The feasibility and practicality of a generic social media library

Fredrik Jonsén
Linköping University
Linköping, Sweden
frejo105@student.liu.se

Alexander Stolpe
Linköping University
Norrköping, Sweden
alest170@student.liu.se

ABSTRACT
Many people today use social media in one way or another, and many of these platforms have released APIs developers can use to integrate social media in their applications. As many of these platforms share a lot of functionality we see a need for developing a library, to contain these, and ease the development process when working with the platforms. The purpose of this paper is to find common functionality and explore the possibility of generalization in this regard. We first look for common denominators between the top social media networks, and using this information we attempt to make an implementation to evaluate the practicality. After the development process we analyze our findings and discuss the usability and maintainability of such a library.

INTRODUCTION
Today a lot of people are using social media in one way or another and it is estimated that there will be around 2.67 billion social media users around the globe by 2018[8]. Most of these social networks have released Application Programming Interfaces (APIs) which developers can utilize to integrate these networks into their software.

Social network usage is growing and has gone from 0.97 billion users in 2010 to 2.14 billion in 2015.[8]. This would account for approximately 29% of the earth's population in 2015, which was 7.347 billion 2015[12]. It is worth noting that this counts created user accounts and not unique users, one person can have several accounts over multiple networks, and accounts may not belong to an actual person, but rather companies, organizations, or bots.

Because of the currently high, and still growing, number of social media users we find it highly likely that we will see an increasing number of applications that involve social media in their software in one way or the other. Out of all the social networks existing today there are twenty that have more than 100 million active accounts[9]. This means that if one would want to create an application that involves a lot of social networks we will have to do a lot of work just to implement all of these into our system.

Purpose
Because of this we see a need for a way to combine these social network APIs in some way to save development time and reduce the amount of duplicate code written in software. The purpose for this project is to create a library that combines the APIs in a modular fashion, where each API serves as a module in order to simplify adding new networks, and make it easier to involve social media in software.

A software library by definition is a set of pre-written code that a developer might add to a project in order to add more functionality or to ease the development process[11].

For our study, we define modularity as the extent to which a program can be divided into modules where[4]:

- Each module has a well-defined interface that describes how the system can interact with it.
- Changes to the underlying functionality of a module does not affect the modules interface.
- Modules can be put together with other modules in different ways to make a complete program

In our case modularity will mostly affect how easy it will be to integrate new social media APIs in the future.

Over time, as new Social Networks enter the market, it will have to be possible to integrate them into our library. As such, the code will have to be maintainable. Maintainability is a very broad term, but in general focuses on how simple the code in itself is to work with, when changes or new additions are necessary. In the Theory chapter we will explore this subject more thoroughly. For the evaluation of the maintainability of our library, we will be making use of the third party tool SonarQube[1].

RESEARCH QUESTION
Is it possible to create a maintainable modular library for Social Network APIs?

As the question pertains to the *possibility* of creating the library, it is also necessary to define what would make us deem it impossible to finalize the library with a satisfactory result. In regards to this, we see two major potential risk factors:

1. It may not be possible to generalize the results of the APIs, but each API will instead require a large mount of exceptions, making a common interface meaningless.
2. The rate of changes to the APIs may be unmaintainable. If we during the course of the relatively short development time find ourselves having to go back multiple times to adjust already implemented functionality due to changes in the API, we will consider the library unmaintainable.

LIMITATIONS
Because there are so many social networks existing today and we have a finite amount of time to complete this project we will focus on a smaller set of APIs to implement into our library. We have set up a few criteria for the APIs so we can find suitable candidates for our library, which are:

[1] https://www.sonarqube.org/

- The social network must be one of the 22 most popular[9].
- It must be relevant for our geographical location, in this case Europe.

Other than this we will judge the API itself subjectively by how good its documentation is, its functionality and its ease-of-use.

THEORY

Code Reuse

A lot of time and money can be saved by reusing code, of which libraries are one form. Although there are some issues with using libraries, the gains from avoiding reinventing the wheel makes writing and using libraries a common practice, in particular in open source software[2].

Writing code which can be easily reused requires a deeper analysis of the problem domain, which may increase the cost and time required compared to developing the same code without reuse in mind, but can drastically decrease the cost of developing systems in the future where the code can be reused [5]. This cost reduction is apparent foremost in terms of direct cost of development, but also in time-to-market, which can be argued to be even more important in the long term[1].

With code reuse there are also several potential issues which have to be kept in mind, both when it comes to the implementation itself but also when it comes to using the implementation. Backwards compatibility between versions is a major topic in itself[6]. There is also the risk of a library being abandoned by it's maintainer. This is especially true for proprietary libraries, where the source code may not be available. It this case, the library might have to be replaced, making all the effort to use the library wasted.

Library Interface Design

When designing our library we will want keep several things in mind. We want to design the library so it's easy and straightforward to use for a developer. As Henning points out in his article[3], about design of APIs, it is very easy to create a bad one, but very hard to create an API that feels natural and easy to work with. APIs, as we know, are a kind of interface for a program to gain access to another program without direct access, and can be compared to the interface of our library.

Henning continues to discuss guidelines for how an API should generally be designed. What feels most relevant to our work is how he describes how the APIs should be designed from the perspective of a user, because when it's done from the implementer's point of view the needs of the user are often forgotten. It's usually best to document first, because when it is done after the implementation the programmer, who wrote the functionality, will usually just dictate what he did, rather than make it is obvious enough for others who are not as familiar with the code.

Maintainability

Maintainability can be defined as the simplicity with which defects can be corrected and the library can be extended or modified to support future requirements.[10]. This can generally be measured by certain quantifiable attributes, such as unit test coverage, lines of code and cyclomatic complexity. [7]. It also includes more subjective aspects, such as how self-explanatory the code is, and how well commented sections of necessarily complex code is.

Social Media APIs

A social media API is an interface through which the software can integrate functionality offered by the social media platform, commonly set up as a REST API. These can be used through registering the application on the social media site, thus acquiring an authentication token which can be used to call different endpoints via HTTP. The services offered varies between the APIs, but tend to share some basic functionality such as publishing posts and sending private messages.

As mentioned, one of the biggest issues with using libraries is the risk of the project being abandoned. This risk increases significantly when the library itself uses APIs which integrate oft-changing platforms, such as the case of Social Media. The rate of change differs between networks, in some cases on average three times a year[2], in other cases several times a month[3], although the impact of the changes varies greatly. At best, a lack of active development simply means missing out on new functionality. In other cases, such as unfixed security vulnerabilities[4], may render the library unusable.

Similiar work

Several similar works, Agorava[5], ASNE[6] and SocialMedia Abstractions[7]serve the same or similar purposes, but all were abandoned before reaching a stable release. In the case of ASNE, the project was abandoned explicitly due to a lack of free time. This shows the issue of a library having only a single maintainer, as the project risks being abandoned by it's developer as soon the project is no longer a priority. The others have no stated reason for the lack of continued development. There are also commercial services[8] which provide this functionality for some popular APIs, but charge money to sign up and use. As it is proprietary it's inner workings are completely opaque, and thus will not be examined by this study.

Encountered issues

Despite being abandoned, we hope to still learn from the problems the similar libraries encountered and solved. For this, we looked into each project's issue tracking (where available) and commits. This was somewhat complicated in that one project, SocialMedia Abstractions, simply had not used issue tracking. Another project, ASNE did use issue tracking, but much of the discussion regarding individual issues was largely in Russian, making it unusable in our case.

[2] https://developers.facebook.com/docs/apps/changelog
[3] https://www.hitchhq.com/twitter/activities
[4] https://github.com/gorbin/ASNE/issues/107
[5] http://www.agorava.org/
[6] https://github.com/gorbin/ASNE
[7] https://github.com/socialsensor/socialmedia-abstractions
[8] https://cloudrail.com

The issues, despite the name, did not always regard bugs. In the vast majority of cases, issued stemmed from users misunderstanding the library documentation, requesting features, or suggesting refactoring of code to increase maintainability. In the case of documentation misunderstandings, these were often solved by simply adding examples. There was also a noticeable difference in the amount of issues pertaining bugs in ASNE, which included no automatic tests, compared to Agorava, which includes a large amount of automatic tests, and had almost no issues regarding logical errors, despite having a much larger code base.

METHOD

To find out if we could create a modular library for social network APIs we structured our project into three different parts. We began with evaluating the most popular networks on a basis of which are relevant for our project and which are appropriate to integrate into our library based on functionality. As the next step we then implemented the library, using SonarQube throughout development to ensure code quality and maintainability stayed high. Lastly we evaluated the library in two different ways. First we analyzed the data we got from SonarQube and see how well our library performs, and then created a small application, in order to test the library's practical usability.

Social Networks selection

To choose which APIs to implement into our library we initially looked at what social media network sites where most popular[9]. Out of these networks we sorted out those which were geographically irrelevant for the focus of this project, which is the European market. We then further narrowed down the list by looking at the functionality of each API. We selected the APIs which shared a lot of common functionality, which made them more appropriate for the core idea of our study, to develop a modular library.

Implementation

When we had chosen APIs to implement we started we started to implement our library. For this we have chosen to work in the Java language, as we found a decent number of pre-existing API-specific libraries we could use in this language. We used the most current version of Java at the time of our project, Java SE 8. We also used several tools, which we will describe briefly.

Maven

Maven[9]is a project management tool. While the tool includes a lot of functionality, we mostly made use of its automated build tool, release-, and dependency management.

IntelliJ IDEA

IntelliJ IDEA[10]is the Integrated Development Environment we chose for our project. It offers integration with Maven, syntax highlighting, static analysis, boilerplate code generation and other functionality to simplify the implementation work.

[9]https://maven.apache.org/
[10]https://www.jetbrains.com/idea/

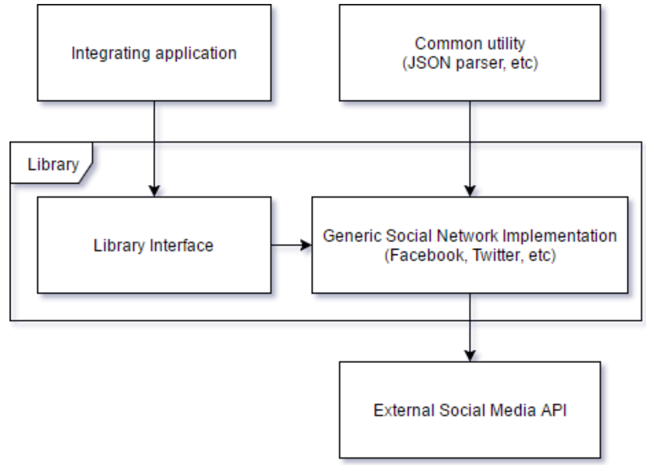

Figure 1. Architectural overview of library and use case

Travis CI

Travis[11]is a free Open Source service for continuous integration. It integrates with github and runs tests automatically for the project when changes are checked in. When done, participants of the project can be notified of the result.

SonarQube

Sonarqube[12]is a service for, in their own words, continuous code quality. As with Travis, it integrates with GitHub and automatically runs a series of checks when code is committed to the repository. This includes analysing the code for general issues, such as a high cyclomatic complexity, missing documentation and dead code, but also more detailed issues which may be easy to miss, such as potential infinite loops, unhandled exceptions, or checking floating point numbers for equality.

Work Method

We chose to work using a test driven development process, where we first wrote tests for our interfaces. During this process we also documented the functionality according to Hennings[3] concept of documenting first, in an effort of making the documentation more understandable from the users perspective. During this period we implemented functionality iteratively for our chosen APIs, implementing one functionality at a time for each as to not be left with only one fully implemented API in our library.

Design

Our library mainly consist of two parts, see figure 1. Where the library interface is what the users are in contact with when interacting with our library. This interface is an abstraction of a generic social network API. The Generic Social Network Implementation is the actual implementation of each social media platform and acts as the middleware between the interface and the external social media platforms. The internals of each implementation might differ greatly, where some already had an existing library that we could plug in and use for our library, while we had to implement

[11]https://travis-ci.org/
[12]https://www.sonarqube.org/

others completely ourselves, using the REST APIs directly. The ones we implemented use common utility tools such as an HTTP client and a JSON deserializer in order to be able to properly communicate with the external platforms.

The already existing libraries we used are:

- twitter4j (http://twitter4j.org/)
- facebook4j (https://facebook4j.github.io/)
- jumblr (https://github.com/tumblr/jumblr)

Evalution

The evaluation of our library consists of two parts. Throughout the project we have used SonarQube to analyze the quality and maintainability of our code, which is the first part. The second part involves creating a small application which utilizes the library. When selecting the functionality of the application we looked at what common denominators was shared between the different plaforms, that might be suitable. This will gave us a hands-on experience on how easy our library is to use in practice and the more analytical aspect gained from the SonarQube analysis.

When integrating our library into our test application, we mainly focused on if we felt that there was anything lacking, or if something was not working as intended. If we felt the need to go back and change the source code of the library, or its interfaces, the result is deemed unsatisfactory.

RESULTS

Social Networks Selection

When looking at social media APIs we found that out of the 22 networks from our list[9], 10 were geographically irrelevant for our study as many of these were niched at the Asian market, primarily the Chinese one. We then narrowed it down further to networks with APIs which shared a lot of common functionality, see Appendix A, making them more appropriate for the core idea of our study. Given that the time of the project is finite we chose to look at the top four platforms with good functionality, where Facebook and Messenger was regarded as a single platform.

- Facebook
- Instagram
- Tumblr
- Twitter

Comparison

While comparing APIs functionality we found out that these differ both in how well developed they are as well as how much functionality they offer. Out of the top four platforms facebook and twitter had the most well developed APIs, as well as libraries developed by the same developer. Making these our top priority. For the last two platforms we chose to prioritize Tumblr over Instagram considering it had an official library written in java, which would make it faster for us to implement into our own library. As such, our final prioritization was as follows:

1. Facebook
2. Twitter
3. Tumblr
4. Instagram

Implementation

In terms of tools used, no implementation compromises had to be made to suit our configuration. Although the library has few dependencies, using Maven made the project significantly more portable, and outside of the initial startup and configuration, handling dependencies proved to be no issue, even when switching between workstations.

For SonarQube, our original intention was to use it for evaluation of the end result, but it proved useful during the implementation as well. When committing the code to our git repository, SonarQube would automatically pull the code and analyze it. As such, we always had a current idea of the state of the quality of the implementation, and we occasionally took the time to correct issues SonarQube reported. A plugin called SonarLint[13], developed by SonarQube, was also available for IntelliJ IDEA, which uses a more basic rule set than the SonarQube service, but allowed us to avoid many of the most common mistakes, such as unnecessary package imports, which may not directly impact the usefulness of the library, but results in cleaner, and more easily worked with, code.

Work Method

Using Test Driven Development helped in many cases to give a better understanding at the problem each function is meant to solve. This was the most useful when writing functions which have no side effects, as it allows for clearer sets of inputs and outputs to and from the function. In essence, writing the tests firsts forced us to write the functions in a way that made them easier to test, and functions which were easy to test also turned out to be easier to understand.

At the start of the project we set out to document first as to make the documentation reach a higher quality. As our library mainly consists of already documented functionality in most cases this were merely duplicate information and did not impact our own documentation in any meaningful way.

Design

For two of our chosen platforms, Facebook and Twitter, third-party library implementations of the respective APIs were available. While using these libraries significantly reduced the workload, some compromises had to be made. One of the two, facebook4j, is completely synchronous, which meant the other implementations had to be synchronous as well in order to be generalized properly.

As the goal was to generalize the Social Network APIs, we had to look for the lowest common denominator. This meant that the classes for users and posts only supported the very most basic functionality, such as getting their id, name and biography. Other functionality, such as getting the friends of a user, had to be relegated to API-specific classes, due to the way friendships work on different networks. An example

[13] http://www.sonarlint.org/

would be Facebook, where friendship is mutual, while the closest equivalent on Twitter is follow, which can be one-sided. On the other hand, Facebook also has a concept of follow, which is not accessible through the API, and thus Facebook friends can not be generalized together with the Twitter follow concept.

In some cases, in particular for Facebook, functionality which is available on the site and library users would expect to be present in the library, is not available in the APIs. This is largely in regards to functionality which could potentially be used to spam or otherwise inconvenience regular users, such as sending friend requests. Some functionality is available, but has additional rules for usage, such as direct messaging. In this case, users are required to start the conversation, to avoid the chat being used as a system for notifications or ads. This does also mean that the usefulness of chat functionality is significantly reduced in the simple REST API we intended the library to be, and we decided to leave it out of the library, as it would be more suitable for libraries with this specific purpose.

Evaluation

Not reached in project yet.

REFERENCES

1. Griss, Martin L. Software reuse: from library to factory. *IBM systems journal*, 32(4), 1993: 548–566.
2. Haefliger, Stefan, Krogh, Georg von, and Spaeth, Sebastian. Code reuse in open source software. *Management Science*, 54(1), 2008: 180–193.
3. Henning, Michi. Api design matters. *Queue*, 5(4), May 2007: 24–36.
4. Kiczales, Gregor and Mezini, Mira. Aspect-oriented programming and modular reasoning. *Proc. ICSE '05*, 49–58.
5. Lim, Wayne C. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5), 1994: 23–30.
6. Raemaekers, Steven, Deursen, Arie van, and Visser, Joost. Measuring software library stability through historical version analysis. *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. 2012, 378–387.
7. SonarQube. Metric definitions. Address: **https://docs.sonarqube.org/display/SONAR/Metric+Definitions**.
8. Statista. Number of social network users worldwide from 2010 to 2020 (in billions). Address: **https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users**.
9. Statista. Leading social networks worldwide as of january 2017, ranked by number of active users (in millions). Address: **https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/**.
10. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, Dec. 2010: 1–418.
11. The Linux Documentation Project. Shared libraries. Address: **http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html**.
12. World Bank. Population, total. Address: **http://data.worldbank.org/indicator/SP.POP.TOTL**.

Appendices

APPENDIX A: API SUPPORT TABLES

	OAuth	Get basic user info	Get profile image	Like/unlike post	Get main post	Get comments on post	Get likes on post	Get users posts	Write post	Publish image	Publish video	Write comment on post	Search post	Search user	Search event	Search organization	Make checkin	create event	Get RSVPs
Facebook	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Twitter	Yes	Yes	Yes	Yes	Yes	No	Count only	Yes	Yes	Yes	Yes	Yes	Yes	Yes	N/A	N/A	N/A	N/A	N/A
Instagram	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Tags	Yes	N/A	N/A	N/A	N/A	N/A
Tumblr	Yes	Yes(blog)	Yes	Yes	Yes	No	Only whole blogs	Yes	Yes	Yes	Yes	No	Tags	No	N/A	N/A	N/A	N/A	N/A

Get user	Age	Email	First name	Gender	Home town	Id	Languages	Surname	Username	Website	Bios/Descript	Uploads count	Following count	Followed count	Likes by user
Facebook	Yes	Yes	Yes	Yes	Yes	App-Specific Scoped ID	Yes	Yes	N/A	Yes	Yes	No	No	No	Pages(AU)
Twitter	Yes	Yes(whitelist needed)	N/A	N/A	Location	Yes	Yes	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Posts
Instagram	N/A	No	Fullname	No	N/A	Yes	No	Fullname	Yes	Yes	Yes	Yes	Yes	Yes	Posts(AU)
Tumblr	N/A	No	N/A	N/A	N/A	Blog hostname	No	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Posts(AU)

Get post	Text body	Creation time	Id	Get comments	Get tags	Get image if present	Edit time	Get attachements	Get likes	Author	Reply To	Geo-tagged	Quote of	Shared count	If shared, original post	Language	Permalink
Facebook	Yes	Yes	Yes	Yes	Part of post body	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Twitter	Yes	Yes	Yes	N/A	Yes	Yes	N/A	Yes	Count only	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Can be generated with id
Instagram	Yes	Yes	Yes	Yes	Yes	Yes	N/A	Yes	Yes	Yes	N/A	Yes	N/A	N/A	N/A	N/A	Yes
Tumblr	Yes	Yes	Yes	No	Yes	Yes	N/A	Yes	Yes	Yes	Yes	N/A	Yes	Yes	Yes	N/A	Yes

Relations	Follow	Unfollow	Show relation user-user	ON/OFF notific.	ON/OFF retweets	Block user	Unblock	List friends	Followers List	Following List	Pending In(AU)	Pending Out(AU)	Blocked list(AU)	Get users friends	Friend request
Facebook	No	No	Yes	Yes	N/A	Yes	Yes	Yes	No	No	No	No	Yes	Yes	No
Twitter	Yes	Yes	Yes	Yes	Yes	Yes	Yes	N/A	Yes	Yes	Yes	Yes	Yes	N/A	N/A
Instagram	Yes	Yes	AU-user	No	N/A	No	No	Yes	Yes(AU)	Yes(AU)	Yes	Yes	N/A	N/A	N/A
Tumblr	Yes	Yes	No	No	N/A	No	No	N/A	Yes	Yes	N/A	N/A	No	N/A	N/A

Direct MSGs	AU				
	Send	Destroy DM	Inbox	Outbox	Get img from DM
Facebook	Yes*	No	Yes	Yes	Yes
Twitter	Yes	Yes	Yes	Yes	Yes
Instagram	No	No	No	No	No
Tumblr	No	No	No	No	No

Figure 2. Overview of functionality supported by different APIs. Green indicates the functionality is fully available through the API. Yellow indicates functionality is partly available. Red indicates functionality is present on the platform, but not available through the API. Gray indicates the functionality is not a part of the platform, and is as such not applicable.
*** For Facebook, direct message conversations must be initiated and kept alive by the conversation target in order to be used through the API.**