UrbanIt: Mobile 3D Git Visualization

Portable Git visualization tool for Apple iPad

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Engineering

presented by
Ciani Andrea

under the supervision of
Prof. Michele Lanza
co-supervised by
Roberto Minelli

September 2015

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Ciani Andrea
Lugano, 11 September 2015

To my parents,
for the support they provided me through my entire life.
To all those who have always believed in me,
but especially to those who have never done.

Technology is nothing. What's important is that you have a faith in people, that they are basically good and smart, and if you give them tools, they will do wonderful things with them.

Steve Jobs

Abstract

Over the past decades, more and more research has been spent studying software evolution and this effort has made valuable data available for further studies. An undeniable source of information to study software evolution is the data gathered from Version Control Systems (VCSs). VCSs are often used to analyze software structure evolution, providing relevant information, but it is exactly the richness of that information that raises a problem: the more versions we consider, the more data we have at hand. This is why choosing a visualization tool that balances expressiveness and functionality becomes deterministic for many developers.

Along the spreading of VCSs many companies made available online services (such as GitHub[1] or Bitbucket[2]) that provide tools or interfaces to handle and represent in many forms software evolution, helping developers to better understand the history of a software system. Even if on personal workstations full-fledged applications and command-line tools are available they are not always at developers disposal. This is the main reason why, with the increasingly distributed nature of software development, having a basic repository analysis always available could be deterministic to steady check the correctness of software developing.

To support the analysis of software repositories on a mobile setting, we devised UrbanIt a tool which takes advantage from such devices, like iPad, providing a simple interface, especially designed to be used with fingers and gestures. UrbanIt brings to developer's hands a first glance in software evolution, always available even when personal workstation cannot.

UrbanIt currently supports Git[3], a popular VCS, offering to users an interface which allows the browsing of software evolution and supports commits comparison, file comparison as well as the possibility to send e-mails, screenshots and projects overview to teammates. The actual version of UrbanIt is thought to be unrelated to programming languages used by developers. This means it visualizes general characteristics, like files dimensions, files types, committers, authors and creation dates and it is thought to be easily extended, representing language-specific features such as class relationships, package analysis and many other useful features.

[1]http://www.github.com
[2]http://www.bitbucket.com
[3]https://git-scm.com

Acknowledgements

My deepest gratitude is to Michele Lanza. I appreciated his vast knowledge and skills in many areas. I have been amazingly fortunate to have a supervisor who gave me the freedom to explore on my own, and at the same time the guidance I needed. He lead me beyond my limits, teaching me how to face problems and obstacles.

I want to say thank you to my friends for their support in the last two years. For our funny moments in the open space and changes of knowledge, skills, and venting of frustration during our master program. You turned my years at USI in to something I will never forget. I have a lot of work to do to be able to return the support I received from so many people.

A special thank to all the REVEAL team, especially to Andrea Mocci, Luca Ponzarelli, Roberto Minelli and Tommaso Dal Sasso for supported me with patience and dedication. I know it wasn't been easy sometimes.

My deepest grateful to Nicolas Lusa and Nicolas Latorre. We was been the "crew of three" in the last year, during the development of our theses. We helped and supported us in the hardest moments and laughed in the best ones. Thanks you for everything.

I would also express my best gratitude to my family, that gave me the opportunity to grow as person. None of this would have been possible without your love.

What makes us who we are should be glorified.

Thank you all.

Contents

Figures

Chapter 1

Introduction

During team software development, one of the most important aspect is synchronizing the developers' work. Indeed, as teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the developers to be working simultaneously on updates. VCSs are meant to handle code changes to a set of data over time, tracking each modification as an incremental series of versions[1]. Each version embeds informations and data about files modified, added or deleted as well as informations about the author of these updates (e.g. e-mail, username and date of the modifications).

These "streams" of data created by the VCSs open new possibilities for code visualization.

Developing and deploying from small to big software projects needs to have always under control the general structure of the system. This is the main reason why modern CVS tools often provide, as their integrated parts, softwares able to visualize the system evolution.

Just to make some examples, GitHub and Atlassian provide additional softwares such as respectively the *GitHub App*[2] and *Source Tree*[3] for desktop computers. If, in one hand, these tools provides a great visual solution for developers, often becoming a "must-have"; on the other hand they are tied to the limited availability of the workstations. This becomes the main limit of such kind of applications and the main reason why is so important to provide a tool that permits developers to access these features everywhere, even when personal workstations are not available.

In a world where mobile devices are becoming day after day more powerful and dynamic[19] we decided to create URBANIT, a tool that aims to make basic code visualization features always available.

URBANIT takes advantage from the great availability that modern portable devices make available to developers making always accessible in their hands an overview on the system evolution, providing also some sharing features that are always useful in a team of developers.

In order to make URBANIT useful and effective, we focused on a set of key ideas:

1. *Easy:*

   To create an interface that is really useful and intuitive for users, we closely followed

[1]Also known as "commit"

[2]`https://desktop.github.com/`

[3]`https://www.sourcetreeapp.com/`

modern Apple guidelines for the best results, as explained in 1.1.1. By choosing an intuitive 3D city representation, with building (i.e., files) and districts (i.e., folders) UrbanIt makes available to users a fast and easy representation of the system evolution.

2. *Portable:*
   UrbanIt aims to achieve this goal by implementing a native iOS app that runs on Apple tablets, providing an embedded, optimized solution, to share screenshots, summaries and comments with teammates.

3. *Generalized architecture:*
   UrbanIt is structured and thought to be general: by analyzing general files features (like committer, author, commit date, file dimension, file type etc...) UrbanIt is able to represent each kind of versioned project, independently from his nature (programming language, file types etc...).

4. *Extensibility:*
   UrbanIt is extensible. The 3D representation provides multiple “dimensions” available to be assigned to multiple kinds of software features. For example is possible to extend UrbanIt by assigning software-related features to the width or length of each building, or by associating colors to different features (currently colors are associated to files types).

1.1 Our Approach in a Nutshell

Developing for mobile devices often requires different skill sets and familiarity with different runtime environments than those more commonly used on desktop PCs[29]. Mobile devices have many restrictions both on computational power, power consumption and available storage space and they must taken in consideration in advance; before starting to write code, or even before to think about the project at all.

Even if mobile devices power is increasing every year we still have a bunch of limitations compared to a desktop pc. This is why UrbanIt is thought optimized from the ground up: this implies to use native code programming (Objective-C), ad-hoc 3D libraries (SceneKit[4]) developing and testing in a native environment (i.e., Xcode and Instruments).

The different mindset a mobile application needs is not only related about technical aspects but it is also related to *how* a mobile application is used.

UrbanIt is thought as mobile application. This means that every component and every step must be adapted taking in mind the limitations just explained. Critical steps, such as algorithms or files transfers, must be developed and defined to be as lightweight as possible; the interface must be as immediate as possible as well as easy to use with a finger instead of a mouse. The differences between a mobile and a desktop application influenced deeply the developing of UrbanIt and was been mandatory having an overview about these approaches in advance. To accomplish these goals we based our approach on the Apple Guidelines[4], especially on three main points (we will discuss the idea further on 1.1.1):

[4]SceneKit is a 3D framework developed by Apple since iOS 8 to provide well a very optimized solution to represent 3D object in Apple devices, such as iPad and iPhone, shaping around the hardware of these devices the OpenGL libraries.

- *Apps respond to gestures, not clicks*: People make specific finger movements, called gestures, to operate in the multi-touch interface of devices. For example, people tap a button to activate it, flick or drag to scroll a long list, or pinch open to zoom in on an image. People are comfortable with the standard gestures. They can choose to use a *single-tap* or a *double-tap*, as well as a *swipe* or using a *pinch* gesture if needed or a *touch and hold*. It is up to the developer use these gesture in a way the people feel natural and intuitive. In example for URBANIT we chose to use a pinch gesture to allow user zoom-out of the 3D view. Each building has associated multiple gestures to recognize whether the user is single-tapping, double-tapping or touch-and-hold then every of these gestures are reflected in URBANIT with a different behavior.

- *Apps are used differently compared to desktop application*. Users want clear interfaces, with big buttons easy to tap (they do not have a mouse available). Lists are scrolled with fingers not with a mouse's scroller. Each aspect of a mobile application must be thought to be used with fingers.

- *Limited user time.* Frequently mobile applications are used for a limited time by their users, much shorter respect to desktop applications. This is the reason why they must accomplish the user needs in as less "taps" as possible. When you design a mobile application you need to have clear in mind the purpose. If the user need too much steps to do something, maybe you need to rethink something.

1.1.1 Mobile Application approach

Interacting with mobile devices users expect to have applications that follow the same *Design Principles* among them, this means that they must follow a sets of written (and unwritten) rules that permits to have a common idea of interaction between the apps. Just to make some examples, Apple summarizes these design principles in five different aspects [4]:

1. *Aesthetic Integrity:* Aesthetic integrity do not measure the beauty of an app's artwork or characterize its style; rather, it represents how well an app's appearance and behavior integrates with its function to send a coherent message.
   People care about whether an app delivers the functionality it promises, but they are also affected by the app's appearance and behavior in strong and sometimes subliminal ways. For example, an app like URBANIT that helps people perform work-related tasks can put the focus on the task by keeping decorative elements subtle and unobtrusive and by using standard controls and predictable behaviors.

   On the other hand, in an app that encourages an immersive task, such as a game, users expect a captivating appearance that promises fun and excitement and encourages discovery. People do not expect to accomplish a serious or productive task in a game, but they expect the game's appearance and behavior to integrate with its purpose.

2. *Consistency:* Consistency lets people transfer their knowledge and skills from one part of an app's UI to another and from one app to another app. A consistent app is not a slavish copy of other apps and it is not stylistically stagnant; rather, it pays attention to the standards and paradigms people are comfortable with and it provides an internally consistent experience.

3. *Direct Manipulation:* One of the key feature of a mobile app is the Direct Manipulation. When people directly manipulate onscreen objects instead of using separate controls to manipulate them, they're more engaged with their task and it is easier for them to understand the results of their actions.

4. *Feedbacks:* Feedback acknowledges people's actions, shows them the results, and updates them on the progress of their task.

5. *Metaphors:* When virtual objects and actions in an app are metaphors for familiar experiences ,whether these experiences are rooted in the real world or the digital world, users quickly grasp how to use the app.

6. *User Control:* People, not apps, should initiate and control actions. An app can suggest a course of action or warn about dangerous consequences, but it is usually a mistake for the app to take decision-making away from the user. The best apps find the correct balance between giving people the capabilities they need while helping them avoid unwanted outcomes.

The approach used to build UrbanIt was based to follow these guidelines to provide the best user experience without compromise usability and features.

1.2 Contributions

In this thesis we present UrbanIt a tool that starting from the idea of a portable tool (with the related *design principles* discussed previously) trying to take advantage from the power, usability and convenience of a native mobile application and evolution's information that a CVSs provide, mixing them together to develop a new experience is this field.

UrbanIt provides the following features:

- Automatic Git repository reading from a given public URL or Zip file.
- Embedded Git repository visualization with 3D interaction.
- Visual comparison of commits.
- Single file evolution with plots and values.
- Multiple files evolution with comparing plots and values.
- Commit information with embedded e-mail module to contact authors and committers.
- Fast and easy screenshot sharing.
- Multiple Git repository tracking with single application.

1.3 Structure of the Document

- In **Chapter 2** we will discuss about related work, software that use visualization techniques applied to similar UrbanIt topic.
- In **Chapter 3** we discuss how the main views of UrbanIt work and how the results are achieved.
- In **Chapter 4** we discuss the general structure of UrbanIt and how the different parts of the tool interacts each other, as well as the main models used by UrbanIt to build the final visualization of the repositories.
- In **Chapter 5** we face a real example of usage of the tool. We discuss how start from a repository and visualize it with UrbanIt .
- In **Chapter 6** we discuss the conclusion regarding our work and about possible future extension of UrbanIt internally or with additional softwares.

Chapter 2

Related Work

Writing code is a human and mental activity. The more familiar we are with a program, the easier it is to understand the impact of any modification we may want to perform [30].

The importance of visualization techniques is undeniable in all field of science, and also in various software engineering activities including comprehension and collaborative exploration. Diagrams, charts and other graphical elements are often used to present quantitative and qualitative properties and their relations. These tools use simple and abstract graphical primitives which could not be found in real world like straight lines, points and circles. They are able to express some attributes of the software successfully but are less useful to present more complex, many dimensional contexts[7].

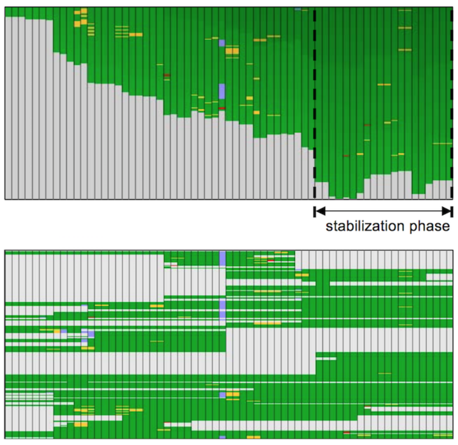


Figure 2.1. CVSScan line status visualization. File-based (top) and line-based (bottom) layouts

In the last years many visualizations were been proposed by researchers and engineers to give the best overview, depending on the domain specific requirements, including the static structure [31] or evolution of repositories, such as the CVSScan [33], shown on figure 2.1[1].

[1]Image taken from "*CVSscan: Visualization of Code Evolution*"[33]

The objectives of software visualization are to support the understanding of software systems (i.e., its structure) and algorithms (e.g., by animating the behavior of sorting algorithms) as well as the analysis and exploration of software systems and their anomalies (e.g., by showing classes with high coupling) and their development and evolution. One of the strengths of software visualization is to combine and relate information of software systems that are not inherently linked, for example by projecting code changes onto software execution traces[9].

Riva *et al.* proposed the usage of colors to distinguish how the stability of the architecture changes on the time among the releases[22], U.Erra and G.Scanniello opted for a different approach using the forest metaphor to ease the comprehension of object oriented software systems[15].

Rysselberghe and Demeyer proposed another kind of visualization[28] related to version control system, meanwhile the *revision towers* of Taylor and Munro [11], used the information gathered from the VCSs to represent the software evolution. Pioneering work on 3D visualization was proposed by Reiss [27].

2.1 The city metaphor

Code analysis often require to investigate in the system's code base to find and understand program elements (e.g., classes, methods) that are pertinent to a specific feature[34]. A good visual metaphor is the key for an effective representation. Many 3D visualizations are undoubtedly appealing, but fail at communicating relevant information about the system and thus fail at supporting program comprehension tasks[36].

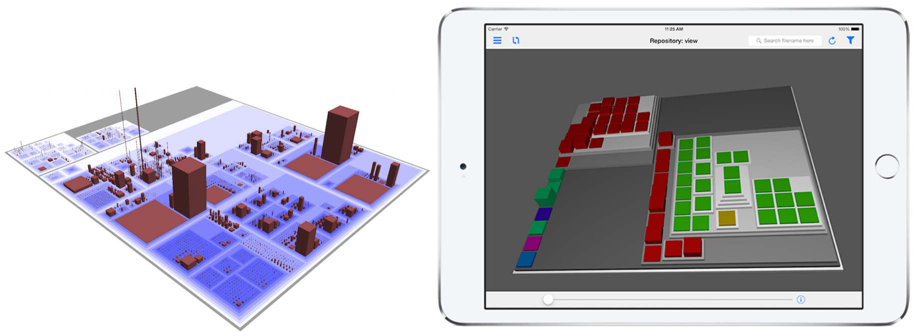


Figure 2.2. The Code City city metaphor compared to the UrbanIt view

URBANIT uses the *city metaphor* to represent the folder content in a visual way. City metaphors were already seen on *Software World* proposed by Knight [26] and Charter *et al.*'s *Component City*[10] while Marcus *et al.*'s sv3d [18] and Balzer *et al.*'s *Software Landscapes*[8] use a similar 3D metaphor to visualize single versions of software systems. Other city metaphor can be found on Ducasse *et al*.[12] in the *SimCity* game to express challenges behind software evolution, which remained an idea. Another valuable example could be the one of Langelier *et al.*'s Verso [21] that used 3D visualizations to display structural information, representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around classes.

The city metaphor was also been associated to localize design problems during the development[38].The effectiveness of such kind of visualization was widely proofed, allowing the user to gather immediate information about the context.

2.1.1 Code City

The core idea of UrbanIt leans on the idea of R.Wettel and M.Lanza called *Code City*[37], shown in figure 2.2[2]. Wettel and Lanza used the city metaphor to build a visualization tool that has as goal to support the analysis of large-scale object-oriented software systems.

From a user interface perspective, CodeCity is a sovereign (i.e., it requires the entire screen[32]) and interactive application. To allow experimenting with visualizations, CodeCity provides a high customizability, by means of view configurations (i.e., property mappings, layouts, etc.) and later through scripting support[35]. The tool, which is freely available, has been released in March 2008 and has reached its fifth version (i.e., 1.04).

Code City focus its functionality by representing systems as cities, where classes are depicted as buildings, and packages as the districts of the city. Although the common idea in between *Code City* and UrbanIt they differ on many aspect:

1. *Usage: Code City* is a visualization tool to support software analysis and gather information from the classes themselves. UrbanIt gather information from Git and it is used to represent the software evolution starting from this source.

2. *Target Device: Code City* is a desktop application meanwhile UrbanIt is thought to be portable, running natively in Apple tablets.

3. *Architecture and programming language: Code City* is build on top of Moose[13] which provides the implementation of the language-independent meta model needed by the tool. UrbanIt builds his own models representations as parts of the tool itself.

4. *3D engine:* Differently from *Code City* that uses open source OpenGL implementation as 3D engine[1] UrbanIt uses an hight optimized engine build on top of OpenGL-ES especially developed for Apple products, developed by Apple itself, called *Scene Kit*[6].

2.2 Colors usage

Another important aspect of UrbanIt is the idea behind the usage of colors. For example a visualization that base its functionality (also) on colors is the one made by A.Marcus *et.al*[24]. They use the *SeeSoft metaphor*[14] to create a new 3D representation for visualizing large software systems.

In the work of Marcus *et.al*, color and pixel maps are used to show relationships between elements of a software system (figure 2.3[3], meanwhile in UrbanIt colors are mainly used to

[2]Image taken from "*CodeCity*" of R.Wette and M.Lanza[37]
[3]Image on the left taken from "3D Representations for Software Visualization"[24]

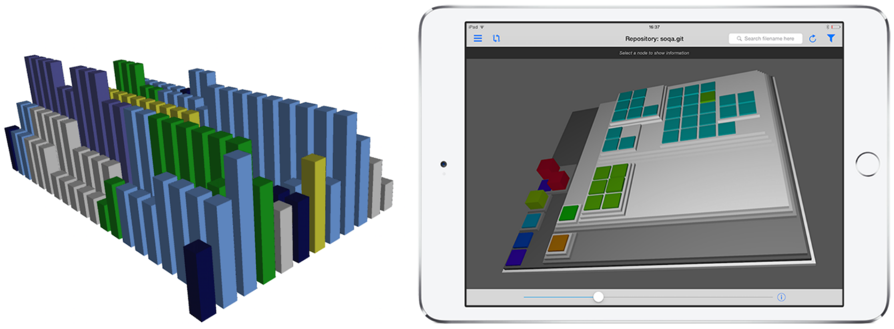


Figure 2.3. On the left the Marcus' implementation[24] where colors are mapped to control structure types, on the right the UrbanIt coloring associated to file types

separate different types of files inside a tracked repository, then is up to the user deciding which of them visualize, depending on the needs.

Guillaume Langelier used colors in his visualization[21] to express the "dangerousness" of software coupling during the development, by associating the natural association of danger to the color *red* and to the *blue* an acceptable coupling between classes.

2.3 The HisMo data model

HisMo[17] is a meta-model in which history is modeled as an explicit entity. *HisMo* adds a time layer on top of structural information, and provides a common infrastructure for expressing and combining evolution analyses and structural analyses.

Taking in consideration only a single version of the system we miss important information related to the evolution of the system. Therefore we used in UrbanIt the *HisMo* data model in order to keep information about multiple versions of a system.

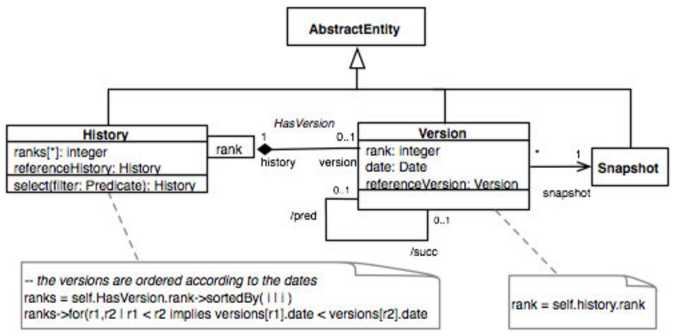

AbstractEntity
History
ranks[*]: integer
referenceHistory: History
select(filter: Predicate): History
rank
HasVersion
1
0..1
history
version
Version
rank: integer
date: Date
referenceVersion: Version
*
1
Snapshot
snapshot
0..1
/pred
0..1
/succ
-- the versions are ordered according to the dates
ranks = self.HasVersion.rank->sortedBy(i | i)
ranks->for(r1,r2 | r1 < r2 implies versions[r1].date < versions[r2].date
rank = self.history.rank

Figure 2.4. The overall structure of HisMo[17]

As *iPlasma*[25] used *HisMo* to have a model that permits the tool to perform searches about logical couplings between modules, *HisMo* represented a valuable addition also to UrbanIt which permitted to search in the history of commits keeping the application fast and building in real time the list of differences founded.

Daren Fang, Hongji Yang and Lin Liu with "*Evolution for the Sustainability of Internetware*" used the *HisMo* to proof how transformation-based techniques rely on transformation tools which are driven by predefined transformation rules or templates, working at either the source code or the meta-model level[16].

Other application on *HisMo* can be found also in the work of Philip Makedonski, Fabian Sudau and Jens Grabowski "Towards a Model-based Software Mining Infrastructure". They use the *HisMo* meta-model to enable the incorporation of historical information related to artifacts[23].

Chapter 3

Main Views

The key points of UrbanIt are the views. They must provide a fast access to the information shown, giving to users at the same time a complete toolset to accomplish the tasks. Every view must be thought in terms of functionality and efficiency. Among the main views of URBANIT we find:

- **URBANIT view**:
  The core view of URBANIT that provides the 3D visualization.
- **Difference view**:
  The view that permits to visualize, using a pair of URBANIT views, the differences between two commits.
- **Evolution views:** There are two different types of evolution view:
  - **Single Difference View:**
    The view that shows the evolution of a single node.
  - **Multiple Difference View:**
    The view that shows the evolution of a set of nodes.

3.1 UrbanIt view

The URBANIT view is the central part of the mobile app interface. It provides the core representation of URBANIT. Given a model (see 4.2.2) it is able to represent it as an interactive 3D model. It is decoupled from the rest of the application.

The URBANIT view provides a delegation pattern for events and observed values for properties. With this approach the controllers can link interface elements to the corresponding code. The view controller of the URBANIT view is able to keep tracks of the highlighted nodes, the selected nodes and multiple interactions, such as a long touch on a specific node, a single tap or double tap. The underneath delegate will implement the right behavior for each interaction.

The URBANIT view makes massive use of the SceneKit framework[6] developed by Apple, released for iOS devices since iOS 8.

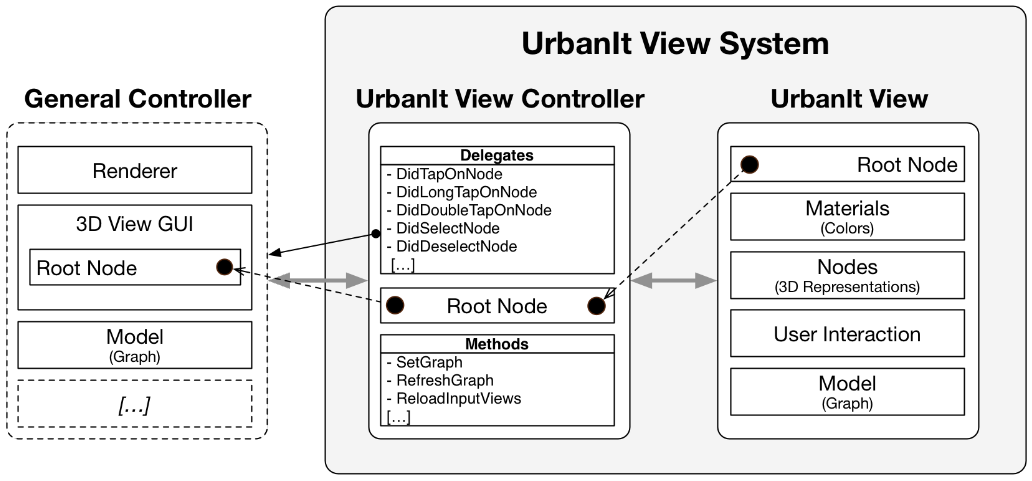


Figure 3.1. The UrbanIt view structure, with the controller between the General Class and the Interaction View.

SceneKit is an Objective-C framework for building apps and games that use 3D graphics, combining a high-performance rendering engine with a high-level, descriptive API. SceneKit supports the import, manipulation, and rendering of 3D assets. Unlike lower-level APIs such as OpenGL that require developers to implement in precise detail the rendering algorithms that display a scene, SceneKit only requires descriptions of developers scene's contents and the actions or animations they want it to perform([20]).
Using SceneKit has multiple benefits respect using directly OpenGL:

- It as highly optimized for Apple devices. With SceneKit you can have a huge number of polygons moved by an optimized renderer.
- It's possible to write SceneKit objets directly in Objective-C, with the same IDE. The objects, materials and particle effects can also be visualized as previews in developing time.
- It expose to the developer the Core Animation frameworks ([20]), allowing to assign animation to the node's properties. We used this feature to add the fading animation on selection, highlighting and hiding.

On the other hand the user interaction is provided by a combination of standard iOS interactions with some additional math. The main challenge about the URBANIT interaction were the coordinates. For each interaction the system provides the *x*,*y* coordinates among the entire device screen. That coordinates must be translates to the selected coordinate object on the URBANIT view. The main step done by the URBANIT view controller for each interaction are the following:

1. Receive the "world" coordinate of the screen.
2. Translate the given coordinate to the coordinate inside the view

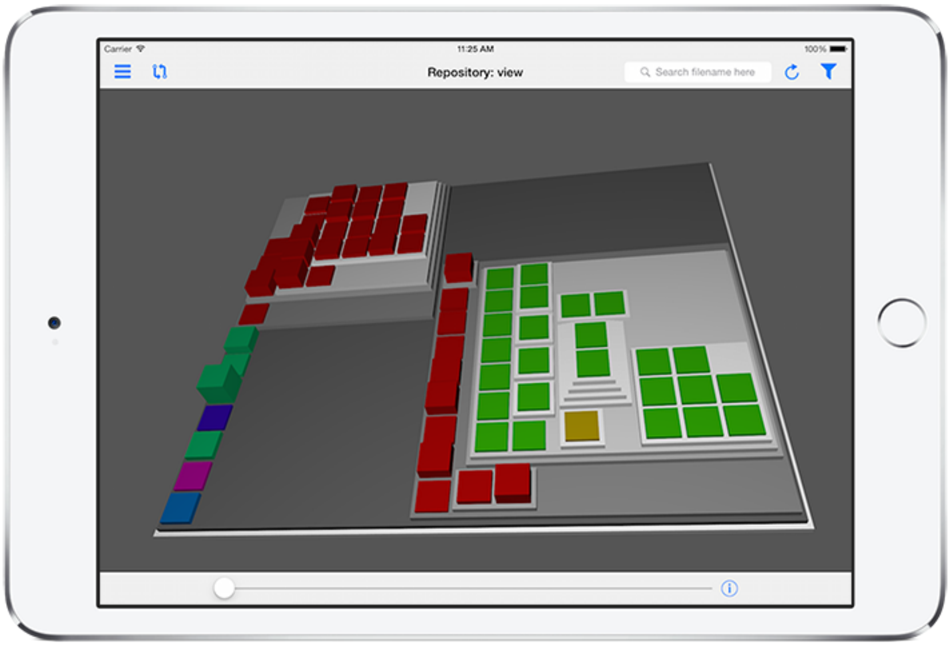


Figure 3.2. An example of UrbanIt view with a repository displayed

3. Retrieve the object that is currently displayed in the translated coordinate
4. Call the delegate's methods associated to that particular event (long tap, double tap etc...) passing the selected event *and* the current controller that received the action (allowing to have multiple URBANIT view displayed in the same time).

The other main challenge we had developing the URBANIT view was the lay-outing algorithm. We needed an algorithms that was able to find the correct position for each building, in real time, as fast as possible, using as less memory as possible and saving as space as possible. To achieve this goal and assemble the final 3D structure we decide to opt for a customized implementation of the *Rectangle Packing* algorithm, applied recursively to all the nodes.

3.1.1 The rectangle packing algorithm

The idea behind the rectangle packing algorithm is to place smaller rectangles inside a bigger container rectangle as tightly as possible.

The implementation we chose uses the concept of "free rectangles" within a give main rectangle. The packed rectangles are always placed in the top left corner of some free rectangle that they completely fit into (see figure 3.3). To get very close to optimal packing the top most of the left most free rectangles the packed rectangle fits into is selected for placing.

In the algorithm initially there is naturally only one "free rectangle" that is the main rectangle itself. After packing the first rectangle in the original free rectangle is removed and there

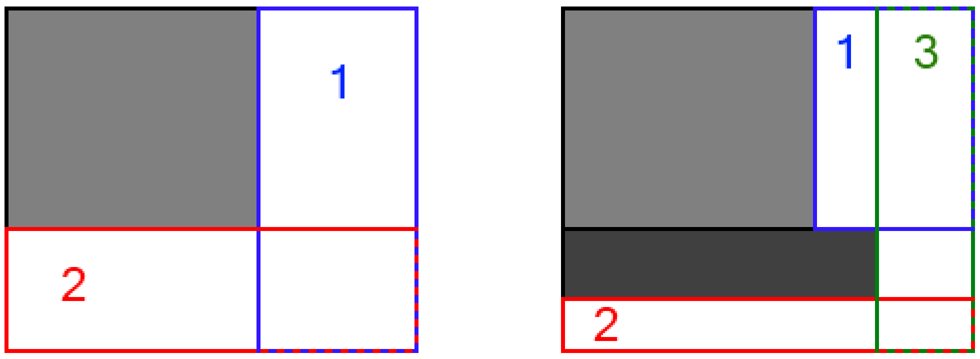


Figure 3.3. The first two steps of the Rectangle Packing algorithm adopted in UrbanIt

are from zero to two new free rectangles, if the packed rectangle is as big as the container there are no more free rectangles, if the packed rectangle is as wide or as tall as the container there is one free rectangle either below or on the right side of it and if the packed rectangle is smaller there is one free rectangle below it and one on it is right side. Packing next rectangle happens the same way, also any other free rectangle the packed rectangle intersects is cut into new smaller free rectangles around the packed rectangle. During the process all the free rectangles that are fully contained by another free rectangle are removed.

Once we have a working algorithm for the rectangle packing we linked the 3D objects to the algorithm rectangles using the unique identifier of the nodes. In this way once the algorithm arranged the rectangles, the final result can be used to position the 3D object in the space.

Of course the final step to obtain the result of a city is to make the algorithm working recursively. If in the starting point the first free rectangle is a fixed empty root rectangle (used also as root of the 3D scene kit hierarchy) in the recursive step the districts become the base free area for the contained nodes.

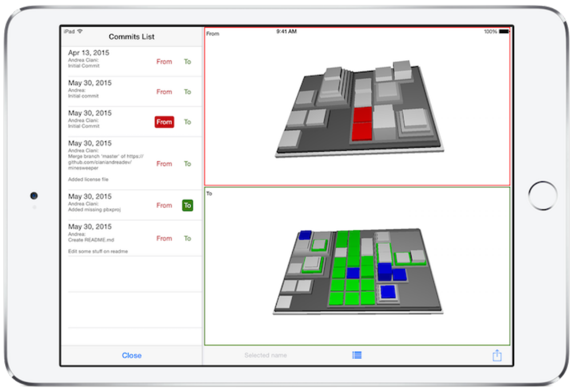
The result obtained by the algorithm was fast and reliable but presented a final problem: the area needed by the districts to contains all the nodes is not known in advance. In one hand the district area cannot be computed until all the buildings are not layouted, on the other hand the layout cannot be computed if we do not have a container.
In order to have a result the was enough appealing and that minimize the amount of empty area we decided to opt for the following solution:

1. We first give to the system a basic district dimension to compute the first layout.
2. We give this basic container to the rectangle packing algorithm to have a result.
3. If all the rectangles fits in the given container we proceed to the next step, otherwise the increase the dimension of the container and we run again the rectangle packing.

4. Once all the buildings are contained in the district (the container) we calculate the bounding box of the contained buildings.
5. We cut away the unneeded space.
6. We save in a temporary buffer the dimension of the container (as a pair "container id" and dimension) to reduce running time in next steps for the next step

3.2 The Difference View

The *Difference View* (figure 3.4) is used to compare two different commits in URBANIT. It is build from an iOS *SplitView*[1] where on the left side is shown a the list of commits ordered by commit time and on the right side there is the comparing result view.



Apr 13, 2015
Andrea Ciani:
Initial Commit
From To

May 30, 2015
Andrea:
Initial commit
From To

May 30, 2015
Andrea Ciani:
Initial Commit
From To

May 30, 2015
Andrea Ciani:
Merge branch 'master' of https://github.com/cianiandreadev/minesweeper
Added license file
From To

May 30, 2015
Andrea Ciani:
Added missing pbxproj
From To

May 30, 2015
Andrea:
Create README.md
Edit some stuff on readme
From To

Figure 3.4. An example of Difference View showing changes between two commits.

The user can select two commits from the menu (the "from" that will be the starting point for the comparison) and the "to" (that will be the ending point of the comparison).

The *Difference View* is obtained using two different instances of the URBANIT *view* (discussed on 3.1) by giving two different sources (the "from" and the "to" commit).
The diff view compares all the paths of the contained nodes before and the dimension of each correspondences then. All the deleted, added and edited nodes are then colored with the corresponding difference.

Files (or directories) that have been removed are colored in red on the upper view (representing the older version), while files that have been added are colored in green in the bottom view (representing the newer version To). All modified files are colored in blue.

For instance, given two commits (see image 3.5) where some changes are happened, the c view is able to search the differences by traversing all the commits in the model (see section

[1]i.e. a view split in two subview; one for a menu, the other (called detail view) is usually used to represent what is selected in the menu

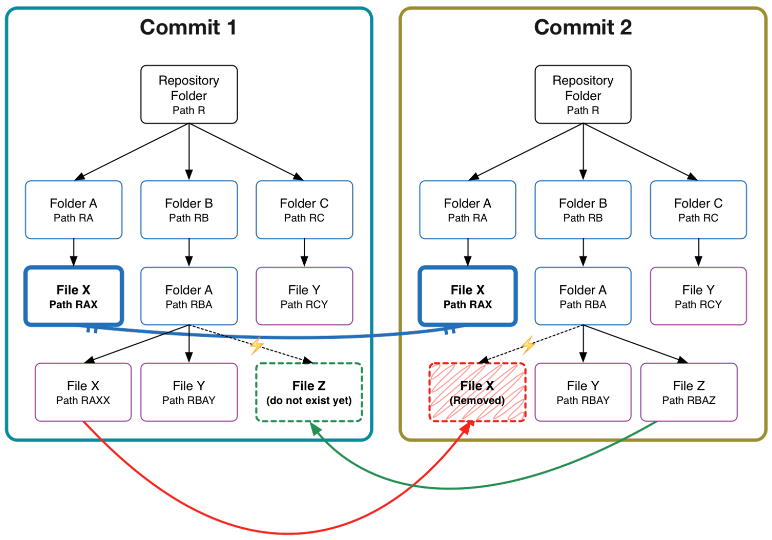


Figure 3.5. The idea behind the difference view

4.2.2).

In order to find all the possible differences (add, edit, remove) the difference view performs two horizontal search (see image 3.5), once for each “direction” (i.e., on image 3.5 would be from commit 1 to commit 2 then from commit 2 to commit 1). As shown on figure 3.5 the algorithm is composed by two steps:

1. First traversal search (from commit 1 to commit 2): The difference view searches for the following differences:

    (a) **File deleted:** (in red on image 3.5) Files that have a path in the commit 1 and are no longer reachable with the same path on commit 2.

    (b) **File edited:** (in blue on image 3.5) Files that exist both on commit 1 and commit 2 *but* they have different sizes.

2. In the second step the *Difference View* changes the order of the search (from commit 2 towards commit 1).

    (a) **File Added:** (in green on image 3.5) Files that are present on commit 2 but are not present in commit 1 means that was been added later.

For the visualization we decided that a moved file or directory means a directory that is deleted from a path and added to another one. This is equivalent to an deletion followed by an addition[2].

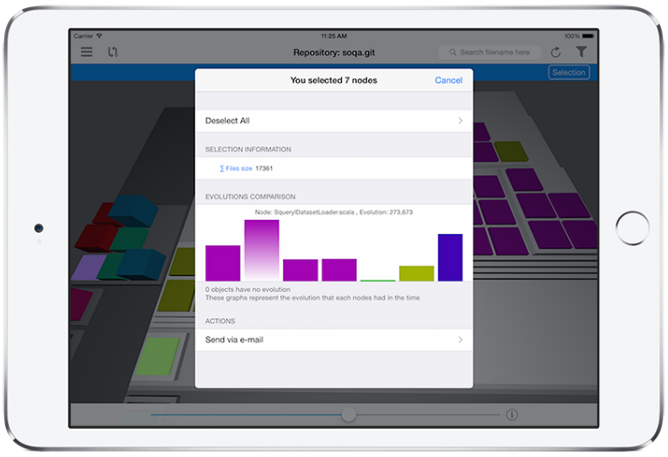
Since every filename founded in the previous algorithm is saved in a specific array the *Difference View* is also able to use the result not only to color the buildings but also to present a

[2]This is also the standard behavior of any Git repository

written list of the file names and writing a pre-configured e-mail with a listing of all the names of the modified/added/deleted files alongside a screenshot attached for a fast sharing of the results. This will be explained in more detail with a practical example on 5.2.4.

3.3 The Evolution Views

In UrbanIt exist two types of Evolution Views. Both perform a search on the history of the evolution (see *HisMo* on 2.3) but one is used to perform the search within a single element, the other one is used to search on the evolution history of a set of elements. Depending on which is used the result will be shown differently.



You selected 7 nodes

Node: SqueryIDatasetLoader.scala , Evolution: 273,673

0 objects have no evolution
These graphs represent the evolution that each nodes had in the time

Figure 3.6. The multiple selection view. Each color of the graph represent the selected node color

When the user analyzes a single node the evolution view will show a chart showing how the selected node is evolved, as well as other additional useful information about the element, such as the current dimension, the relationship with other nodes and the containing folder.

On the other hand, when the view is on multiple selection mode the Evolution View shows instead a comparison between all the selected elements. In this case the chart will represent how each node is evolved respect to the others. The evolution shown in this case is not in terms of dimension but in terms of the absolute variation of dimension:

$$\sum_{t=0}^{m} |E(n,t)| \tag{3.1}$$

Where $E(n,t)$ is the *Evolution* (variation of dimension) of node n on time t.

$$E(n,t) = D(n,t) - D(n,t-1) \tag{3.2}$$

Where $D(n,t)$ is the dimension of the node n at time t.

From 3.1 and 3.2 we have:

$$\sum_{t=1}^{m} |D(n,t) - D(n,t-1)| \tag{3.3}$$

The formula 3.3 is applied to each node of the history (that is obtained from the HisMo model, as explained in 2.3), for all the node dimension (e.g, the file size).

A visual representation of the result is on figure 5.9 and the *Soqa* example, in chapter 5.2.3.

Chapter 4

The UrbanIt Architecture

In this chapter we will discuss about the internal structure of URBANIT.

Starting from the server until the mobile application, trough the internal model representation URBANIT is thought to be extendible. This was a mandatory feature that extends the visualization power of URBANIT.

Even if currently URBANIT shows general Git information (e.i., the author, the committer, the file size etc. . .)it also allows to develop more specific visualization.

To fulfill these requirements we needed an architecture composed by decoupled components and well defined communication protocols.

In URBANIT every component can be extended easily, starting from the Java Git reader (see 4.3), the server side or the iOS application. We will discuss every component of the URBANIT structure later in this chapter.

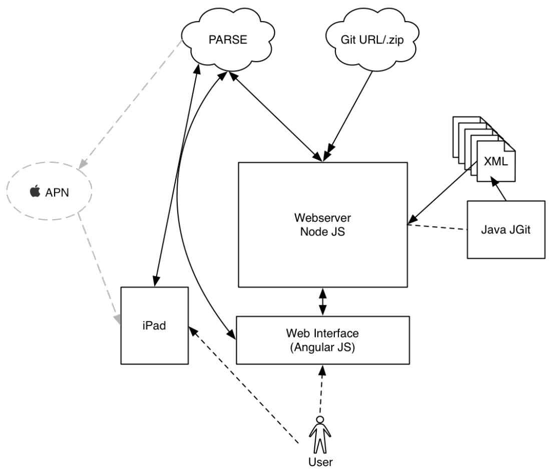


Figure 4.1. The general structure of UrbanIt

4.1 The Structure in general

The UrbanIt structure is presented like in figure 4.1:

As shown in figure 4.1 the user can interact both with the front-end (image 5.1) as well as with the iOS application (image 5.1.4). The first one allows to add/remove a repository and see the list of which was been already uploaded. Trough the front-end new users can create the accounts.

The second part of the UrbanIt architecture is the back-end. Here the repository are downloaded and then analyzed by the Java Git Analyzer application (see4.3). Hence, the models(4.2.1) are created and they become ready to be used by the final application.

On the left side of image 4.1 is also shown a semi transparent part representing the connection between the *Apple Push Notification service* (APNs) and the Parse server. This is a very complex part that require some extra features[1] that we preferred not to implement from scratch.

4.1.1 The Backend

The backend is the part of UrbanIt responsible to generate the files needed by the application to build the final visualization.

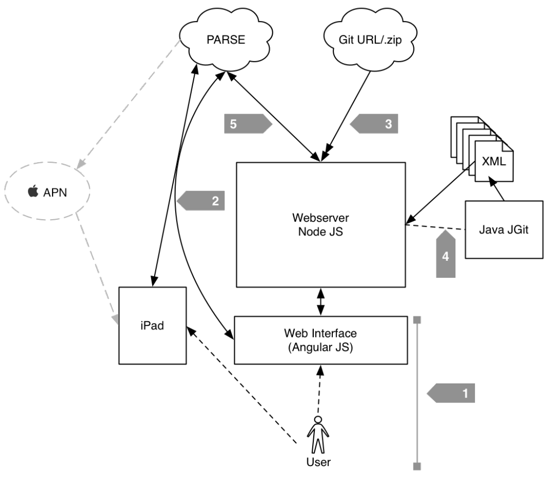


Figure 4.2. The structure of UrbanIt backend with the step performed in order

Referring to image 4.2 we will now discuss separately each step performed by UrbanIt at server side.

1. **The user login/register on the Web Interface:** At this step, since the web interface in developed in *Angular JS*, the communication with the server are significantly reduced.

[1]APN require that the server emitting the push notification has a fixed IP, a registered SSL certificate and that provides a custom implementation of push notification protocol[2].

The requests are directly computed by the web browser.
Every time a user adds or remove a repository the changes are updated on Parse. The communication between the Web Interface and the URBANIT are limited to the information about the git repository to add (see point 3).

2. **Communication with Parse:** Altrough most of the the operations needed to build the web interface are done on the client side, the login process still requires the credential being checked by Parse. These request are sent directly from the web browser to Parse (via TLS 1.2). If the credential are accepted a new token is created and saved locally on the client web browser.
   The token will be kept alive for a week or until the user manually logout from the web browser. The token is mandatory for each changes requested on the user account and on the Parse server.

3. **Add a new repository:** In this step the user provides a repository to the server, trough a `.zip` file or with a public repository URL. Depending on the source the user has. The server will unzip the file (if the `.zip` version in provided) or it will clone the public repository (if the URL version in provided).
   Each repository is saved locally on the server until the user choose to delete the repository. When the repository files are ready will be then invoked the Java Git Analyzer (step *4*) or, if something goes wrong, a new request is sent to Parse to send a push notification to the user.

4. **Java Git Analyzer:** The Java Git Analyzer is a tool that we develop using the *JGit* library[2]. This tool is able to read a `.git` folder contained in a given path (passed by the server at step 3), to analyze it and retrieving information about the committer, commit time, files dimension etc... (see 4.3). When the process ends a callback is sent to the server, or, if an error occurs, a push notification is sent to the user device.

5. **File uploading:** Once the server receives the callback from the Java Git Analyzer it starts the uploading of the generated files to the Parse server. Together with each file the server attaches additional metadata about the user. This is the way how the files are related to the requesting user (see section 4.2.4).

4.1.2 The Application Structure

As we can expect the operations performed by the iOS application are less then the operations performed by the server. This is because we prefer to relieve the mobile application load.

We can split the operations made by the mobile application in to three different sets:

1. **The login operations:** *Step 1* on figure 4.3. In this step the application mirrors what is done in step 1 by the backend previously. It requests username and password from the user and sends them to the Parse server, that will eventually response with a token.

   In this step, respect what was done by the server, the mobile application also registers the user device to the APNs.

[2]`https://eclipse.org/jgit/`

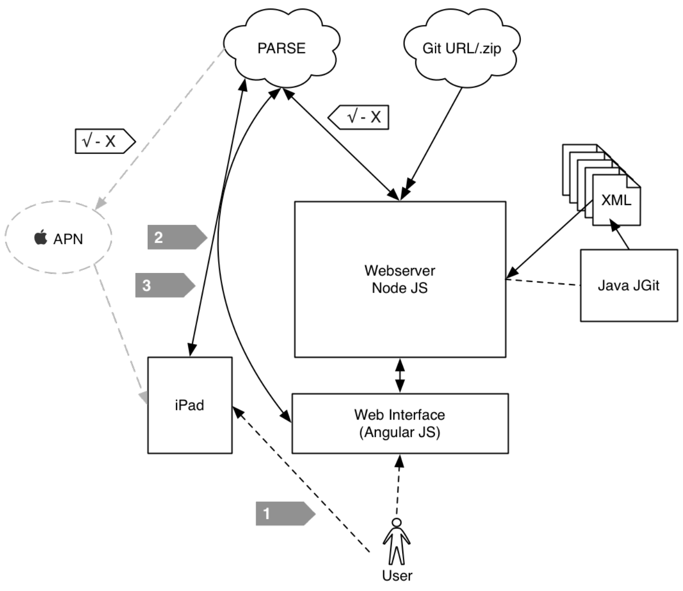


Figure 4.3. The structure of UrbanIt application with the step performed in order

This step is mandatory in order to make possible the correct forwarding of the push notifications. Since the notification target are linked to the user account they will be filtered depending on the account that is currently logged in the application. This means that if the user has multiple devices (i.e., more iPads) the notification is sent in all the iPads where the user logged in with the URBANIT app; but if the user has different accounts associated to the iPads the notification will be sent only to the pertinent account.

2. **The file operations:** *Step 2 and 3* on figure 4.3. The application can perform two different kinds of operations. One is to download the model (see 4.2.1) that contains all the informations about the commits, the other one is querying the Parse database to retrieve information (i.e., the list of the repositories stored in the server, the last user login, the token expiration date etc...). These operations are essential for the functionality of the application and can also be performed in background when needed.

3. **The push notification operations:** *White boxed arrows* on figure 4.3. These operations are push-notification related and we will not discuss in to deep here. They are mandatory for a correct functionality of the APNs and are performed only in particular situations, such as during the login procedure or during the app loading time.

4.2 The Models

When we do visualization we try to use a picture to communicate some kind of information. We can use graphs, diagrams or animation as channel to show what we want to convey. It is also true that the same graphs, diagrams and animations can visualize different information depending how we chose to use them. Currently URBANIT represent general Git informations (1.1), but the same visualization used in it can be extended with additional information.

To achieve this goal URBANIT uses specifics model to represent the information; models that are made with the specific purpose to be easily extendable with additional features, such as additional VCSs (e.g., `subversion`[3]) or with additional data (i.e., by using the width and the length of the building to assign them specific values).

One of the main models used in URBANIT are the *Graphs*. Since the starting point of URBANIT are repositories, this implies to have a set of folders and files, that are usually represented as graphs in file systems.

For the server side of URBANIT we needed a model that was lightweight, due the limited bandwidth of the mobile app, but in the meantime extendable and easy to build, as well as well supported then we decided to opt for the XML.

XMLs are easy to read (especially for testing purpose) and are easy to parse, thanks by the huge support by companies and communities with libraries as well as very flexible to further changes.

We will then discuss about other models present in URBANIT such as the database model (Chapter 4.2.4) and the *HisMo* model (Section 2.3).

4.2.1 The UrbanIt XMLs

The XMLs are used to represent the information gathered by the server from the repository. They are lightweight and are a perfect candidate to be exchanged between the server and the mobile application, which has a limited bandwidth.

Every XML used in URBANIT has a precise architecture that can be divided in two functional part:

- *The repository information*: This part is delimited by the `<Information>` tag. We used this section to save all the information we need about the a snapshot, such as:
  - **Author and Author e-mail** Who is the author of this commit and his/her e-mail
  - **Committer and Committer e-mail:** Who is the committer and his/her e-mail
  - **Comment:** The comment that the author wrote at commit time
  - **Creation Date:** The *timestamp* of the represented commit.
  - **SHA:** The *SHA* of the Git commit
- *The project representation:* The main part of each XML file is delimited by the `<directoryTree>` tag. Here are saved the two main objects of each project: The *folders* and the *files*.
  for the folders we save the name, meanwhile, for the files are saved multiple values such as:
  - **Dimension:** The file's dimension on the file system, expressed in Bytes.
  - **Name:** The file name.

[3] `https://subversion.apache.org`

  - **Type:** The file type.

With this simple architecture each XML file carries the information needed to represent the snapshot of each project version (or commit) tracked by the repository.

4.2.2 The UrbanIt Graphs

The graphs are used to represent the directory structure in the code. A single graph is made starting from a set of classes. Some of them are mandatory to build a correct representation, others are used as convenience classes. Among the main classes we found:

- The `CSGraphTree:` This class represent the basic shape of a Tree Graph. It permits to have a reference to the graph's root, and has a number of useful methods to retrieve nodes.

  We also extended this class to contain additional information about a single Git commit. These information can be retrieved any time to have VCS's related data. One of the most useful methods provided by this class is

  ```
  depthSearchOnTreeWithFunction:(void(ˆ)(CSTreeGraphNode *node))nodeBlock
  ```

  that permits to perform a given block of code for each node that belongs to the graph, using a depth search algorithm.

- The `CSTreeGraphNode:` This class represent a single graph's node. The most important properties and methods of this class are:
  - Property `identifier:` Is a graph-unique identifier. This identifier can be used to distinguish a specific node.
  - Relation Properties: Relation properties such as `parent`, `siblings`, `children` are mandatory property for each node that are built to handle references to the other nodes.
  - Property `Path:` This property contain an instance to the node's `CSGraphPath`.

- The `CSGraphPath:` The `CSGraphPath` is used to represent a node's path inside a graph. By following the node's ancestors the `CSGraphPath` is able to create unique identifier.

4.2.3 The HisMo applied

The *HisMo* models was used to develop an architecture able to analyze the entire evolution of given repository, creating final objects that are then queried to retrieve the overall evolution of a specific node (ora a set of them) as well as the status in a specific time (commit or version).

With *HisMo* we was also been able to handle the gaps between the files evolutions such as, for instance, a file that is deleted and re-inserted later. *HisMo* shaped this behavior giving us the possibility to visualize this gap correctly.

Using *HisMo* was been mandatory both to do a correct handling of the corner cases as well as to have a structured model that can be queried to retrieve evolution-related information.

In UrbanIt *HisMo* is composed by the following classes:

- *Snapshot:* In *HisMo* this entity is a placeholder that represents the entities whose evolution is studied i.e., file, package, class, methods or any source code artifacts. The particular entities are to be sub-typed from Snapshot.
  In UrbanIt a snapshot represent a single file. It also has information related to the files dimension and creation date that belongs to the Snapshot. All the data are gather from the XML (see 4.2.1).
- *Version:* A Version adds the notion of time to a Snapshot by relating the Snapshot to the History. A Version is identified by a time-stamp and it knows the History it is part of. A Version can exist in only one History. Based on its rank in the History, Version has zero or one predecessor and zero or one successor. Each Version has a reference to the so called reference Version which provides a fixed point in the historical space. For each node we have an associated version.
- *History:* A History holds a set of Versions. The relationship between History and Version is depicted with a qualified composition which depicts that in a History, each Version is uniquely identified by a rank. From a History we can obtain a sub-History by applying a filter predicate on the set of versions. Each History has a reference to the so called *Reference History* which defines the historical space.

  A single node History are crated by quering all the graphs and analyzing each node's path. The nodes with the same paths will be added to the same History. All the histories are handled by a singleton using the path as id.

4.2.4 The Database Model

The Parse server stores data and information about the users and files of UrbanIt, working as remote persistent system (see the UrbanIt structure on 4.1.1).

Parse uses a relational database with tables and relations. The structure that we choose for UrbanIt (see image 4.4) is very simple and intuitive.
In few words a `user` can have multiple repository associated and multiple installation. Each installation has information about a "copy" of UrbanIt installed on the device. This table contain information related to the devices only. Each repository can have one or more file associated (one file for each commit). Parse also requires that the file representation will be stored in a separated table.
We will explain in more detail the fields of the tables:

- `User`: The table that store information about the each registered user.
  - *ID*: The element unique key
  - *username*: The username chose by the user
  - *password*: The hash value of the password chosen by the user.
  - *authData*: Data that can be used together with ACL to perform different type of file access.
  - *creation/updated date*: Date of the object creation or last update
  - *ACL*: Permission of the current table (read/ readwrite).

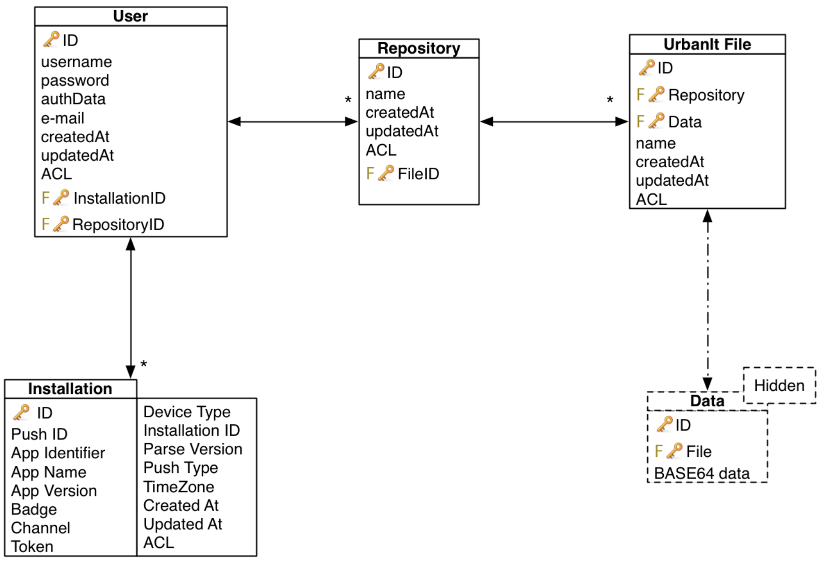


Figure 4.4. UrbanIt database schema

- `Installation`: Data related to the app and user device
  - *ID*: The element unique key
  - *pushID*: Identifier for the push notification related to the specific device
  - *App Identifier*: The unique identifier related to the application. It's the same used by the application in the AppStore.
  - *App Name*: Used for the push notification to show the correct app name
  - *App Version*: The last app version found in the device
  - *Badge*: The badge information related to the push notification. It's used to show the little number that could be shown the in the application icon. Not used in URBANIT but mandatory for push notification service
  - *Channel*: The push notification channel. It's used to filter the recipients of the push notifications. In this case we associated this value to the UserID.
  - *Token*: The token associated to the device. This is sent by the application in every request and compared with the one stored here in order to know if it's valid or not.
  - *DeviceType/ InstallationID*: Information related to the device. Useful to distinguish single device on statistics.
  - *Parse Version*: The Parse API version used by the application
  - *Push Type*: Type of push notification requested. This is fixed to *Apple APNs* but can be extended for further implementations
  - *TimeZone/CreatedAt/UpdatedAt*: Date of the object creation or last update with relative time zone

  - *ACL*: Permission of the current table (read/ readwrite).

- `Repository`: The table where are stored information about the uploaded repositories. Each user can be associated one or more repositories but each repository belongs to only one user. Each repository can have one or more `UrbanIt File` associated. (For `createdAt`,`updatedAt`,`ACL` description refer to precedent table descriptions)
  - *ID*: The repository unique ID
  - *Name*: The repository name
- `UrbanIt File`: This table stores information about each file uploaded **but not the file itself**. Data are stored in a specific separated table with a *one-to-one* relation to this table. (For `createdAt`,`updatedAt`,`ACL` description refer to precedent tables descriptions).
  - `ID`: The unique id of the file
  - `Name`: The file name chosen by the server. Usually is something like `commit_[number]`. The number close to the commit string is useful to check the correct ordering of the commits by the application.

4.3 The Git Reader

When a given repository is sent to the webserver we need to create the models (4.2.1) that are going to incapsulate the information needed to the application to generate the visualization. Such information must be gathered from the repository and saved in the model. To accomplish this goal we implemented the *Java Git Reader*, an application that has as unique goal: reading the repository and generate the output model.

The Reader is composed by three main parts:

1. *The Git Analyzer:*
   This part reads an input repository and generate a set of objects that can be browsed to generate the output. It is build using the JGit[4] library. JGit is a reimplementation of Git using Java. This library allowed us to have accessor methods to the most common commands of Git such as `Rebase` and `Revert`, largely used to browse the repository. Indeed, starting from the `HEAD` of a given commit, using the `Revert` command we was been able to browse in each commit and generate the output object. We decided to use a third-party library for this step in order to have a full-fledged set commands to browser easily and efficiently a Git repository, with the possibility to rely on an library tested and maintained by the community, avoiding to go too much in the deep of the Git details.
2. *The Information Objects:*
   These objects have as main goal to represent information about commits. Each of the generated object contains informations about a specific commit. These objects works independently from the JGit library. This mean that they can be filled from JGit or from another source, decoupling the part related to the JGit library and the output generator. Each object gather two different sets of information (both of them related to the same commit). The first one is CVS-related (i.e., the commit date, the commit author, the

[4]https://eclipse.org/jgit/

commit committer), the second one is files-related (the dimensions of the files, the files name, directories etc...). The Information Objects are the first disassembling step of the repository in to programming objects.

3. *The Output Generator:*
The output generator reads the information objects and create the XMLs 4.2.1. Since each of the objects contains information about a single commits and about all the files stored in the commit, the output generator browse recursively each object generating step by step the output file, containing all the information gathered from the Git Analyzer, formatted according to the a pre defined grammar.

With these three parts the git reader is able to take in pieces a given Git repository, creating an output set of files with all the information needed to the final application to create the visualization, maintaining a structure that is flexible and easy to extend.

Chapter 5

UrbanIt Usage Scenario

URBANIT needs to store files and information related to user that is logged in the system, hence, the first step a new user have to do is creating a new account. This step will be done trough the URBANIT web interface and it will be discussed in section 5.1.1.

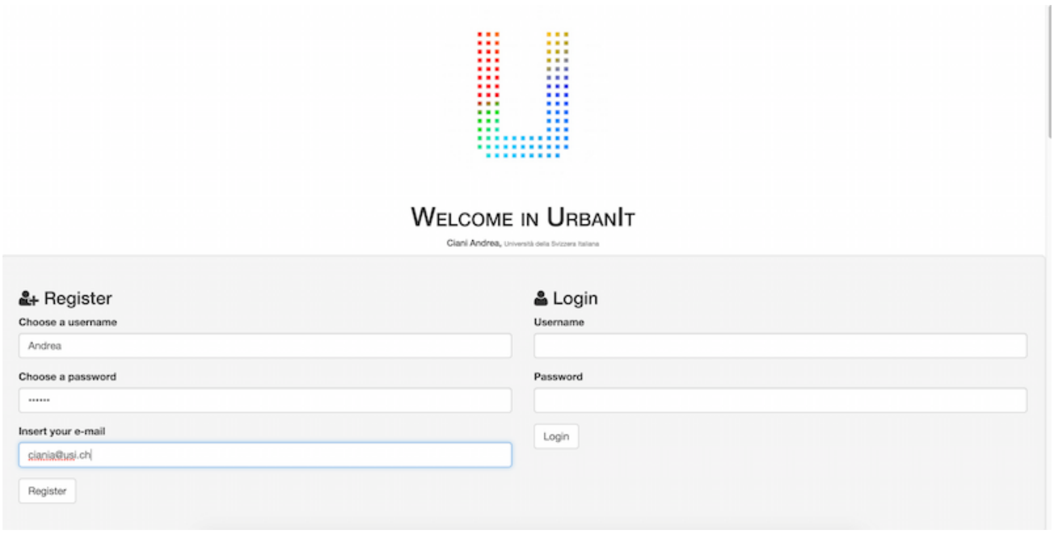
Once created a new account the user can to give to the system the first repository that needs to be analyzed. This step will be done trough the web interface too, both giving a URL of a *public* repository or trough a compressed file (i.e. an `.zip` file) of a versioned folder. This step will be explained with more details in section 5.1.2.

After provided a new repository, the backend service will read it to generate the XMLs (discussed on section 4.2.1) needed by the client application. When the process ends (event with a success with a failure) a feedback is sent to the user trough a *push notification* to the user's device, describing the final result of the given analysis[1]. These features are addressed later in 5.1.3

Finally we will explain how is possible to download the information generated in the previous steps on the devices and how to start using URBANIT. We will then face a real example (the *Soqa* repository) on Chapter 5.2.[2]

5.1 Starting Up

The register page (figure 5.1) is the first step a new user pass trough. By giving an unique *nickname* and a *password* (and optionally an *e-mail address*) the user can create a new URBANIT account.


WELCOME IN URBANIT

Ciani Andrea, Università della Svizzera Italiana

Register

Choose a username

Andrea

Choose a password

••••••

Insert your e-mail

ciania@usi.ch

Register

Login

Username

Password

Login

Figure 5.1. The register page at `http://urbanit.inf.usi.ch`

5.1.1 Creating a new user

At each user is associated an unique `id`. This `id` is created by the web-service and is necessary to distinguish the users.

The unique identifier is always shown in the *dashboard* on the top bar, as shown on figure 5.2.

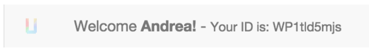

Welcome **Andrea!** - Your ID is: WP1tld5mjs

Figure 5.2. The information shown by the dashboard about the current logged in user

The dashboard (fig 5.3) allows users to add a new repository, provides a list of the repository added and eventually permits to delete them from the server.

5.1.2 Adding the repository

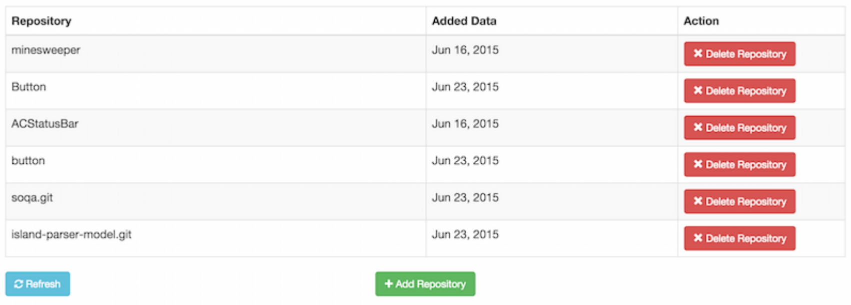
Once the user created the repository he/her can add a new repository from the dashboard.

The module for adding a new repository (fig 5.4) permits to use an URL of a *public* repository or a `.zip` file.

It's important to keep in mind that the URL provided **must** be accessible publicly because the server does not permit to handle login behaviors (see 6.1 for more detail). The `.zip` upload is provided for all such repositories that are not accessible with a public URL. The server in that

[1]This can be a success alert or a description of the failure reason.

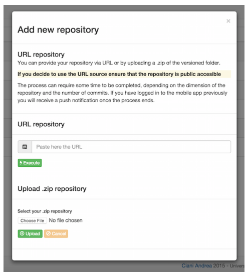
[2]*Soqa* is an internal project of the *Università della Svizzera Italiana* and it's a good candidate for this test since it has a good evolution on the time, a quite complex structure and my kind of different files to analyze.



Repository	Added Data	Action
minesweeper	Jun 16, 2015	✖ Delete Repository
Button	Jun 23, 2015	✖ Delete Repository
ACStatusBar	Jun 16, 2015	✖ Delete Repository
button	Jun 23, 2015	✖ Delete Repository
soqa.git	Jun 23, 2015	✖ Delete Repository
island-parser-model.git	Jun 23, 2015	✖ Delete Repository

Figure 5.3. The dashboard with some repository added

case will unzip the given folder, searching on the root of the resulting project the *.git* folder that contains the information about the CVS.



Add new repository

URL repository

You can provide your repository via URL or by uploading a .zip of the versioned folder.

If you decide to use the URL source ensure that the repository is public accesible

The process can require some time to be completed, depending on the dimension of the repository and the number of commits. If you have logged in to the mobile app previously you will receive a push notification once the process ends.

URL repository
Paste here the URL
Execute
Upload .zip repository
Select your .zip repository
Choose File No file chosen
Upload
Cancel
Ciani Andrea 2015 - Univers

Figure 5.4. The module used to add a new repository

5.1.3 Parsing the repository

Since this procedure is quite invisible to the user (see 4.3) we adopted a useful method to give to the users a feedback.

During this procedure if the given repository was successfully analyzed it is added to the list of the repositories in the dashboard (image 5.3) then a push notification is sent to the user device.

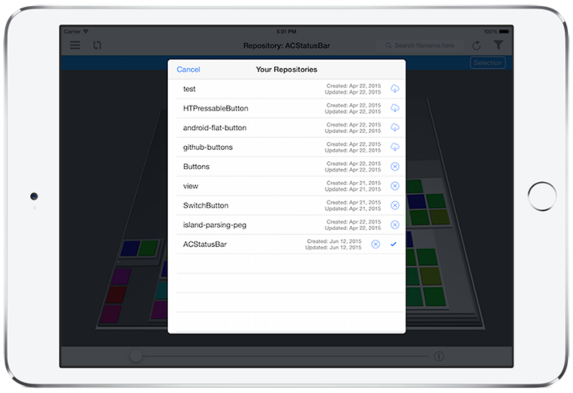
The push notifications are used both to notify the correct fulfillment of the process, as well as description alert if something went wrong.

5.1.4 Download on device

Once the files was been created and uploaded on the server the used dice is able to download them locally and use them to create the visualization.

Once the user has chosen to download one of the repository available the device will keep all the file on the **sandbox**[3] of the app.

[3]For security reasons, iOS places each app (including its preferences and data) in a sandbox. A sandbox is a set of



test	Created: Apr 22, 2015 Updated: Apr 22, 2015
HTPressableButton	Created: Apr 22, 2015 Updated: Apr 22, 2015
android-flat-button	Created: Apr 22, 2015 Updated: Apr 22, 2015
github-buttons	Created: Apr 22, 2015 Updated: Apr 22, 2015
Buttons	Created: Apr 22, 2015 Updated: Apr 22, 2015
view	Created: Apr 21, 2015 Updated: Apr 21, 2015
SwitchButton	Created: Apr 21, 2015 Updated: Apr 21, 2015
island-parsing-peg	Created: Apr 22, 2015 Updated: Apr 22, 2015
ACStatusBar	Created: Jun 12, 2015 Updated: Jun 12, 2015

Figure 5.5. The download view with some available repositories. Notes the different status icon on the right of each of them

For a performance reason when the user chooses to remove a repository. This folder is used to keep files that are no longer needed by the application but allows them to be easily restored in any moment, avoiding to download them again. If the device is running out of free space these folders will be emptied automatically by the operating system.

Once the repository files are downloaded on the device the project can be selected and URBANIT will generate the representation immediately.

5.2 A real example: The Soqa Repository

In this chapter we present a preliminary evaluation of UrbanIt. To do that we will use the *Soqa* repository. On the app side, the repository is shown as in figure 5.7. We will discuss in more detail every component of the interface.

5.2.1 The interface

On the top of figure 5.6 is shown a toolbar with the name of the repository visualized. On that toolbar are also available a list of buttons that allow the user to perform different operations on the repository. From the left to the right of the toolbar we find:

- **The menu button**: This button shows the URBANIT setting menu. Here the user can change settings related to the visualization (like the offset between the buildings, the anti aliasing of the renderer), about the active account, the repository list and many other useful controls.
- **The Difference View**: The *Difference View* (see also 3.2) is used to compare two different commits. We will explain further how it can be used in more detail.

fine-grained controls that limit the app's access to files, preferences, network resources, hardware, and so on. As part of the sandboxing process, the system installs each app in its own sandbox directory, which acts as the home for the app and its data.[3]

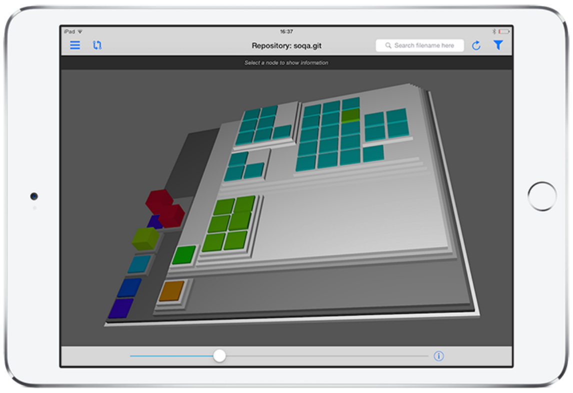
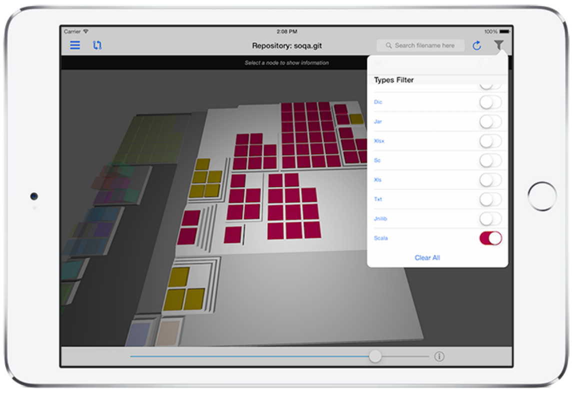


Figure 5.6. The Soqa repository with some filtered elements. On the right we can see the filter popup.

- **The repository name**: The current repository name is steady shown at the center of the toolbar, always visible to the user.
- **Search control**: The search control permits to search a file by giving its name (or a substring of it). The view will highlight all the files their name is equal or contain the given string.
- **Refresh button**: The refresh button cleans the interface and repositions the "city" in the middle of the screen. The refresh button can also be used to apply some changes when the setting are modified (for example the renderer anti-aliasing or buildings offsets).
- **Filter button**: The filter button opens a popup with a list of all the file extensions founded in the repository. Each extension has its associated color. By disabling an switch the view will represent all the file that have that extension as semi-transparent buildings (see image 5.7).



Types Filter

Dic ○
Jar ○
Xlsx ○
Sc ○
Xls ○
Txt ○
Jnilib ○
Scala ●

Figure 5.7. The starting point of a repository: The city visualization.

5.2.2 Single node evolution

By **long tapping** on a specific element the user can see more details about the node. By

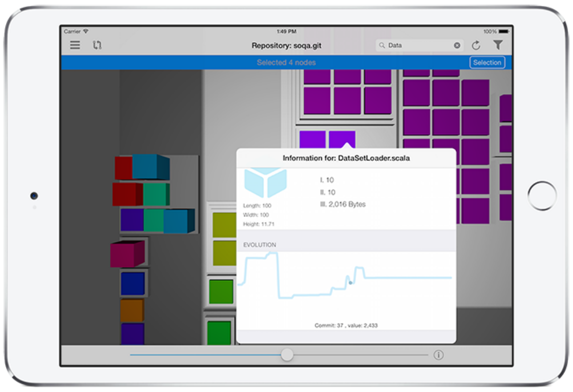


Figure 5.8. The Single Node detail view. The plot represent the node's evolution on the time. The darker dot is the current commit

querying the Hismo structure (see 2.3) this view is able to plot the node's evolution. A darker dots is added as tag of the current commit. The user can also drag the finger on the graph to query the plot and see for each commit what would be the associated value.

The third dimension of each node (the *Height*) is associated to the file's dimension (in Bytes). The *Single Node evolution view* shows both the raw value (shown as third dimension "*III*" in figure 5.8) as well as the related building dimension in the view (called "*Height* " in figure 5.8).

5.2.3 Multi node evolution

When the user **double taps** on a building, it becomes selected. The selection of multiple buildings will set the status bar on "multiple selection mode", becoming blue. When the status bar is on *multiple selection mode* it permits to show the *Multiple Node Evolution View*.

This view has a specific plot that compare all the evolution of the selected nodes (image 5.9).

In this view the users have many different tools available:

- **Deselect All**: Intuitively this button permits to deselect all the selected nodes.
- **File Sizes Sum**: A sum of all the third dimensions (files size) of all the selected nodes.
- **Evolution Comparison and Share**: The plot represent the variation of the nodes (see 3.3). Like in the *Single Node Evolution View* the user can drag the finger on the plot's bars to query the plot. By tapping on the share button, the user has also available a fast, precompilated mail module, with a summary of all the nodes evolutions written into, as well as a screenshot of the plot.

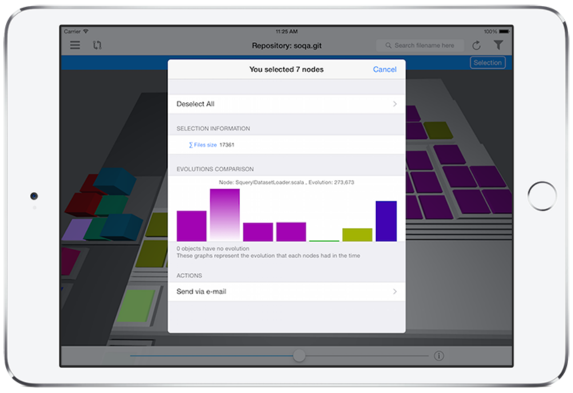


Figure 5.9. The multiple selection view. Each color of the graph represent the selected node color

5.2.4 Commit comparison with the difference view

One of the most important feature of UrbanIt is the possibility of making comparison. Since now we have seen how to compare evolution nodes, now we will see how is possible to compare two commits.

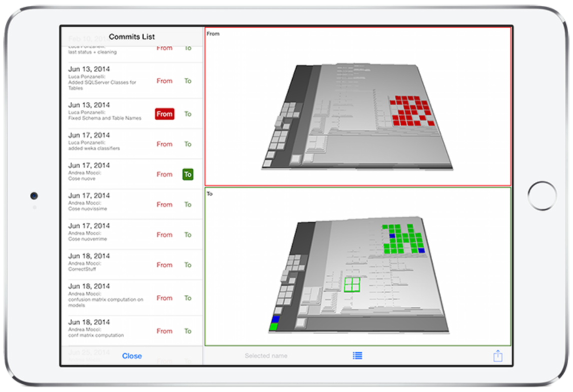


Figure 5.10. The difference view in action with the Soqarepository

The image 5.10 shows how the *Difference View* comapres two different commits of the *Soqa* repository.
In the image are selected two commits and as shown the starting commit is made on the Jun 13th by “Luca”. Under the name is shown the commits comment written by the author. That commit is represented in the top view, shown with red edges.

The destination commit is made by “Andrea” on Jun 17th. This commit is represented on the view on the bottom, with green edges.

As discussed on the chapter 3.2 the building colored in red represent files that are no longer available on the destination commit (deleted), the buildings colored with green are files added in the destination commit and the blue ones are files that have different dimension between the two commits.

The difference view permits also to know the filename of each building by tapping on each of them (the name will be displayed on the little box on the bottom-left corner) and a list of the deleted, added and modified files, just by tapping on the list icon in the center of the bottom bar.

As always URBANIT permits also to share screenshots via a pre-configured mail module that can be created by tapping on the action button on the bottom right corner. This module can be modified by the user starting from a generated list of the modified files (added and removed) and with a screenshot of the two views as attachment for the recipients.

Chapter 6

Conclusion

How discussed in Chapter 1 even if on personal workstations full-fledged applications and command-line tools are available they are not always at the developers disposal. This is the main reason why having a basic repository analysis always available could be deterministic to verify that a software is being developed correctly.

UrbanIt aims to solve such problem by providing a fast and easy to use tool that brings 3D evolution visualization to modern portable devices. In particular the contributions of UrbanIt are:

- Automatic Git repository reading from a given public URL or `.zip` file.
- Embedded Git repository visualization with 3D interaction.
- Visual comparison of commits.
- Single file evolution with plots and values.
- Multiple files evolution with comparing plots and values.
- Commit information with embedded e-mail module to contact authors and committers.
- Fast and easy screenshot sharing.
- Multiple Git repository tracking with single application.

In the following, we discuss how we tackled those points in this thesis, through the implementation of our approach.

In Chapter 3 we discussed how UrbanIt provides many different views to allow the user to query the information and creating a visual representation of the result (i.e., the *difference view* creates two views with colored buildings, the *evolution view* represents with plots the files evolution etc...).

In Chapter 4 we discussed how UrbanIt provides a dynamic structure allowing future extensions and making the tool adaptable to different visualization sources. In particular we explained on 4.1.1 how the backend works and could be extended, on 4.1.2 how the mobile application is structured and on 4.2 what are the main models used in UrbanIt, creating the interface for each component of the tool.

Finally in Chapter 5 we validated URBANIT with a real example: the *Soqa* repository, a *Git* repository of medium size containing enough commits and file types, that exploit the URBANIT features, providing speed and interactivity and using state of the art technologies on mobile devices.

6.1 Future Work

We showed how we achieved the contributions stated in Dection 1.2 by implementing our system: URBANIT.

We discussed how URBANIT provides developers a useful portable tool to have always available a glance in to the project evolution, with the possibility to share easily and quickly comments and screenshots to teammates, comparing single files or entire commits with few taps.

Even though we can claim that those features meet the requirements posed by the contributions we stated, there are still enhancements to be done:

- *Automatic Repository Update:*
  A very helpful feature in URBANIT would be the possibility to automatically notify the server when an update happen to the repository, refreshing all the data saved online and on the application. Since in URBANIT is already using a push notification service, it can be extended to notify the user to an incoming update in the repository. This can be achieved in URBANIT by restructuring the server side.

- *Desktop Version:*
  In the last few years Apple tried to bring closer the world of mobile devices and the world of Macintosh desktop computer, by developing personal computers operating systems (i.e., Apple Yosemite) and mobile operating systems (iOS) with technologies that make easier the communication between mobile and desktop pc.

  URBANIT could take advantage from these technologies to provide a more complete experience both on mobile and desktop pc, allowing developers to do on the desktop machines advanced features that can be reflected to the mobile app. To make the extensions easier in that sense we decided to use in URBANIT third-party frameworks that are available also for Macintosh computer.

  The desktop application could open new possibilities such as:

  – **Adding/Removing local private repositories:** With the desktop application the users could read the local private repositories locally, without use the web application, that actually is not able to read non public repositories. Indeed a desktop application can bypass this problem because it would have directly access to a local repositories, saving the server the effort and the limit to clone from a public URL.

  – **Complete environment:** Apple encourage developers that make mobile application for iOS to develop the respective desktop application for Macintosh. This create a continuos experience for users that use such tools every day.

  – **Responsiveness:** Native apps are more responsive than web apps, partly due to lower level access to the machine and partly due to not having to talk to a remote server.

- **Continuos experience:** If we image an hypothetical usage of UrbanIt from a developer, for example on the train from home to work, is feasible guess that once arrived at work he/she wants to continue was he/she was doing. This can be achieved by a new entry technology developed by Apple in 2015, called *Handoff*[1]. This technology hit this target allowing the user to continue a work started in the mobile version on UrbanIt to the desktop application.

- *Other VCSs:*
  As described in this document UrbanIt is thought to be extensible. Adding other VCSs (such as *Subversion*[2]) would expand the value of UrbanIt.

  To add a new VCS in UrbanIt an adaptation on the server side would be required. Once the generated files are conform to the same features described on 4, the mobile application will be able to visualize additional repositories.

- *Additional Filters:*
  On chapter 5.2 we showed how nodes can be filtered using the UrbanIt filter. These filters use specific Objective-C protocols [5] to create a flexible implementation that is capable to adapt to different purpose. Just to make an example a user may need to visualize some set of file type with the same color (for example all the images .png, .jpg, .tiff in blue) and all the text file in red (.doc, .docx, .pages, .tex). This feature can be achieved by creating a filter object conform to the filter protocol, that describes this behavior.

[1] `http://www.apple.com/ios/whats-new/continuity`
[2] `https://subversion.apache.org/`

Bibliography

[1] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takashima, and Y. Yamamoto. A case study of the evolution of jun: an object-oriented open-source 3d multimedia library. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 524–533. IEEE Computer Society, 2001.

[2] Apple. Remote notification. *Apple Developer Reference*, 2013.

[3] Apple. The ios environment. *Apple Developer Reference*, 2014.

[4] Apple. ios human interface guidelines. *Apple Developer Reference*, 2015.

[5] Apple. Programming with objective-c, 2015.

[6] Apple. Scene kit. *Apple Developer Reference*, 2015.

[7] G. Balogh and A. Beszédes. CodemetrpolisâĂŤa minecraft based collaboration tool for developers. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.

[8] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. 2004.

[9] J. Bohnet, S. Voigt, and J. Döllner. Projecting code changes onto execution traces to support localization of recently introduced bugs. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 438–442. ACM, 2009.

[10] S. M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for informed decision making; from code to components. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 765–772. ACM, 2002.

[11] M. C.Taylorand. Revision towers. In I. C. Society, editor, *VISSOFT 2002 (1st International Workshop on Visualizing Software for Understanding and Analysis)*, pages 43–50, 2002.

[12] S. Ducasse and T. Gırba. Being a long-living software mayor—the simcity metaphor to explain the challenges behind software evolution. In *Proceedings of CHASE International Workshop*, volume 2005. Citeseer, 2005.

[13] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an agile reengineering environment. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 99–102. ACM, 2005.

[14] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *Software Engineering, IEEE Transactions on*, 18(11):957–968, 1992.

[15] U. Erra and G. Scanniello. Towards the visualization of software systems as 3d forests: the codetrees environment. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 981–988. ACM, 2012.

[16] D. Fang, X. Liu, H. Yang, and L. Liu. Evolution for the sustainability of internetware. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, page 17. ACM, 2012.

[17] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Software Maintenance: Research and Practice (JSME)*, 2006.

[18] D. Gračanin, K. Matković, and M. Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[19] A. Holzer and J. Ondrus. Trends in mobile application development. In *Mobile wireless middleware, operating systems, and applications-workshops*, pages 55–64. Springer, 2009.

[20] A. Inc. Core animation. *Apple Developer Reference*, 2015.

[21] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *The 20th international conference on Automated Software Engineering*. ACM Press, Nov, 2005.

[22] H. G. M. Jazayeri and C. Riva. Visualizing software release histories: The use of color and the third dimension. In I. C. Press, editor, *ICSM 1999 (16th IEEE International Conference of Software Maintenance)*, pages 99–108, 1999.

[23] P. Makedonski, F. Sudau, and J. Grabowski. Towards a model-based software mining infrastructure. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–8, 2015.

[24] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, page 27. ACM, 2003.

[25] R. W. Marinescu, Mihancea. iplasma: An integrated platform for quality assessment of object-oriented design. In *ICSM (Industrial and Tool Volume)*, 2005.

[26] T. Panas, R. Berrigan, and J. Grundy. A 3d metaphor for software production visualization. *IEEE*, page 314, 2003.

[27] S. P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995.

[28] F. V. Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In I. C. S. Press, editor, *ICSM 2004 (20th IEEE International Conference of Software Maintenance)*, pages 328–337, 2004.

[29] P. Smutnỳ. Mobile development tools and cross-platform solutions. In *Carpathian Control Conference (ICCC), 2012 13th International*, pages 653–656. IEEE, 2012.

[30] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[31] A. Telea, A. Maccari, and C. Riva. An open visualization toolkit for reverse architecting. In *Program Comprehension, 2002. Proceedings. 10th International Workshop*, pages 3–13. IEEE, 2002.

[32] M. Van Welie and G. C. Van der Veer. Pattern languages in interaction design: Structure and organization. In *Proceedings of interact*, volume 3, pages 1–5, 2003.

[33] L. Voinea, A. Telea, and J. J. Van Wijk. Cvsscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56. ACM, 2005.

[34] J. Wang, X. Peng, Z. Xing, and W. Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 762–771. IEEE Press, 2013.

[35] R. Wettel. Scripting 3d visualizations with codecity. In *FAMOOSr'08: Proceedings of the 2nd Workshop on FAMIX and Moose in Reengineering*, 2008.

[36] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 231–240. IEEE, 2007.

[37] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.

[38] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 155–164. ACM, 2008.