Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

DUMITRU COTET

MASTER THESIS

CRAWLING CODE REVIEW DATA FROM PHABRICATOR

Submitted on 4 June 2019

Supervisors: Michael Dorner, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nuremberg, 4 June 2019

License

Nuremberg, 4 June 2019

Abstract

Modern code review is typically supported by software tools. Researchers use data tracked by these tools to study code review practices. A popular tool in open-source and closed-source projects is Phabricator. However, there is no tool to crawl all the available code review data from Phabricator hosts. In this thesis, we develop a Python crawler named Phabry, for crawling code review data from Phabricator instances using its REST API. The tool produces minimal server and client load, reproducible crawling runs, and stores complete and genuine review data. The new tool is used to crawl the Phabricator instances of the open source projects FreeBSD, KDE and LLVM. The resulting data sets can be used by researchers.

Contents

List of Abbreviations

API	tab application programming interface
CLI	command-line interface
FreeBSD	free and open-source Unix-like operating system
HTTP	hypertext transfer protocol
JSON	JavaScript object notation
JSON-RPC	Remote procedure call protocol encoded in JSON
KDE	free and open source software community
LLVM	low Level Virtual Machine, now a compiler infrastructure project
OSS	open source software
REST	representational state transfer
SCM	source code management

1 Introduction

Code review is an important practice in software quality assurance. The process is that one or several people check a program mainly by reading parts of the source code. Code reviews should be scheduled for new modules and major modifications of existing code. and minor revisions should be reviewed solely by the change reviewer, rather than multiple engineers (Kolawa & Huizinga, 2007). Kolawa and Huizinga (2007) also state that the main goals of code reviews are to find defects, to find better solutions, to help in knowledge transfer, and to generally improve internal code quality and maintainability.

In order to facilitate the code review process, various code review tools are used. These tools allow people to collaboratively inspect and discuss changes, storing the history of the process for future reference. Phabricator is a suite of web-based software development collaboration tools, including a code review tool. It is available as free software under the Apache License 2.0. Phabricator is used in the development process of a few big open source projects (e.g. LLVM, Freebsd, KDE), but also in some companies (e.g. Asana, Quora, MemSQL) (Tsotsis, 2011).

Because of the significance and accessibility of code review data from Phabricator, extensive research can be performed on that data. Phabricator provides a REST API with JSON responses for straight forward access to the code reviews stored by Phabricator. However, there are huge amounts of data to be downloaded and this operation cannot be performed over one single API call.

At the moment, there are no publicly available code review datasets from any of the projects using Phabricator. There are also no crawling tools for easy downloading of this data. In order to easily access and download all the available code review data from Phabricator, a small program was written that takes into consideration all the unusual aspects of the Phabricator API. This Phabricator crawler was named Phabry.

The requirements Phabry are presented in the chapter 2 of this thesis. The code review tool Phabricator and its API are described in chapter 3. Chapter 4 discusses the design and implementation of the program. Chapter 5 focuses on the evaluation of the crawler. Also, to determine the potential of the downloaded datasets, some data analysis was performed in this chapter. Finally, chapter 6 summarises the thesis with a conclusion.

2 Requirements

Writing effective product requirements is the first step towards a successful release. The requirements act as a starting point for the product and outline its purpose and how to use it. One of the most important aspects of the requirements is to identify the relevant stakeholders. This thesis is a part of ongoing research that is still in its early stages. Thus, the stakeholders are limited to the developers and researchers of the project for the time being.

2.1 Functional requirements

The functional requirements define what the program must do. They describe the behaviour of the program as it relates to its functionality.

F1: Crawler based on Phabricator

- The software downloads the code review data from a specified web resource using Phabricator's REST API

The crawler must also be flexible and require minimum adaptation to easily support future modifications of the API.

F2: All available data is downloaded

- Download all available data and save it in text files

All code review data that is available via the provided API must be downloaded and saved as is in text files, the only change accepted is formatting and indentation according to JSON syntax.

F3: Complete data

- Store complete and unaltered data

The crawler must download and store the data in exactly the same condition as it comes from the Phabricator server. This would ensure that the user receives the very same data that is stored by Phabricator.

F4: Single-threaded and minimal server load

- The crawler must be single-threaded for minimal server load and to not hinder other users of the system.

F5: Reproducible behaviour

- Every run of the crawler must be reproducible

In order to make the behaviour reproducible, every crawling execution must be accompanied by a configuration file and a log file. This way, both files can be attached to the resulting data of the crawling and allow anybody to replicate the run.

F6: Incremental crawling

- The software must support incremental crawling, i.e. to crawl the content which has been added/modified since last successful crawl.

As compared to full crawling, incremental crawling drastically reduces server and network load in cases when it can be used. Also, it is usually a lot faster as well.

F7: Anonymisation of personal data

- Replace the names and emails with hashes in order to protect the identities behind every person involved.

The purpose of the crawler is to download and save the code review data according to some predefined criteria, while keeping the identities in the data anonymous.

2.2 Non-functional requirements

As opposed to the functional requirements, the non-functional ones define *how* the program must perform. The non-functional requirements elaborate a performance characteristic of the system.

N1: Efficiency

- The program must not be memory and CPU intensive.

Since the crawler does not effectively perform any CPU intensive tasks, an acceptable CPU usage would be less than 20 %.

N2: Execution time

- One crawling run must be finished in a reasonable amount of time

In order to keep the program usable, the full execution time, i.e. the complete downloading of all available code review data, should be done in less than 24 hours.

2.3 Evaluation scheme for requirements

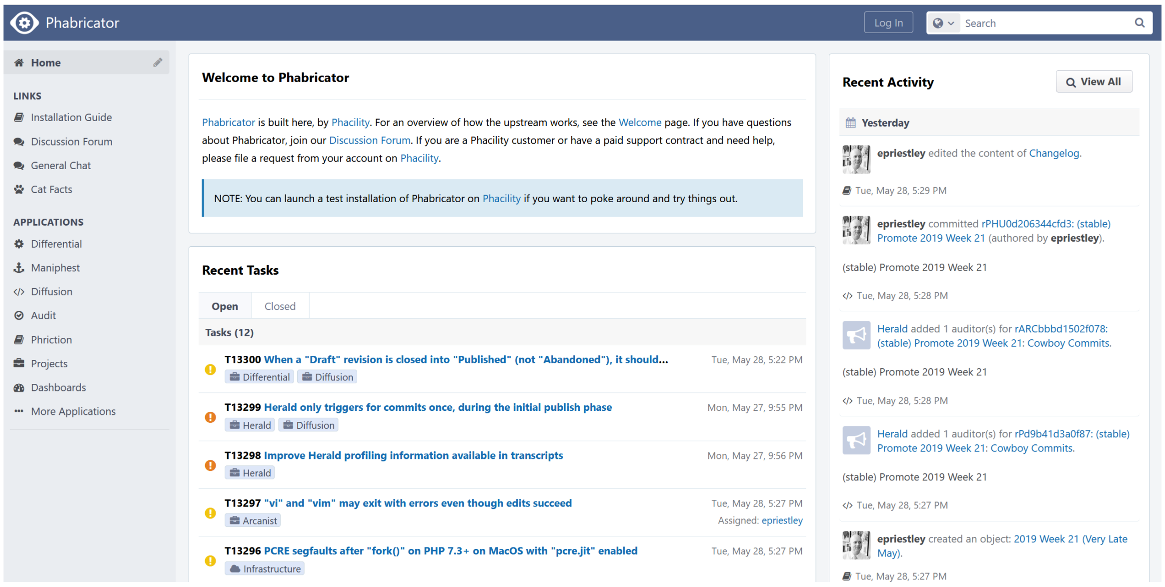
Ensuring that the requirements set in the previous section are met is the primary goal of this project. It is important to verify the functionality of each section of code in a white box style using unit tests, particularly at function level, to guarantee a code coverage of 100% or as high as possible. Additionally, a set of manual tests needs to be performed in order to evaluate the program in a black box style. That is to test that the crawler achieves the results expected by the stakeholders, that it is stable to use and that it meets the requirements. It must be acknowledged, however, that each Phabricator host may contain tens of thousands of code reviews, therefore the scope of the crawler can not guarantee that Phabricator's API will return all its available data, as it depends solely on the implementation of the API.

3 Phabricator

This chapter covers the research of Phabricator and its API in order to identify its possibilities and usage procedures.

Phabricator is a set of tools for developing software and includes applications for code review, repository hosting, bug tracking, project management, and more (“Phacility - Home”, 2019). Phabricator is powerful, fast, scalable, and completely open source. It can be downloaded and installed on any hardware for free, or launch a hosted instance with the company behind the product - Phacility (“Phacility - Home”, 2019).

Figure 3.1 shows the home page of Phabricator’s own instance of Phabricator, where the development of Phabricator is tracked. Phabricator supports two similar but separate code review workflows: ”review” and ”audit” (“User Guide: Review vs Audit”, n.d.). According to the same page, a review occurs in the tool called Differential, before changes are published, whereas an audit occurs in the Diffusion tool, after changes are published. For the purpose of this thesis, only the reviews are considered.



Welcome to Phabricator

Phabricator is built here, by Phacility. For an overview of how the upstream works, see the Welcome page. If you have questions about Phabricator, join our Discussion Forum. If you are a Phacility customer or have a paid support contract and need help, please file a request from your account on Phacility.

NOTE: You can launch a test installation of Phabricator on Phacility if you want to poke around and try things out.

Recent Tasks

Tasks (12)

T13300 When a "Draft" revision is closed into "Published" (not "Abandoned"), it should... — Differential, Diffusion — Tue, May 28, 5:22 PM

T13299 Herald only triggers for commits once, during the initial publish phase — Herald, Diffusion — Mon, May 27, 9:55 PM

T13298 Improve Herald profiling information available in transcripts — Herald — Mon, May 27, 9:56 PM

T13297 "vi" and "vim" may exit with errors even though edits succeed — Arcanist — Tue, May 28, 5:27 PM — Assigned: epriestley

T13296 PCRE segfaults after "fork()" on PHP 7.3+ on MacOS with "pcre.jit" enabled — Infrastructure — Tue, May 28, 5:27 PM

Recent Activity

Yesterday

epriestley edited the content of Changelog.

Tue, May 28, 5:29 PM

epriestley committed rPHU0d206344cfd3: (stable) Promote 2019 Week 21 (authored by **epriestley**).

(stable) Promote 2019 Week 21

Tue, May 28, 5:28 PM

Herald added 1 auditor(s) for rARCbbbd1502f078: (stable) Promote 2019 Week 21: Cowboy Commits.

(stable) Promote 2019 Week 21

Tue, May 28, 5:28 PM

Herald added 1 auditor(s) for rPd9b41d3a0f87: (stable) Promote 2019 Week 21: Cowboy Commits.

(stable) Promote 2019 Week 21

Tue, May 28, 5:27 PM

epriestley created an object: 2019 Week 21 (Very Late May).

Tue, May 28, 5:27 PM

Figure 3.1: Phabricator home page. Retrieved May 29, 2019, from https://secure.phabricator.com/

3.1 Differential – review workflow

The Phabricator “Differential User Guide” (n.d.) describes how reviews work:

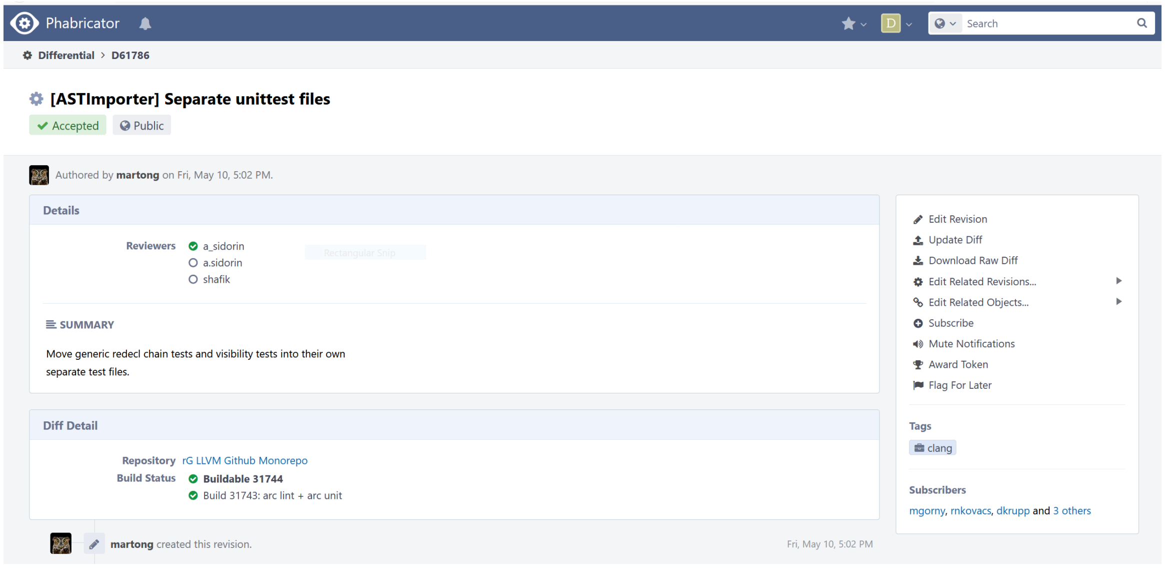
– An author prepares a change to a codebase, then sends it for review. They specify who they want to review it (additional users may be notified as well). The change itself is called a ”Differential Revision”.

– The reviewers receive an email asking them to review the change.

– The reviewers inspect the change and either discuss it, approve it, or request changes (e.g., if they identify problems or bugs).

– In response to feedback, the author may update the change (e.g., fixing the bugs or addressing the problems).

– Once everything is satisfied, some reviewer accepts the change and the author pushes it to the upstream.



[ASTImporter] Separate unittest files

✓ Accepted | Public

Authored by **martong** on Fri, May 10, 5:02 PM.

Details

Reviewers: ✓ a_sidorin, ○ a.sidorin, ○ shafik

SUMMARY

Move generic redecl chain tests and visibility tests into their own separate test files.

Diff Detail

Repository: rG LLVM Github Monorepo

Build Status: ✓ **Buildable 31744**, ✓ Build 31743: arc lint + arc unit

Tags: clang

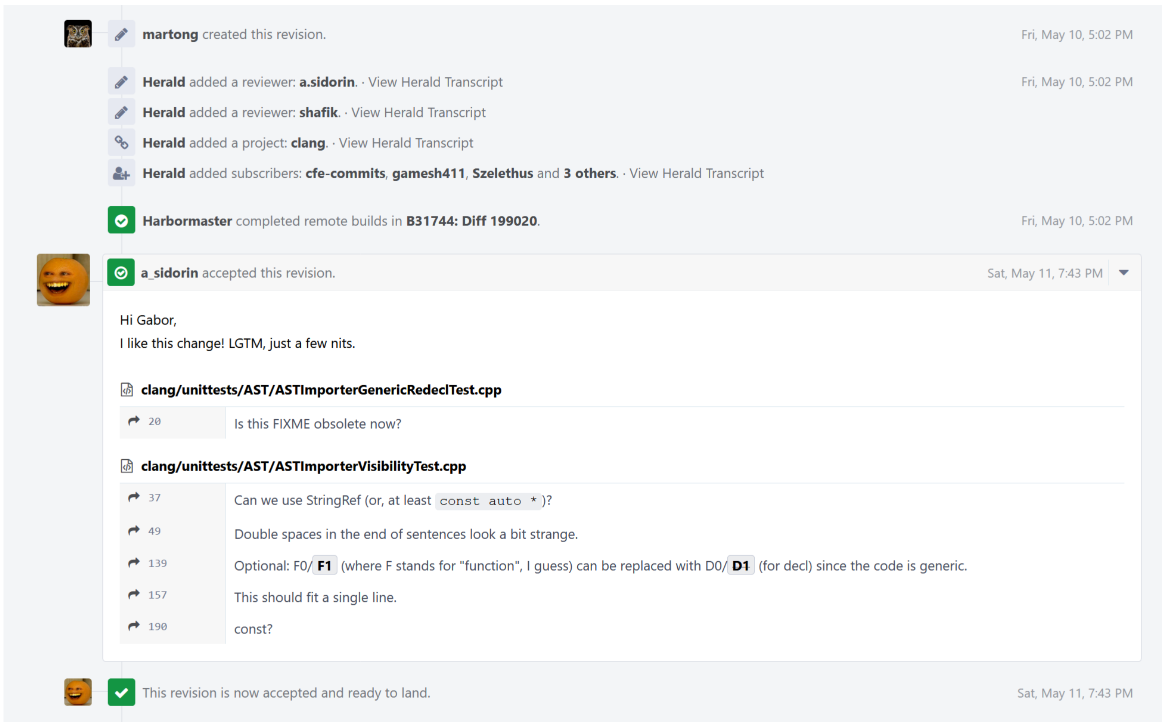
Subscribers: mgorny, rnkovacs, dkrupp and 3 others

martong created this revision. Fri, May 10, 5:02 PM

Figure 3.2: The user interface of a differential revision. Retrieved May 12, 2019, from https://reviews.llvm.org/D61786

Figure 3.2 shows the differential user interface with the summary of the differential revision, retrieved from the public Phabricator host of the LLVM project. The top of the page has the ID of the revision (D61786) and then right below it there’s the title of the revision ([ASTImporter] Separate unittest files), the status (Accepted), and the visibility (Public). Next below is the name of the author and the date and time the revision was created. Then follows the ”Details” section with the reviewers and a short summary of the revision. Lastly, on the bottom of the figure, there is the ”Diff Detail” that has the ”Repository” and the ”Build Status” of the code change attached to this revision - called a ”Diff”. This screenshot did not fit the whole page of the differential user interface and it is described further below.

Every Differential revision is an object in Phabricator. In order to preserve every activity occurring with an object, like creation, edit, comment, etc., a ”transaction” for each activity is created by Phabricator (“Conduit - transaction.search”, n.d.). Figure 3.3 shows all the transactions for the differential revision and is a continuation of the same page shown in Figure 3.2.



martong created this revision. Fri, May 10, 5:02 PM

Herald added a reviewer: **a.sidorin**. · View Herald Transcript Fri, May 10, 5:02 PM

Herald added a reviewer: **shafik**. · View Herald Transcript

Herald added a project: **clang**. · View Herald Transcript

Herald added subscribers: **cfe-commits**, **gamesh411**, **Szelethus** and **3 others**. · View Herald Transcript

Harbormaster completed remote builds in **B31744: Diff 199020**. Fri, May 10, 5:02 PM

a_sidorin accepted this revision. Sat, May 11, 7:43 PM

Hi Gabor,
I like this change! LGTM, just a few nits.

clang/unittests/AST/ASTImporterGenericRedeclTest.cpp

20	Is this FIXME obsolete now?

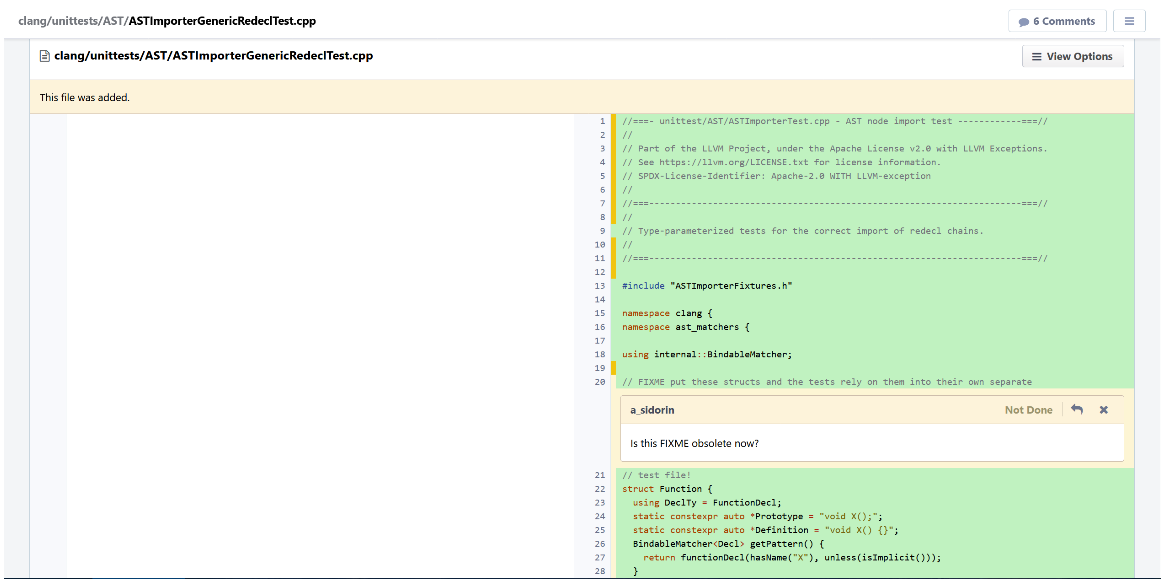
clang/unittests/AST/ASTImporterVisibilityTest.cpp

37	Can we use StringRef (or, at least `const auto *`)?
49	Double spaces in the end of sentences look a bit strange.
139	Optional: F0/**F1** (where F stands for "function", I guess) can be replaced with D0/**D1** (for decl) since the code is generic.
157	This should fit a single line.
190	const?

This revision is now accepted and ready to land. Sat, May 11, 7:43 PM

Figure 3.3: The transactions of a differential revision. Retrieved May 12, 2019, from https://reviews.llvm.org/D61786

One of the most useful features of Differential is the ”Inline Comments”. “Differential User Guide: Inline Comments” (n.d.) explains how Differential allows reviewers to leave feedback about changes to code inline within the body of the diff itself and can be used to discuss specific parts of a change. An example of an inline comment is shown in Figure 3.4. As it can be seen, the status of the comment is ”Not Done”, which means that the author hasn't updated the change according to the feedback in the comment. After the author updates the change, the reviewer that made the comment has to review the change again and comment accordingly. If the reviewer accepts the change, they can mark the comment as ”Done”, see Figure 3.5. After the changed is reviewed and accepted, the inline comments are preserved as part of the activity in the transactions. Finally, the status of the revision changes to ”Published” when the accepted changes are published in the main code.


clang/unittests/AST/ASTImporterGenericRedeclTest.cpp

6 Comments

clang/unittests/AST/ASTImporterGenericRedeclTest.cpp

View Options

This file was added.

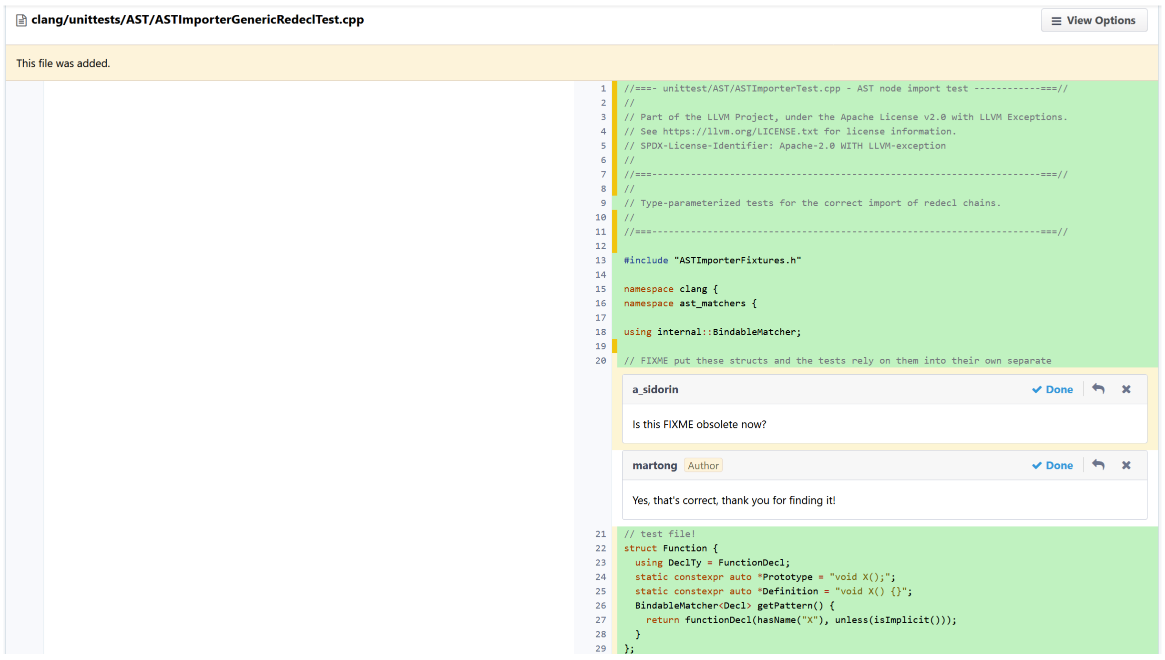
```
//===- unittest/AST/ASTImporterTest.cpp - AST node import test ------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
//
// Type-parameterized tests for the correct import of redecl chains.
//
//===----------------------------------------------------------------------===//

#include "ASTImporterFixtures.h"

namespace clang {
namespace ast_matchers {

using internal::BindableMatcher;

// FIXME put these structs and the tests rely on them into their own separate
```

a_sidorin — Not Done

Is this FIXME obsolete now?

```
// test file!
struct Function {
  using DeclTy = FunctionDecl;
  static constexpr auto *Prototype = "void X();";
  static constexpr auto *Definition = "void X() {}";
  BindableMatcher<Decl> getPattern() {
    return functionDecl(hasName("X"), unless(isImplicit()));
  }
```

Figure 3.4: An inline comment inside a ”diff”. Retrieved May 12, 2019, from https://reviews.llvm.org/D61786


clang/unittests/AST/ASTImporterGenericRedeclTest.cpp

View Options

This file was added.

```
//===- unittest/AST/ASTImporterTest.cpp - AST node import test ------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
//
// Type-parameterized tests for the correct import of redecl chains.
//
//===----------------------------------------------------------------------===//

#include "ASTImporterFixtures.h"

namespace clang {
namespace ast_matchers {

using internal::BindableMatcher;

// FIXME put these structs and the tests rely on them into their own separate
```

a_sidorin — ✓ Done

Is this FIXME obsolete now?

martong Author — ✓ Done

Yes, that's correct, thank you for finding it!

```
// test file!
struct Function {
  using DeclTy = FunctionDecl;
  static constexpr auto *Prototype = "void X();";
  static constexpr auto *Definition = "void X() {}";
  BindableMatcher<Decl> getPattern() {
    return functionDecl(hasName("X"), unless(isImplicit()));
  }
};
```

Figure 3.5: An inline comment after it was accepted. Retrieved May 20, 2019, from https://reviews.llvm.org/D61786?id=199020#inline-548812

3.2 Conduit API

According to “Conduit API Overview” (n.d.), Conduit is the HTTP API for Phabricator and is roughly JSON-RPC: a JSON blob is usually passed, and a JSON blob is usually sent back, although both call and result formats are flexible in some cases. The “Conduit API Overview” (n.d.) also describes the primary ways to make Conduit calls:

– Web Console: The Conduit application provides a web UI for exploring the API and making calls;

– `ConduitClient`: This is the official client available in `libphutil`, and the one used by `arc`;

– `arc call-conduit`: this `arc` command can be used to execute low-level Conduit calls by piping JSON in to stdin;

– `curl`: a call can be formatted with basic HTTP parameters and cURL and the console includes examples which show how to format calls;

– Other Clients: There are also clients available in other languages that can be checked in the Community Resources[1] page.

The Community Resources refers to a Python library that can work with Phabricator Conduit API. It will be analysed for compatibility in the next chapter.

After consulting the list of available Conduit API methods[2], two methods were selected that provide the required functionality:

– `differential.revision.search` – read information about revisions;

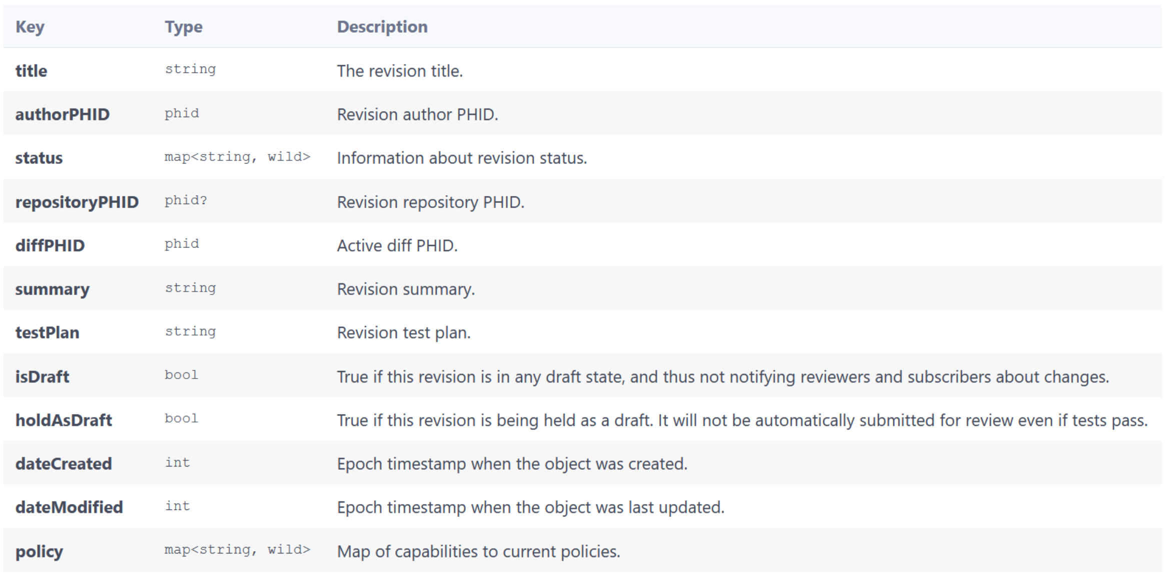
– `transaction.search` – read transactions and comments for an object.

The first method provides a way to download the revision data with essential information, but without the activity. In order to get all the available transactions for each revision, the second provided method must be used.

3.2.1 The Conduit method – `differential.revision.search`

The page “Conduit - differential.revision.search” (n.d.) describes the usage of this API method, it being the primary method to list, query or search for Difrential revisions. It can be used, therefore, to get all the available revisions from the Phabricator database. “Conduit - differential.revision.search” (n.d.) indicates that the revision objects are returned as a list of dictionaries in the `data` property of the results and each dictionary has some metadata and a fields key, which contains the information about the object, see Figure 3.6, while the result may look like in Listing 3.1.

[1]https://secure.phabricator.com/w/community_resources/
[2]https://secure.phabricator.com/conduit/



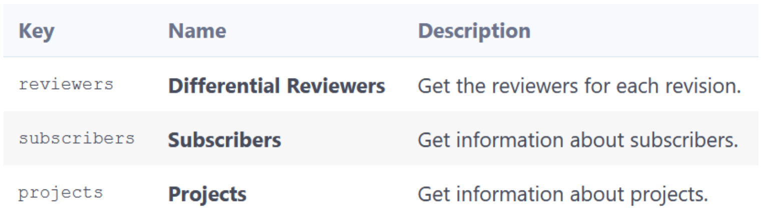
Key	Type	Description
title	string	The revision title.
authorPHID	phid	Revision author PHID.
status	map<string, wild>	Information about revision status.
repositoryPHID	phid?	Revision repository PHID.
diffPHID	phid	Active diff PHID.
summary	string	Revision summary.
testPlan	string	Revision test plan.
isDraft	bool	True if this revision is in any draft state, and thus not notifying reviewers and subscribers about changes.
holdAsDraft	bool	True if this revision is being held as a draft. It will not be automatically submitted for review even if tests pass.
dateCreated	int	Epoch timestamp when the object was created.
dateModified	int	Epoch timestamp when the object was last updated.
policy	map<string, wild>	Map of capabilities to current policies.

Figure 3.6: The result fields of a review from a call to `differential.revision.search`. Retrieved May 20, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/

```
{
  ...
  "data": [
    {
      "id": 123,
      "phid": "PHID-WXYZ-1111",
      "fields": {
        "name": "First Example Object",
        "authorPHID": "PHID-USER-2222"
      }
    },
    {
      "id": 124,
      "phid": "PHID-WXYZ-3333",
      "fields": {
        "name": "Second Example Object",
        "authorPHID": "PHID-USER-4444"
      }
    },
    ...
  ]
  ...
}
```

Listing 3.1: The result of a review from a call to `differential.revision.search`. Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/

By default, only basic information about objects is returned, but more extensive information can be accessed by using available attachments to get more information in the results, as shown in Figure 3.7 (“Conduit - differential.revision.search”, n.d.).



Key	Name	Description
reviewers	**Differential Reviewers**	Get the reviewers for each revision.
subscribers	**Subscribers**	Get information about subscribers.
projects	**Projects**	Get information about projects.

Figure 3.7: The attachments field of a review from a call to `differential.revision.search`. Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/

Another important aspect to consider is that queries are limited to returning 100 results at a time and if more results are required, additional queries need to be made (“Conduit - differential.revision.search”, n.d.). The result structure contains a cursor key with the information needed in order to fetch the next batch of results, according to “Conduit - differential.revision.search” (n.d.), and after an initial query, it will usually look like in Listing 3.2.

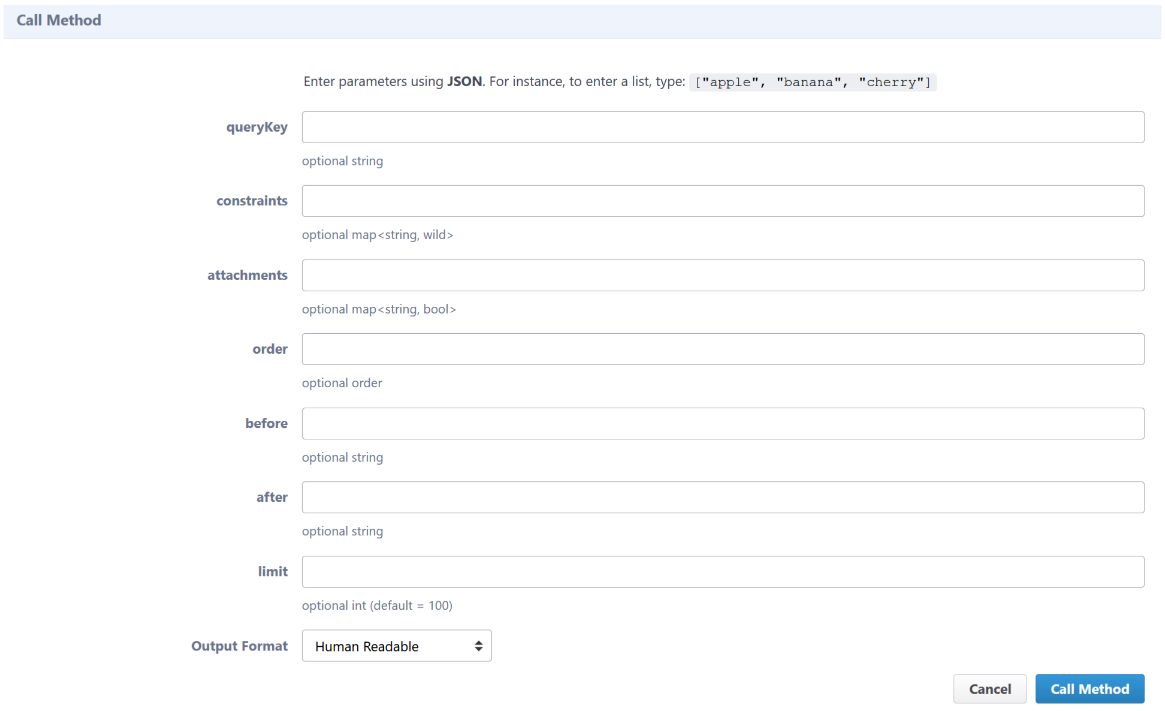
```
{
  ...
  "cursor": {
    "limit": 100,
    "after": "100",
    "before": null,
    "order": null
  }
  ...
}
```

Listing 3.2: The cursor key structure in a result from the first call to `differential.revision.search`. Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/

The fields `limit` and `order` describe the effective limit and order the query was executed with, and are usually not of much interest, but the `after` and `before` fields give cursors which can be passed when making another API call in order to get the next (or previous) page of results (“Conduit - differential.revision.search”, n.d.). To get the next batch of results, the API call has to be repeated with all the same parameters as the original call, but the `after` cursor received from the first call needs to be passed in the `after` parameter when making the second call, as explained by “Conduit - differential.revision.search” (n.d.), and the second call result would have a cursor structure like in Listing 3.3.

```
{
  ...
  "cursor": {
    "limit": 100,
    "after": "200",
    "before": "101",
    "order": null
  }
  ...
}
```

Listing 3.3: The cursor key structure after a second call to `differential.revision.search`. Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/



Call Method

Enter parameters using **JSON**. For instance, to enter a list, type: `["apple", "banana", "cherry"]`

queryKey	optional string
constraints	optional map<string, wild>
attachments	optional map<string, bool>
order	optional order
before	optional string
after	optional string
limit	optional int (default = 100)
Output Format	Human Readable

Figure 3.8: The Web UI for calling the method differential.revision.search. Retrieved May 21, 2019, from https://reviews.llvm.org/conduit/method/differential.revision.search/

Finally, "Conduit - differential.revision.search" (n.d.) provides a Web UI for making calls using this API method, but in order to use it, one has to be authenticated. Unfortunately, new accounts cannot be registered ("Phabricator Login", n.d.). However, this method can be also tested with another Phabricator source, like LLVM, which provides a way to register as a new user. The functionality of the Conduit API is the same and the the Web UI for the method `differential.revision.search` can be found at https://reviews.llvm.org/conduit/method/differential.revision.search/.

Figure 3.8 shows the UI of calling the method, along with the type of each parameter. Besides providing an easy way to make an API call, this UI also gives usage examples with different clients, including encoding the parameters. This feature will be useful later when designing the Phabricator crawler. For example, Listing 3.4 and Listing 3.5 show the provided examples after introducing the start and end dates of creation as constraints, including all attachments.

```
$ echo '{
  "constraints": {
    "createdStart": 1552176000,
    "createdEnd": 1552262399
  },
  "attachments": {
    "subscribers": true,
    "reviewers": true,
    "projects": true
  }
}' | arc call-conduit --conduit-uri https://reviews.llvm.org/ \
 --conduit-token <conduit-token> differential.revision.search
```

Listing 3.4: Usage example for `arc call-conduit`. Retrieved May 21, 2019, from https://reviews.llvm.org/api/differential.revision.search

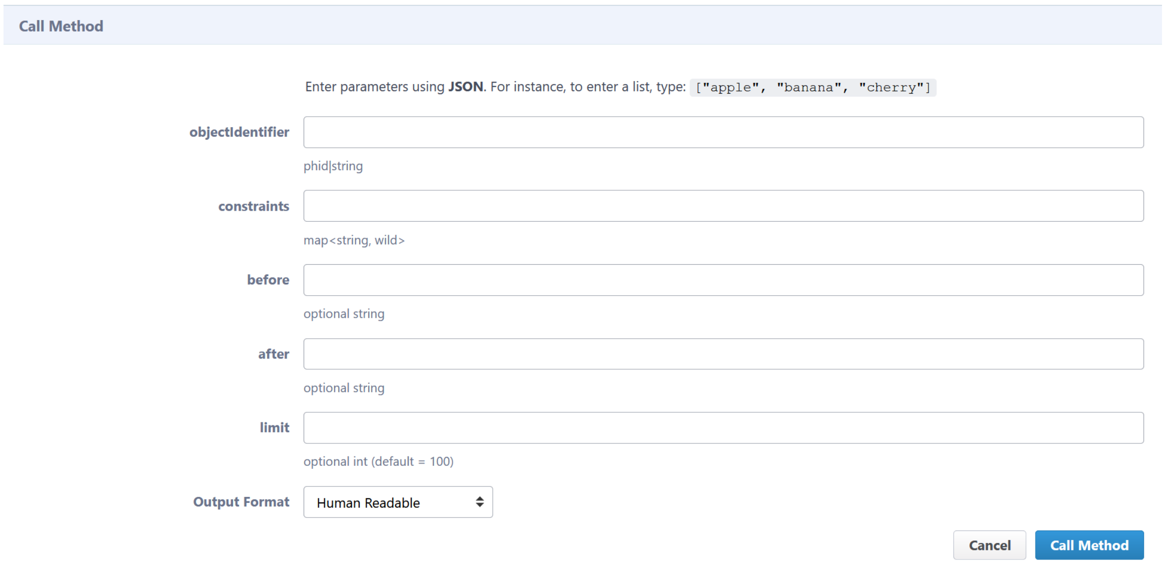
```
$ curl https://reviews.llvm.org/api/differential.revision.search \
    -d api.token=api-token \
    -d constraints[createdStart]=1552176000 \
    -d constraints[createdEnd]=1552262399 \
    -d attachments[subscribers]=1 \
    -d attachments[reviewers]=1 \
    -d attachments[projects]=1
```

Listing 3.5: Usage example for `cURL`. Retrieved May 21, 2019, from https://reviews.llvm.org/api/differential.revision.search

As seen in the previous examples, one would need an `api-token` in order to have access to this method via other clients. This token can be generated in the `Conduit API Tokens` panel in `Settings` (“Conduit - differential.revision.search”, n.d.).

3.2.2 The Conduit method – `transaction.search`

When an object (like a revision) is edited, Phabricator creates a ”transaction” and applies it, then the list of transactions on each object is the basis for essentially all edits and comments in Phabricator “Conduit - transaction.search” (n.d.). Since the method `differential.revision.search` does not return any data on edits, comments, etc., `transaction.search` is a method that provides valuable for research information. This method returns the transactions for one revision at a time, therefore since the method `differential.revision.search` returns 100 revisions in one call, there are needed to be made another 100 calls for each revision in order to get all the transactions for this batch. Figure 3.9 shows the Web UI for calling the method.


Call Method

Enter parameters using **JSON**. For instance, to enter a list, type: `["apple", "banana", "cherry"]`

objectIdentifier	phid|string
constraints	map<string, wild>
before	optional string
after	optional string
limit	optional int (default = 100)
Output Format	Human Readable

Figure 3.9: The Web UI for calling the method `transaction.search`. Retrieved May 21, 2019, from https://reviews.llvm.org/conduit/method/differential.revision.search/

The description of the method "Conduit - transaction.search" (n.d.) does not provide the object fields from the result like for `differential.revision.search`, but the fields can be interpreted relatively easy from the result itself. Also, unlike the previously mentioned method, `transaction.search` has one mandatory parameter -- `objectIdentifier`, which can be found in the result of the previous call under the name "phid". After calling the method, a result with a similar structure -- a list of dictionaries in the `data` property and each dictionary has some metadata and a fields key, which contains the information about the object, see Listing 3.6.

```
{
  "data": [
    {
      "id": 1517222,
      "phid": "PHID-XACT-DREV-boih33ogvg63pcs",
      "type": "status",
      "authorPHID": "PHID-USER-6ilmjx4ipykswr2bg4ro",
      "objectPHID": "PHID-DREV-6qktcmyat6okdtfht5k4",
      "dateCreated": 1558769696,
      "dateModified": 1558769696,
      "comments": [],
      "fields": {
        "old": "needs-review",
        "new": "accepted"
      }
    },
    ...
  ]
}
```

Listing 3.6: Example result from calling `transaction.search` in LLVM

As with the `differential.revision.search` method, the UI gives usage examples of `transaction.search` with different clients, see Listing 3.7 and Listing 3.8.

```
$ echo '{
  "objectIdentifier": "PHID-DREV-6qktcmyat6okdtfht5k4"
}' | arc call-conduit --conduit-uri https://reviews.llvm.org/ \
--conduit-token <conduit-token> transaction.search
```

Listing 3.7: Usage example for `arc call-conduit` in LLVM. Retrieved May 21, 2019, from https://reviews.llvm.org/api/transaction.search

```
$ curl https://reviews.llvm.org/api/transaction.search \
    -d api.token=api-token \
    -d objectIdentifier=PHID-DREV-6qktcmyat6okdtfht5k4
```

Listing 3.8: Usage example for `cURL` in LLVM. Retrieved May 21, 2019, from https://reviews.llvm.org/api/transaction.search

4 Design and Implementation

This chapter covers the design aspects and the implementation of the crawler and also gives an overview of the workflow.

4.1 Design

The first important consideration regarding the design of the crawler is to choose the programming language in which to develop it. A few aspects to consider are:

- needs to have a Conduit API implementation or, alternatively, HTTP functionality for API calls;
- cross-platform;
- ability to read and save files;
- easy to read and use.

Taking into consideration these aspects, Python was chosen based on its popularity as a scripting language and that it's also quite suitable for data mining. The last aspect is important because the last chapter of this thesis will provide a demo analysis of the downloaded datasets. "Phabricator - Community Resources" (n.d.) provides a list of community-maintained clients for the Conduit API, one of which is for Python. However, during the initial research, it became clear that this library for Conduit API was not entirely suitable for all the use cases required by the crawler. Therefore, it was decided to go with the direct requests using HTTP. The recommended Python library for handling HTTP requests is `requests`.

Next step is to comply with the requirement **F5**. In order to do this, the functionality to read the input parameters from configuration files has to be implemented first. This can be achieved using a combination of reading `json` formatted files and the library `argparse`. Then, the logging of errors can be implemented using `logging` library.

The functionality of the system does not involve extremely complex classes and interdependencies. The intention is to have one class that covers all the functionality for downloading and saving the code review datasets according to some input parameters, while the logging and parsing of the configuration file and command line arguments would be performed outside of the said class. The class was named also Phabry and the UML class diagram is shown in Figure 4.1. This makes the code easier to read and understand, but also makes it reusable so that anyone willing to use a specific part of the class can import it in their code.

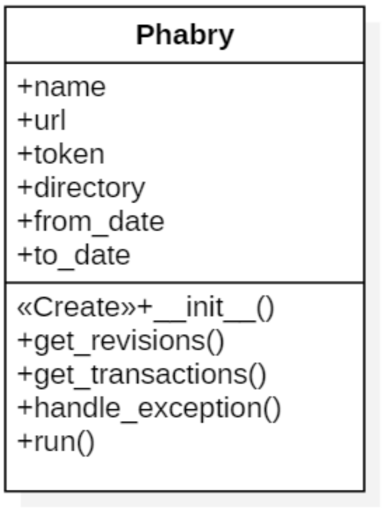


Figure 4.1: UML Class diagram

Figure 4.2 shows the UML activity diagram of the system with the overall work flow. The general idea is that the script loads the configuration parameters, requests the first batch of 100 revisions and saves them into a file. Then for every revision ID, it requests the transactions, saves them into a file, checks if there are any transactions left and requests and saves them again. Then if there are any revisions left to download it goes through the same process until done. Also, if at any step there happens to be an error, it is logged into the log file saved in the same base directory.

4.2 Implementation

This section will describe the implementation details of the source code, along with all the complications that arose during development and the ways they were overcome.

The first action in the script is loading the input parameters from configuration file or from command line arguments. An important aspect to consider in this situation is the approach to loading the parameters. For example, they could be loaded exclusively from either a configuration file or from command line arguments, or they could be loaded from both. In the latter case, it would be also necessary to decide the order of choosing the loading methods. The chosen approach for Phabry is to load the input parameters from both a configuration file and command line arguments. The next few paragraphs describe this procedure more thoroughly.

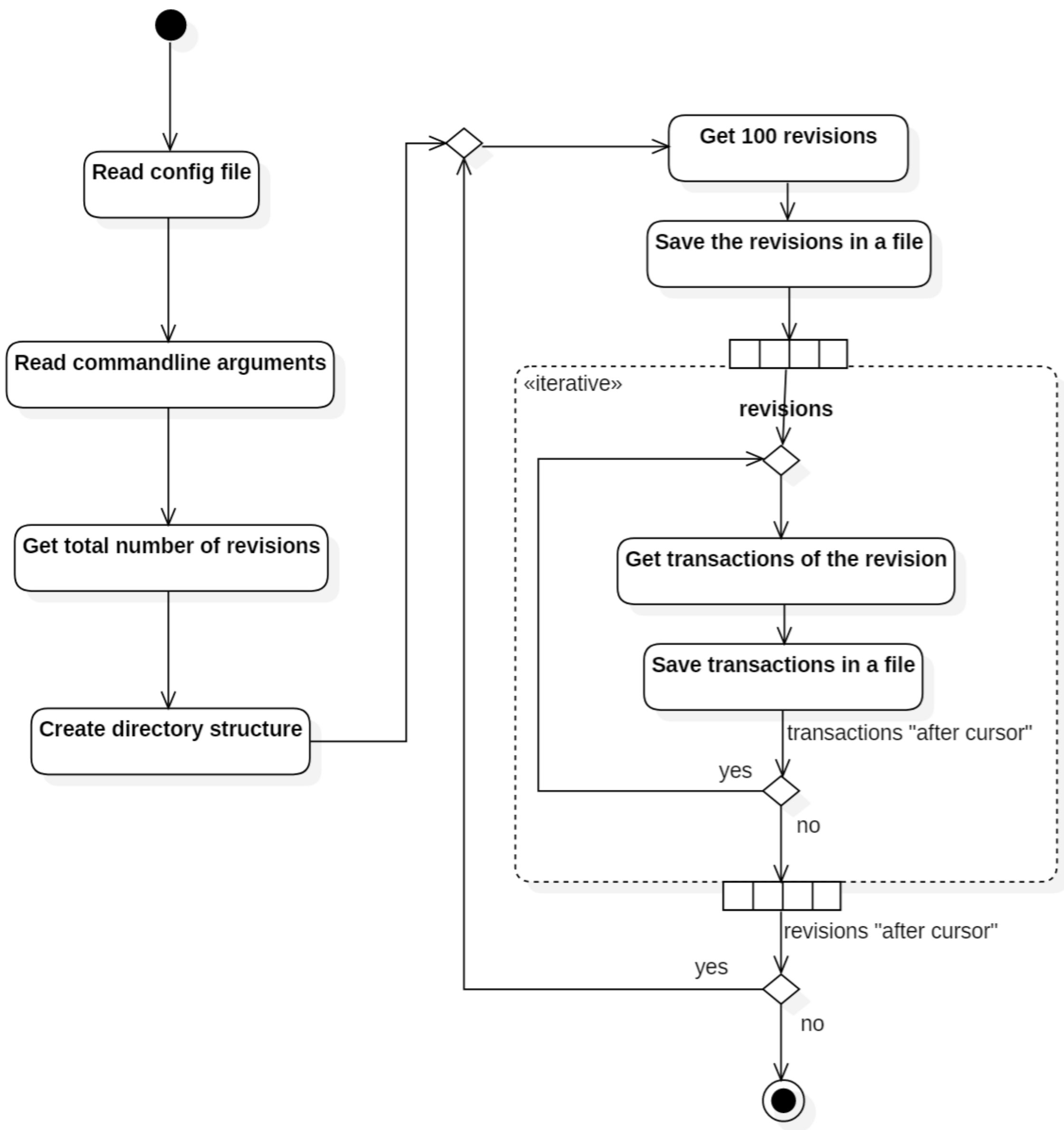

Read config file
Read commandline arguments
Get total number of revisions
Create directory structure
Get 100 revisions
Save the revisions in a file
«iterative»
revisions
Get transactions of the revision
Save transactions in a file
transactions "after cursor"
yes
no
revisions "after cursor"
yes
no

Figure 4.2: UML Activity diagram

At the beginning of the script, a constant is defined with the name of the configuration file: `CONFIG_FILE = 'config.json'` When running the script, the user can also provide a different config file by using the `-c` command. Therefore, if both files exist, then the one provided by the user is the one used, see Listing 4.1

```
# Parse any conf_file specification
# add_help=False so it doesn't parse -h and print help.
conf_parser = argparse.ArgumentParser(
    description=__doc__, # printed with -h/--help
    formatter_class=argparse.RawDescriptionHelpFormatter,
    # Turn off help, to print all options in response to -h
    add_help=False)
conf_parser.add_argument("-c",
    dest='conf_file',
    default=CONFIG_FILE,
    help="Specify config file",
    metavar="FILE")
args, remaining_argv = conf_parser.parse_known_args()
```

Listing 4.1: Setting the `config` file as a command line argument

It was written this way, so that the script can show the option to provide a `config` file along the other options, but also be able to load the parameters from the file and then parse any other arguments from the command-line, if they exist.

The next step is reading the parameters from the configuration file, if the file exists. The code snippet in Listing 4.2 shows how the script opens the file and tries to load a `json` object as a python dictionary. The keys are also converted to lower case for convenience. If the file is not properly formatted, then a helpful error message is shown to the user.

```
defaults{}
if args.conf_file:
    try:
        with open(args.conf_file) as f:
            config = json.load(f)
            defaults.update({k.lower(): v for k, v in config.items()}
                )
    except FileNotFoundError:
        print('Config file not found: ' + args.conf_file)
    except json.decoder.JSONDecodeError as e:
        print('Config file parsing failed. Please format it as json
            object')
```

Listing 4.2: Loading of the `config` file

It is then time to create the parser for the command line arguments. The dictionary `defaults` with any values from the previous step are passed as defaults to this new parser. This means that if a specific parameter is provided as a command line argument, it overwrites the value found in the config file. There are also provided helpful information for each argument to be shown in the help message. Because it was not possible to specify the required arguments in the function `add_argument()` so that it would work for both command line and config arguments, additional checks for the required arguments `name`, `url` and `token` were written. If any of the required arguments is missing, an error is thrown and a help message is displayed, see the implementation in Listing 4.3.

```
# Parse the rest of arguments
parser = argparse.ArgumentParser(parents=[conf_parser])
parser.set_defaults(**defaults)
parser.add_argument('--name', help='REQUIRED Directory name for the
                                   Phabricator source')
parser.add_argument('--url', help='REQUIRED Phabricator api URL')
parser.add_argument('--token', help='REQUIRED Phabricator api token')
parser.add_argument('--basedir', default='./phabry_data/',
                    help='Base directory name')
parser.add_argument('--start', default=None, help='Start date dd-mm-
                                   yyyy')
parser.add_argument('--end', default=None, help='End date dd-mm-yyyy'
                                   )
args = parser.parse_args(remaining_argv)
if None in (args.name, args.url, args.token):
    parser.print_help()
    print("phabry.py: error: the following arguments are required:",
                                       end='')
    if args.name is None:
        print(" --name", end='')
    if args.url is None:
        print(" --url", end='')
    if args.token is None:
        print(" --token", end='')
    exit()
```

Listing 4.3: Setting the input parameters from command line arguments

After the input parameters are collected and normalized, they are passed further to an object to be used as constraints for the API calls. In order to make the code less convoluted and more importantly reusable a class named `Phabry` was written to encompass all the functionality. The next paragraphs will describe the attributes and methods of the class.

The class diagram was shown in Figure 4.1, but the detailed code of the constructor can be seen in Listing 4.4. The constructor method is used to initialise an instance of the class Phabry with the given values. It is also used here to create the directory structure using the `basedir` value and the name of the Phabricator data source. If the arguments for start and end date are provided, it is needed to convert them to the UNIX epoch format using the function `timestamp()`, and since the Conduit API accepts only integer but the function returns a `float`, it is also needed to convert them to `integer`. It is also important to mention that the datetime has to be converted to the UTC timezone first, because they are saved in this format on the server. The function for configuring the logging is called, which is further described in Listing 4.5.

```
def __init__(self, name, url, token, from_date=None, to_date=None,
                                   basedir='./phabry_data/'):
    self.name = name
    self.url = url
    self.token = token
    self.directory = os.path.join(basedir, name)
    if from_date:
        from_date = datetime.datetime.strptime(from_date, '%d-%m-%Y')
```

```python
            self.from_date = int(from_date.replace(tzinfo=datetime.timezone.utc).timestamp())
        else:
            self.from_date = None
        if to_date:
            to_date = datetime.datetime.strptime(to_date, '%d-%m-%Y')
            self.to_date = int(to_date.replace(tzinfo=datetime.timezone.utc).timestamp())
        else:
            self.to_date = None
        if self.url == "https://phabricator.kde.org/api/" and (self.from_date or self.to_date):
            print('KDE does not accept a date range. Proceeding to get all revisions.')
            self.from_date = None
            self.to_date = None
        os.makedirs(os.path.join(self.directory, 'revisions'), exist_ok=True)
        os.makedirs(os.path.join(self.directory, 'transactions'), exist_ok=True)
        configure_logging(self.directory)
```

Listing 4.4: The constructor method of the class Phabry

Since the logging functionality is covered by the requirement **F5**, it is considered important for this project. The function `configure_logging()` is defined outside of the class definition and defines the format and logging levels of the messages, creates a log file in the data directory and saves all the messages there.

```python
log = logging.getLogger('phabry')

def configure_logging(data_dir):
    global log
    log.setLevel(logging.DEBUG)
    log_name = os.path.join(data_dir, 'phabry.log')
    formatter = logging.Formatter('%(asctime)s %(levelname)-8s %(message)s')
    file_handler = logging.FileHandler(log_name)
    file_handler.setFormatter(formatter)
    log.addHandler(file_handler)
    return log
```

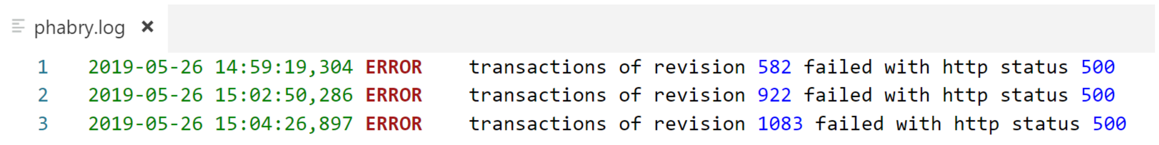
Listing 4.5: The logging configuration

The next function `handle_exception()` is defined within the class `Phabry` and is used to log different messages depending on the type of `Exception`, see Listing 4.6. One notable thing is that the method is defined with the function decorator `@staticmethod`, which transforms the method into a static method to not receive an implicit first argument. In this case, the argument `self`, i.e. the object instance, is not implicitly passed as the first argument because it is not needed here.

```python
@staticmethod
def handle_exception(exception, object_type):
    if isinstance(exception, requests.exceptions.RequestException):
        if exception.response is not None:
            log.error('%s failed with http status %i',
                      object_type, exception.response.status_code)
        elif exception.errno is not None:
            log.error('%s failed with error: %s - %s',
                      object_type, exception.errno, exception.
                                                   strerror)
        else:
            log.error('%s failed with error: %s', object_type,
                                              exception)
    elif isinstance(exception, json.JSONDecodeError):
        log.error('Reading JSON for %s failed', object_type)
    elif isinstance(exception, Exception):
        log.error('Unknown error occurred for %s: %s', object_type,
                                              exception)
```

Listing 4.6: The method `handle_exception()`

Figure 4.3 shows a few lines from the log file after a real run of the script on the FreeBSD Phabricator instance.



```
1   2019-05-26 14:59:19,304 ERROR    transactions of revision 582 failed with http status 500
2   2019-05-26 15:02:50,286 ERROR    transactions of revision 922 failed with http status 500
3   2019-05-26 15:04:26,897 ERROR    transactions of revision 1083 failed with http status 500
```

Figure 4.3: The log file

It was described earlier in subsection 3.2 that calls of the Conduit API methods `differential.revision.search` and `transaction.search` can return maximum 100 objects at a time, which means that each of these API methods will be called hundreds or even thousands of times, because a Phabricator database for the biggest projects contains tens of thousands of revisions. As a result, it was decided to split each API method call into separate class methods.

The first described method is `get_revisions()`, which acts as an abstraction layer for the API method `differential.revision.search`. The method described in Listing 4.7 has three parameters: `after_cursor` , `order`, `limit`, last of which has the default value 100. The parameter `after_cursor` holds the value of the `'after cursor'` received from the previous API call and used to make the next API call to retrieve the next batch of 100 revisions. In order to make the API call, the `requests` library is used. After a lot of frustrating attempts to make the API call using the obvious choice of HTTP method GET, which is usually used to request data from a specified resource, it became clear that the only way to make it work is using the HTTP method POST, which is normally used to send data to a server to create/update a resource. This is an undocumented feature of the Conduit API and has no obvious reasoning behind it. One supposition is that Conduit has all the API calls implemented with the HTTP POST method for the sake of simplicity. As a result, the function `requests.post()` is used to make a POST call, taking the API URL and data as arguments. The data in this case are the arguments for the Conduit method.

Taking as a basis the example given by the Conduit API Web UI for cURL, see subsection 3.2, the data argument was composed. The `order` is used so that it would be possible to to call the method with `"newest"` to get the id of the most recently created revision for displaying the download status, which is described later, and also with `"oldest"` for the usual calls to get all the revisions starting from the oldest. The `limit` is also used for downloading the newest revision when it is set to 1 so that it wouldn't download unneeded data, in the rest of the cases it is set to the default 100. The `createdStart` and `createdEnd` constraints are not mandatory for this API method, therefore they are added only when the user decides to specify a time frame. Without specifying these dates the method returns all the revisions available in the database.

```python
def get_revisions(self, after_cursor, order, limit=100):
    data = {'api.token': self.token,
            'attachments[subscribers]': 1,
            'attachments[reviewers]': 1,
            'attachments[projects]': 1,
            'order': order,
            'after': after_cursor,
            'limit': limit
            }
    if self.from_date is not None:
        data['constraints[createdStart]'] = self.from_date
    if self.to_date is not None:
        data['constraints[createdEnd]'] = self.to_date
    response = requests.post(self.url + 'differential.revision.search
                                       ', data=data)
```

Listing 4.7: The method `get_revisions()(first part)`

If for some reason the request was bad, i.e. a 4XX client error or 5XX server error response, it can be raised with `raise_for_status()` which will raise a Python exception with the appropriate message, see Listing 4.8. If the request is successful, the success code 200 is returned and this function does nothing. If there was no exception raised, then the result contained in `response.text` should contain a string instance containing a JSON document, which is de-serialised to a Python object using `json.loads()`. If Conduit encountered an error, then the HTTP response is still 200, but the `error_code` and `error_info` from response.text are not empty and contain the error message. This error can be raised using the same exception as for HTTP - `requests.exceptions.RequestException` for simplicity and then it will be handled accordingly in the code that calls this method `get_revisions`. Finally, if there are no errors, `'revisions'` is returned. `json.loads(response.text`.

```
    response.raise_for_status()
    revisions = json.loads(response.text)
    if revisions["error_code"] is not None:
        raise requests.exceptions.RequestException(
                revisions["error_code"], revisions["error_info"])
    return revisions
```

Listing 4.8: The method `get_revisions()` (second part)

The next method described in Listing 4.9 is `get_transactions()`, which handles the Conduit API method `transaction.search`. It uses the `revision_phid` returned in the previous function and since there also could be more than 100 of transactions for a revision, the `after_cursor` is passed as well. The rest is similar to the previous function with de-serialising the response, raising the exceptions and returning the `transactions`.

```
def get_transactions(self, revision_phid, after_cursor):
    data = {'api.token': self.token,
            'objectIdentifier': revision_phid,
            'after': after_cursor
            }
    response = requests.get(self.url + 'transaction.search', data=
                                   data)
    response.raise_for_status()
    transactions = json.loads(response.text)
    if transactions["error_code"] is not None:
        raise requests.exceptions.RequestException(
                transactions["error_code"], transactions["
                                                      error_info"])
    return transactions
```

Listing 4.9: The method `get_transactions()`

The last method discussed is `run()`, which runs the whole crawling process. Listing 4.10 shows how the variable `after_cursor` is defined as an empty string so that on the first call it would not act as a constraint and return the very first available objects. The variable `first_rev_id` is used to store the ID of the oldest revision to display the progress correctly. In order to show the progress, the total number of revisions is needed. This can be retrieved using the method `get_revisions()` with an empty `after_cursor`, the newest order and the limit set to 1 for a bit of micro-optimisation to get only the needed latest revision. The method call is enclosed in a `try` and `except` block to catch any exception and print a message that the process cannot continue if this first call failed. This is because if this initial call fails, it means that every other call regarding the revisions would fail as well. Therefore, some input data must be changed. The ID is stored in a variable called `last_rev_id`.

```
def run(self):
    after_cursor = ''
    first_rev_id = 0
    print('Writing revisions to ' + self.directory)
    try:
        last_revision = self.get_revisions('', 'newest', 1)
        last_rev_id = last_revision['result']['data'][0]['id']
    except Exception as exception:
        print("Getting the latest revision failed. Cannot continue
                                                further.")
        raise exception
```

Listing 4.10: The method `run()` (first part)

The next calls to actually retrieve the revisions starting from the oldest should continue until there are no more revisions left, when the `after_cursor` will be returned as `null` by the last API call, therefore the whole process is enclosed in a `while` loop with `after_cursor` as a condition to not be `None`, the Python keyword for `null`. The code snippet from Listing 4.12 is also enclosed in a `try` and `except` block to catch the HTTP and JSON decoding exceptions and stop the whole execution, because it cannot continue further, see Listing 4.11.

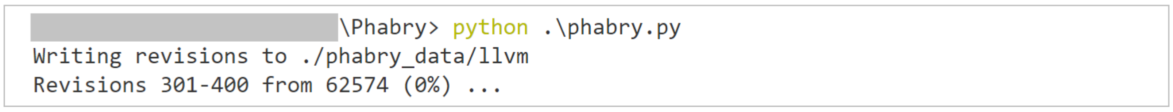
```
    while after_cursor is not None:
        try:
            ...
        except (requests.exceptions.RequestException, json.
                                                JSONDecodeError) as
                                                exception:
            after_cursor = '0' if after_cursor == '' else
                                                        after_cursor
            print("Getting revisions after {}. Cannot continue
                                                        further.".format(
                                                        after_cursor))
            raise exception
```

Listing 4.11: The method `run()` (second part)

The code snipped in Listing 4.12 is enclosed in the `try` statement from Listing 4.11. The first step is calling the method `get_revisions()` with the last available `after_cursor` and the `'oldest'` order. The variables `after_cursor`, `current_first_rev_id` and `current_last_rev_id` are set to their corresponding values from the `revisions`. The download status shows the ID range of the currently downloading revisions, the total number of revisions and the progress in percentage. Figure 4.4 shows the progress status as displayed after starting the `phabry.py` script. The lines 12-14 from the Listing 4.12 show how the revisions file name is formed from the IDs of the first and last revisions from the last call, and how the contents of `revisions` are saved as JSON format with an indentation of 2 to a text file in the `revisions` folder.

```python
revisions = self.get_revisions(after_cursor, 'oldest')
after_cursor = revisions['result']['cursor']['after']
current_first_rev_id = revisions['result']['data'][0]['id']
current_last_rev_id = revisions['result']['data'][-1]['id']
if first_rev_id == 0:
    first_rev_id = current_first_rev_id
print('Revisions', str(current_first_rev_id) + '-' + str(
                                  current_last_rev_id),
        'from', str(last_rev_id), '(' +
        str((current_first_rev_id - first_rev_id) * 100 //
            (last_rev_id - first_rev_id)) +
        '%) ...', end='\r')
file_name = str(current_first_rev_id) + '-' + str(current_last_rev_id
                                  ) + '.json'
with open(os.path.join(self.directory, 'revisions', file_name), 'w')
                                  as json_file:
    json.dump(revisions, json_file, indent=2)
```

Listing 4.12: The method run() (third part)



```
[redacted]\Phabry> python .\phabry.py
Writing revisions to ./phabry_data/llvm
Revisions 301-400 from 62574 (0%) ...
```

Figure 4.4: The progress status shown during the process

After receiving one batch of 100 revisions and saving them to a file, the transactions for each revision have to be retrieved under a for loop. The code snippet from Listing 4.13 follows the same pattern as the one for getting revision with a few notable differences. First difference, there is a `file_count` to be used when saving multiple transactions files for the same revision. Another one is that if any `Exception` happens to be caught, the process is not stopped, but continues further and only the error message with the current revision ID are saved in the log file using the previously described method `handle_exception()`.

```python
for rev in revisions['result']['data']:
    after_cursor_transactions = ''
    file_count = 0
    while after_cursor_transactions is not None:
        try:
            ...
        except Exception as exception:
            Phabry.handle_exception(exception, 'transactions of
                                         revision '
                                   + str(rev['id']))
            after_cursor_transactions = None
```

Listing 4.13: The method run() (fourth part)

The code snippet from Listing 4.14 follows the `try` statement from Listing 4.13. As can be seen, the procedure is very similar to the one for revisions, with the difference that if a revision has more than 100 transactions, then more than one call of `get_transactions()` is needed and then every file name has the form of transaction ID plus file count. For example, if the revision with ID 45 has 123 transactions, the file names would be `45_0` and `45_1`. The transaction files are saved in the directory `transactions`.

```
transactions = self.get_transactions(rev['phid'],
                                     after_cursor_transactions)
after_cursor_transactions = transactions['result']['cursor']['after']
file_name = str(rev['id']) + '_' + str(file_count) + '.json'
with open(os.path.join(self.directory, 'transactions', file_name),
             'w') as json_file:
    json.dump(transactions, json_file, indent=2)
if after_cursor_transactions is not None:
    file_count += 1
```

Listing 4.14: The method `run()` (fifth part)

This concludes the implementation of Phabry. The script can be run in the command line with Python. As described above, the input parameters can be provided in a configuration file or via command line arguments. The source code of the script, the test file and an example `config` file were uploaded on Github in a public repository[1].

[1]https://github.com/dimonco/Phabry

5 Evaluation

5.1 Requirements evaluation

The section 2.3 mentions that the requirements have to be met and therefore they need to be verified with tests. The Phabry script covers a list of Phabricator data sources, each with thousands of revisions and even more transactions. It is practically impossible to guarantee a test coverage of all the possible outcomes and circumstances. Nonetheless, a set of unit tests were written to assess the basic functionality of the script. Unit testing is done by testing each class method separately and independently from others.

The unit testing was performed using Python's `unittest`. The tests use real input parameters and download actual data from a Phabricator host, but use method stubs and mock objects to assist with testing a method in isolation. Listing 5.1 shows the main class of the unit test file, which is also called Phabry.

```
class Phabry(unittest.TestCase):
    @patch('os.makedirs')
    @patch('phabry.configure_logging')
    def setUp(self, mock_makedirs, mock_logging):
        mock_makedirs.return_value = True
        mock_logging.return_value = True
        self.phabry = phabry.Phabry("llvm",
                                    "https://reviews.llvm.org/api/",
                                    "api-token", "11-05-2019",
                                    "12-05-2019")
```

Listing 5.1: The `unittest` class definition

Listing 5.1 shows how the functions `os.makedirs` and `phabry.configure_logging` are substituted with mock functions in order to control the values and to not use any real files. The `setUp()` method is defined so that the test runner will run it prior to each test. One test example is provided below. The method `test_get_revisions()` tests the Phabry method `get_revisions()` by calling it with the provided parameters and asserting that the result is the same as the one expected, see Listing 5.2. This method does not require any other mock functions or values.

```
def test_get_revisions(self):
    revisions = self.phabry.get_revisions('')
    self.assertEqual(len(revisions['result']['data']), 6)
    self.assertEqual(revisions['result']['data'][0]['id'], 20236)
```

Listing 5.2: The unittest class definition

The class Phabry from the main script contains 4 main methods, all of which are covered by individual tests. However, covering the main script with unit tests is only a part the whole testing process. The rest of the functional and non-functional testing was performed manually. In the following paragraphs it is described how each requirement defined in the chapter 2 was tested.

F1: Crawler based on Phabricator The main script can download code review data defined as "differential revisions" in Phabricator from any publicly available repository that supports the Conduit API. It was tested with 3 Phabricator hosts: LLVM, FreeBSD and KDE. The script is also flexible and should require minimum modifications in case of future API changes.

F2: All available data is downloaded This requirement is a bit tricky to test because there is no way to test how much data there is in the database and that indeed all the data was provided during the API calls. Nonetheless, there were taken measures to ensure that the API methods used by the script use the right parameters to download all the data that the Phabricator host provides, in particular the correct use of the `'after cursor'` parameter that acts as a bridge between every API call until there are no more revisions or transactions left to download. Also, the response from each API call was saved in separate files as is, thus minimizing the possibility of data loss.

F3: Complete data Phabry downloads and saves the data in exactly the same condition as it comes from the Phabricator host. The only exception is that the JSON data comes as one line of text that is very hard to visually inspect, therefore it was saved as one key-value pair per line with proper indentation.

F4: Single-threaded and minimal server load The main script uses only one process and makes the API calls synchronously, i.e. each next call was made only after the previous one was finished. Thus, the impact on the Phabricator host was kept to a minimum.

F5: Reproducible behaviour Phabry supports the supplying of input parameters through command line arguments or via a configuration file. Each error encountered during the process is saved in a log file or shown directly on the screen, depending on the severity of the error. These points ensure a reproducible behaviour.

F6: Incremental crawling Incremental crawling was not implemented because it was considered as over-optimisation due to these two aspects: the script normally needs to be run only once to download the whole dataset and doesn't necessarily need daily updates. And then even if an update is needed at some point, it takes only a few hours to download the whole dataset again, therefore making the incremental crawling redundant, especially since it is cumbersome to implement correctly. Nonetheless, it is possible to specify a time frame as input parameters and download only a subset of the data, thus being able to update the dataset with the latest data.

F7: Anonymization of personal data Then data downloaded by the script does not contain any personal data such as names or emails, only the authors hashed ID's are available.

N1: Efficiency Phabry is very lightweight and uses the CPU at 1-2% and maximum 20 MB of RAM. The downloaded data has the size of 1 KB − 200 KB per API call, therefore the impact on network usage is minimal.

N2: Execution time The execution time depends on the Phabricator host and on the total number of revisions and transactions. Three hosts were tested and the results are provided in the Table 5.1.

Phabricator host	**FreeBSD**	**KDE**	**LLVM**
Number of revisions	20400	21500	62700
Size	290 MB	295 MB	1170 MB
Download time	3 hours	1 hour	14 hours

Table 5.1: Overview of datasets downloaded in May 2019

5.2 Dataset analysis

Phabry was used to collect code review data from 3 open source Phabricator hosts. Table 5.1 shows a brief overview of the datasets from each instance. Appendices A and B show excerpts from the files with the corresponding data. The dataset from FreeBSD was chosen for the analysis. The main tools used to analyse the data are *Jupyter Notebook*[1] for live code execution and visualisation of text via browser, and *pandas*[2] as a high-performance tool for data structures and data analysis. The aim is to get an insight into the data and to create a demo Jupyter notebook on how to process Phabricator data for further analysis.

[1]https://jupyter.org/
[2]https://pandas.pydata.org/

One important aspect of pandas is that it works best with two-dimensional heterogeneous data, however the review data crawled from Phabricator does not perfectly fit the two-dimensional criterion. For example, while every revision comes as a dictionary, the `'fields'` key is itself a dictionary that can contain another dictionary. Also, the `'attachments'` key contains dictionaries with lists of other dictionaries. The same goes for the transactions. Therefore, it is required to do some preprocessing in order to be able to analyse the data with pandas. One way to achieve the desired structure is to create new tables from the nested dictionaries. For example, a table `reviewers` was created with every reviewer attached to each revision in a many to one relation. See Appendix C for the demo code. As with any demo, it can be further improved and adapted to the desired functionality.

Phabricator host	**FreeBSD**	**KDE**	**LLVM**
Number of revisions	20479	21477	62721
Date of the first revision	17-12-2013	05-02-2015	09-07-2012
Percentage of closed revisions[3]	91,9 %	93,1 %	90,6 %
Percentage of published over closed revisions[4]	88,6 %	89,9 %	89,3 %
Number of reviewers	682	759	1664
Mean value of reviews per reviewer	30,3	28,3	37,7
Number of authors	675	1201	2509

Table 5.2: Dataset statistics and metrics

The demo code was used to import the revisions and transactions into pandas data frames and analyse the code review data to get some statistics. Table 5.2 shows the gathered metrics for a general overview of the datasets. Another metric that can be tested is the workload for individual reviewers. Figure 5.1 shows the distribution of revisions per reviewer and it follows the Pareto distribution, as would be expected. These results reinforce the that a small group of experienced reviewers can handle the review process. It can also be observed that out of the three projects, KDE seems to be the most extreme and even if the total numbers of reviewers and revisions are comparable to those of FreeBSD, there are a lot fewer reviewers that deal with the absolute majority of revisions.

[3] *Open* revisions - *in review* or *accepted* but not *published* yet
[4] *Closed* revisions can be either *published* or *abandoned*

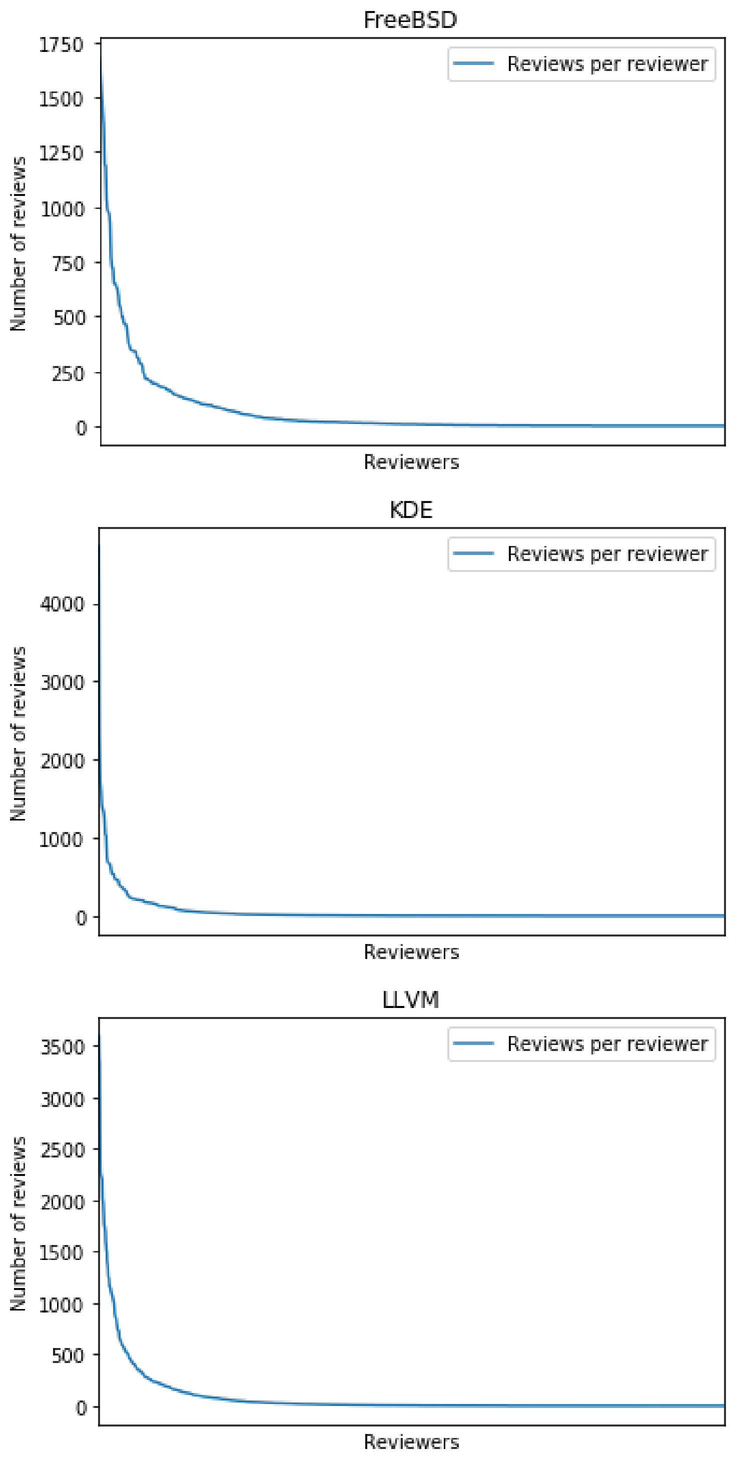


Figure 5.1: Number of reviews per reviewer

6 Conclusion

Code review is an important part of the development of a product. It is widely recognised as one of the best methods to improve code quality, to reduce the number of bugs and to help in knowledge transfer. Nonetheless, It is crucial to continue the research of code review and ways to improve it.

The purpose of this thesis was to develop a tool that crawls code review data from Phabricator instances of open source projects that support it. All the requirements were took in consideration when developing the crawler, named Phabry, and a set of unit tests was written and manual testing was performed for quality assurance. Phabry can crawl code review data from the supported hosts and saves all the data formatted as JSON into text files for further analysis.

The Phabricator datasets crawled from 3 open source projects: FreeBSD, KDE and LLVM were used for an initial data analysis. An overview of the analysis was presented. The demo on how to process Phabricator code review datasets was provided as a Jupyter notebook to help in future research.

Appendix A   Excerpt from a revisions file

```
{
  "result": {
    "data": [
      {
        "id": 2,
        "type": "DREV",
        "phid": "PHID-DREV-rcox32omctpd3g5pzqsj",
        "fields": {
          "title": "Enable LLDB by default on platforms where we build Clang",
          "authorPHID": "PHID-USER-axkijn4pfdxlezgbnt7g",
          "status": {
            "value": "published",
            "name": "Closed",
            "closed": true,
            "color.ansi": "cyan"
          },
          "repositoryPHID": null,
          "diffPHID": "PHID-DIFF-lazxnqfctfserhgzysxg",
          "summary": "",
          "testPlan": "Check that lldb is built if no WITH_ or WITHOUT_ in src.conf",
          "isDraft": false,
          "holdAsDraft": false,
          "dateCreated": 1387310092,
          "dateModified": 1557876634,
          "policy": {
            "view": "public",
            "edit": "users"
          }
        },
        "attachments": {
          "subscribers": {
            "subscriberPHIDs": [
              "PHID-USER-sfbxp2cksgub2ywlvupr",
              "PHID-USER-q5lmute3rwskeizvu5gf"
            ],
            "subscriberCount": 4,
            "viewerIsSubscribed": false
          },
          "reviewers": {
            "reviewers": [
              {
                "reviewerPHID": "PHID-USER-iv3vop5y36l265phkqzr",
                "status": "accepted",
                "isBlocking": false,
                "actorPHID": null
              },
              {
                "reviewerPHID": "PHID-USER-sfbxp2cksgub2ywlvupr",
                "status": "accepted",
                "isBlocking": false,
                "actorPHID": null
              }
            ]
          },
          "projects": {
            "projectPHIDs": []
          }
        }
      },
      ...
     {
        "id": 104,
        "type": "DREV",
        "phid": "PHID-DREV-4apqrggjn2p534ndkram",
        "fields": {
          "title": "Add DOCS and EXAMPLE options to biology/mafft",
          "authorPHID": "PHID-USER-lwgald3qwndle6ytbupv",
          "status": {
            "value": "published",
            "name": "Closed",
            "closed": true,
            "color.ansi": "cyan"
          },
          "repositoryPHID": "PHID-REPO-kevfhl5245mevhmor36u",
          "diffPHID": "PHID-DIFF-zfhql5h3s5tyep65mdon",
          "summary": "Add DOCS and EXAMPLE options to biology/mafft\n\nPR: ports/190161",
          "testPlan": "http://poudriere.ircmylife.com:13780/data/latest-per-pkg/mafft/7.149/",
          "isDraft": false,
          "holdAsDraft": false,
          "dateCreated": 1401295492,
          "dateModified": 1401509954,
          "policy": {
            "view": "users",
            "edit": "users"
          }
        },
        "attachments": {
          "subscribers": {
            "subscriberPHIDs": [],
            "subscriberCount": 0,
            "viewerIsSubscribed": false
          },
          "reviewers": {
            "reviewers": [
              {
                "reviewerPHID": "PHID-USER-26snrek27a4tph6sxbot",
                "status": "accepted",
                "isBlocking": false,
                "actorPHID": null
              },
              {
                "reviewerPHID": "PHID-USER-7dwczumdatyyxdjeiazl",
                "status": "added",
                "isBlocking": false,
                "actorPHID": null
              }
            ]
          },
          "projects": {
            "projectPHIDs": []
          }
        }
      }
    ],
    "maps": {},
    "query": {
      "queryKey": null
    },
    "cursor": {
      "limit": "100",
      "after": "104",
      "before": null,
      "order": "oldest"
    }
  },
  "error_code": null,
  "error_info": null
}
```

Appendix B Excerpt from a transactions file

```
{
  "result": {
    "data": [
      {
        "id": 10234,
        "phid": "PHID-XACT-DREV-gqh6xxdp6quyp27",
        "type": null,
        "authorPHID": "PHID-USER-ycbrujqwoepzdb6clww2",
        "objectPHID": "PHID-DREV-m7b22drp5cp77n7ydrew",
        "dateCreated": 1408547665,
        "dateModified": 1408547665,
        "comments": [],
        "fields": {}
      },
      {
        "id": 29,
        "phid": "PHID-XACT-DREV-hwcq3sbpvtaect6",
        "type": "comment",
        "authorPHID": "PHID-USER-ycbrujqwoepzdb6clww2",
        "objectPHID": "PHID-DREV-m7b22drp5cp77n7ydrew",
        "dateCreated": 1399735586,
        "dateModified": 1399735620,
        "comments": [
          {
            "id": 10,
            "phid": "PHID-XCMT-lybeamzvp7zbtehwtigm",
            "version": 3,
            "authorPHID": "PHID-USER-ycbrujqwoepzdb6clww2",
            "dateCreated": 1399735620,
            "dateModified": 1399735620,
            "removed": false,
            "content": {
              "raw": "Ah, well, ‘USES=perl5‘ should have gone the argument way to begin with, but doesn’t matter.\n\n
              Also, I only put this here because I wanted to test ‘arc‘ and see how it worked, and it was the first
              thing handy 0:-)\n\n‘USES‘ right now is missing the ability to do:\n\n  USES=perl5:build\n
              .if ALSO_RUN\n USES+=perl5:run\n
              .endif\n\nand a few ports need to do things like this before this could even go further."
            }
          },
          ...
        ],
        "fields": {}
      },
      ...
    ],
    "cursor": {
      "limit": 100,
      "after": null,
      "before": null
    }
  },
  "error_code": null,
  "error_info": null
}
```

Appendix C Demo code for processing the code review data

```
import pandas as pd
import os
import json
from datetime import datetime

revlist = []
reviewerslist =[]
revdir = os.path.join('phabry_data', 'freebsd', 'revisions')
for filename in os.listdir(revdir):

    if filename.endswith('.json'):
        with open(os.path.join(revdir, filename)) as f:
            rev = json.load(f)
            rev = rev['result']['data']
            for row in rev:
                for key in row['fields'].keys():
                    if key in ['title', 'authorPHID', 'repositoryPHID', 'diffPHID', 'summary']:
                        row[key] = row['fields'][key]
                    if key in ['dateCreated', 'dateModified']:
                        row[key] = datetime.utcfromtimestamp(row['fields'][key]).strftime('%Y-%m-%d %H:%M:%S')
                    if key == 'status':
                        row[key+'Value'] = row['fields'][key]['value']
                        row[key+'Name'] = row['fields'][key]['name']
                        row[key+'Closed'] = row['fields'][key]['closed']
                row.pop('fields', None)
                for r in row['attachments']['reviewers']['reviewers']:
                    r['phid']=row['phid']
                reviewerslist.extend(row['attachments']['reviewers']['reviewers'])
                row.pop('attachments', None)
            revlist.extend(rev)
revisions = pd.DataFrame(revlist)
reviewers = pd.DataFrame(reviewerslist)

transdir = os.path.join('phabry_data', 'freebsd', 'transactions')
tralist = []
for filename in os.listdir(transdir):
    if filename.endswith('.json'):
        with open(os.path.join(transdir, filename)) as f:
            rev = json.load(f)
            rev = rev['result']['data']
            for row in rev:
                    row['dateCreated'] = datetime.utcfromtimestamp(row['dateCreated'])
                    row['dateModified'] = datetime.utcfromtimestamp(row['dateModified'])
            tralist.extend(rev)
transactions = pd.DataFrame(tralist)
```

References

Conduit - differential.revision.search. (n.d.). Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/differential.revision.search/

Conduit - transaction.search. (n.d.). Retrieved May 21, 2019, from https://secure.phabricator.com/conduit/method/transaction.search/

Conduit API Overview. (n.d.). Retrieved May 20, 2019, from https://secure.phabricator.com/book/phabricator/article/conduit/

Differential User Guide. (n.d.). Retrieved May 10, 2019, from https://secure.phabricator.com/book/phabricator/article/differential/

Differential User Guide: Inline Comments. (n.d.). Retrieved May 12, 2019, from https://secure.phabricator.com/book/phabricator/article/differential_inlines/

Kolawa, A. & Huizinga, D. (2007). Automated defect prevention: Best practices in software management. (p. 260). Wiley-IEEE Computer Society Press.

Phabricator - Community Resources. (n.d.). Retrieved May 22, 2019, from https://secure.phabricator.com/w/community_resources/

Phabricator Login. (n.d.). Retrieved May 22, 2019, from https://secure.phabricator.com/auth/start/?next=%2F

Phacility - Home. (2019). Retrieved May 10, 2019, from https://phacility.com/

Tsotsis, A. (2011). Meet phabricator, the witty code review tool built inside facebook. Retrieved from https://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/

User Guide: Review vs Audit. (n.d.). Retrieved May 10, 2019, from https://secure.phabricator.com/book/phabricator/article/reviews_vs_audit/