to do one thing in a project.
- Designer do many iterations before submitting a design, any tools need to be designed to handle a large number of changes, over 10 versions for each file.
- Reverting to a previous stage is something that happens occasionally on a project. The more interesting part, we think, is the partial revert as this is something we suspect more common. At the moment the most common way to revert is with the undo command but this is linear and presents the problems we have discussed in 2. For bigger reverts we assume they are using an older version of a file copying over objects to the current version and manually merging them.
- Keeping track of the files seems to be the biggest issue at the moment. People are having problems organizing the history. The locations of the different versions come from this problem and limits the use of a version control. Useful would be having a complete history if you cannot find something. The merging issue is expected because of the nature of graphic projects. We suspect that most designers have not used the tools described in 2.

3.2 Online user observation

In the next stage of our study we looked online on how people work. We have chosen 8 videos from Youtube of how people design a webpage. We have gone for this approach and not an onsite observation in order to minimize the bias that could occur during the observation while doing their work. While this bias is also present when someone is recording their work we think it is smaller as the people that record such videos usually do them to teach other people how to do the same or want to showcase their skills. Last reason for choosing this approach is that it makes reviewing of the observation much easier, as other researchers can re-watch the same videos.

Video URL	Uses Layers	Organizes Layers	Notes
`http://y2u.be/xC01YXtpvwU`	YES	YES	Design starts from top to bottom, does an organization of the design as it works. Does a final cleanup of the layers at the end once he is pleased with the solution. Copies over objects to reused them. Uses a grid layout when designing. Work is grouped on small region of the file
`http://y2u.be/HpvTJmQYa9c`	YES	YES	Uses screenshot as guides. Design from top to bottom. Starts with navigation Reuses common design assets like social network icons. Uses a grid layout to organize information. New objects are created from duplicating old objects and changing them. Doing so preserves the style. Work is grouped on small region of the file Experimentation and revert with multiple styles.
`http://y2u.be/2k-052E5_BI`	YES	NO	Items are done in Illustrator and brought into Photoshop for final assembly. Design from top to bottom. Grid used to layout items.

`http://y2u.be/u23sN24fMJs`	YES	YES	Designs start with background, then moves from top to bottom Assets are prepared in a separate file Experimentation takes part all over the project not only in one zone Columns are used to layout the content. Groups of objects are created by duplicating an existing object. Users uses the History Panel. Design from left to right.
`http://y2u.be/3sCshDAqINI`	YES	YES	Old site used as starting point and for reusing of common assets such as the logo. Grid used to organize content. Assets are preprocess and assembled in Photoshop. Design from top to bottom. Application state are represented with layers. Groups of items are creating by duplicating the starting item and modifying the content.
`http://y2u.be/HRPE-5RarFo`	YES	NO	Design from top to bottom Uses grid to organize content.
`http://y2u.be/V0iTsH6CjKc`	YES	YES	Uses grid to layout design Design from top to bottom Reuses a rectangular as spacer to space out items Layers used to market different states of the application

http://y2u.be/iGdS1CZ5bUs	YES	YES	Design starts with logo Design from top to bottom Assets are prepared in separate files Content organized by columns Groups of items created by cloning States represented with layers Layer cleanup at the end

Table 3.1: Designing a Webpage - User Observation

We considered a user organized if he renames the layers, groups and moves them into folders.

From these 8 examples we can conclude that users: use layers to organized content, the design starts from top to bottom, grids are used to position items. Work consists of many chumps; each chump is focused only on a small portion of the design. Each chump has around 1-3 layers. Most experimentation focus on fonts, colors and positioning.

3.3 User interview

We have interviewed 2 users. Please refer to our annex for the interview questions: one user had over 10 years' experience using Photoshop and is the lead designer of a startup and the other is an intermediary user with 3 years' experience working as a freelancer. The daily routine of a freelancer is different from a full time employee as he also has to manage the business aspects of the job. From our interview we found he usually works on 2 projects a day and, sometimes, has to do small updates on client requests. In case of the lead designer, he has to collaborate at least with another person on a project. Tasks are usually split by expertise, some artist focus more on branding and logo creations while other focus more on user interface. As a general rule there is someone that assembles the whole work and presents it to the account. Review meetings are held with the account manager and the whole design team plus representatives from the development group. It is not uncommon in a review meeting for an alternative design or an older version to be present, as clients like to experiment before going for an idea. For team of designers having automatic file synchronization is the most important feature, currently this is accomplished through a cloud solution. When asked about versioning they have a daily backup solution. They perform manual saves on important events that they may need referencing later. This is something similar we have also observed with the freelancer. He will save a new copy of the file and name it for that even. Unlike the team he uses Dropbox to keep his file saved. Both user's express frustration with organizing the files, the guy working in the agency details their naming structure and also the struggling of enforcing it. We asked if they have multiple users working on the same file. And he said yes but not at the same time. When ask why, he confirms it is hard to keep track of the changes. They prefer to work on separate files, create smart objects and merge them in the complete file. Later changes can be done by just updating the smart objects. The intermediary users did not mention the smart objects but he said he keeps layers around. This behavior can also be observed in Brad2015's interview. Going deeper in the review of the design process we found that the freelancer used a notebook to take notes while lead designer uses an online tool to mark a jpg of the design. Asked how they organized their work, they answered with layers, groups and smart objects. We asked the more experienced users if they heard of version control, and he confirm it, we asked why they do not use git. His replay was it is too slow for our job. We asked what are his pains with the current cloud solution that host the files. Restoring a version we need only a part of it, at the moment we need to duplicate the file, restore the old version open the two files and manually combine them, was his answer. The freelancer said locating the exact file, Dropbox does not really have a fast way to search old versions. I wish I could tag them.

We revisited the same users with our prototypes for feedback. Please refer to the evaluation phase for their answers.

Picture and Name	Details	Goal
[Pexels, a] John	Role: designer Background: 28 years old, bachelor in Fine Arts Uses: Photoshop, Illustrator John has to integrate other team members' work. Primary task is to design UIs for web apps.	"I want my work to have meaning"
[Pexels, b] Suzy	Role: freelance designer Background: 26 years old, self-thought Uses: Photoshop, Illustrator, Outlook, FreshBook Suzy wants her design files to be organized. Wants to impress her clients.	"I want to finish my work before 5pm so I can have more time for myself"

Table 3.2: Personas

3.4 Personas

With the information gathered we started doing our personas. We split our users into two categories depending on their job status: freelancer or full time employee. We consider this an important factor of how versioning control system will play out. For freelancer's history is more important while a team member may value collaboration more.

Bellow we have our two personas:

3.5 Summary

We conclude by presenting a list of some the feature that any version control needs to have to be feasible in the design workflow:

- Quick saving of changes
- Fast and easy navigation and locating a version
- Easy comparison between version
- Low learning curve and if possible be integrated the in the tool that design is being created
- Low "initialization" cost, setting up a new project should be as easy as creating a new folder
- Automatic saving, while from our interviews we found that users want manual save the practice says that this not always what is need it. A tag system we believe better suits the user plus also eliminates the problem of finding good names for each version
- As a consequence of automatic save we believe that the undo system should be integrated in the version control system
- Easy automatic merge operations with the option of creating a manual merge
- Facilitate collaboration and offer real-time multiple user editing
- Powerful filters to identify changes made by a specific user, on a specific zone and a specific group/layer/object

Chapter 4

Implementation and Design

The goal of this thesis is to explore if a version control tool that tracks changes on a layers basic is a valuable approach. We want to design a tool that takes the ideas from 2.6 and adds in the insights we observed from our pilot study. We start by going over our initial design assumption, that designers want an easy to setup and use versioning control system. From that we develop a paper prototype inspired by current systems, we did a review of that system with the user, going over the actions that arouse in a typical workplace, we follow by a video prototype, to see the user interface and user interactions in action. This prototype was again reviewed and feedback collected. We complete by showcase a 3rd prototype that descripts the entire set of actions. This last prototype we used in our final user evaluation. Each of the 3 prototypes were evaluated and their feedback was incorporated in the following prototypes, following the classic DIA cycle.

4.1 First Iteration

4.1.1 Concept

Based on our user study results3, we decided to build a tool that was integrated in the current workflow and used the history from a cloud solution such as Dropbox and displays them in an easy to manage solution. We took inspiration from tools already on the market such as pixelapse.com and combined it with the best user interface guide from [Norman] and [Lidwell] to eliminate some of the shortcomings, such as the lack of branching, notations and multi user support. In this initial version we will explore the design requirements testing the users needs in the current workflow.

For displaying the history, we selected a tree structure as it offers more flexibility and it is a better mapping for branches. Interactions where design for an external tool, the user will occasional used this to locate the desire versions, download it and merge it with the current version.

4.1.2 Implementation

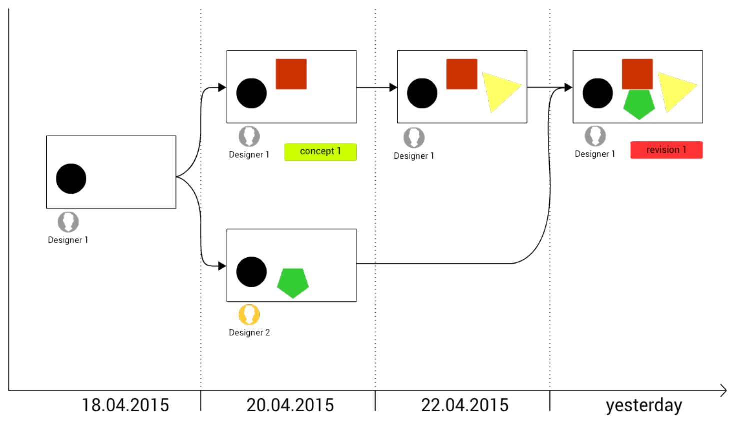


Figure 4.1: Initial Idea for a Version Control for designers

In this first prototype the history is displayed as a tree. Each save state is represented by a node in this tree, bellow we will see the author of that state and a list of tags. The tag list supports multiple colors to make identification much easier. The tree is displayed horizontally from left to right, the root being on the left side. We chose this approach and not a vertical display as we expect users to have multiple branches and it is a more natural fit for a timeline. Having a horizontal timeline also allows us to display more states as all modern monitors are in an aspect ratio of 16x9 or 21x9 [Steam].

We decided to not include a naming option to each state and instead used a tagging system, as it is very hard to produce manifold names for each version. With this system we allow the flexibility of having names and reusing them without being forced to name each state.

All states are grouped by the creation date from left to right, from our interviews with the designers they suggested that they saved around 5-7 versions a day.

Setup of this system, was just creating a new project and selecting a folder from the cloud service as the root of that project. Each individual design file will then be shown and the user could click a file to see his history.

4.1.3 Evaluation

We presented the prototype to two of our designers, one is a freelancer designer with more than five-year experience working with small clients and the other is a lead designer at a digital agency.

Initial feedback was encouraging the like, the idea of using the Dropbox history to display the file history, as this will not have major changes to their workflow. They presented some doubts of how branches can be shown when dealing with a Dropbox history. The history is created from any saved version of a file in a Dropbox folder, they expressed some worries as sometimes they save incomplete work or other times they forget to save complete states. They preferred the tagging system as opposed to a traditional naming system, they will like the system to have an option to display all tags per projects and of course filtering by multiple tags.

We tested the system with a paper prototype, having around twenty saved states each represented by a piece of paper. Even with this small number of saves, we found that navigation is tricky. We are thinking of implementing some keyboard shortcuts to move faster through the history. Arrow navigation is a natural fit here, users can use the four arrows keys to move up, down, left, and right. Home will go to the first version, and holding the shift key with one of the keys will move ten saves, this is a familiar concept from the Adobe Creative Suite. The number of saves being displayed at once can also be a problem, while it is nice to see more "work", it is however harder to understand. Using previous research about the user's working memory [Kuorelahti] we are thinking of limiting the display to around ten saves per view.

Another interesting interaction we observed is that the user expects to swipe through the history, pan around similar to any touch interface.

Users asked what happens when they download their history, how do they restore, and how is this reflected in the system. All of these actions have to be manually performed by the users, when asked how they perfum them, they said they copy over in the current file stuff they need and follow by "integrating" them in the current design. This "integration" is usually positioning, scaling and some grouping

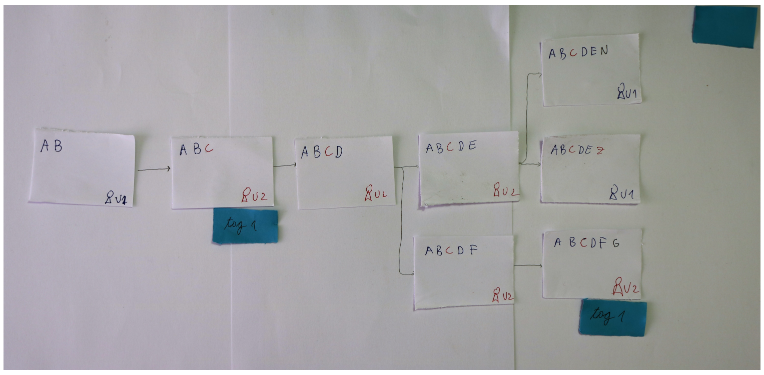


Figure 4.2: First Iteration of the paper prototype

4.2 Second Iteration

4.2.1 Concept

For our second iteration we decided to embed the external tool directly into a design authoring environment like Photoshop, while this approach will limit the number of supported formats, each authoring environment needing a separated plugin, we believe the positives outweigh the negatives. One advance of this approach is that users don't need any extra setup process, creating a file automatically will setup the versioning control environment, restoring can be done by simply selecting a version from the history, no need to download and override the current file. We also changed how versions are committed, the system has an automatic save option. We implemented some filters to improve navigation, the users can filter changes by users, region and tool that created them. Last thing we considered is a replay function, so the users can see how the design progressed. For this replay function we also implemented a calendar view so the changes can be viewed on a daily range. For this iteration we implemented a paper prototype but also a video prototype so we can get a better feel of how the system will function.

4.2.2 Implementation

One important design decision we had to consider is when we should save the revision to our history. Should we save it when the user saves the file or should we implement an automatically save system. We asked our users what they prefer and most said manual save. While we believe that all users want manual control over their tool, workplace experience has proven this to be not that case. As most common productivity tools (like Microsoft Office, Adobe Suite) offer auto save features plus a method to recover files if the app crashes, we believe there is a place for an automatic system. By eliminating the need to commit in the system we will also bring the learning curve down, this is one of the main reasons that has made version control tools adoption slow. Going with this automatic save system we need to decide what we save and when we save. A simple solution is save every minute and ignore the changes the versions that have no new changes. While simple to understand this model may have a different number of changes per version. A version can have just one change while another a complete set. To eliminate any confusion, the system saves on each new change. This approach is easy to understand by the user but creates a new problem, saving on every change creates a complete history but a very long history. Thus filtering and grouping is key to facilitate navigation. By default, the system always showcases the previous version so the users know the action of an undo command. After that it displays the version with the last minute of work, last hour of work until the beginning of the day followed by a day grouping. Click on a group of items maximizes the navigation to show all the corresponding versions, so clicking a day will show a list of groups by hours, hours a list of groups by minutes until we get to the individual version. All changes are saved as the user works, so he can see iteratively how the history is being built. If volume of works is not sufficient to be grouped by hours they are shown in a list. The arrows between history notes are not shown if no branch is presented, this makes for a cleaner look of the navigation.

In any stage of this exploration we can use our region filter

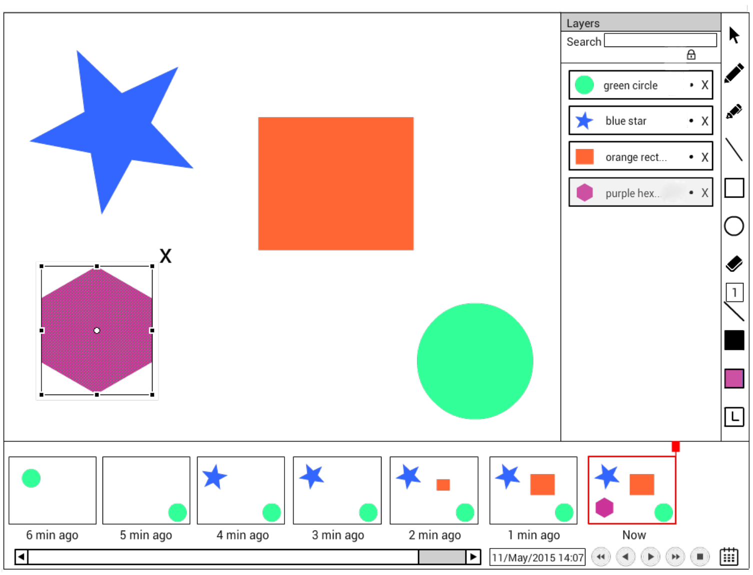


Figure 4.3: UI of Versioning Control system from the Video Prototype

to remove any versions that don't have any changes in that region. This filter result can be further refined by showing only changes made by a specific user or tool. If we consider the time of change as a filter the system has a total of 4 four classes of filters to narrow down the desired version.

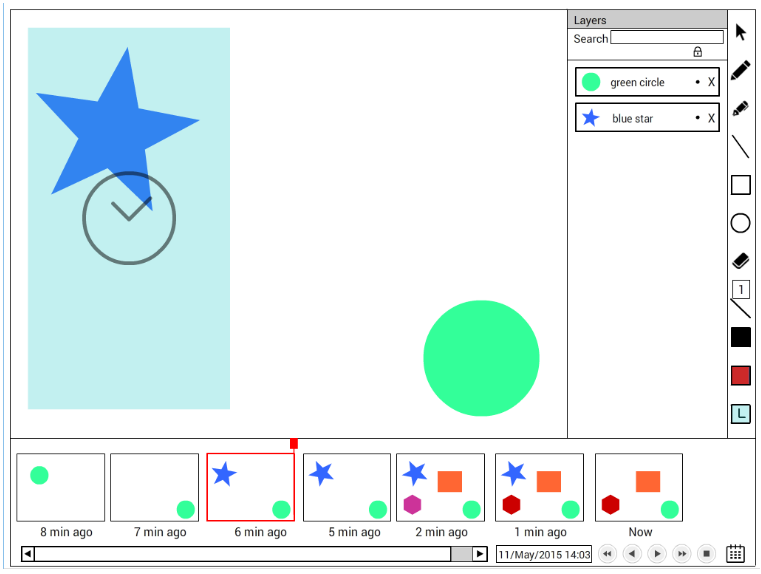


Figure 4.4: Region filter from the Video Prototype

In our video prototype we implemented a playback feature, the user can play forward and backwards through the navigation. Playback speed is one change per three seconds, and can be sped up by holding the shift key. To produce a smooth motion, animations are automatically created between two consecutive versions. Thus an easy to follow action for the user is achieved.

We implemented a compare feature so the user can compare different versions, the system allows for scaling and panning when using the compare tool. And to highlight differences between two versions, the system fades to grey in the common parts leaving only the difference in color.

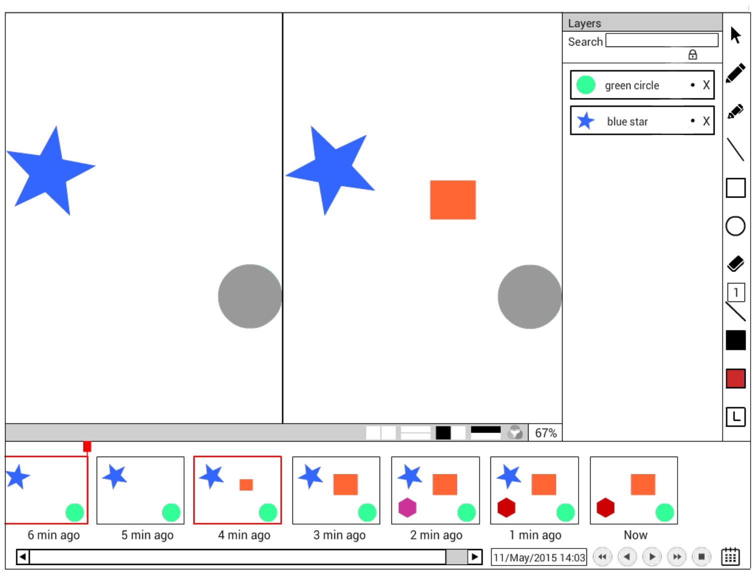


Figure 4.5: Compare UI from the Video Prototype

To create a new branch, users need to go back to a previous version and start making a change, the system will then create a branch for him.

Mergers are done in a similar manner as the system described in 2.6, with users selecting regions from different versions that they want present in the resulting, merged, version.

The calendar view show a bird's eye view over the working month, a user can select a day or a range and in the left side a video will be generated of all work that has been done. The use case for such a feature is to swiftly remind the user what he was working on. This can happen for a number of reasons as designers are moved from one project to another or simply go on a holiday. Another use case for these features is as a learning tool similar to what was presented in [Fitzmaurice] or as an estimate assessment, project managers and designers can look over the history to estimate how much time a similar change may take.

4.2.3 Evaluation

4.2.3 Evaluation We again presented the prototype to two of our designers, this time first the paper prototype followed by the video version. The initial feedback was encouraging they welcome the move as a plugin integrated directly into the design tool. The automatic saving features presented some doubts, "But I want to choose when I save", tags will give some control back, and as they get familiar with the tools the instinct of always saving not to lose work will disappear. The freelancer who had more experience working with google docs welcomed the change. One important question that arose what happens with the undo model. What happens if the designer undoes a change and then continues working? Will that change be presented in the history or be skipped completely? Our initial assumptions were for it to be removed from the history as to keep the history as light as possible. But after further discussions we concluded that this information could be useful further down the line, and could remove one of the drawbacks of having a linear undo history model, that after undoing a change and adding new changes the initial action is lost. We discuss if this undo changes should be kept forever in the history or cleared on exiting the application, we concluded that it will be best they are kept.

The filters were welcome, the region filter being considered the most useful of the four, however, after further discussion we observed that designers like to organize an "object" on a layer or a folder, while we can draw a region over that object to see the history, it will have to be included on all the objects that overlap that region. So it was concluded that filtering based on "object" will be of great use.

The Comparison Tool was also welcomed, suggestions included the ability to present multiple views of the history, so the user can work while viewing a previous version of the file or a different branch. The playback function while fun to use, will probably not be used that often, however, it maybe be useful as an educational tool. The same skeptic feedback regarding the calendar view, is nice to have but they don't consider using it daily. Problems were observed also with the merge tool, which follows a similar design to one from [Chang], selecting just the region to be merged is not that easy to use. We suspect the problem is visibility, so user training is necessary before this feature can be used with confidence. We also need to represent this merge better in the history navigation, currently this is represented as two arrows intersecting into a new history node.

4.3 Third Iteration

4.3.1 Concept

For our final iteration, we decided to merge the undo model with the history model. Changes are now tracked on a layer basics and not on a global file level. The "global" timeline is composed of a multiple layer timeline, following a similar pattern described in [W. Keith Edwards, Takeo Igarashi]. If a user undoes an action, it will still presented in history as a separated branch, drawn with a semitransparent option to indicate it was generated from an undo action. We updated the layer view to include a button to filter by layer option. We removed the calendar view to streamline the navigation, and update how the history items are been grouped. We will change the previous click-to-expanded mechanics with a scroll to review more changes. We also consider making the versioning control panel a detachable panel, to facilitate multi display workplaces, a common account according to our designers.

4.3.2 Implementation

For this last iteration we implemented only high quality screenshots of the whole system actions. We opted for this option in order to represent all the system actions. We will continue by presenting each of this screen and explaining the changes that we did compared to the previous iterations.

Main screen

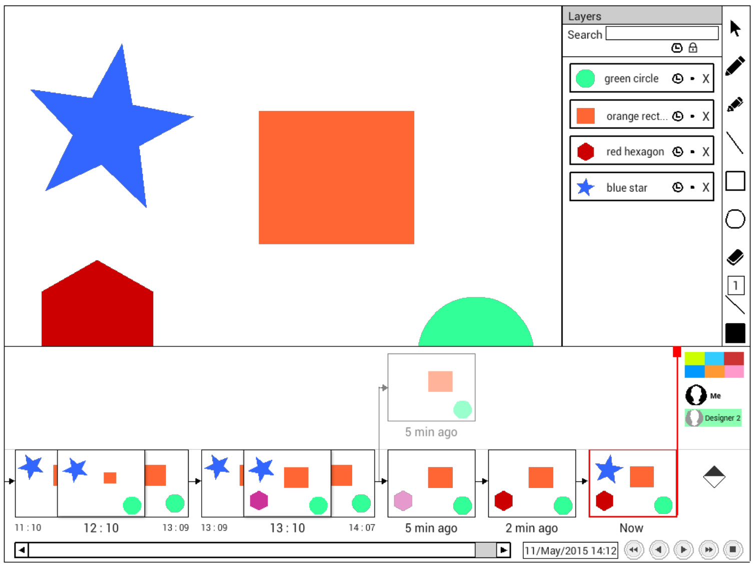


Figure 4.6: UI of Versioning Control System from the final iteration

Starting with the main UI, we made the panel resizable, so that the users can decide how many branches they wish to show. We removed the calendar view, added on the right side a preview of all the tags in the projects. They are displayed as small colorful rectangles, if the user over one of this tags the name of the tags name appears. Users can add a new tag to a version by simply highlighting a node and choosing “Add New Tag”. Below that we add it to the list of users who had worked or are working currently on these projects. By default, the first user shown is the current user, followed by the collaborators. To identify changes easily each user is assigned a color. As soon as a new user starts working on the file his changes are highlighted with that color. For information on multiple users please check 4.3.2.

As previously mentioned undo operations are now saved and displayed as faded notes, the users can still select this node to see the version and even continue working. For information on undo please check 4.3.2.

We updated the layers view to add a filter option, represented by a clock icon. By default, the system shows the complete history, the presents of the clock icon means the system shows history nodes that contain changes on that layer. Clicking on any layer turns off the clock icon from the other layers, filtering the history to only changes that are contained within that layer. User can select multiple layers to show only changes that affect those layers. If the reverse/opposite is required, then ALT should be held while pressing the icon, this will turn off the icon for the current layer or folder of layer and not update the icon of the other layers. We decided to go with this approach as showing changes for one layer is a more common use case than showing changes that exclude a layer. A global clock icon is present on top of the layer panel, this allows for easy reset of the layer filtering system so the users can easy go back to full history with one click. The layer filter can be combine with the region and tool filter to provide even finer control.

To avoid a large number of history nodes being shown the system automatically groups them by hour and day. To explore this grouping the user needs to hover over the node and scroll to reveal the content. More information about the history panel can be found in the following subsection 4.3.2.

Now that we have a high overview of how the whole system works in the following subsection we go over in detail of each component of the system, how it works and how it was designed to help the user use the design history.

History Panel

In this section we go over the details of how the history panel works, how the history is constructed and what changes have been done following the previous iteration.

We begin with how the history of the file is being conducted. As we mention in the beginning we removed the global history of the file and replaced it with a list of layer-histories. The global history is now constructed by sticking tougher the histories similar to [W. Keith Edwards, Takeo Igarashi]. To see this in action let us look at the following example. We have a project with 3 layers, Layer 1, Layer 2 and Layer 3. The user starts working on the first layer does 2 changes, we will call them A and AB. He then continues working with the second layer, making 3 changes, we will name them 1, 12 and 123. When this is finished he continues working the last layer making again 3 changes, labeled I, II and III. Finally, he goes back to the first layer making 2 last changes with the labels ABC and ABCD. The system holds 3 histories one for each layer, the first history for layer 1 will contain A, AB, ABC and ABCD. The second history will contain 1, 12 and 123 and the last history will have I, II and III. Besides this the system also holds information when the user has changed layers, who has made the change on the layer, timestamp, tags etc. With all of this information we can now construct a “global timeline” that showcase all the changes that user has done, in the following order: A, AB, 1, 12, 123, I, II, III, ABCD and ABCD. We can see this process in the following diagram:

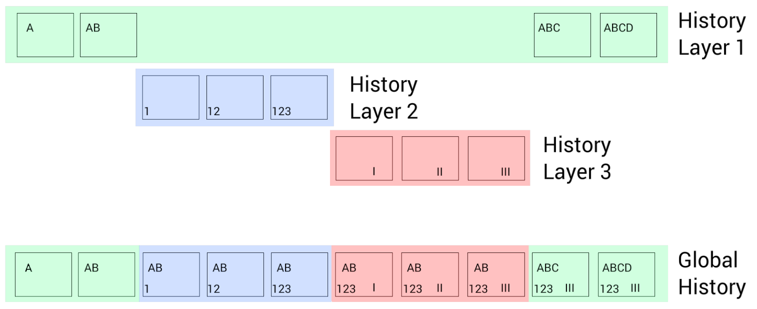


Figure 4.7: Construction of the global history

Now that we know how the history panel is being constructed let us explorer how the user can navigate it. As we have seen from the main screen the changes are now grouped, we scroll over a group to expand it, this expansion is not linear, and is made over the point of hovering, so that the user can choose which part he wants to focus on. Time information is presented for the center node of the group and also for the endings of the group. With the expansion more time information is being displayed. Horizontal scroll or the arrows keys can be used to navigate forward or backward through the history. From our past test we have seen that sometimes it is hard to see what changes have been done, so for this version we introduce a delta feature that displays for the node just the changes from the previous version. And to make sure this change is visible, we zoom in and center it in the frame. The current selected version is highlighted with a red frame, and the playback needle is next to it. This highlights change is always showcased in the main working area, selecting two or more versions can be done holding the shift key. Doing so will automatically open the comparison tool. We are looking into offering multiple version editing, this could be an interesting subject to research after the current version of tool gets implemented. The playback needle is used with the playback feather, and its purpose is to give a movie like feel to the history. The user can use it to fast scroll over the navigation, similar to what he can do with the arrow navigation. This extra element is also present to improve the visibility of the history panel, as keyboard shortcuts require an extra degree of learning. To facilitate quick movement around the different nodes we suggest the following keyboard shortcuts, holding Ctrl while using the left arrows will move to the first change of a group, right arrow + Ctrl will move to end, while holding arrow up will move to the middle. Holding arrow down will expand the group one level.

The last component of the history panel is the filtering system. The system comes with four classes of filters, each of them are applied to the result of the previous one, in a pipeline manner. This four classes of filters are, spatial filters, location filters, user filters and tool filters. The spatial filter will return changes that happen in the specific area, the location filter will return changes that happen on

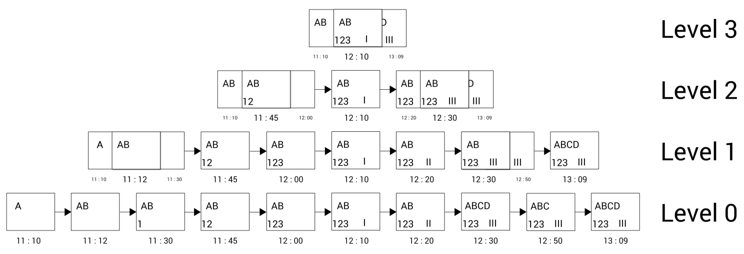


Figure 4.8: Expanding the timeline by scrolling over groups of versions

the specified layer or folder, the use filter will only show changes made by a set user and finally the tool filter will only show changes realized with that specific tool. These filters can be combined to produce even more powerful sorting methods. For example, we can ask for all changes done on a layer, in this region, by a set user with the movement tool. The output of this filter should limit the number of presented changes and with the improved scroll system should make finding a change easy. Popular changes can be tagged for even easier location, limiting the overall time the user spends on browsing. An important feature given the fact that the application implements an automatic save, creating a new node for each change.

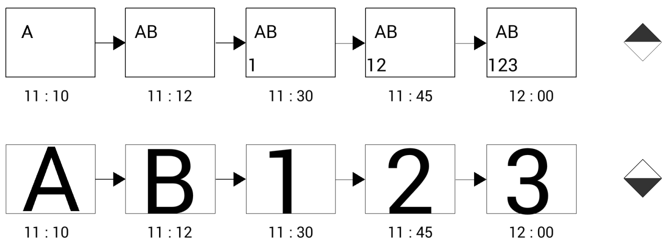


Figure 4.9: The two modes of displaying the timeline; show preview of the change or only the difference between it and the previous state

Undo Model

One important contribution, is how the system handles undos. Unlike many commercials system is doesn't have a global linear undo history, but a layer undo history. This change eliminates most of the limitations of the classic undo system describe in 2.5, but without the complexity of a full on a selective undo system. A selective undo will provide more control but at the cost of higher cost of learning, we believe this trade off provides the best of worlds with the minimal disadvantages.

When the users press the keyboard shortcut for undo command, the system will check if a layer is select, if so it will run the undo command on the layer history, a new state will be created and save to history. If no layer is selected the undo command will run on "the global timeline" similar to a classic undo command and the result will be saved in the history.

Because the undo command creates a new version in the history, the previous version is not lost, the only exception to this rule is if that users follows the undo command with a redo command, in this case to avoid any duplicate the actions of this two command are removed from the history.

Having the previous undo versions saved in the history, allows us to comeback even after the file has been closed. To difference them from the traditional save nodes, they are presented in the history panel as semitransparent nodes. Any changed to these nodes will automatically promote them to a branch, giving the user the possibility to continue work on an old idea even if it was undo.

One last feature we will explore, is the ability to simultaneously undo 2 or more layers, shift selecting two or more layer will create a "global" timeline, with undo command running on this timeline. The same effect can be obtaining by undoing on a folder of layer basics. From our observation in chapter 3 we saw that user group similar UI components in a folder option, with this feature they have an "undo" option on an object detail.

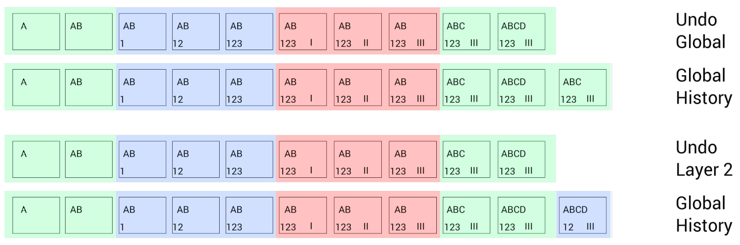


Figure 4.10: Global undo and layer undo to see how the global timeline is being built please check 4.3.2

Branching and Merging

One distinct features of the proposed version control system is how branches are handled. Unlike previously analyzed systems, branches are not created by a branch command on the history, instead they are automatically created when we choose to modify a node who already has a child. This change cuts down on the learning curve, presenting a more natural way of dealing with branches. Other changes include the removal of names for branches, changing the way branches are displayed and how users switch between them, removal of branch deletion and how merges are handled. Branches are also created when two or more users work on the same node, or their working state already has a child. At this point it is important to discuss how the system's architecture works. As we described in Chapter 2 there are two possible architectures a distributive approach and a centralized approach. The system utilizes a mixture of them, by default there has to be a root repository, this serves as a backup and as a place to centralize changes, provide analytics and to manage user access. When a user first creates a file the plugin will contact the server and initialize a repository, at the same time it will create a local repository that is a mirror on the main repository without the analytics functions. Changes are then streamed between the local repositories, queued and finally sent to the server repository and any collaborators repository. The main server repository also plays a role of rendezvous, helving the different users communicate between them. One important feature in the future for this server could be as a DRM manager, the local server could be encrypted and allow access only while client is connected to the DRM server.

The main propose of the branches is to allow for different versions of the same design. We already see the need for this feature, as many designers simply create new versions of the designs a layers, groups them in a folder and finally hides them. Only to be seen when that version is being need it. With the current branch model, once a branch is created, the design in the new branch acts like a new file. This can be a problem as we see in the following example. Let us say we are working on a banner design for a client. In this banner we have some text and some pricing information. But the client once to target multiple markets, and needs the text translated in multiple languages and the price updated to fit the local market. With current solution the designer, creates a base design in English, duplicates the text and finally updates that text with proper translation. To switch between the versions, he just hides or shows the proper text. If try to do that in our system, we will create a new branch and in this branch we will update the text. All good if don't want to do any other changes to the common part of the design. If we want to do so, we need constantly remerge this part of the file to the branch. The alternative the system present is something called layers in sync, this feature allows for a layer to modify in one branch and update in other branches. The conditions for this feature, is that the branches need to have a common ancestor, that we can refer when this background merge to happen.

One last design decision we need to discuss is the omission of branch names and branch deletions. As we mention before the aim of this tool is to provide a quick and easy to use versioning control system for designers, branch names make more sense in the world of software development, in the design workplace there are more a niche. We believe that the powerful tagging system, offers an alternative workflow. As for branch deletion, we decide to remove the possibility of removing any item from the history, we believe this safe guards the users from any work lost, and because of the strong navigation system we don't lose so much time looking through uninteresting versions.

Last thing we need to discuss is how the merging system works. In the previous iteration we used a merge system similar to [Chang], while this is sufficient for a global history it doesn't fit with the current system design. For this last iteration we implemented a new merge system.

Merges operate on a layer basics now, to perform a merge the user can drag a layer/a group of layers from a version to another version or drag a version on top of an another version. In the first case the layer/layers are compared to the ones in the new version. If they are not found, they are automatic copied, if there are found the user has one of the following two options. He can copy the new layers while keeping the old layers, or he can try a merge with the system described in [Chang]. In the second case, the system first compares the list of layers from both versions. If the layer appears in both version and contain the same data, they are copied over to the new version, if the data is different we have the same 2 choices as we presented in the previous case. Layers that are not contain in one or the other version are shown in a checkbox list, so the user can choose what goes in the new version.

One important distinction between the other systems is that a merge operation is not static, we can remerge two versions, create a new merge node that will be treated as new branch. All merge nodes are shown with a green frame, click on one of them will highlight two faded arrows showcasing the new version from which this node has been created.

Collaboration

As we mentioned in previous subsections, the system offers support for multiple users with the option of multiple uses working on the same time. This is accomplished with branches, as soon two or more users work on the same version the system creates a new branch. It is the job of the users to decide who will merge the final version. To facilitate understanding all nodes created by another user are automatically colored with the user color. As we mentioned in the navigation panel the user can filter nodes by the user who created them. The system also saves the undos of all users and any other branches or tags they have created. Because the history is built iteratively the user can see in real time what his colleague is working on. The feature of sync layers minimizes the need to merge files. We imagine a workflow where each user works on a separated layer to avoid any conflicts, at the end one of the users merges the other changes and finally tags the final versions. We first observed this workflow while interviewing designers, they mentioned that for some projects the have colleagues doing special parts of the project like in the illustration, logo design, etc. Later someone takes them and integrates them in the final design.

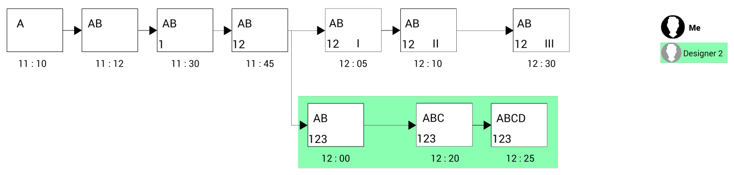


Figure 4.11: Multi-user support through branches

4.3.3 Evaluation

We went back to our initial two designers and showcased the final version of the prototype. We've shown the final design and explained with a simple paper prototype how their actions will flow in the new design. To our surprise the layer history model was not hard to understand, one of the designers even said why there is no undo action on a layer in Photoshop now. The navigation ordering changes were also welcomed, the like the idea of scrolling to see more content, but they presented a problem what happens when a group contains too many changes, how should the system handle this, should it create new subsection or change the scroll speed. This is an interesting question that needs to be tested with a prototype before it can be answered. The way branches are created presented some doubts, they were expecting a command action, but after presenting the drag and drop approach they considered this much faster to use. Dragging a layer from one version to another was the most applauded change, as they noted this is a common use case in many projects. The users suggested some changes on the tagging system. They wished to set up some profiles in the project with a template text, i.e. Version # and that the system would Show an Option to add a tag from that profile with the Name : ”populate”. This will be also useful when working on a ’milestone’ base. In the end they were happy how the project progressed and would like to try out a working prototype.

Chapter 5

Summary and future work

In this last chapter we summarize our proposed system and the contribution we bring the field of HCI. We continue by discussing the next step, implementing a limited prototype, challenges of set prototype and future directions for research based on the idea of using layers to track changes.

5.1 Summary and contributions

We started this thesis with a question about why designers aren't using the same tools as developers to manage their work. We see the challenges of building a Version Control System, why a text based approach is easier to implement and why it is not applicable to a binary format.

We explore the common language of a Version Control System and why it creates a steep learning curve and looks on how current systems manage history for graphic design. We see the advancements of having a Version Control System and how it can be used for much more than a simple backup, having applications in other use-cases, e.g. training and collaboration [Fitzmaurice].

We conducted a study and found out that most current designers are not using a complete Version Control System, instead relying for alternative solutions like Dropbox with the corresponding set of limitations.

Based on our observation, we found that Designers use Layer-Groups to organize their work. Further, we also found that some designers don't even organize the layers, and when they move them they only partially organize them. This allows the omission of features from traditional Version Control Systems such as commit messages, branching names etc. Instead, we chose a simpler approach to minimize the initial learning effort, i.e. an automatic save feature, which reduces the time needed for setup of a new repository.

The current branch model is better suited for designers, as opposed to the more "static" approach, which programmers have been thus far accustomed to. The proposed merging system is also easier to understand as it treats changes on a layer based system, instead of the full document. The ability to drag and drop part of a file from a previous change entry and merge it into the current version automatically. This would greatly benefit users, showcasing one of the big advancements of having a nonlinear file history.

We combine the 'undo' system with the Version Control, allowing users to reuse 'undo' states and offering a more flexible 'undo' system than the traditional linear model.

Ultimately we showcase how collaboration can be introduced into the system, allowing the users to simultaneously work with minimal overlap conflict and due to the "merge" system the time spent is reduced considerably.

5.2 Future work

As we previously mentioned, the next step will be to implement the system and compare it to other systems. The first task will be to assess feasibility of the auto-save system.

That is, whether or not the current navigation structure is sufficient to handle an auto-save system, with a bounded resulting numbers of changes. Other strategies could be tested: calculating the pixel delta between two changes and using the resulting value to create groups with similar delta values, or assigning a score to each type of change, grouping changes with the similar score. We have already begun modifying an open source painting application to record each change. The next step will be using these changes to test different navigation structures. As discussed in our architecture section, the system needs to be distributive, however it requires a tracker to synchronizes changes and exchange the user information and access. An iterative construction of the history is necessary in order to avoid the long time needed for saving the different versions. Changes can be arrayed and only differences could be sent when doing synchronizations.

We carried out tests of the layer undo system onto a paper prototype, which has shown promising results, very similar with [W. Keith Edwards, Takeo Igarashi] .

Even if doesn't come with Version Control System it is still a useful tool in eliminating some of the problems of the linear model without implementing a full object history model or a selective undo model.

Future work is needed to determine if the current set of filters is enough. The proposed layer filter system should help reduce the searching time and in combination with the other filters can produce fast results with minimal number of steps. The implementation will need to exploit this; hashmaps will be needed when constructing the history to optimize filter running processes. This idea of tracking changes on parts of the file instead of the hole file can be applied to other environments as well, that use similar objects.

We believe that the proposed solution would offer designers a user-friendly and easily approachable system, that would make their endeavor a more pleasant experience. We hope that this paper would inspire future research on this topic.

Appendix A

Designer VCS questionnaire

- What tools do you use?
  - Photoshop
  - Illustrator
  - CorelDRAW
  - InDesign
  - Other
- On how many projects do you work on a given day?
  - 1
  - 2
  - 3
  - More than 3
- How many files do you have to open on a given project?
  - 1
  - 2
  - 3
  - 4
  - More than 4

- Do you save old versions of the files?
  - Yes
  - No
- How do you manage old versions?
  - I save a local copy with a different name, like version1, version2 etc
  - I save my files on a file hosting service like dropbox, google drive, creative cloud etc
  - I use git / cvs / team foundation server
  - I don't save old versions
  - Other
- Are there multiple people working on the same file?
  - No
  - Yes, one more person
  - Yes, two people
  - Yes, more than two people
- On average how much time do you spend on a project?
  - Less than a week
  - A few weeks
  - A few months
  - More than year
  - Other
- Have you been forced to revert to a previous version?
  - Yes, often (more than 3 times per file)
  - Yes, sometimes (at least once per project)
  - Not really (once on some projects)
  - No
- Have you been force to revert some parts of the file to a previous version?
  - Yes
  - No

- What's the most tedious thing when reverting a file or a part of the file?
  - Locating the file
  - Merging the files (if you are restoring parts of it)
  - Keeping track of the new file resulting from the restoring
  - Other
- On average how many iterations do you need, to come up with the final design?
  - Just one
  - 2
  - 2-5
  - The design is never done

Appendix B

Versioning Control Design Exploration Interview

- For how long have you been working as a designer?
- What made you choose this field?
- Please describe a typical day at your job?
- On how many projects do you work during a day?
- How many files do you think you have to open daily?
- Is there someone else working on the same files?
- If yes, how do you collaborate?
- How do you keep track of the files?
- Do you keep different versions of the same file? Why?
- How do you review a design? What is the creative workflow?
- How do you structure your work in a file?
- Did you have to go back to a previous version? Why?
- What ist the most annoying part of this history tracking?

- Where do you get inspiration for your design?
- How you use that inspiration?
- What did you work on yesterday?

Bibliography

Hsiang-Ting Chen, Li-Yi Wei, Chun-Fa Chang. Nonlinear revision control for images.

Kirby Ferguson. Everything is a Remix. `http://everythingisaremix.info/watch-the-series/`. [Online; accessed 12-June-2016].

Tovi Grossman, Justin Matejka, George Fitzmaurice. Chronicle: Capture, exploration, and playback of document workflow histories.

Antti Oulasvirta, Sakari Tamminen, Virpi Roto, Jaana Kuorelahti. Interaction in 4-second bursts: the fragmented nature of attentional resources in mobile.

Brad A. Myers, Ashley Lai, Tam Minh Le. Selective undo support for painting applications.

Jill Butler, Kritina Holden, William Lidwell. *Universal Principles of Design, Revised and Updated*.

Donald A. Norman. *The Design of Everyday Things*.

Robert Nystrom. *Game Programming Patterns*.

Pexels. Stock man photo. `https://www.pexels.com/photo/man-couch-working-laptop-7066/`, a. [Online; accessed 30-June-2016].

Pexels. Stock woman photo. `https://www.pexels.com/photo/macbook-apple-woman-computer-7362/`, b. [Online; accessed 30-June-2016].

Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato. *Version Control with Subversion*.

Jun Rekimoto. Time-machine computing: A time-centric approach for the information environment.

Ben Shneiderman. Creativity support tools accelerating discovery and innovation.

smartinsights. Mobile marketing statistics compilation. `http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/`. [Online; accessed 30-June-2016].

Steam. Steam hardware and software survey: June 2016. `http://store.steampowered.com/hwsurvey`. [Online; accessed 30-July-2016].

Scott Chacon, Ben Straub. *Pro Git*.

Slippery Thong. Every Designer in The World. `http://imgur.com/VbWttOp`. [Online; accessed 12-June-2016].

Anthony LaMarca, Elizabeth D. Mynatt W. Keith Edwards, Takeo Igarashi. A temporal model for multi-level undo and redo.

Laura Wingerd. *Practical Perforce*.

Typeset August 1, 2016