Closing the gap between the Web and Peer to Peer

Final Report

Steffan Norberhuis
1509306

Quinten Stokkink
4016270

Faculty Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

17 June 2013

Summary

During the course of Q4 of the 2012-2013 academic year, we have worked on the project of a Peer-to-Peer web browser. After a two week orientation phase, a six week programming phase and finally another two week reporting phase we deliver our product. The final result of this endeavor is an integrated web browser in the Tribler platform [1], which supports automatic retrieval and distribution of resources encountered on the web page.

Acknowledgements

During our project we were assisted by a number of people, who were crucial to its success. We would like to thank Dr.ir. J.A. Pouwelse for being our supervisor and making sure we were on track with our project and at the same time delivering quality. We would like to thank Ir. E. Bouman for helping us with the technical side of programming with Tribler, assessing the viability of our project and of course letting us mess with Tribler in the first place. We would also like to thank the rest of the Tribler team for keeping up with our issue on GitHub and providing feedback on our ideas. We thank Dr. M.A. Larson for being our general supervisor and assessing our final product. Lastly we would like to thank our retired collegue M. de Vries for his input the first part of the project.

Contents

1 Project description 5
1.1 Actors 5
1.2 Mission 5
1.3 Assignment 6

2 Methodology 6
2.1 Strategy 6
2.2 Available materials 7
2.3 Workflow 7

3 Orientation phase 9
3.1 Milestones 9
3.2 Research 9
3.2.1 Python 9
3.2.2 Tribler 10
3.2.3 wxPython 10
3.3 Architecture 11

4 Eternal Webpages phase 11
4.1 Milestones 11
4.2 Architecture 12
4.2.1 WebBrowser class 12
4.2.2 SiteRipper module 13
4.3 Code quality 15
4.3.1 Code quality 15
4.3.2 Test coverage 16
4.4 Licenses 16

5 TUPT phase 16
5.1 Milestones 16
5.2 Architecture 17
5.2.1 Webbrowser module 18
5.2.2 TUPT module 18
5.2.3 Infobar module 30
5.2.4 Plug-in module 30
5.3 Code quality 30
5.3.1 Code quality 31
5.3.2 Test coverage 32
5.4 Licenses 32

6 Testing and documentation phase 33
6.1 Bug hunting 33

6.2 Documentation 33
6.2.1 UML diagrams 33
6.2.2 SIG deliverable 34
6.2.3 Final report 34

7 Evaluation and conclusion 34
7.1 Customer satisfaction 34
7.2 Self evaluation 35
7.3 Project time sinks 35
7.4 Future work 36
7.5 Conclusion 37

A Docstrings 38

B Class diagrams 39
B.1 Eternal Webpages 39
B.2 TUPT 40

1 Project description

In this section we will provide a general overview of our project. This section will contain an objective view of what the assignment was when we took it on and how it developed over time. Next to that we will go over the actors and their interests, our personal visions and the materials we had available to us.

1.1 Actors

During our project we came in contact with several actors who have played a role in our project. In this subsection we will explain who these actors were and how they influenced our project.

Product owner
The product owner, Ir. E. Bouman of the Tribler team, was (naturally) interested in expanding Tribler functionality. This meant for us, that we were confronted with a healthy amount of ideas, which we had to interpret to get a good direction for the project. The overall direction we took with our project was essentially the distilled version of all of these ideas.

Supervisor
Our personal supervisor, Dr.ir. J.A. Pouwelse of the TUDelft Parallel and Distributed Systems group, took an interest in the planning of our project. This lead to us creating clear targets for ourselves so we knew exactly where we were headed and how we were going to get there. By having a clear set of targets we could also commit ourselves to delivering a quality product.

Tribler team
The Tribler team is a non-profit research group from the Delft University of Technology and Vrije Universiteit Amsterdam. The team consists of professors, phd students and software engineers from both universities. The group is funded by several different entities like the European Union CORDIS FP7 research program[2], Petamedia[3] and the I-Shareproject[4].

The Tribler team had an interesting role to fill in our project. With the status updates we provided via GitHub came immediate feedback by other Tribler team members. This meant we could immediately see if a certain decision we made would be 'acceptable'.

1.2 Mission

Our mission was to close the gap between (anonymous) web browsing and P2P by creating a fully functional web browser which also supports (anonymous) P2P video streaming. We take pride in being able to report that our final deliverable prototype does just this, it binds P2P sharing to a fully functional web browser, which is then able to stream the found resources. But there will be more on that subject in the rest of the report.

1.3 Assignment

After our initial research we produced a report stating concretely what our product would be. This product would be able to rip images and Youtube videos from sites and use torrents to distribute and fetch them. It would also find and replace direct links to these types of resources.

We altered our course when it became clear after the first 3 weeks we were basicly done with our assignment, and then we went above and beyond the original target.

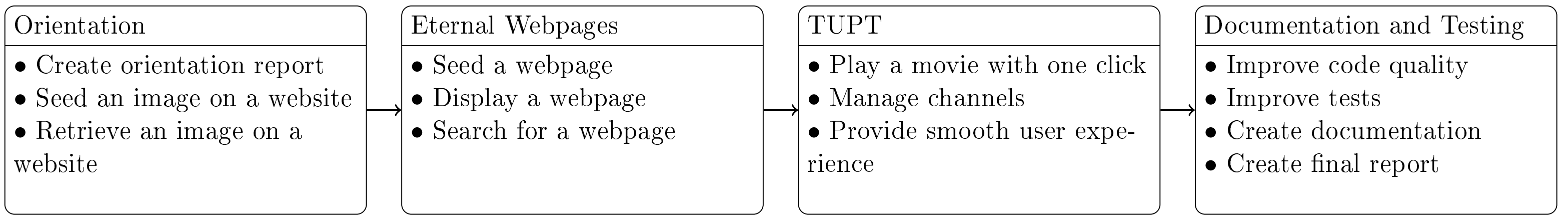


Figure 1: Overview of the product phases in our project

In the end our product had gone through 4 phases: the orientation phase, the Eternal Webpages phase, The Ultimate Piracy Tool (TUPT) phase and finally the documentation and testing phase. This led to two different products: the Eternal Webpages product and the TUPT product. The rest of this report will summarize the contents, the design decisions and the research for each phase. The overall picture of this process is given in Figure 1.

2 Methodology

In this section we describe our approach to the methodology of the project. We will first go over the software engineering outline of our methodology. Lastly we will describe the workflow we utilized.

2.1 Strategy

Our project employed an agile approach using a methodology resembling the Scrum methodology [5]. We held weekly meetings with our TU Delft supervisor and company supervisor. In these meetings we discussed our progress of the past week or demostrated the progress made on the software, our plans for the following week and what functionality we prioritized. We used milestones to track project features and issues to track what needed to be done for a specific feature.

As our team consisted of only two developers more process organization was not needed. We were in constant communication sitting next to each other. We always discussed when a task was finished and what task would be worked on next. The meetings with our supervisor and product owner made sure all our supervisors were up-to-date on the state of our product.

2.2 Available materials

To produce our product we were granted several materials.
Firstly we were granted a workspace by the Parallel and Distributed systems department of the TUDelft [6] in the lab on the 9th floor. Our actual workspace became the public Drebbelweg labs though, because it was simply much less busy there.

The GitHub issues of the Tribler project [7] are publicly accessible and here we could communicate our efforts with the Tribler team and receive feedback. We ended up using this mostly during the initial stages of the project.

Lastly our own materials were our personal lap-tops, running the Ubuntu OS. We ended up using the Eclipse IDE [8] with the PyDev plug-in [9] to develop our product. We also made use of the GitHub platform for version control of our code, other deliverables and our product backlog [10]. Our project utilized a private repository for project documents (like this one), and a public repository for the actual code and technical documentation. The public repository is a fork from the actual Tribler repository.

2.3 Workflow

We used the GitHub issuetracker to statisfy our need to track our progress and product backlog. The GitHub issue tracker allows you to create issues and categorize them using milestones and labels. This allowed quick overview of what a issue was for, what kind of issue it was and it allowed to only view relevant issues.

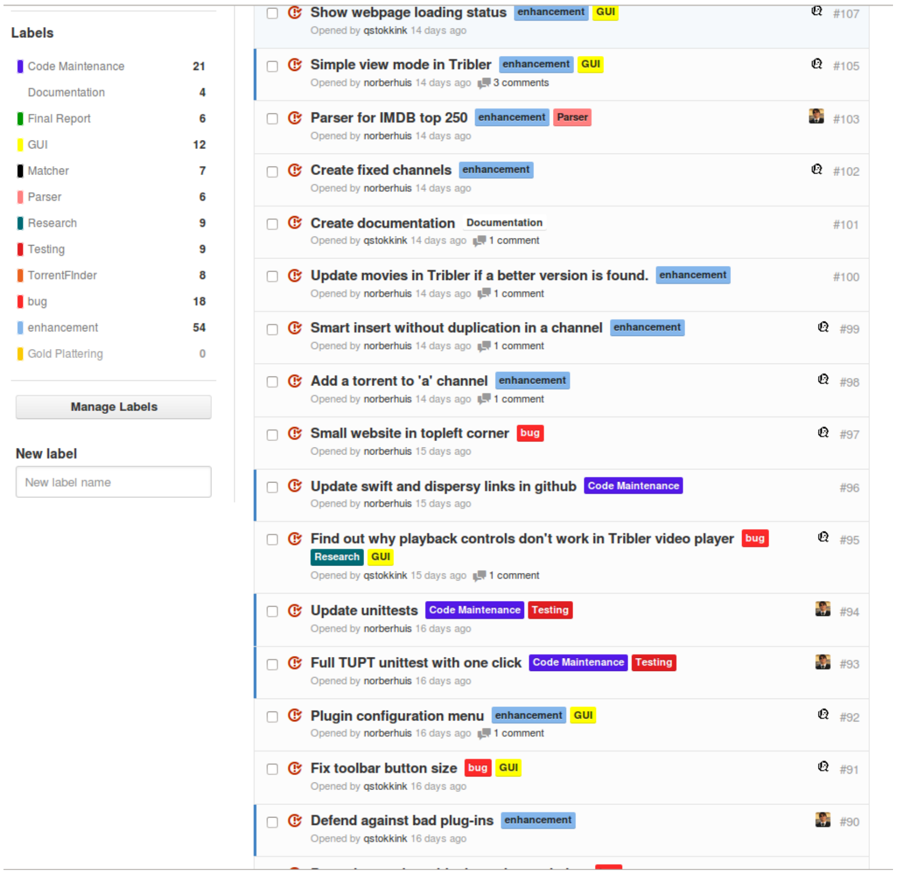
Every issue was tracked using a milestone in GitHub. The milestones were used to distinguish complete additions of functionality and iterations. After completing the milestone, the code was at a stage of being ready to be released. Milestones were not exactly completed in a week's time, but closely followed a week rotation.

Next to the milestones we used labels to categorize issues. We used the following labels:

- Bug: a bug in the software.
- Code maintenance: Refactoring or cleaning up code to increase the maintainability.

- Enhancement: Addional functionality.
- Testing: Testing or adding unit tests for a part of the software.
- Documentation: Documenting the usage or technical description of the software.
- Research: Perform research to determine what technology or architecture should be used for the next part of the project.

We also used secondary labels to indicate for what part of the software the issue was for. A page from our list of closed issues is shown in Figure 2.



Labels

Code Maintenance	21
Documentation	4
Final Report	6
GUI	12
Matcher	7
Parser	6
Research	9
Testing	9
TorrentFinder	8
bug	18
enhancement	54
Gold Plattering	0

Manage Labels

New label

New label name

- **Show webpage loading status** enhancement GUI — #107
  Opened by qstokkink 14 days ago
- **Simple view mode in Tribler** enhancement GUI — #105
  Opened by norberhuis 14 days ago 3 comments
- **Parser for IMDB top 250** enhancement Parser — #103
  Opened by norberhuis 14 days ago
- **Create fixed channels** enhancement — #102
  Opened by norberhuis 14 days ago
- **Create documentation** Documentation — #101
  Opened by qstokkink 14 days ago 1 comment
- **Update movies in Tribler if a better version is found.** enhancement — #100
  Opened by norberhuis 14 days ago 1 comment
- **Smart insert without duplication in a channel** enhancement — #99
  Opened by norberhuis 14 days ago 1 comment
- **Add a torrent to 'a' channel** enhancement — #98
  Opened by norberhuis 14 days ago 1 comment
- **Small website in topleft corner** bug — #97
  Opened by norberhuis 15 days ago
- **Update swift and dispersy links in github** Code Maintenance — #96
  Opened by norberhuis 15 days ago
- **Find out why playback controls don't work in Tribler video player** bug Research GUI — #95
  Opened by qstokkink 15 days ago 1 comment
- **Update unittests** Code Maintenance Testing — #94
  Opened by norberhuis 16 days ago
- **Full TUPT unittest with one click** Code Maintenance Testing — #93
  Opened by norberhuis 16 days ago
- **Plugin configuration menu** enhancement GUI — #92
  Opened by norberhuis 16 days ago 1 comment
- **Fix toolbar button size** bug GUI — #91
  Opened by qstokkink 16 days ago
- **Defend against bad plug-ins** enhancement — #90
  Opened by qstokkink 16 days ago

Figure 2: A screenshot of a page of our list of closed issues.

With the TUPT part of the project we really started to work on different parts of the system in parallel. The Eternal webpages project consisted of only two parts, whereas the TUPT project can be divided in seven parts. To increase code quality and share knowledge with each other we started to use pull requests inside GitHub. This allowed us to easily manage code reviews and comment on the code without disrupting the other in his workflow.

3 Orientation phase

In the orientation phase we were set on easing our way into Tribler. We would start out with distributing images in the first phase and at the end of the project handle playing and distributing Youtube video files.

3.1 Milestones

In the orientation phase our milestones were:

- Seed an image on a website using Tribler
  This sprint would result in a prototype that could identify an image on a webpage by its tag name, download the image by its src location, create a torrent file for this resource and lastly seed the file using Tribler.
- Retrieve an image on a website using Tribler
  This sprint would result in a prototype that could identify an image on a webpage by its tag name, create a query for Tribler based on the src location and then search for and download the corresponding torrent file.

3.2 Research

In the orientation phase we oriented ourselves on the different packages we had to use if we were to use Tribler. In the following subsections we will lay out the discoveries we made while evaluating the new software/API's we were going to have to use.

3.2.1 Python

Because the Tribler source code is written in Python, we would also use Python to extend Tribler's functionality. Python is a programming language with a main distinguishing feature of being dynamicly typed [11]. With our team having primarily experience with staticly typed programming languages, it took some time to get used to, but we adapted to it and made use of its advantages in the end.

To develop our Python code we utilized the Eclipse IDE together with the PyDev package. This allows for a easy project overview and easy execution of the code. Our initial research on how to set-up this toolchain was even included on the official Tribler GitHub wiki (which can be viewed on https://github.com/Tribler/tribler/wiki/Compiling-Tribler-from-sources-under-Eclipse).

Default standards for Python are lacking for object oriented programming. Function names are expected to be be in underscore_format [12], which hardly anyone has adopted.

This has resulted in that pretty much everyone defines their own standards. We did this as well, with respect to the rest of the code we were going to work with. We adopted the standard from the wx package we were using, by using UpperCamelCase for method names and class names and lowerCamelCase for variable names. When appropriate, we tried to define private member variables.

The great danger that comes with using Python, that we would like to point out, is the fact that most syntax errors are only detected at runtime. This means the language is very sensitive to type-o's. To deal with this, it is even more important for all python code to be thoroughly tested, checked and reviewed.

3.2.2 Tribler

Tribler is a Peer-to-Peer file sharing platform made by the Tribler team from the Parallel and Distributed systems department of the TUDelft [6]. Tribler supports decentralized torrent tracking and is in the process of supporting anonymous torrenting [1]. The Tribler features we use, are its integrated video player (based on a VLC python wrapper), its library for creating torrents, its functionality to stream torrents and its channel management capabilities.

The streaming torrent functionality is based on a library called 'Swift' and is able to download the torrent as sequentially as possible [13]. While this streaming functionality is fully supported, the videoplayer cannot handle playing this stream correctly for all video formats. The Tribler team is currently working on the next beta version of Tribler that will actually support streaming any video file, making our tool that much more useful.

The channel functionality is based on a library called 'Dispersy', which takes care of synchronizing communities around a channel filled with torrent files [14]. We use this functionality to keep the Tribler platform clean of duplicate torrent files and make sure they are stored in an appropriate location.

3.2.3 wxPython

This is a package of wx bindings for Python. There are several versions of the wxPython bindings released [15]. The version of wxPython used by Tribler is version 2.8. The official, stable release is the 2.8 release. This release is the version used in the Ubuntu package manager.

The official development wxPython is the 2.9.4.0 release. The development release of 2.9.4.0 is also the release that makes the wx WebView class available, which we used for implementing a web browser in Tribler. This meant that we had to port Tribler from wxPython 2.8 to wxPython 2.9.4.0. The amount of work that went into porting Tribler to this new wxPython version was very minimal however and did not cause any major instability to the existing code.

The downside to using wxPython 2.9.4.0 is that Linux users will have to compile the package themselves, which we would like to rank as non-trivial. We ourselves spent several days compiling the package before it worked.

While wxPython has the up-side of being completely cross platform, another downside to it, is that it is very unstable. The threading in wxPython is particularly unsafe. Any parallel GUI calls will cause the back-end to segfault out of execution or shut down by error.

3.3 Architecture

The first proof of concept we created in the orientation phase was able to parse the HTML structure (called the DOM tree) of a webpage and locate an image based on a *name* value given to the image. A python package called BeautifulSoup [16] would handle walking through the HTML structure. When the given *name* identifier was encountered, we would retrieve the object's *src* field value. The url pointed to by this *src* field would then be downloaded and put into a torrent file. This torrent file was then registered (and therefore shared) through Tribler.

4 Eternal Webpages phase

In the Eternal Webpages phase we were focused on the technical aspect of resource (re)location and identification. The problems we faced in this phase were on a more intellectual level than simply creating a torrent file for some image. Now we had to resolve conflicts and store resources properly.

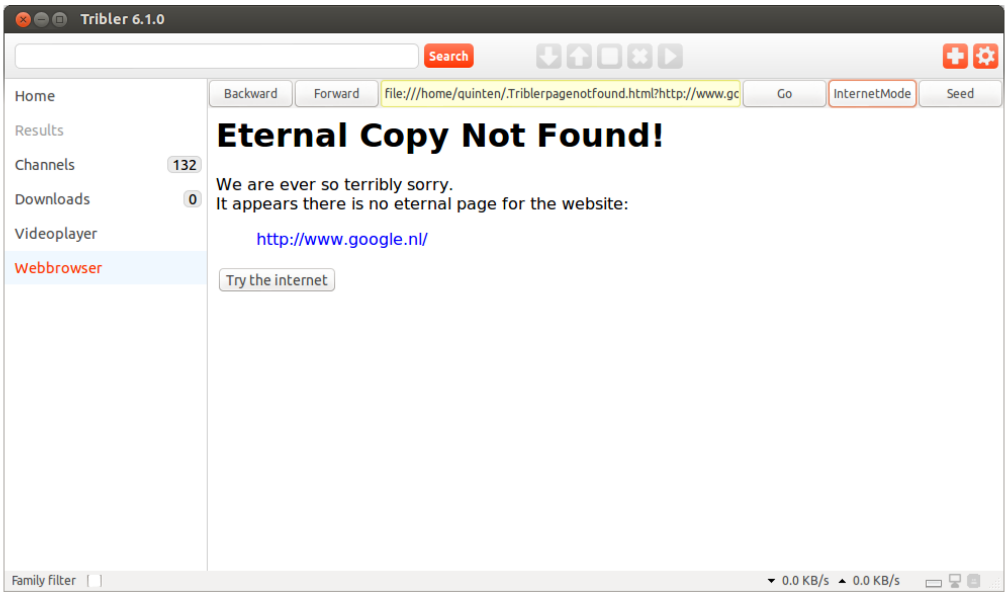
4.1 Milestones

- Seed a WebPage.
  This sprint would result in a prototype that could, with the click of a button, seed the webpage that was currently being viewed in the webbrowser. The webpage would then have all its resources downloaded and added to a tar file, which would be seeded on Tribler.

- Display a Web Page retrieved from Tribler
  This sprint would result in a prototype that would open a webpage from a local tar file by unpacking it and then displaying it in the webbrowser using the local content.

- Search for a WebPage in Tribler
  This sprint would result in a prototype that would use an offline and online viewing

mode where the offline viewing mode automaticly downloads a torrent according to a specified url. This sprint built on the previous two.



file:///home/quinten/.Triblerpagenotfound.html?http://www.gc

Eternal Copy Not Found!

We are ever so terribly sorry.
It appears there is no eternal page for the website:

http://www.google.nl/

Figure 3: A screenshot of the Eternal Webpages product in action.

4.2 Architecture

The achitecture of the Eternal Webpages extension can be divided into two parts: the WebBrowser class and the SiteRipper module. The extension is built into Tribler as part of the GUI: an extra tab is added to the menu that allows users to go to our part of the system.

4.2.1 WebBrowser class

The WebBrowser class is responsible for displaying the webbrowser itself and the seed-button. It consists of one GUI class that encompasses a wxWebView object and provides all browsing functionality needed. The wxWebView object natively supports HTML5 and provides us with everything we need in order to display webpages. The webbrowser object can be manipulated in a way that you can load WebPage objects from a file. See Figure 3 for the final result.

When we found out that wxPython 2.9.4.0 did not support resource handlers, which were essential to ripping out all the resources from a web page, we had to switch to a version of wxPython that did support it. This is how we came to use wxPhoenix, the latest iteration of wxPython.

The wxPhoenix package is an in-development package that has not been released yet and has to be downloaded from the versioning repository from wxPython [17]. It is easier to compile than the development release of wxPython and has better documentation. The wxPhoenix package was named Phoenix, according to the site, because it broke everything and was reborn better. A lot of code of wxPhoenix is refactored and there is no backwards compatibility. From an objective standpoint this is the truth, the wxPhoenix package architecture is made much better than the wxPython 2.9.4.0 package. Sadly, this major refactorization of code meant that Tribler broke down completely. Even after a week of bug hunting, Tribler still wasn't completely stable when we left the Eternal Webpages project.

4.2.2 SiteRipper module

The SiteRipper module is responsible for being able to store and retrieve webpages in torrentfiles.

Seeding

To be able to seed a webpage all resources have to be known. This is why a ResourceSniffer object detects all resource calls made by the WebBrowser when browsing and records all the resource URLs on particulair webpage. The actual resources are not saved, but only their URLs. This is a design choice we made with the fact in mind that most web pages would not be seeded, therefore keeping track of all the actual resource files for every web page loaded would hurt the performance of web browsing too much.

After the seedbutton is pressed a new folder is made. Inside that folder the HTML source and all resources of the webpage are saved to file. After everything is downloaded the folder is compressed to a tarfile and the tar is added to a torrent. This torrent is then added to Tribler and Tribler starts to seed the torrent.

To map a URL to a local file resource efficiently, we can not simply store a resource with the name of the URL. Storing an additional mapping file per web page package is inefficient. Trying to modify the links in a web page itself is infeasible because of javascript dynamicly building links. Therefore we came to the conclusion that storing resources by MD5-hash was the best way to do real-time mapping of resources. This meant that there was no need for additional files, file names are short and there is very little chance of resources being unlocatable.

See Figure 4 for a activity diagram of seeding a webpage.

View mode based web browsing

View mode based web browsing is the idea that you can freely change modes when viewing a web page and change the way a webpage is retrieved accordingly. WebBrowser

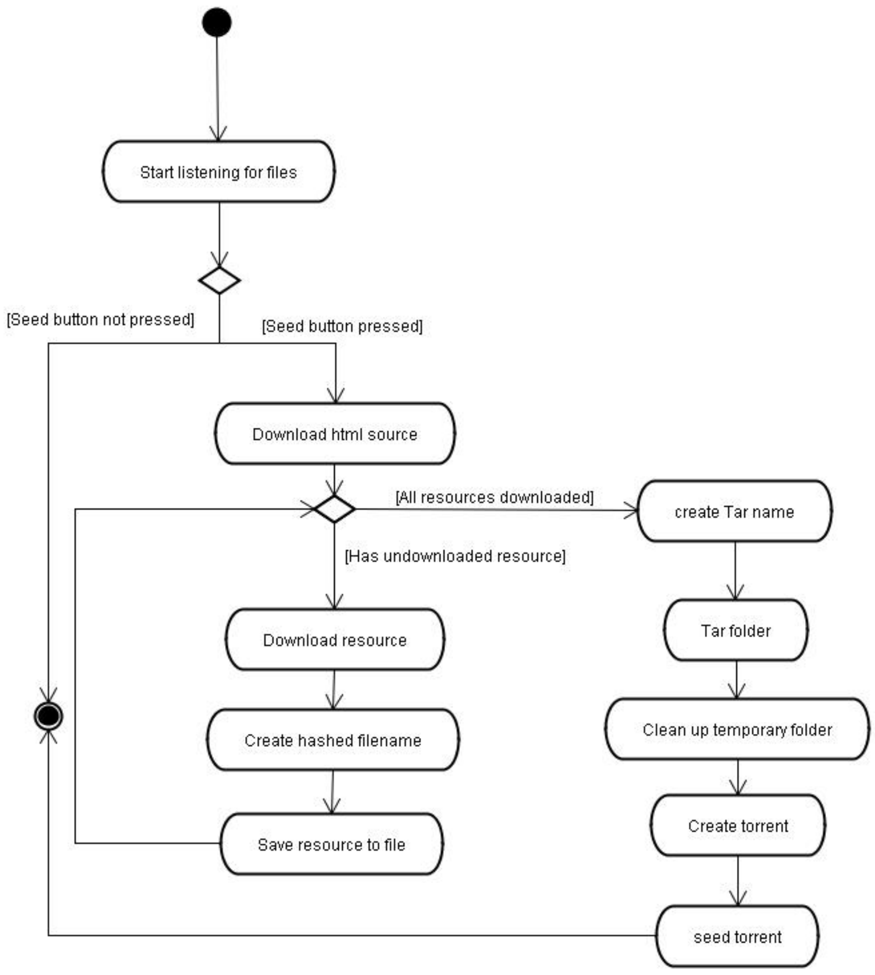


Figure 4: Activity diagram of seeding mode webbrowsing.

has two viewmodes: internet mode and swarm mode.

In internet mode the webbrowser functions like normal. All webpages' HTML sources are retrieved using HTTP by URL. The webpage starts showing as soon as the HTML source is downloaded. Any links to resources in the HTML source are downloaded using HTTP and these resources are dynamically shown. This is the same way any other webbrowser would display a webpage and everything is handled by the backend of wxWebView.

The other view mode is swarm mode. In this mode URLs would be transformed into torrent identifiers and downloaded from Tribler. Inside the torrent a tarfile is stored that contains everything that is needed to display the webpage. The tarfile inside the torrent is unpacked and the HTML source inside is loaded into the wxWebView object. Any resource calls are now not satisfied by downloading the resource from the internet, but this time the URL to the resource is translated to its corresponding hashfunction and the file is retrieved from file. See Figure 5 for a activity diagram of webbrowsing.

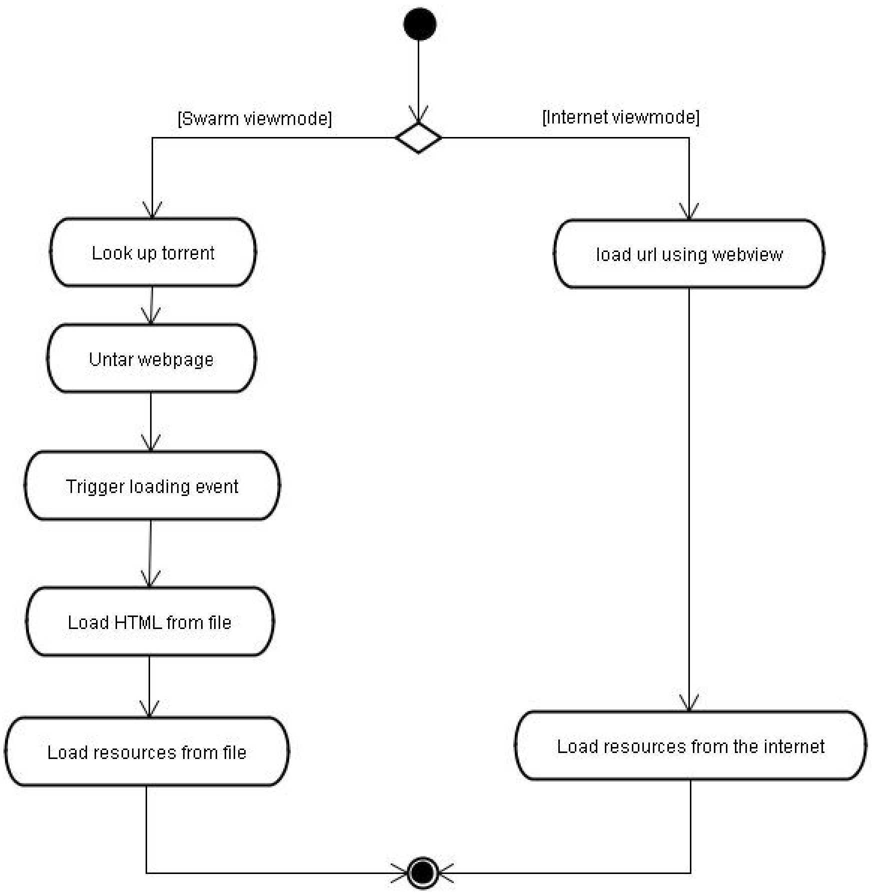


Figure 5: Activity diagram of viewmode based webbrowsing.

4.3 Code quality

The Eternal Webpages product was the first of the two products we created. This means we were still in the phase of learning about Python while we were creating the product. While this does not mean code quality and test coverage are terrible, they are definitely worse than in the TUPT product.

4.3.1 Code quality

Our own opinion of the code quality of the Eternal Webpages package is that the code comments (also known as docstrings) are proper, but the code itself is slightly lacking.The code blocks are too large and too complex. The coupling between the classes is too high. Lastly the package coupling is also too high. As mentioned before, wxPheonix also makes Tribler too unstable.

The SIG feedback for the Eternal Webpage package is that there exists a block of code in our WebBrowser initialization code, which is too long. We agree with the SIG in their finding and we split this GUI code into several different methods in the final version of our product.
This is an issue we did know about, but ignored in favor of other higher priority jobs. This is bad etiquette on our behalf and because of our laxity the same issue existed and had to be changed in the TUPT package as well .

4.3.2 Test coverage

Our own opinion of the test coverage we provided for the Eternal Webpage package is that it is lacking very much. This is mostly due to the fact that we have high package coupling though, and can therefore not easily seperate functionality into unit-tests. Most code also needs real world interactions with a webpage. This also limits the ability to unit test. If anyone were ever to continue work on this package, this would have to be improved.

4.4 Licenses

The Eternal Webpages product does not utilize any functionality other than that, which was already included in Tribler. This means that the Eternal Webpages project shares the Tribler LGPL Open Source license. We do not impose additional restrictions to our product.

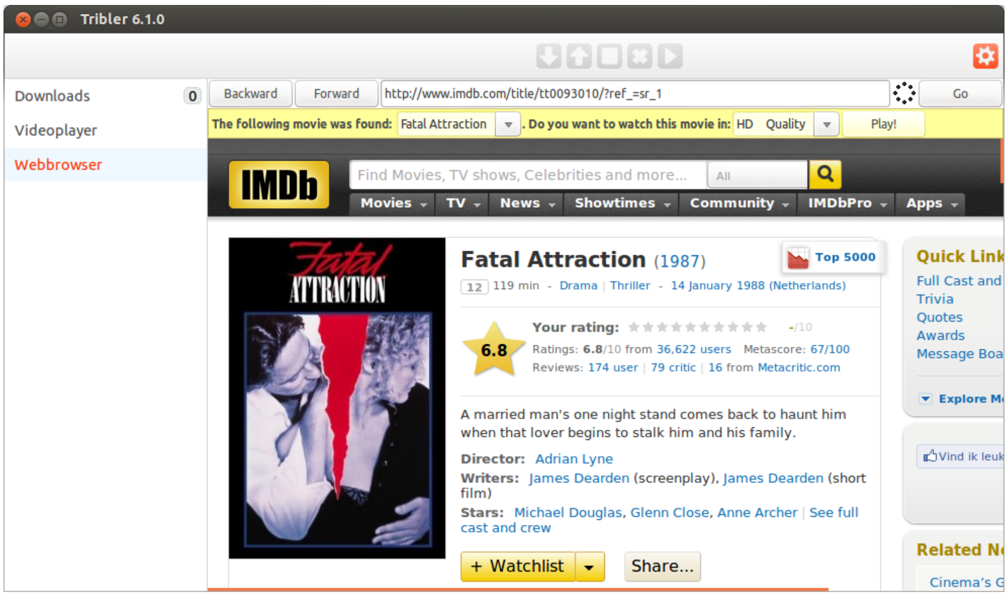
5 TUPT phase

The TUPT phase was by far the largest phase in terms of goals and targets. The software we implemented in this phase was more end-user centric and the back-end was more complicated. Our personal goal in this phase was to create something that people really wanted to use (see Figure 6 for the result).

5.1 Milestones

In the TUPT phase our milestones were:

- Increase stability of Tribler



http://www.imdb.com/title/tt0093010/?ref_=sr_1

The following movie was found: Fatal Attraction . Do you want to watch this movie in: HD Quality

Fatal Attraction (1987)

12 119 min - Drama | Thriller - 14 January 1988 (Netherlands)

6.8 Your rating: -/10
Ratings: 6.8/10 from 36,622 users Metascore: 67/100
Reviews: 174 user | 79 critic | 16 from Metacritic.com

A married man's one night stand comes back to haunt him when that lover begins to stalk him and his family.

Director: Adrian Lyne
Writers: James Dearden (screenplay), James Dearden (short film)
Stars: Michael Douglas, Glenn Close, Anne Archer | See full cast and crew

Figure 6: A screenshot of the TUPT product in action.

This sprint would result in improvements done to the software that would increase the stability of Tribler and our extension.

- Be able to play a movie on a website with one click
  This sprint would result in a prototype that could view any webpage and the webpage would be automatically be parsed for possible movies. The parsed movies would be corrected according to multiple matchers and be ready to stream by using torrents provided by several torrent providers.

- Add torrents to channels in Tribler
  This sprint would result in downloaded movie torrents to be added to Tribler channels without duplication between the channels. This functionality would be based on the correct movie meta data.

- Streamline user experience using TUPT
  This sprint would result in a prototype that would better inform the user about what was going on while the user waited. It would show website loading status and torrent retrieval loading status.

5.2 Architecture

The TUPT architecture is fairly simple and consists of four parts: the Webbrowser module, the TUPT module, the Infobar module and the plug-in module.

5.2.1 Webbrowser module

The Webbrowser module is still responsible for displaying the webpages. Listeners can be registered to it and these will be notified if a webpage is loaded. This Observer pattern allows decoupling of the Webbrowser class and any class dependent on state changes in Webbrowser in a generic way. The pattern was implemented because TUPT control needs to be notified when a new webpage is being loaded.

In the TUPT phase we switched back to wxPython 2.9.4.0 instead of wxPhoenix for stability reasons. Fixing Tribler to work with wxPhoenix was not our main objective and the Eternal Webpages prototype was too slow in loading web pages.

5.2.2 TUPT module

The TUPT control is the core class of the module that controls the flow of retrieving the movie options, showing the options to the user and processing the user interaction to start playing a movie. The TUPT control listens to the Webbrowser and will recieve the page HTML source when a new page is loaded.

The flow that the TUPT control class controls consists of the following steps:

- Parse the HTML source for movies.
- Correct the metadata of the movies.
- Find torrent corresponding to the movies.
- Display the movie options to the user.
- Start streaming the movie.
- Add the movie to the Tribler channel.

For every step the TUPT control class does not execute the step itself, but only calls the appropriate class. This follows the single responsibility principle and allows functionality to be easily reused in other places without any trouble.

See Figure 7 for a activity diagram of the TUPT control.

Parsing

Parsing is done by the ParserControl class. The control will determine if it has a plug-in that can parse the webpage. If it has a plug-in that can parse the website it will run that plug-in on the webpage HTML source. This is done defensively by not allowing the plug-in to crash the whole program and checking the results of the plug-in on correctness.

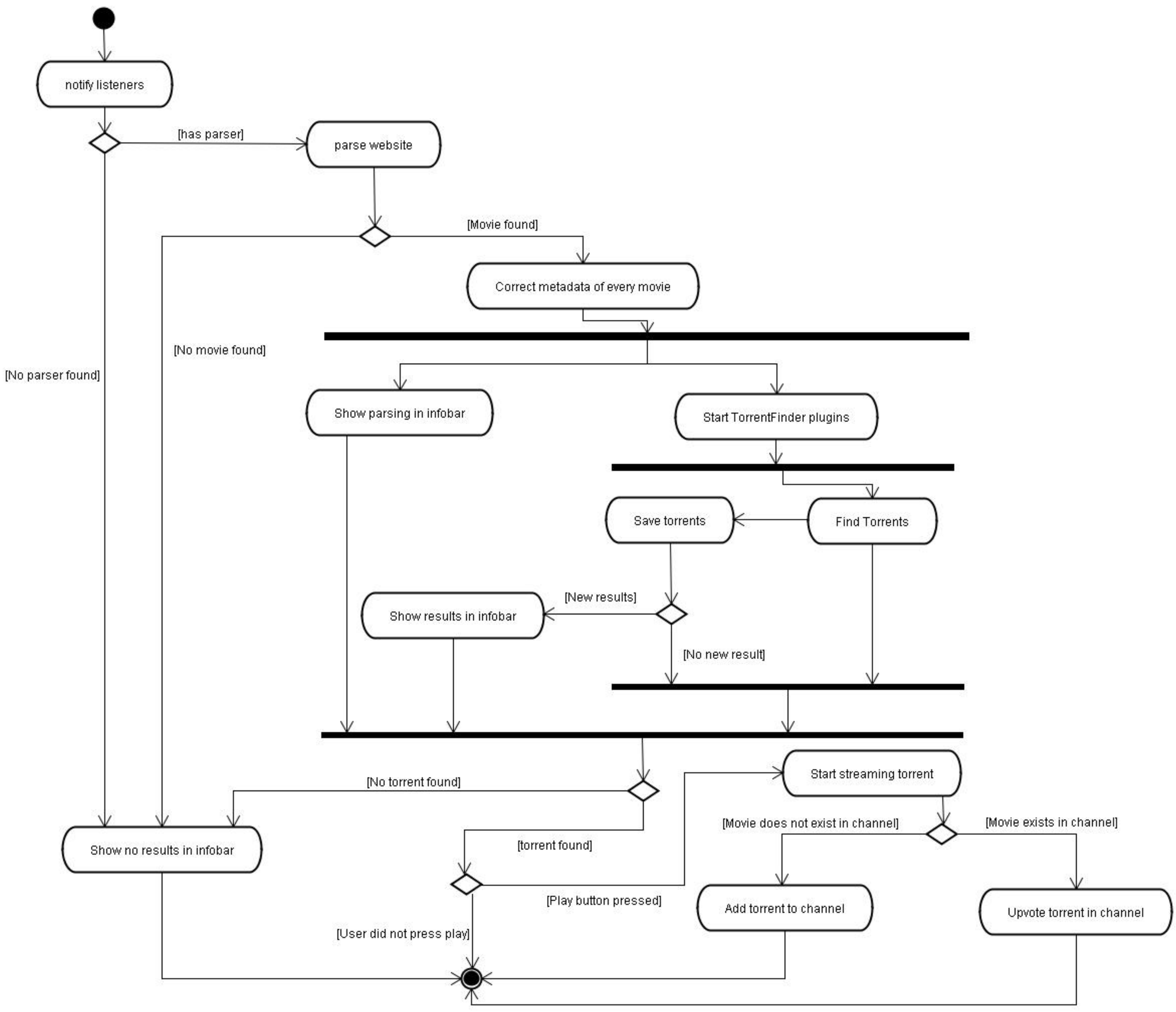


Figure 7: Activity diagram of general TUPT functionality.

Parser plug-ins have to implement the IParserPlugin interface. This will ensure the plug-in has all the functionality needed by the ParserControl.

We created a parser plugin that can parse imdb.com by using the IMDbPy package. The IMDbPy package is a screen scraping package for imdb.com [18]. It can process different imdb web pages and find movies specified on a certain page. We would use this functionality to parse results for specific query URLs and to try to figure out if a current web page being viewed, contained movies.

Correcting metadata

The movie metadata correction is done using multiple sources. Multiple sources can return different values for a single metadata attribute. The final value is determined using a trust-based frequency mechanism. This system allows an end-user to specify a trust for a certain movie database. For example see Table 1 : website A might be correct 90% of the time, website B is trusted for 60% and website C has so many spelling errors and other problems that it is only 20% trustworthy.

Table 1: Example of multiple metadata providers.

(a) Attribute-value pairs of A (trust 0.9)

Attribute	**Value**
title	Example
releaseYear	1947
discs	5

(b) Attribute-value pairs of B (trust 0.6)

Attribute	**Value**
title	exaMple
releaseYear	1946
actorCount	12

(c) Attribute-value pairs of C (trust 0.2)

Attribute	**Value**
title	example
releaseYear	1947
add	Visit my website!

What happens next is the construction of a frequency table per attribute-value pair for the movie. For every attribute a plug-in claims to have for a movie, a value is voted upon with the trust we have for a plug-in. A minimum trust of 0.5 is chosen to filter out very untrustworthy sites from making it into the corrected metadata. An example of such a frequency table is given in Table 2.

In the example the different and conflicting values given by plug-ins A, B and C from Table 1 are inserted. The rows that have insufficient trust value to make it to the final table have been greyed out. When the unwanted values have been removed the resulting

Table 2: Trust-based multi-source metadata correction.

Attribute	**Value**	**Trust**
title	Example	0.9
	exaMple	0.6
	example	0.2
releaseYear	1947	1.1
	1946	0.6
discs	5	0.9
actorCount	12	0.6
add	Visit my website!	0.2

Table 3: Final movie metadata

Attribute	**Value**
title	Example
releaseYear	1947
discs	5
actorCount	12

metadata for a movie is left over. The final metadata table for our example is shown in Table 3.

The MatcherControl class takes care of this functionality. It runs defensivly every matcher plugin and records the results given by the plug-in. After all plugins are run the final values are determined and returned. We created two plugins that use a source to correct metadata. The IMDbMatcher plugin and TheMovieDbMatcher plugin, that respectively use imdb.com and themoviedb as a source to correct metadata. See Figure 8 for an activity diagram of the MatcherControl.

Torrent finding

The software has to ensure only quality torrents are presented to the user, because if the torrents presented to the user are of a too low quality, it will hurt the user experience. This will also hurt the image and adoption of our product. We thought of two ways to ensure quality: downloading and reviewing by the user and user voting.

Downloading and reviewing would work as follows. First only the first few megabytes of a torrent's video file are downloaded. This file will then be opened and screened to see if it was not all black, if it has sound and if the pixel ratio on screen is good enough. The big drawback of doing it like this, is that it takes a very long time to evaluate a torrent, in the range of several minutes. The time it would take to evaluate several torrents is too long to be a viable option.

Manual voting requires users to vote for a movie after they have watched it. This is

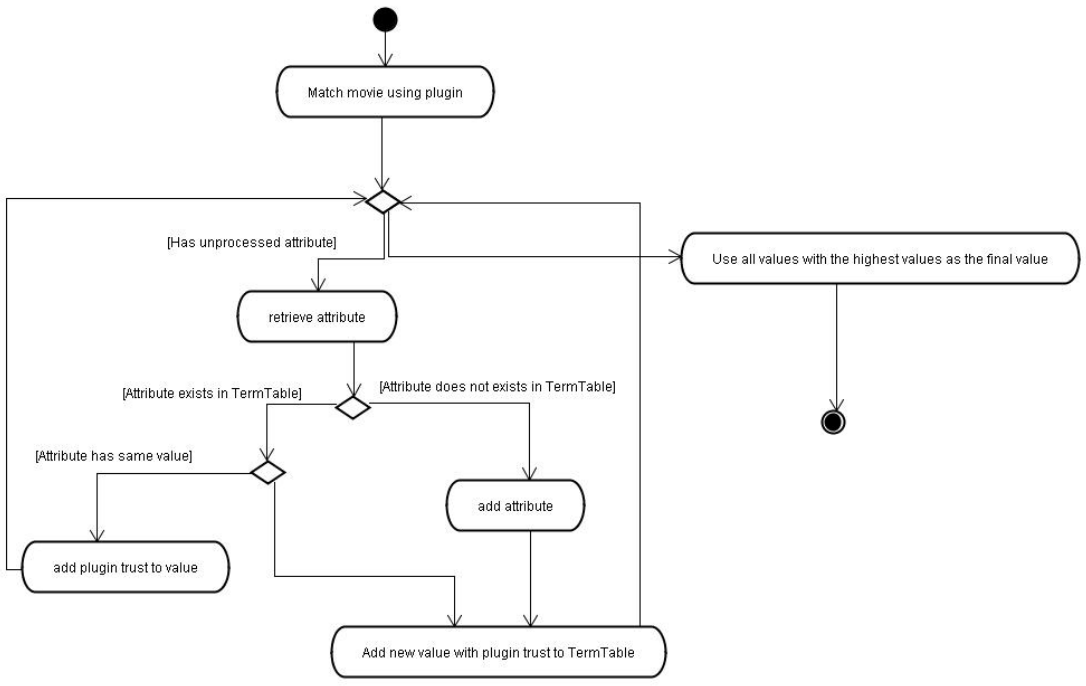

Match movie using plugin
[Has unprocessed attribute]
Use all values with the highest values as the final value
retrieve attribute
[Attribute exists in TermTable]
[Attribute does not exists in TermTable]
[Attribute has same value]
add attribute
add plugin trust to value
Add new value with plugin trust to TermTable

Figure 8: Activity diagram of the Matcher sub-package.

dependent on users both voting correctly and voting at all. An automatic system could be also implemented, which would autovote after a user has watched a movie for a certain percentage. This special voting system would be unique to Tribler: so every movie would have to be downloaded, viewed and ranked through Tribler, before we could determine whether the torrent is any good. After this initial set-up the voting makes it very easy to identify good torrents. The drawback is that this set-up may take days, weeks or even years: making new entries very slow as well. Another drawback is that the voting system is limited to Tribler and it does not take into account other valuations of users not using Tribler.

In the end however, it occurred to us that a voting system is already in place and this voting system transcends Tribler. The amount of peers connected to a torrent (seeders and leechers) can be seen as a way people vote on the quality of a torrent. A good torrent will have the crowd to support it. Any torrent that has inferior quality will also have inferior support. This method is much faster than the other two methods and it also uses input from peers outside Tribler. It still has the same drawback as the voting systems inside Tribler. If there is no crowd then our tool can not guarantee getting a quality torrent. But this drawback is far less because the crowd is much larger then the voting system inside Tribler would have.

Torrent ranking

To construct a torrent ranking algorithm we had the following inputs to utilize: the number of seeders for a torrent, the number of leechers for a torrent, the name match rate of a torrent's name with the movie data, the name match rate of a torrent's name with some user defined terms and lastly the user's trustedness of the plug-in. We will describe these inputs and how we take them into account in the next paragraphs.

Matching
In this context the match rate of a list of terms with a single name is determined by the summation of the matching rate of every term in the list with the single name:

$$name\ match\ rate = \sum_{i=0}^{n} ratio(term[i], name)$$

Where the $ratio()$ function is defined on $[0.0, 1.0]$.
Note that the *name match rate* may be anywhere in the range of $[0.0, n\rangle$ for a list of terms that is very much alike itself. If all terms were fully disjunct however, the *name match rate* would be in the range of $[0.0, 1.0]$.
Take, for example, the disjunct set T's matching to the string "abc":

$$\begin{aligned} T &= [\text{``}a\text{''},\text{``}b\text{''},\text{``}c\text{''}] \\ name\ match\ rate &= \sum_{i=0}^{2} ratio(T[i], \text{``}abc\text{''}) \\ &= \tfrac{1}{3} + \tfrac{1}{3} + \tfrac{1}{3} = 1 \end{aligned}$$

However if T is not disjunct:

$$\begin{aligned} T &= [\text{``}a\text{''},\text{``}b\text{''},\text{``}ab\text{''}] \\ name\ match\ rate &= \sum_{i=0}^{2} ratio(T[i], \text{``}abc\text{''}) \\ &= \tfrac{1}{3} + \tfrac{1}{3} + \tfrac{2}{3} = 1\tfrac{1}{3} > 1 \end{aligned}$$

This means that some terms in the term table have more weight than others when matching. This is not expected behavior of a matching algorithm, but in our case we assume the term table is disjunct enough, for simplicity of the algorithm.

It is possible to fix this behaviour though. In our usage, the strings to match to are dynamic and the term table T stays the same. Therefore the term table could have the term covariance calculated at initialization to preserve algorithm speed.

Speed estimate
To estimate the potential speed a torrent may achieve, we can use statistical analysis. Let u be the average upload speed of a peer in a swarm, S the amount of seeders and ,

L is the amount of leechers. Assume a leeching peer's download progress be uniformly distributed over the interval $[0, 1\rangle$. Assume a leeching peer will attempt to upload every part of the file the leeching peer itself has.

Now the expected value of the download progress for each leeching peer is 0.5 (or 50%):

$$E[U] = \mu = \frac{1}{2}(\alpha + \beta) = \frac{1}{2}(0 + 1) = 0.5$$

This means that on average a leeching peer can upload 50% of a file at upload speed u. In turn this means that the average upload speed of a leecher l for en entire file is $u \times 0.5 = 0.5u$. This makes the total estimated available download speed for a torrent equal to the sum of the expected download speed from the seeding peers and of the expected download speed from the leeching peers:

$$\textit{download speed estimation} = uS + 0.5uL = u(S + 0.5L)$$

We note that this estimation is based on a uniformly distributed download completion rate, whilst in reality it may be more of a normal distribution skewed to the right. Additional research is required to confirm this.

We also note the fact that starting leechers are not fully connected to the swarm yet. This means we would discover more peers with higher completion rates. This would further shift the estimated download speed to the right.

All notes combined, we assume our $1.5u$ download speed estimate is too low, but we do not have the resources to pursue this problem further.

User terms
This algorithm also takes into account some trusted terms set by the user. In the context of movies this may be a quality term like “1080p”. This would prioritize a torrent with the name “Some Movie 1080p” over another movie named “Some Movie 720p”. We have chosen to let the user define these quality terms, for they may shift over time. This also leaves room for expansion to different content other than movies. For example we might prefer series from the “BBC” over any others.

Torrent finding algorithm
Finally, to tie everything together we will multiply the user trust, the name match rate of the torrent's name with the movie data and the name match rate of the torrent's name with the user terms with the estimated download speed for a torrent.

Here we make sure that the different matchings can only increase the rank of a torrent. Hence, not having the special term “BBC” will not decrease the rank of a torrent, or for example not having a movie description. What should be able to decrease the rank of a torrent, is the trust the user has in the torrent provider. For example, if we trust a website for 0%, we don't want it in our list at all, even if it has a near infinite amount of seeding peers.

The final function for determining a torrent's rank then becomes:

$$rank = trust \times (1 + name\ match\ rate(movie, torrent)) \times (1 + \\ name\ match\ rate(user\ terms, torrent)) \times download\ speed\ estimation$$

It should be noted that this rank is in units of avarage download speed per torrent swarm. For simplicity we assume that this average speed is the same for all torrents. In reality however, it won't be because different torrents will attract people with different internet speeds.

The TorrentFinderControl class is responsible for finding torrents of a movie and ranking them. It uses different plug-ins that will all retrieve torrents from a different source. Every plug-in is run defensively so that no crashes in a plug-in will crash the rest of the software. The torrents are ranked using an algorithm we will describe later. We created three plug-ins that use Kat.ph, Fenopy and Tribler's overlay network as sources for torrents.

See Figure 9 for a activity diagram of the TorrentFinderControl.

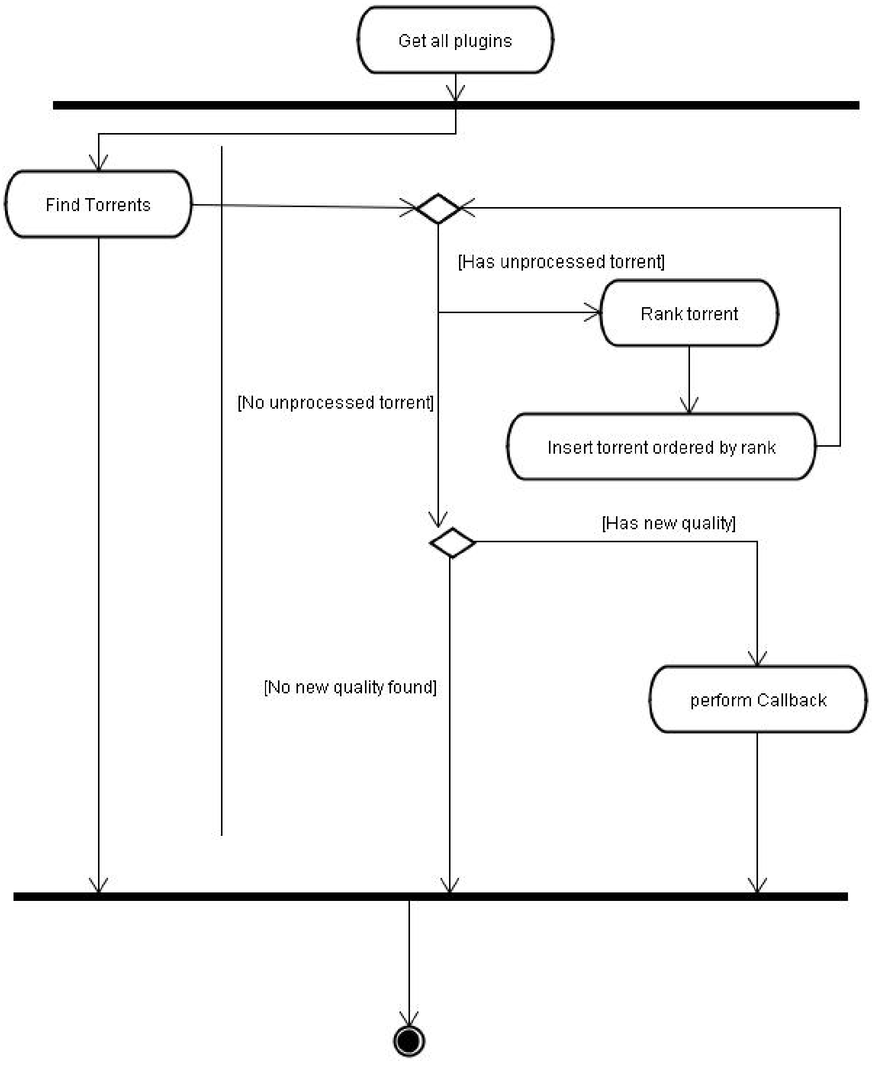


Figure 9: Activity diagram of the TorrentFinder sub-package.

Display the movie options to the user

The TUPT control uses the infobar module to inform the user of the status of the TUPT extension. The module has several states and the module itself checks which state it is in. This is done using the information stored in the MovieIterator object. Its states are:

- Parsing: a movie was found but there are no torrents yet.
- Movie: a movie was found and a torrent was found for this movie.
- No Results Found: no movie was found or no torrent was found.

When a TorrentFinder plug-in has finished finding torrents from its source, it will call-back to the TUPTControl if it has new results. The TUPTControl will instruct the TorrentInfoBar to update its state and show the new options to stream.. After all TorrentFinder plug-ins have run for all movies found on the webpage, the TUPTControl will do a final call on the TorrentInfoBar. If no torrent was found the infobar will go from parsing state to no result found state. See Figure 10 for a activity diagram of the infobar.

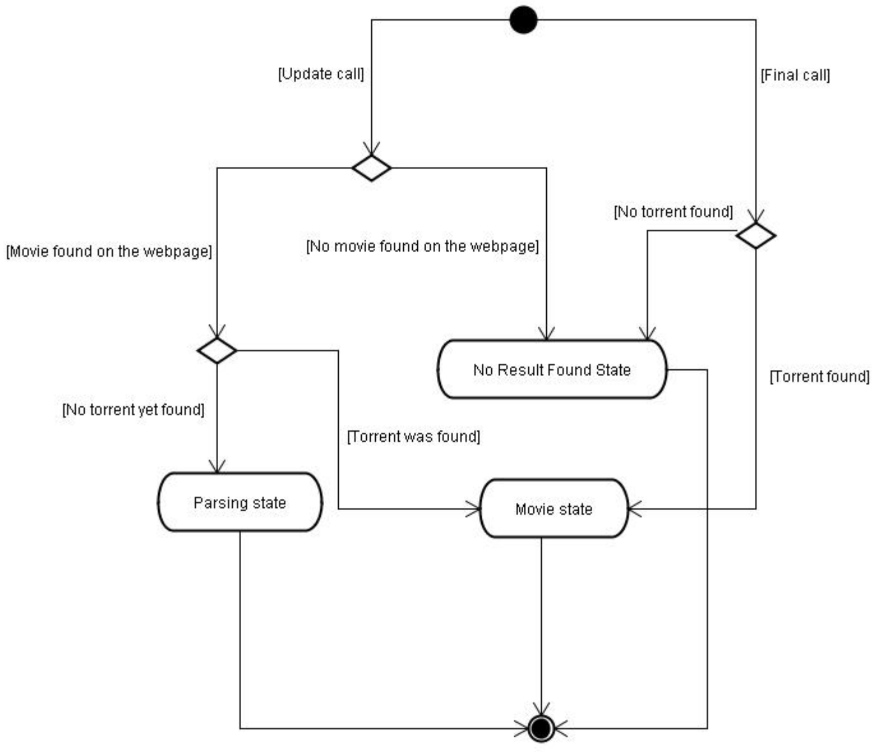


Figure 10: Activity diagram of the TorrentInfoBar.

Streaming a torrent

After the user has pressed the play button the TorrentInfoBar will return the selected torrent to the TUPTControl. The TUPT control will download the torrent regardless if it is a magnetlink or a normal torrent file. The torrent is then added to Tribler like a regular download. After the torrent download is started, Tribler is called to start streaming the movie.

Channel insertion

To distribute torrents within Tribler, every user has his own channel. This channel (if it is open to outside moderation) can then be moderated by outside sources. Even though the front-end of Tribler does not support more than one channel per user, the back-end supports multiple channels per user. We have modified the Tribler database to support additional hidden channels, whilst retaining the original Tribler personal channels per user.

The channels we created are open to outside moderation to allow for easier merges between channels.

The way channels are managed is by first searching for an apt channel to store a torrent in. To do this, we need a resource that has completely correct metadata (so it shows up the same for all clients) and then distill a universal name for it. So, no matter where the resource is collected from, the identifier will still be the same.

When we have our unique identifier for a channel, one of three things can occur. The channel might either *(a)* not exist, and we need to create it ourselves, *(b)* exist already (either because we made it or someone else made it) or in the worst case *(c)* there are multiple matching channels to insert into.

In the third case *(c)* we simply select the channel with the most torrents to insert into. If it so happens that we own our own version of the channel and we are not the most popular channel, then we insert all the torrents (that are not already in the other channel) into the other channel. Now that our peer knows of this other channel, it is also easier to disperse the other channel, quickly making it the only channel of its kind through a survival of the fittest strategy.

Once we have a channel to insert our torrent into, we can inspect the channel to see if the torrent already exists. One of three things may happen now. Either *(a)* the torrent does not exist yet and we can insert it, *(b)* the exact torrent resource we found already exists as our universal name or *(c)* another torrent with the same universal name exists, but it does not point to our found torrent resource.

In the third case *(c)* we need to find out which of the resources is the best. The best torrent is determined by the highest amount of seeders for the torrent. If this turns out to be our torrent, we remove the other torrent in the channel and substitute it with our own.

See Figure 11 for a activity diagram of channel insertion.

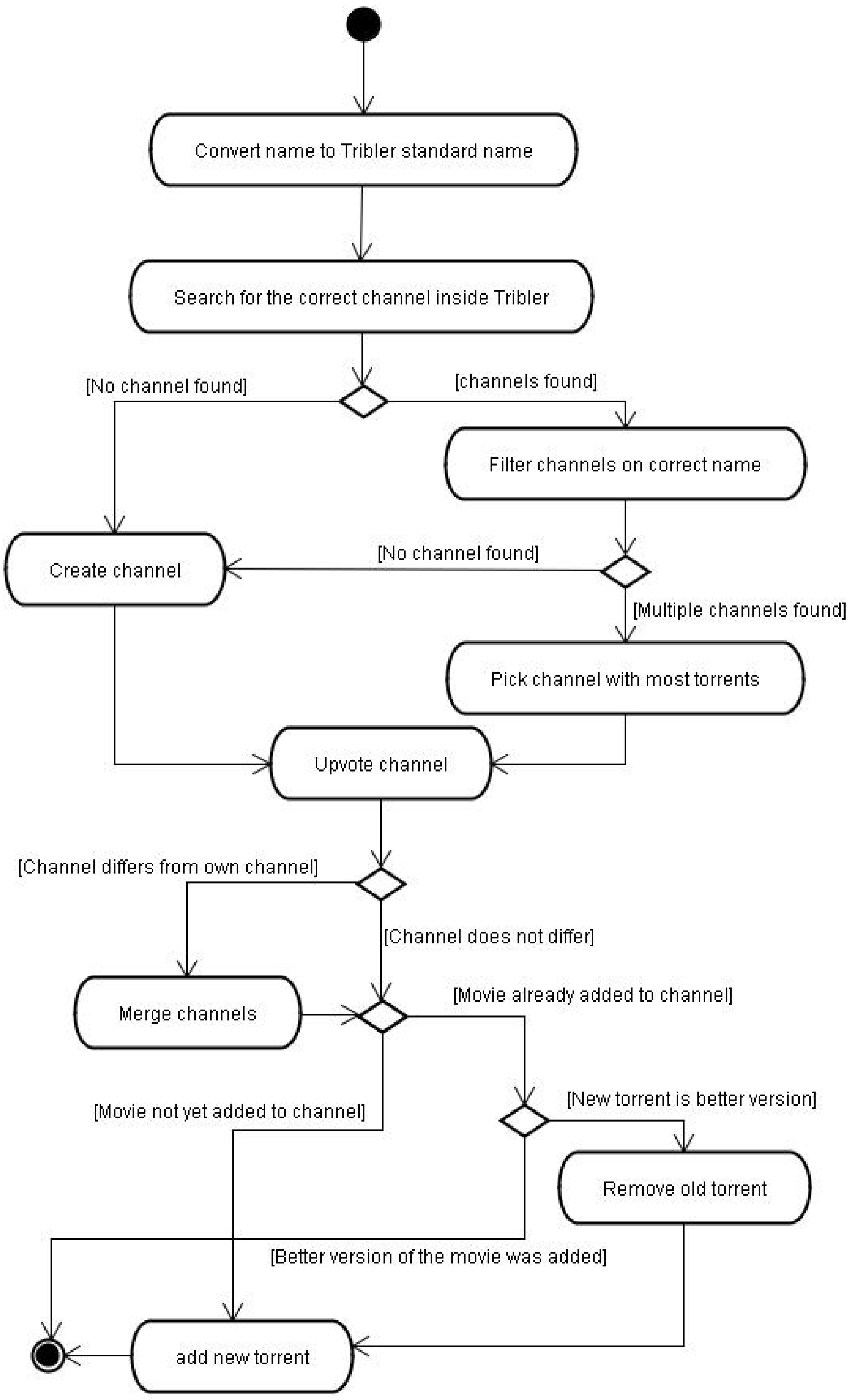

Convert name to Tribler standard name
Search for the correct channel inside Tribler
[No channel found]
[channels found]
Filter channels on correct name
Create channel
[No channel found]
[Multiple channels found]
Pick channel with most torrents
Upvote channel
[Channel differs from own channel]
[Channel does not differ]
Merge channels
[Movie already added to channel]
[Movie not yet added to channel]
[New torrent is better version]
Remove old torrent
[Better version of the movie was added]
add new torrent

Figure 11: Activity diagram of the ChannelInsertion sub-package.

5.2.3 Infobar module

The infobar module is a small GUI element that displays the results found by the TUPT extension and allows the user to choose a movie and quality and start playing the movie. The infobar gets updated asynchronously by the TUPT control. The infobar is implemented in a generic way that will allow to display any other information, but is currently only used by the TUPT control.

5.2.4 Plug-in module

We decided on a plug-in structure for two reasons. The plug-in structure allows for easy additional functionality added by other developers. Developers can easily create other plugins that our tool can use if they implement the simple interface. A parser plug-in for a new website or a torrentfinder plug-in for another torrent source can be easily added this way. The second reason is that because of licensing concerns we had to make sure that code can be easily run seperated. This allows several licenses to be applicable to different parts of the system.

Because we were now supporting plug-ins we needed a plug-in package for python. For plug-in support we then decided to use the Yapsy python plug-in support package.

Yapsy is a Sourceforge project that supports plug-ins for python [19]. Yapsy searches for plug-in metadata files in a programmer specified folder and then loads the plug-in module from the metadata file into the current environment. This also allowed us to easily add metadata of our own to plug-ins. The implementation of Yapsy was very easy.

The downside to Yapsy is that it exposes the entire inner architecture of our project and lets plug-ins do whatever they want. This means we can not guarentee our users that running user scripts is safe.

The PluginManager module handles all functionality corresponding to the plug-ins and has been built in such a generic way that it can handle any new additon of a plug-in category without any effort. It facilitates all interactions with Yapsy to provide the plug-in functionality. The plug-ins are fetched based on a category and returned as a list of objects. The objects can then be called to perfom plug-in functionality. A configuration tab has been added to the Tribler configuration menu that allows easy administration of the plugins.

5.3 Code quality

The second product we delivered, the TUPT package, has a much better code quality and test coverage than the first deliverable. This is because we started off with a better design than we had for the Eternal Webpages product. This meant we could decouple classes better and therefore test them better.

5.3.1 Code quality

We believe that the code quality we delivered for the TUPT package is very high. Besides that, we believe the architecture we designed reflects a nice separation of activity flows in its package design, the functionality can be easily extended and components can be easily reused. Class and package coupling is low. Most of the files are not bloated and methods are not longer then necessary. Code comments (docstrings) are of high quality and standardised throughout the files. For the definition of these standardized docstrings see Appendix A.

In the TUPT phase we also started using pylint to check our code quality. Pylint is a code analysis tool that can be automatically run. The first time running it introduced alot of warnings and errors. Most of these are small refactorings to adhere to the code standard, but it also found two bugs and good refactoring opportunities. This proved the value of running code analysis tools on our code. The results for the major classes are shown in Table 4 .

We did consciously disable particulair pylint warnings in lines of code. This was alway done with a good explanation why we disabled that warning. An example is that we catch a too general Exception, but this was done because of defensively execution plug-ins. We also globally disabled the warning that states that a method not referencing its class could be used as a function. We believe this rule clutters the namespace too much and is too much of a

Table 4: Pylint ratings

Filename	**Rating**	**Work**
TUPTControl	9.77	refactor amount of attributes
WebBrowser	8.99	several refactorings need to be done
MovieChannelControl	9.51	several refactorings need to be done
MatcherControl	10	no work needed
ParserControl	10	no work needed
TorrentFinderControl	10	no work needed
WebBrowser	9.92	refactor method name to code standard.
PluginManager	10	no work needed

The feedback the SIG gave us about our code quality is the fact that there is GUI code in the WebBrowser class which is too long. We agree with this finding of the SIG and will split this functionality into several different methods in the final version of our product. The SIG also mentioned the fact that our *MovieTorrentDef classes had high duplication. We also agree with this finding and will use an abstract class instead of an interface for use by the different *MovieTorrentDef classes in the final version of our product.

5.3.2 Test coverage

Table 5: Test results

Test name	**Classes**	**Coverage (%)**	**Execution time (s)**
test_PluginManager	PluginManager	86.7	0.006
test_ChannelControl	MovieChannelControl	44	0.003
test_IMDbMatcherPlugin	IMDbMatcherPlugin	87.8	2.381
test_MatcherControl	MatcherControl	86	5.634
test_TheMovieDBMatcherPlugin	TheMovieDBMatcherPlugin	100	2.416
test_IMDbParserPlugin	IMDbParserPlugin	98	0.696
test_ParserControl	ParserControl	85.5	0.001
test_SortedTorrentList	SortedTorrentList	100	0.009
test_TorrentFinderControl	TorrentFinderControl	84.5	0.020
test_TUPTControl	TUPTControl	50	0.019

The test coverage we provided with the TUPT package is rather high. Most tests score over 80% coverage, and all tests score over 44% (see Table 5). The only thing we could see, which could be held against us, is the fact that some unit-tests are slow because they require internet files. We did this because we needed to parse the imdb website which is copyrighted material and we did not want to upload this to GitHub. The unittests now also function as a good test to see if the software can parse the imdb webpage after real-world changes, but it slows down the tests.

5.4 Licenses

The TUPT product uses third party libraries, next to those used in Tribler. The integrated plug-in manager Yapsy is distributed under the simplified BSD license. This means the integrated Yapsy package does not pose any more strict rules on the license than the LGPL Open Source license already used by Tribler.

The IMDbPy package, which is used in a plug-in, is distributed under the GPL 2 license. The code can be easily seperated and run separately, because of this it does not impose restrictions on the Tribler license. This will hold for any other user plug-in. Because it can be seperated it does not have any effect on the Tribler license.

Since we do not impose any additional restrictions, the product retains the LGPL Open Source license already in use by Tribler.

6 Testing and documentation phase

To conclude the programming we did for this project, we ended with the testing and documentation phase. In this phase we would make sure that all the deliverables were of sufficient quality through additional testing and complete documentation. The additional testing consisted of code reviews (coding style, code complexity, etc.) and system testing. This quality assurance would cost us about two weeks of our time.

6.1 Bug hunting

A big part of our testing and documentation phase consisted of getting our TUPT product "Grandma Proof". In other words, we wanted to deliver an uncrashable application. We delved into the reports our tools gave us on test coverage and coding style to check for possibly problematic sections of code. In a handful of cases this actually led to us finding sections of code that behaved normally under tests, but were actually wrong.

The other part of our bug hunting was intense system testing. This meant that we would abuse our product as much as possible to discover possible errors. Even though our system is very resilient, the wx GUI package is somewhat unreliable with threaded calls. Through intense testing we could then discover deadlock situations caused by the GUI back-end and avoid these situations.

6.2 Documentation

Because our product invites expansion, improvement and end-user interaction, we wanted to make sure that the documentation we delivered with our product was of sufficient quality. In this phase we made sure our technical documentation was delivered with the product, in the form of UML class and activity diagrams. We also made sure we documented our work in a SIG technical documentation deliverable and in this report.

6.2.1 UML diagrams

A big part of understanding any code is understanding the architecture. To this end, we made a complete package of class diagrams and activity diagrams, for both the Eternal Webpages and the TUPT package. All of the activity diagrams can be found in their respective chapter in this report. The class diagrams for the Eternal Webpages project can be found in subsection B.1. The class diagrams for the TUPT project can be found in subsection B.2. We did not use any generation tool for these class diagrams and activity diagrams. We felt it would be more clear what the functionality was, if we separated the different aspects of our code by hand. It took us two days to produce all of the diagrams for this project.

6.2.2 SIG deliverable

To produce the SIG deliverable we had to make sure we had documentation on our code and a separated version of our code. Separating our code from the Tribler code was no big issue. During our project we had made sure that our code was in our own folders as much as possible. We had just a single class that existed inside a Tribler folder. We also wanted to make sure that the tests we delivered were runnable. Due to a cascaded dependency, intially only 3 out of 10 tests were runnable without the Tribler code. We then set out to decouple a single class, which instantly boosted our testable amount to 8 out of 10 tests runnable without Tribler. The final 2 test classes were nested so deeply into Tribler library code, that we felt they were not worth decoupling.

In the end, the SIG deliverable contained a document that covered the projects architecture (including the UML diagrams we made) and the test suite, the separated code for the Eternal Webpages project and the TUPT project and lastly the test code. It took us 3 days to complete the SIG documentation.

6.2.3 Final report

The bulk of the testing and documentation phase was spent creating this report. With all of the work we put into our project, we wanted to make sure we covered everything we had done. This led to the fact that we had a lot of text to write. Writing all of the text we needed to write took us 5 days. We then submitted a draft to our personal supervisor and when we were reviewing our text, we discovered we had a lot of text with little structure. This lead to the fact we would spend another three days rewriting our report to adhere to a better report structure. All in all we went through another 29 GitHub issues on our private repository, just for the final report.

7 Evaluation and conclusion

In this section we will focus on some more global subjects of our project. We will talk about both the customer's satifaction and our own satisfaction after the project. Lastly we will discuss the time sinks we ran into and the future work that can be pursued when working with our deliverables.

7.1 Customer satisfaction

To assess the satisfaction of the client, we will base ourselves on the reactions of the client during our meetings. The first reaction we got on our progress, during the second meeting, was "Well then, you've done everything we wanted, what are you going to do the coming weeks?". This tells us that we have gone above and beyond the initial project's expectations. However, over time expectations shift and in pursuing our final goal of getting our product “Grandma proof” (uncrashable) we do feel we have let our client down. So whilst we do feel we have risen above and beyond the original project, we do feel disappointed we did not manage to provide the so called icing on the cake.

This does not mean we failed or let our client down however, quite the contrary. We have both, even though we are “just” Bachelor students, been offered jobs at Tribler to continue our work. This means the client must have been impressed enough by the quality of our coding and the progress we have made, that it meets the standards of a professional programming company.

7.2 Self evaluation

The goals we had specified for ourselves in the orientation phase of the project were: to learn more about peer-to-peer communication and to prove ourselves as software engineers. We feel we have accomplished both goals.

During our work with Tribler we came in contact with several cutting edge peer-to-peer techniques. By utilizing the Tribler functions we also got to learn a lot about how decentralized tracking works, how decentralized communities are synchronized, how torrents are sequentially downloaded instead of by random piece and many more topics.

During this project we have written over 4000 lines (including tests and comments), changed 131 files, closed 130 GitHub issues and made 404 commits in 6 weeks with a two-man team. To produce so much content - which is all subject to our own high quality standards - while working with experimental packages, is a personal accomplishment.

7.3 Project time sinks

During our project we finished most of the features rather quickly (within a day or two). There were also parts of the project that did not go quite as swimmingly however.

A big hindrance during our project was using the wxPython bindings. The functionality we were using was still relatively new and in the case of wxPython 2.9.4.0: only half-implemented. The main problem with the wxPython bindings were that they would segfault or mystery error out of execution because of bad threading policy on its back-end. It was an absolute nightmare to figure out where, how and why crashes related to the GUI happened.

The better implementation of wxPython (wxPhoenix) was also a big time sink. Instead of being broken on the end of wxPhoenix, the entirety of Tribler became unstable. With no access to GUI testing tools, we had to probe around the entire Tribler environment to discover any refactorizations. This is made extra difficult by python being dynamicly typed, so not even name refactorizations were caught at compiile time.

7.4 Future work

The code we leave Tribler is very much a platform for them to expand on. We expose a great many things they can further build upon or research. In this section we explain the different things that can be altered or expanded upon in our project.

Eternal Webpage versioning
A big problem that remains to be solved for the Eternal Webpages product is how to differentiate versions of a website inside the swarm. If the content of a webpage is changed, then the swarm should be notified that a new version is available and the newest version should be preferred. It should not necessarily be guaranteed that the newest version is always displayed, but a balans should be found. This is a very difficult problem, but a very relevant one for the Parallel and Distributed systems department of the TUDelft. Problems arise very quickly when a malicious user inserts fake versions into the network.

Eternal Webpage integration
A remaining potential expansion of the Eternal Webpage product is to seemlessly integrate the entire chain into the web browser. This would mean pages are automaticly checked for availability from the internet and retrieved from the swarm if they are not available. Also, pages should be automaticly seeded in some smart fashion.

Eternal WebpageAuthenticating
Authenticating the webpages retrieved through Tribler is a problem. Currently the software has no safegaurds to ensure the webpage retrieved from Tribler is a correct replica of the webpage on the internet. Malevolent hackers can start sharing webpages that resemble webpages, but include harmfull code. A way to authenticating the webpages without going directly to the webpage itself should be designed.

TUPT plug-in development
The TUPT product thrives on having as many quality plug-ins as possible. Especially matching plug-ins and torrent finding plug-ins are useful for getting the best quality search results.

TUPT categories
The TUPT product is created in such a way that it should be fairly easy to expand it to categories other than movies. It could be used to disperse texts, books, subtitles, television series and more.

TUPT torrent ranking
The torrent finding algorithm used by TUPT is effective, but also quite primitive. Researchers with interests in ranking algorithms could probably come up with better and more sophisticated solutions than the heuristic we used.

TUPT paid services
The TUPT package can be extended to allow for paid services to include quality content.

This might be a pay-per-play construction or a subscription construction.

TUPT global parser
An extension of the TUPT product can be to create a web page parser that is capable of interpreting text and finding potential resources. This is on the level of Human-machine interaction and would require an application to understand the text on a page and find out where on the page a movie is referenced.

7.5 Conclusion

Looking back on our progress we believe we have succeeded in achieving our goals. The client and we ourselves are pleased with our end products. We overcame programming blocks in the form of malfunctioning packages and complicated problems, within a very short time span. And, in the end we delivered a quality product, which we firmly believe will attract users to Tribler. Most importantly, we achieved all of this, with just a two-man team.

The products we leave may not be perfect, but this is not the point of their creation. One of the best things about what we delivered is that they provide a lot of potential for expansion and research. This is the ultimate intellectual legacy.

A Docstrings

In our delivered code we have utilized a standardized docstrings (see Table 6). We chose a standardized format to ensure that all our comments were of high quality. We also made sure not to put too much information in a docstring, as to keep the docstrings maintainable.

Table 6: Standardized docstrings

Type	**Standard**	**Example**
Class	"""	**class** **MyClass:**
	Classname	"""
	Class description	MyClass
		This is an example description.
	Package dependencies	Filling a second line here.
	"""	
		Depends on: Test package, ExtendedMath package
		"""
Method	"""	**def** **SomeMethod**(input1, input2):
	Method description	"""
		Perform some arithmetic on 'input1'.
	Expected input types	Also prints "Hello World".
	Return value(s)	Args:
	"""	'input1' (str): This is a very long description
		for input argument 'input1'
		'input2' (SomeClass): Input2 representing something.
		Returns transformed value of 'input1' (str)
		"""

B Class diagrams

This appendix contains all the class diagrams we made for the Eternal Webpages package (B.1) and the ones we made for the TUPT package (B.2). These diagrams are here for reference only, if you wish to know about the activities and interactions of these classes you should check out the Design section of this paper.

B.1 Eternal Webpages

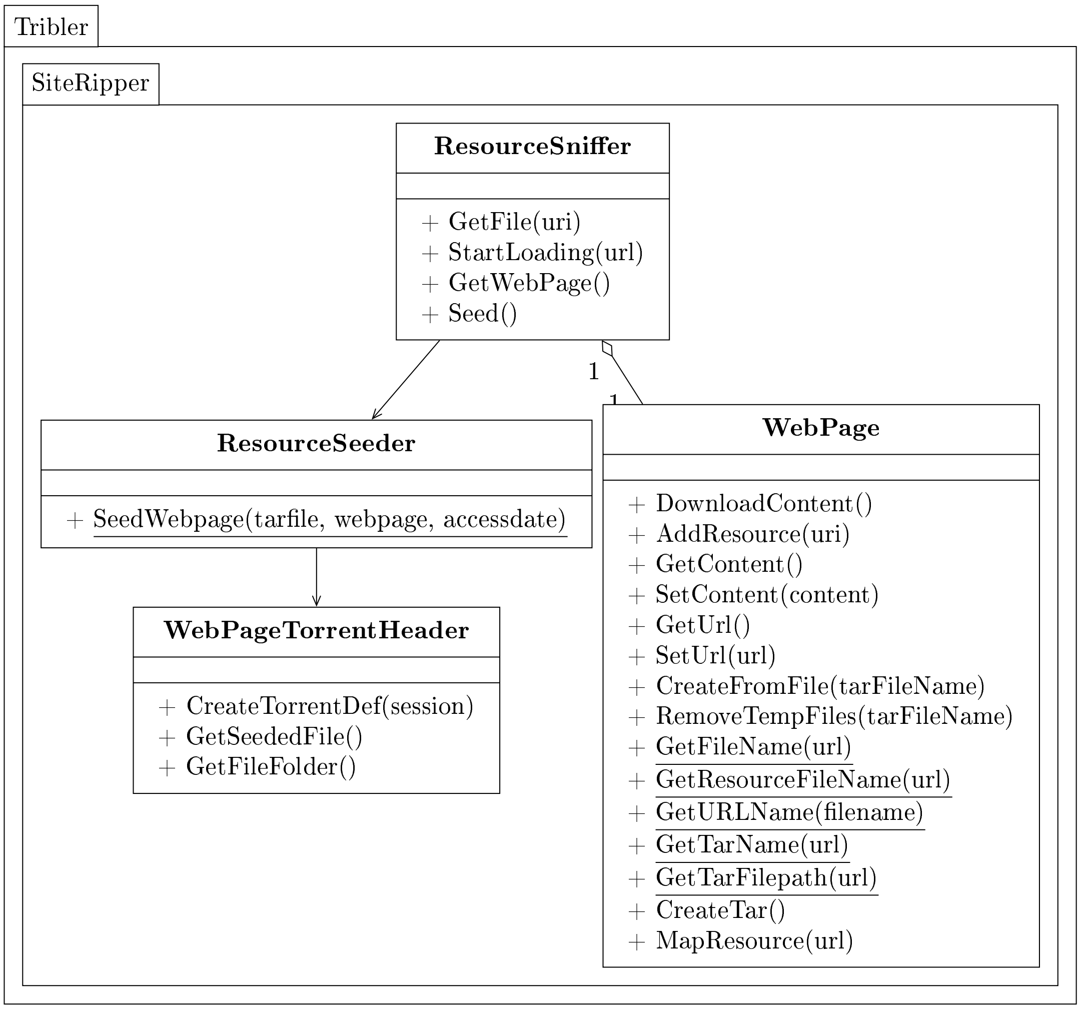


Figure 12: Class diagram for Eternal Webpages.

B.2 TUPT

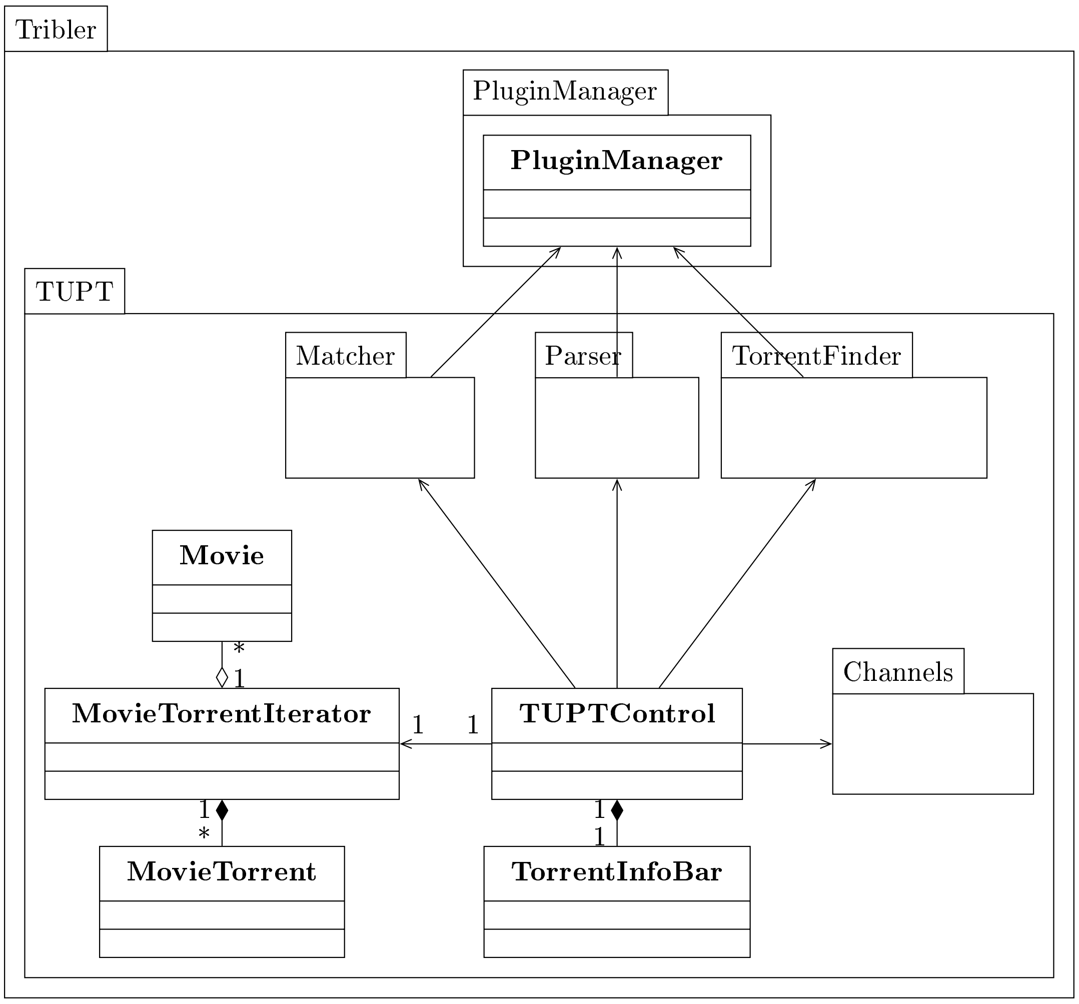


Figure 13: Class diagram for the TUPT top package.

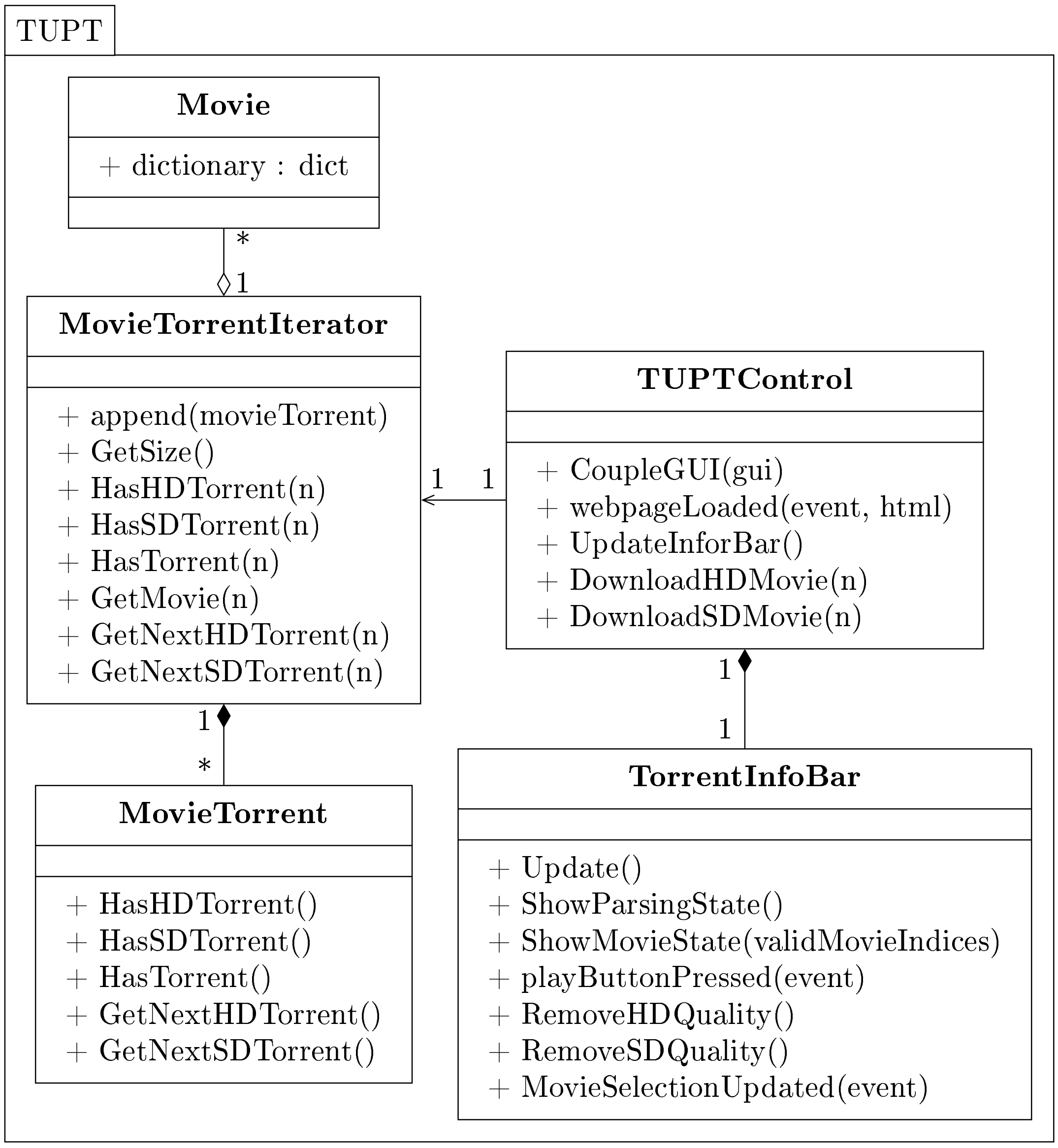
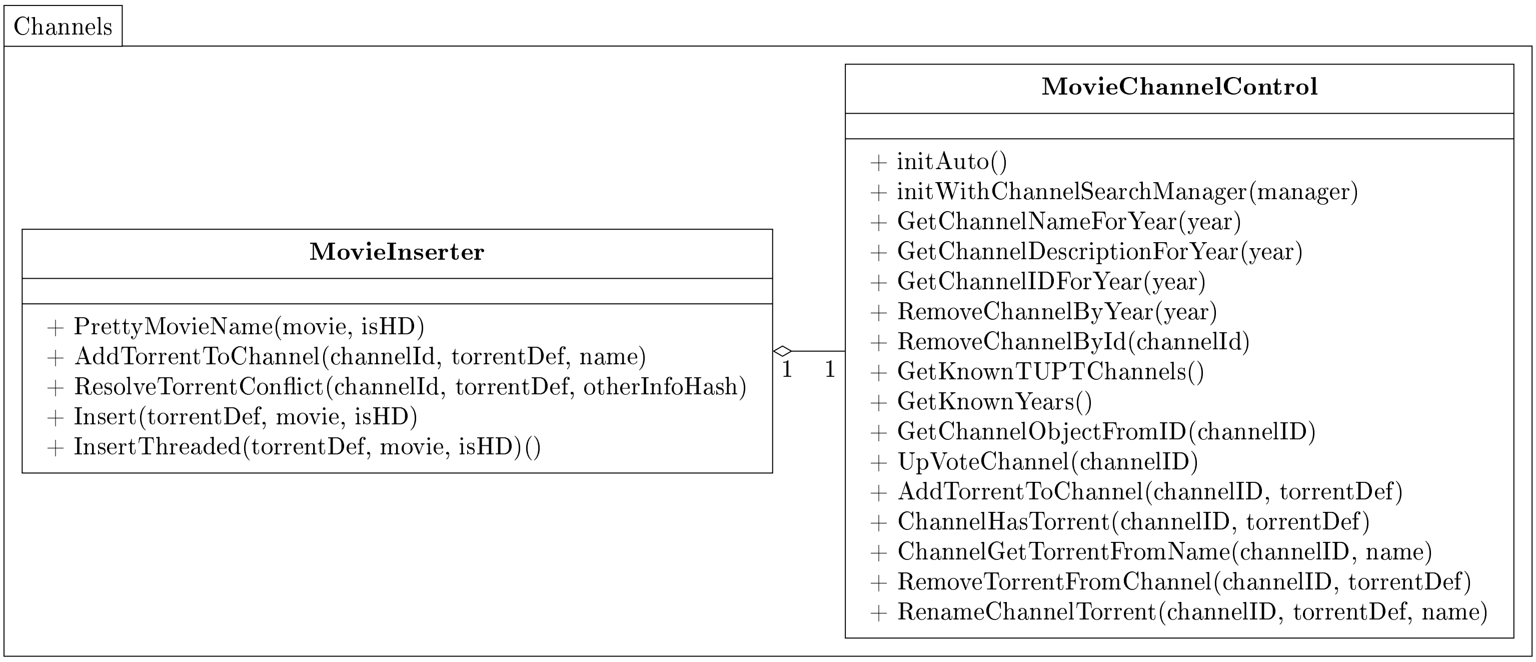


Figure 14: In-depth class diagram for the TUPT top package.



Channels
MovieChannelControl
+ initAuto()
+ initWithChannelSearchManager(manager)
+ GetChannelNameForYear(year)
+ GetChannelDescriptionForYear(year)
+ GetChannelIDForYear(year)
+ RemoveChannelByYear(year)
+ RemoveChannelById(channelId)
+ GetKnownTUPTChannels()
+ GetKnownYears()
+ GetChannelObjectFromID(channelID)
+ UpVoteChannel(channelID)
+ AddTorrentToChannel(channelID, torrentDef)
+ ChannelHasTorrent(channelID, torrentDef)
+ ChannelGetTorrentFromName(channelID, name)
+ RemoveTorrentFromChannel(channelID, torrentDef)
+ RenameChannelTorrent(channelID, torrentDef, name)
MovieInserter
+ PrettyMovieName(movie, isHD)
+ AddTorrentToChannel(channelId, torrentDef, name)
+ ResolveTorrentConflict(channelId, torrentDef, otherInfoHash)
+ Insert(torrentDef, movie, isHD)
+ InsertThreaded(torrentDef, movie, isHD)()
1
1

Figure 15: Class diagram for the TUPT Channels package.

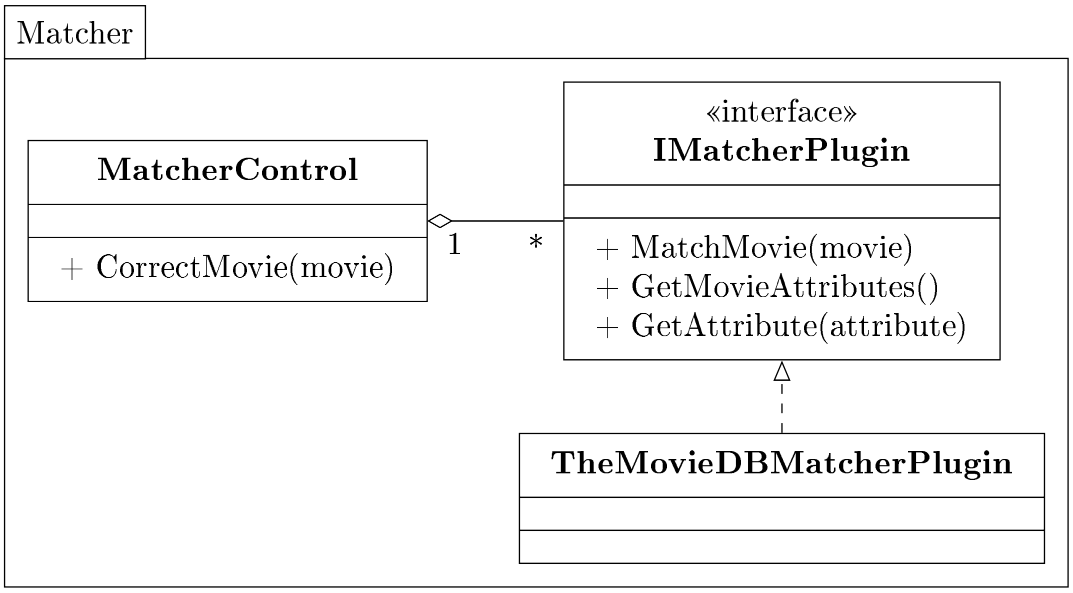


Figure 16: Class diagram for the TUPT Matcher package.

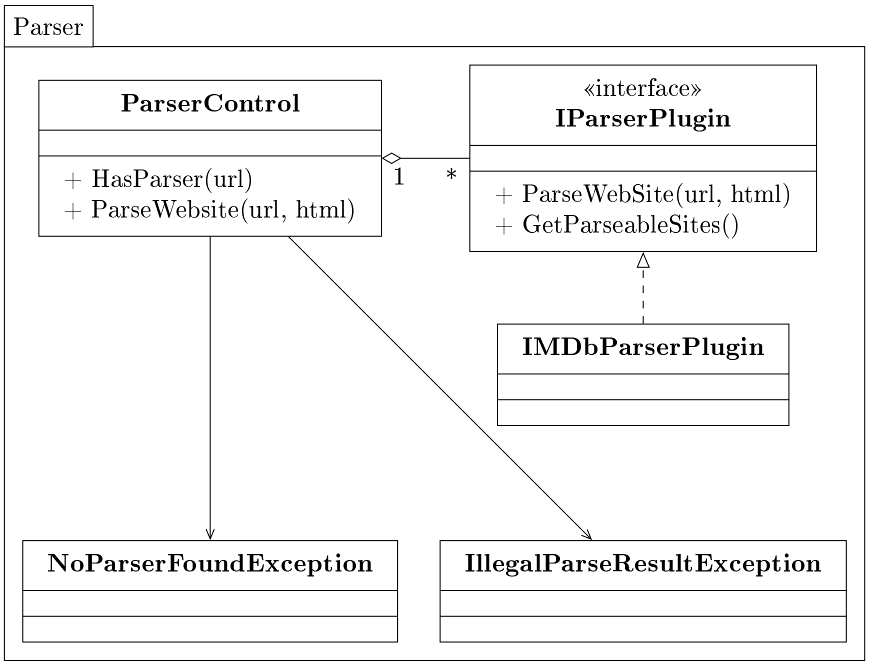


Figure 17: Class diagram for the TUPT Parser package.

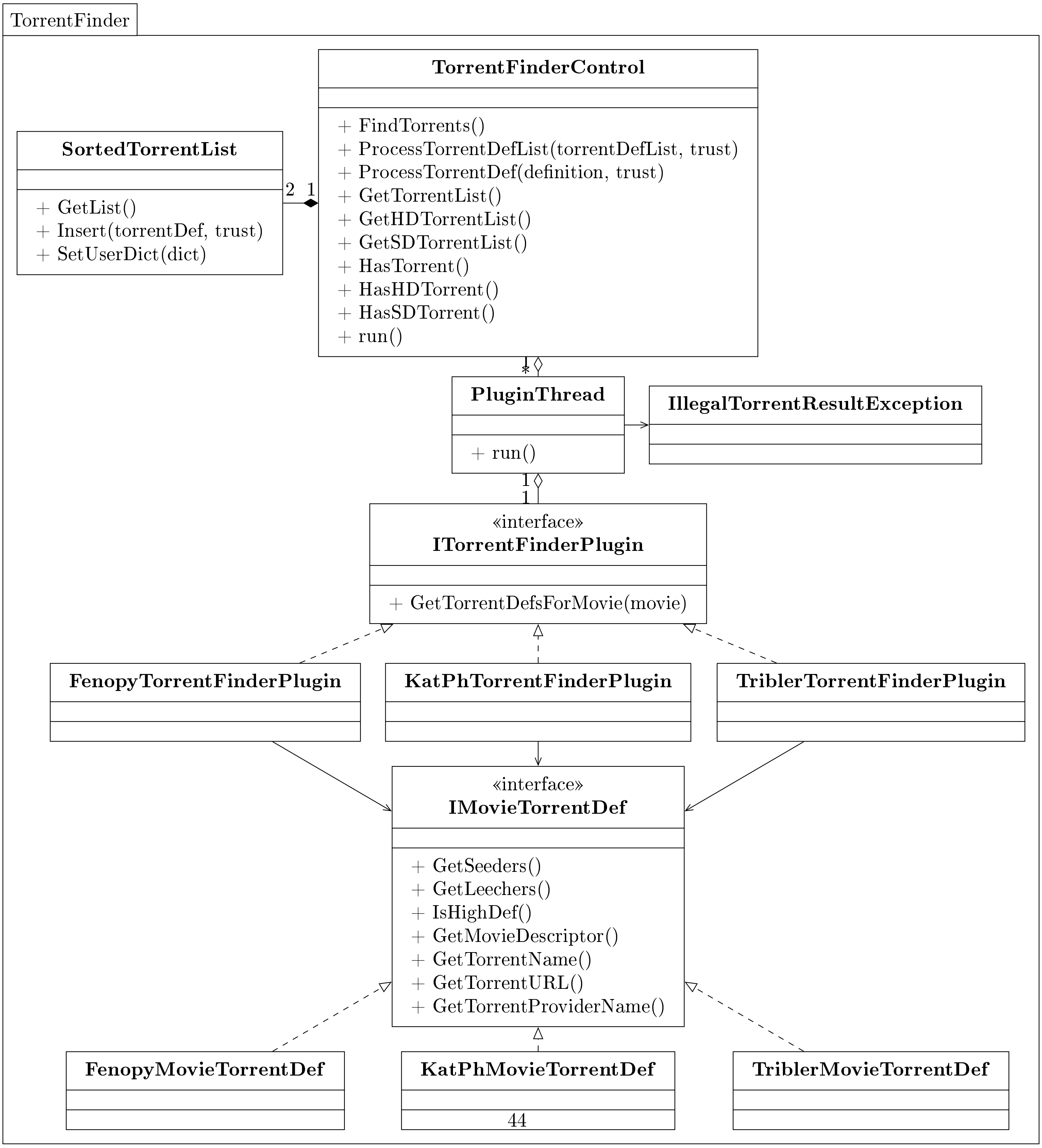


Figure 18: Class diagram for the TUPT TorrentFinder package.

References

[1] Tribler. Available at: http://www.tribler.org/, [Accessed 21 June 2013].

[2] European union cordis fp7 research program. Available at: http://cordis.europa.eu/fp7, [Accessed 21 June 2013].

[3] Petamedia. Available at: http://www.petamedia.eu/, [Accessed 21 June 2013].

[4] I-share project. Available at:www.freeband.nl, [Accessed 21 June 2013].

[5] Scrum. Available at: http://www.scrum.org/, [Accessed 21 June 2013].

[6] Tudelft department of parallel and distributed systems. Available at: http://www.pds.ewi.tudelft.nl/, [Accessed 21 June 2013].

[7] Tribler/tribler - github. Available at: https://github.com/Tribler/tribler, [Accessed 21 June 2013].

[8] Eclipse ide. Available at: http://www.eclipse.org/, [Accessed 21 June 2013].

[9] Pydev. Available at: http://www.pydev.org/, [Accessed 21 June 2013].

[10] norberhuis/tribler - github. Available at: https://github.com/norberhuis/tribler, [Accessed 21 June 2013].

[11] Python. Available at: http://www.python.org/, [Accessed 21 June 2013].

[12] G. van Rossum. Style guide for python code. Available at: http://www.python.org/dev/peps/pep-0008/, [Accessed 21 June 2013].

[13] libswift. Available at: http://libswift.org/, [Accessed 21 June 2013].

[14] N. Zeilemaker B. Schoon J. Pouwelse. Dispersy bundle synchronization. Technical report, Delft University of Technology, January 2013. ISSN 1387-2109.

[15] wxpython. Available at: http://www.wxpython.org/, [Accessed 21 June 2013].

[16] Beautiful soup. Available at: http://www.crummy.com/software/BeautifulSoup/, [Accessed 25 June 2013].

[17] wxphoenix. Available at: http://wiki.wxpython.org/ProjectPhoenix, [Accessed 21 June 2013].

[18] Imdbpy. Available at: http://imdbpy.sourceforge.net/, [Accessed 21 June 2013].

[19] Yapsy. Available at: http://yapsy.sourceforge.net/, [Accessed 21 June 2013].