Refactoring Legacy Python Code to Patterns
Towards a Reduction of Code Smell in the Gajim Jabber Client

SEMINAR PAPER
BY
STEPHAN ERB

WITHIN THE COURSE OF STUDIES OF
APPLIED COMPUTER SCIENCE
AT THE
UNIVERSITY OF COOPERATIVE EDUCATION KARLSRUHE

JANUARY 8, 2010
Abstract

Refactoring is the process of improving software internally without affecting external behaviour. Refactoring is difficult and takes a lot of time. Unfortunately, legacy code is valuable and worth to be refactored.

This paper shares some experiences gained during refactoring of Gajim, a Jabber client written in Python. It is shown why refactorings do not work in isolation but require a context to be effective. It is explained why refactorings work well together with design patterns and principles.

In the course of this discussion three concrete refactorings from the Gajim environment are presented.
Contents

1 Introduction 1

2 The Gajim Project Environment 2
  2.1 XMPP Basics ................................................. 3
  2.2 Architecture Overview ..................................... 4
  2.3 Need for improvement ....................................... 5
    2.3.1 Known issues ........................................... 5
    2.3.2 An Example .............................................. 5
  2.4 Motivation .................................................... 6

3 Context-aware Refactoring 7
  3.1 Four different Paths ......................................... 7
    3.1.1 Do nothing ............................................... 8
    3.1.2 Rewrite .................................................. 8
    3.1.3 Evolve ................................................... 8
    3.1.4 Let evolve ............................................... 8
  3.2 The role of Design Patterns and Design Principles ............... 9
  3.3 Refactoring Activities ....................................... 9
    3.3.1 Find areas of code to improve ......................... 10
    3.3.2 Decide on the refactorings to perform ................... 10
    3.3.3 Safeguard the planned changes ......................... 11
    3.3.4 Make the planned changes ............................... 12
    3.3.5 Assess the value of the refactoring .................... 12

4 Refactoring Cases 13
  4.1 Decouple Hard-Coded Features with Plugins .................. 13
    4.1.1 Adding new behavior .................................... 14
    4.1.2 Altering existing functionality ......................... 16
    4.1.3 Further Plans ........................................... 16
  4.2 Introduce Null-Object and more ................................ 17
  4.3 Convert Procedural Design to Objects ......................... 18
    4.3.1 Current Design .......................................... 18
    4.3.2 Object-Oriented Alternative ............................ 20
    4.3.3 Transition from Old to New ............................. 21
    4.3.4 Evaluation ............................................... 24

5 Lessons Learned and Future Work 25

Acronyms i
# Contents

List of Figures  
Listings  
Bibliography
1 Introduction

 Legacy code is code from the past. It has a lot of words but is maintained because it works. It is code without tests, code which smells and lacks appropriate structure. It is code that might not be completely understood and which nobody generally wants to maintain and work with.

 But apart from these matters, legacy code is always one thing: valuable. It is valuable because of the knowledge which flew into the code base with every modification since it was crafted, most notable in the form of bug fixes. Rewriting legacy code from scratch would sacrifice this knowledge. Instead, as nicely described by Spolsky [2000], it is advisable to refactor the existing code.

 Goal of this paper is to share some experiences gained during refactoring of a legacy python program. Central contribution is in the presented approach to legacy code. It is based on the observation that refactorings need a context. It uses design patterns and design principles as guidance in the course of continuous refactoring and tries to assist in answering the important questions of where to start and how to proceed.

 Martin Fowler defines refactoring as “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” [Fowler, 1999]. The general topic of reducing software complexity by incrementally improving internal software quality is a research domain referred to as restructuring, with refactoring being the specific case for object oriented systems [Mens and Tourwé, 2004]. The presented approach does not cover any new ground in this realm. Instead, it just tries to assist in setting direction in a program which has not seen any fundamental, structural improvements for years.

 Subject to discussion is the code base of Gajim¹, an Open Source Jabber/XMPP instant messenger client written in Python. Gajim and reasons why it should be improved are presented in the next chapter.

 Describing the general approach to legacy code in chapter 3, chapter 4 then focuses on three concrete, design pattern-driven refactorings. It is shown why those are applied and why they are effective. The paper then concludes with a short retrospective.

 As refactorings are language-dependent [Goldman, 2002], some design conclusions and most of the presented tools in this paper might be Python specific.

¹http://www.gajim.org/
2 The Gajim Project Environment

All experiences which lead to this paper were gathered in the context of the Gajim project.

Gajim is an Open Source instant messenger. It is meant to be a full-featured and easy to use client for the Extensible Messaging and Presence Protocol (XMPP)\(^1\), an open standard for real-time communication.

![A screenshot of the Gajim main window.](image)

Figure 2.1: A screenshot of the Gajim main window.

Gajim is a desktop application based on the GTK+\(^2\) window toolkit. It integrates into the GNOME\(^3\) desktop environment, though it does not depend on it. It is written in Python and runs on BSD, Microsoft Windows and Gnu/Linux.

The project was started in 2004 and as of today consists of roughly 60000 lines of code.

\(^1\)http://xmpp.org/

\(^2\)http://www.gtk.org/

\(^3\)http://www.gnome.org/
2.1 XMPP Basics

XMPP is a generic XML routing protocol, based on a distributed client-server architecture. In essence, it provides a way to send small pieces of XML from one entity to another. Entities are either server side components or clients. The XML messages they pass around are called stanzas.

XMPP entities are identified by so called JabberIDs (JIDs). A JID consists of a domain and a user portion, very similar to an e-mail address (e.g., romeo@montague.com).

Clients label their connection to a server with a resource identifier. Using these resources, it is possible to be simultaneously connected from different clients or locations, all using the same JID. For example, one can be connected with a cell phone (romeo@montague.com/mobile) and a desktop computer (romeo@montague.com/home) at the same time.

![Figure 2.2: The decentralized architecture of the XMPP Network. Clients connect to servers from different domains, which in turn connect to each other.](image)

Figure 2.2 shows the direct federation model as used by XMPP. When Juliet wants to contact Romeo, she sends her messages to the Capulet’s server. The server detects that Romeo is located at a different domain and establishes a connection to montague.com to forward the messages. The Montague’s server is then responsible to deliver it to one of Romeo’s clients.

<table>
<thead>
<tr>
<th>Stanza Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;message /&gt;</code></td>
<td>Fire-and-forget messages as a simple mechanism to push information from one to another.</td>
</tr>
<tr>
<td><code>&lt;presence /&gt;</code></td>
<td>A specialized broadcast, informing about the network availability of entities (online, offline, etc.).</td>
</tr>
<tr>
<td><code>&lt;iq /&gt;</code></td>
<td>A request-response (Info/Query) mechanism similar to what is known from HTTP.</td>
</tr>
</tbody>
</table>

Table 2.1: Different stanza types as defined by RFC 3920

For more information beyond this very simplified introduction, see [Saint-Andre et al., 2009].
2.2 Architecture Overview

Figure 2.3 shows the basic architecture of Gajim using the block notation of the Fundamental Modeling Concepts (FMC)\(^4\).

![Diagram of Gajim architecture](image)

Figure 2.3: FMC Block diagram of the basic fundamental architecture of Gajim.

Servers send and receive messages to and from Gajim. When Gajim receives and detects one of the three stanza types as described in table 2.1, it is the responsibility of the embodied XMPP Library to forward those messages to registered Protocol Handlers. Those analyze the stanzas, classify them and perform or dispatch appropriate actions and events. Event Handlers then act upon the generated events.

Event Handlers control the UI and operate, compared to Protocol Handlers, on a higher abstraction level. For example, if the Filetransfer Protocol Handler detects, by inspecting the received XML, that another peer tries to send a file to the user, it generates and dispatches an INCOMING_FILETRANSFER event. Based on the event information an Event Handlers opens a dialog, asking the user to accept or reject the incoming filetransfer. Alternatively, if the user has configured Gajim not to interrupt him with popup dialogs, then the incoming filetransfer will only be indicated in the Roster\(^5\) window.

\(^4\)http://www.fmc-modeling.org/
\(^5\)XMPP refers to a buddy or contact lists as a roster
The decision of the user to accept or reject the filetransfer is carried out by a responsible Protocol Action on the Connection. The action is merely a method which builds a reply, serializes it to XML and sends it back to the server.

2.3 Need for improvement

Improvements and issues need to be communicable. Unfortunately, it is much more difficult than it seems to effectively communicate why a software design needs to be improved and how this can be practically achieved.

2.3.1 Known issues

The following lists the most obvious design flaws in Gajim. More details, especially why those are issues, follow in chapter 4 along the refactoring cases.

- Gajim has almost no automated tests. Many new features are accompanied by regressions.
- The main module of Gajim defines over 30 global variables. This hampers testability and makes the code harder to understand because dependencies of modules and classes are not visible at the interface level.
- All event handlers are hardcoded in a single class. Though different subsystems are interested in certain events, there is only one handler per event which then has to accumulate all the different responsibilities. Many handlers are therefore difficult to understand and extend.
- XMPP-Accounts are realized as strings, making it impossible to carry responsibility.
- A connection uses multiple inheritance in such a way that all Protocol Handlers and all Protocol Actions are united in a single object. That way (beside potential namespace problems) the design gets to forgiving: It is possible to use methods or attributes which are neither defined in a class nor in a super class. Instead, it is sufficient that those exist in any class the connection inherits from.

2.3.2 An Example

The Personal Eventing Protocol (PEP) is a widely implemented extension of the core XMPP standard. It enables users to easily publish personal information, like their currently played tune or their current activity, to interested subscribers.

While this concept exists to publish to contacts in a user’s roster, it is not yet specified for Multi-User Chats (MUCs), allowing participants in a chat room to publish information to other

---

6 called a XMPP Extension Protocol (XEP)
7 think of XMPP based IRC
participants. Matthew Wild proposed a potential solution and showed the feasibility of his approach in a server and client side proof-of-concept implementation [Wild, 2009]. The client side was implemented in Gajim. Unfortunately this turned out to be more cumbersome than anticipated.

Gajim can show retrieved PEP information in chat windows and in the roster next to a corresponding contact. The information is both shown in form of little icons and within a tooltip.

![Figure 2.4: Tooltip showing an Activity published via PEP.](image)

In order to retrieve PEP information from MUC participants and in order to show the retrieved information in the groupchat windows, a lot of existing functionality needed to be reimplemented. There was no way to reuse the existing implementation without first refactoring it.

Another PEP related, user contributed patch\(^8\) confirmed that reuse in Gajim is not easy enough: To support a new PEP type, enabling Gajim to show the current geographic location of contacts, a lot of existing code was copied and slightly adapted.

Infrequent contributors just tend not to reuse, if reuse is not easy enough.

### 2.4 Motivation

The project leading to this seminar paper is based on a simple idea: Improve Gajim. It focuses on the issues mentioned above and follows three basic insights\(^9\):

1. It is easier to change code and to add new functionality if the code is clean and well factored.
2. It is easier to reorganize code if one does not try to change its functionality simultaneously.
3. One should not refactor for the sake of refactoring.

It is therefore the goal to contribute refactorings to Gajim in order to unlock opportunities for further structural improvements and to make it easier for other developers to contribute new features.

---

\(^8\)http://trac.gajim.org/ticket/1981
3 Context-aware Refactoring

I’ve just spent a year designing and implementing a reasonably complex system. Now I know a lot more than I did a year ago. The system works, but has a lot of warts. Some fundamental architectural decisions were wrong. [...] Am I stuck forever? Am I on the inexorable path to bit rot? Or am I just being a spoiled brat, and I should take my lumps and keep evolving this system, since it meets its customer needs? [Picard, 2003]

Believing that a code base needs to be improved and having committed to actually improve it, one has to decide on where to start and how to proceed.

At a first glance refactoring seems fairly easy and straightforward: Following well described steps to make code better. Some of the steps are even very well automated in an IDE. In a legacy application however, the first few simple refactoring seem to be merely drops in the ocean, not making a substantial contribution to the improvement of the overall code quality.

To consider the context of a refactoring helps to assess and to communicate its value. The refactoring is no longer seen in isolation, but instead fits into a bigger picture. A central element of a context is the aspired, improved situation. If refactoring are meant to define structural improvements, those can be effectively captured with design patterns. With the context in mind one can finally follow the different refactoring activities with confidence.

3.1 Four different Paths

Jim Little exposes different paths for reworking software [Little, 2003]:

1. **Do nothing**: Keep on going as before.
2. **Rewrite**: Decide on a new architecture. Write it and switch over after it is finished.
3. **Evolve**: Determine a new architectural direction and incrementally migrate towards it.
4. **Let evolve**: Make changes and refactor as needed. Let the architecture evolve on its own.

These paths provide the context for techniques like refactoring. Very much like reengineering patterns, they frame a process which starts with the detection of symptoms and ends with the refactoring of code to arrive at a new solution [Demeyer et al., 2008].

Choosing and communicating these paths is essential. To be able to grasp what is going on, all people involved in a project need to be aware of the underlying plan driving a refactoring. For example, it might just not seem worth to extract a block of statements from a method which is over 500 lines long, unless it is clear how this area of code is supposed to look like in a few weeks or months.
3.1.1 Do nothing

This approach is not about reworking. Doing nothing is not just an acknowledgment of the existing problems, but also a commitment to new.

The long-term efforts to maintain and enhance the code base will most probably outweigh the short-term savings. If the code is going to be thrown away soon anyway, this might be a viable approach as it is simple and cheap.

3.1.2 Rewrite

Rewriting an application, component or module means to start from scratch with a new architecture or design in mind. Once it is finished, it is swapped in as a replacement for the existing solution.

Rewriting is often easy and fun, but also involves a lot of risks. Many rewriting attempts are too ambitious and never finished [Spolsky, 2000].

However, this path tends to be feasible for smaller amounts of work (with small being a range from a couple of minutes to a few of days). Joshua Kerievsky gives the advice to perform these reimplement and replace tasks in a test driven manner and refers to them as test-driven refactorings [Kerievsky, 2005].

3.1.3 Evolve

The idea of evolutionary reworking is to gradually move towards a better design or architecture. Similar to the Rewrite approach a new direction is chosen, but this time the switch is meant to be incremental. Code block after code block is ported to the new architecture.

The mantra of this approach is to seek improvement but not perfection. The new architecture should therefore be as simple as possible, as it can always be adapted if needed. Following this path is low risk and effective, but can lead to patchy code using two or more different architectures at the same time. It requires a lot of discipline to stay on course, especially when there are many contributors.

3.1.4 Let evolve

By doing changes and refactoring as needed, a system can evolve on its own.

In this path a guiding architecture is replaced by a process of emergent design, driven by refactorings. Changes are made as needed. This means if code does what it is supposed to do and has no serious bugs, it is not changed. However, when a bug needs to be fixed or a feature added, the code is refactored in order to support this effort.
Compared to the other paths, this one is the less disruptive. It takes time to have a clearly visible improvement, but at one point one might be astonished at how much the code has improved, as Michael Feathers [2004] points out.

A disadvantage of this approach is the lack of guidance. If the team is large each team member has to fully commit to this approach and not to fall back to the much easier path to do nothing instead.

### 3.2 The role of Design Patterns and Design Principles

Design patterns provide a means to describe a program structure at a high abstraction level [Gamma et al., 1995]. They can be used to quickly envision and communicate solutions for design problems without getting lost in details.

Given these qualities, design patterns can serve as building blocks for castles in the sky - visions of how a cleaner, better version of the code might look like - as they are needed by the Rewrite and the Evolve path. This should not be confused with a Big Design Up Front. There is no need for such a vision to be perfect or to be even realized one day. Reworking is a process; most likely the vision will therefore change over time and incorporate the knowledge gained within the course of the ongoing changes.

Furthermore, if a language of choice - in this case Python - supports an idiom\(^1\) natively, there is no need to spell out a pattern explicitly. For example, the Strategy pattern [Gamma et al., 1995] is invisible in languages with first-class functions. However, the underlying ideas of the pattern remain.

Design principles like the Dependency Inversion Principle or the Open-Closed Principle [Martin, 1996, 2000] are the root of design patterns. They are at least if not even more important than design pattern and allow to question a proposed design. In refactoring they help to point out why one solution should be preferred over another.

### 3.3 Refactoring Activities

Though there is no master plan to success, it can be helpful to think of it as a sequence of distinct activities:

1. Identify where the software should be changed and why.
2. Determine suitable refactorings.
3. Provide means to safeguard the planned changes.
4. Make the changes using small iterations.
5. Assess the value of the refactoring.

\(^1\)The most import built-in features of Python are presented in [Gregorio, 2006]. Booth [2003] goes further and looks at a few of the most common design patterns and how they relate to Python.
3.3.1 Find areas of code to improve

Martin Fowler and Kent Beck came up with a list of indicators for bad design which they do call Code Smells [Fowler, 1999]. Kerievsky [2005] and Martin [2008] further complete this list with several additional smells.

These catalogues are a great source of inspiration when deciding which code to improve and which refactorings to apply, but refactoring by abolishing random code smells might not yield fundamental improvements as fast as desirable. Some smells are more severe or profitable than others and should therefore be addressed first.

Stevenson and Pols [2004] give the important hint to always consider the customer. They propose to always ask the users what the problem is and to aim to deliver value to the customer as this clarifies priorities and prevents one from starting on a dead end of the project. Demeyer et al. [2008] coin this idea as Most Valuable First.

The following are more technical ways to find potential areas for improvement:

- Software metrics can be used to quickly identify interesting code paths or subsystems. McCabe’s Cyclomatic Complexity, for example measured with PyMetrics\(^2\), can help to find methods or modules which bear too much responsibility.
- Martin Fowler calls duplicated code the biggest stinker. Fortunately, a great portion of it can be automatically detected with appropriate tools. Clone Digger\(^3\) is especially useful because it operates on abstract syntax trees and can therefore also detect cloned and slightly modified code (e.g., it detects a duplicate pattern even if the variable names are different).
- Version control systems can help to find files which have been subject to many changes in the past and will therefore likely be changed in the future as well. Improvements in those areas might pay off soon. Examples for useful statistics are Changed Lines per File per Year\(^4\) or Commits per File per Year\(^5\).

3.3.2 Decide on the refactorings to perform

Fowler and the other authors have directly linked code smells to refactorings which can help to abolish them. Nevertheless, before one applies such a refactoring it is advisable to analyze it according to its particular purpose and effect in the own, given domain. One must consider the current context and make sure to fix the problems, not symptoms.

Jain [2009] gives an example for this. He argues that duplicate code is a symptom and should be handled as such: "When one tries to eliminate duplicate code, at one level, there is no literal code duplication, but there is conceptual duplication and creating a high order abstraction is an effective way to solve the problem."

Not surprisingly, valuable refactorings require a good portion of domain knowledge

\(^2\)http://pymetrics.sourceforge.net/
\(^3\)http://clonedigger.sourceforge.net/
\(^4\)in Mercurial: \texttt{find src/ -name ".*\.py" \textbf{-print} \textbf{-exec}\ hg\ churn\ \textbf{-f} \texttt{"NY\ Y\ s\ O\ \}\;}
\(^5\)in Mercurial: \texttt{find src/ -name ".*\.py" \textbf{-print} \textbf{-exec}\ hg\ churn\ \textbf{-f} \texttt{"NY\ Y\ s\ c\ O\ \}\;}
3.3.3 Safeguard the planned changes

Refactorings need to preserve behavior by definition. Automated tests therefore play an important role.

Feathers [2004] proposes to look for an inflection point. An inflection point is a point where all changes to a set of classes are visible. If a change is not visible at the inflection point, it is not visible at all. One can think of it as an interface or boundary at which the external behavior can be asserted with tests and under which the internal behavior can be changed.

In addition, Feather proposes to make use of effect sketching to narrow inflection points. The idea is to sketch a diagram tracking the effect of changes to attributes and methods on the rest of the class.

Call and dependency graphs are not that fine grained as effect sketches but can be used in a similar fashion to get a broader understanding of the surrounding code.

- snakefood\(^6\) visualizes Python module dependencies. It makes it easy to find users of modules.
- construct\_call\_graph\(^7\) generate a call graph by statically analyzing python code.
- pycallgraph\(^8\) also constructs call graphs but collects the information dynamically at runtime.

Once found, it is often not easy to cover inflection points with tests. Most legacy code was not written with tests in mind, leading to a Refactor-Test-Dilemma: In order to refactor safely, tests are required. But in order to write these tests, one must refactor the code and break dependencies first.

When performing the initial changes to get tests in place, code analyzers can help by at least detecting the most obvious problems. There are several such tools for python. The most common and useful ones are: the static analyzers pylint\(^9\) and pyflakes\(^10\), pychecker\(^11\) which actually executes code and gathers information at runtime and pyntch\(^12\) which is again a static analyzer but with type checking support.

A useful tool helping to cover inflection points with tests is pythoscope\(^13\), a unittest generator. Given an entry point into the application, it executes the code, gathers information at runtime and derives testcases.

---

\(^6\)http://furius.ca/snakefood/
\(^7\)http://blog.prashanthellina.com/2007/11/14/generating-call-graphs-for-understanding-and-refactoring-python-code
\(^8\)http://pycallgraph.slowchop.com/
\(^9\)http://www.logilab.org/857
\(^10\)http://divmod.org/trac/wiki/DivmodPyflakes
\(^11\)http://pychecker.sourceforge.net/
\(^12\)http://www.unixuser.org/˜euske/python/pyntch/index.html
\(^13\)http://pythoscope.org/
3.3.4 Make the planned changes

Before starting with the refactoring itself, it can be helpful remove extreme ugliness and obvious clones as well to make the code conform to the style guidelines.

Refactorings do not always work as expected. Sometimes something was missed when inspecting the code, making it impossible to apply a planned change. In these cases it is important to be able to away the existing modifications without any effort.

When working with a Distributed Version Control System (DVCS) it can be useful to clone the respective repository a second time locally and to refactor only within the local clone. If one step or a whole series of steps shall be thrown away, then those are simply not pulled back into the main repository which is used to push the changes upstream. Every commit can then document a baby step of the refactoring without disturbing back and forth of backed out changes. Patch management systems like *quilt*\(^\text{14}\) help to implement a similar strategy on top of centralized version control systems.

3.3.5 Assess the value of the refactoring

Refactoring should not be done for its own sake. Analyzing the effect of refactorings helps to keep this in mind. In addition, often many new opportunities for subsequent improvements become visible.

\(^{14}\)http://savannah.nongnu.org/projects/quilt
4 Refactoring Cases

This chapter covers three different refactoring attempts towards design patterns and design principles.

4.1 Decouple Hard-Coded Features with Plugins

A few Gajim developers and many Gajim users desire a plugin system. It shall allow developers to concentrate on core functionality while more specific behavior can be outsourced to user-contributed plugins.

This section is a survey of what needs to be done to introduce a plugin system in Gajim.

An initial implementation was started by Mateusz Biliński in a separate branch\(^1\). As mentioned in the Known Issues in section 2.3.1, the core system relies on several global variables as entrance points for the various subsystems. Unfortunately no concrete plugin API does exist, yet. By now, plugins import the main module defining all these globals and can from there on access the whole system. As such plugins would make further evolution of the core system almost impossible (without regularly breaking the existing plugins), the plugin branch is not about to be publicly available soon.

The current plan however is to start using the plugin system internally. By transforming existing, hard-coded features over to plugins, there are three benefits:

1. By re-implementing real-world features it should become apparent which usecases the plugin API needs to support. The API could grow incrementally with each moved feature.
2. Moving existing features over to plugins gives the chance to refactor those along the transformation.
3. The core system gets leaner with every removed, hard-coded feature. Less code is likely to be easier to deal with in subsequent refactorings.

A closer look at the potential plugins yields two slightly different types of requirements:

1. Add new behavior, using the existing functionality. For example, register a handler so that mail notifications send by the Google Talk servers can be detected and signaled to the user via a popup or a system notification.
2. Alter existing functionality. This contains the entire range from expanding acronyms before those are send, up to transparently translating or de/encrypting messages.

\(^1\)http://hg.gajim.org/gajim/branch/plugin-system
4.1.1 Adding new behavior

Figure 4.1 shows the basic architecture of Gajim as already known from page 4. All components which are to be extended through plugins are dyed dark gray.

Plugins should be able to:

- extend the GUI with enhancements like plugin specific menu items.
- register Protocol Handlers to act upon received XML of certain types and define Protocol Actions so that new XMPP protocol extensions can be implemented.
- register Event Handlers so that plugins can act upon already or newly defined events.

Extending the GUI requires to hook into code construction and composing the UI widgets. It is therefore only a special case of altering existing functionality, covered in the next section.

Registering of custom Protocol Handlers is directly supported by the XMPP Library used in Gajim. Adding Protocol Actions instead requires reworking of existing Gajim infrastructure. This will be explained at the end of section 4.3.
A remaining question is how plugins can contribute *Event Handlers*. Handlers are simple methods on a single class, named *Interface*. Each method is responsible for a single event. The dispatch method of the Interface class simply looks up the responsible handler method and calls it, passing the event as a parameter.

The basic idea is to let the dispatch method call several registered handler methods instead of just one. The required steps to alter the design are pretty much described by Kerievsky [2005] in his refactoring *Replace Hard-Coded Notifications with Observer*.

However, a look into the code reveals that some existing events and handler methods might need to be reworked, so that plugins can work correctly. The changes are not that big, but difficult to cover with tests.

Listing 4.1 shows the handler which is invoked when a *ROSTER_INFO* event is dispatched, signaling that the roster has changed server side. Particular interesting is what happens in the lines 14 and 21: It is not possible to simply trigger a redraw of a contact in the roster. Instead, the contact has to be removed and then re-added because data structures within the roster window depend on the group information stored in a contact instance.

```
def handle_event_roster_info(self, account, array):
    """ Handle the "ROSTER_INFO" event: Roster entry of a given JID changed. """
    jid, name, sub, ask, groups = array
    contacts = gajim.contact_manager.get_contacts(account, jid)
    if contact_removed_us_from_his_roster(sub, ask):
        ...
    else:
        # it is an existing contact that might has changed
        re_place = False
        # If contact has changed (sub, ask or group) update roster
        old_groups = contacts[0].groups
        if contact_has_changed(contacts, sub, ask, groups):
            re_place = True
            self.roster_window.remove_contact(jid, account, force=True)
        for contact in contacts:
            contact.name = name or ''
            contact.sub = sub
            contact.ask = ask
            contact.groups = groups or []
            if re_place:
                self.roster.add_contact(jid, account)
                for group in old_groups:
                    self.roster_window.draw_group(group, account)
        else:
            self.roster_window.draw_contact(jid, account)
```

**Listing 4.1:** *Event Handler* method responsible for the *ROSTER_INFO* event. (This version is slightly modified to be easier to understand.)
4.1.2 Altering existing functionality

Plugins should be able to change existing behavior in some predefined spots. A pattern solving this problem is the Interceptor Pattern [Schmidt et al., 2000].

The pattern consists of merely three different types of components: Dispatchers, Contexts and Interceptors.

The application provides a number of hook/interception points where additional procedures can be called. Interception points are managed by Dispatchers. If a hook is called they delegate to registered Interceptors. The information needed and/or altered by Interceptors are stored in interception point specific Contexts.

For example if a message shall be sent, a context will be built containing the receiver and the message to send. Afterwards the dispatcher is triggered to call all registered interceptors passing them the context as a parameter. The plugin can now alter the message within the context before it is finally send over the wire.

![Diagram](image)

Figure 4.2: FMC Block diagram showing the compositional system structure of the Interceptor pattern.

4.1.3 Further Plans

The previous two sections covered the requirements and the potential implementation of the plugin system. They were meant to serve as a castle in the sky, which can be taken into account when some code needs to be adapted or re-designed. Following the Evolve path, the move towards the new design was meant to be done incrementally, code block after code block.

However, the last weeks have shown that this switch does not just happen on its own: Some features could have been implemented as plugins but were instead hardcoded in the old manner.
The new infrastructure was not yet in place (e.g., there was no ready to use implementation of the Interceptor pattern available) and it was not considered to first provide it and to then continue the actual feature.

This leads to the conclusion that the infrastructure must be provided along the idea, not later on.

4.2 Introduce Null-Object and more

The title of this section is as unclear as the purpose of the refactoring. It was not clear which exact value it should deliver. Not surprisingly the described refactoring did not work out well.

One of the central domain concepts in Jabber chat clients are contacts. They have a JID, a name, a status, a subscription state and other data. Most contact information has its source in the server side contact list (the roster); some other information is client side only.

Contacts in Gajim are of the type Contact and managed by the ContactManager class. Clients of the manager have to deal with Null-values which are returned when no Contact instances for a given JID exist (e.g., a user types in a JID he wants to chat with, but the JID is not in his roster and therefore also not yet known by the ContactManager).

Introducing the Null-Object pattern would not have been too complicated. The difficult part of the refactoring described by Kerievsky [2005] is the intermediate step to replace all repeated checks for a null value with an isNull call on the introduced Null-Object. With Python this becomes easy: It is sufficient to implement the __nonzero__(self) method on the Null-Object which is then used for truth value testing and by the built-in operation bool().

Reading the contact code to prepare the changes revealed that a contact is represented by a simple data class: A class merely consisting of attributes without methods. “Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility”. Falsely inspired by this statement of Martin Fowler, the initial goal to introduce a Null-Object slipped out of sight. It was not considered that a contact really just ought to be a data structure and not a full blown object. Instead, it was from now on believed that both the Contact and ContactManager class needed to undergo a bigger reworking effort.

Inspecting the API of the ContactManager showed that it was not serving a single purpose but had several responsibilities. Following a tactic proposed by Feathers [2004] to start introducing the Single Responsibility Principle (SRP) at the implementation level rather at the interface level, the class was brought under test and splitted into several smaller classes.

Along with the idea that an account should really be an object and not just a primitive string, the design was altered to what is shown in figure 4.3.

Unfortunately, it was not obvious how to move on afterwards. It was hoped that it would be easy to move methods over to the Contact and the newly created Account class, along the way of other refactorings; that somehow this refactoring could be a catalyst for others. Two months

\(^2\) or almost without, in the case of Gajim
and several refactorings later no method was moved over. The issues within the API remained as well. They were all postponed to fix along with other refactorings.

It would have been easier to do this refactoring if it had concrete requirements: E.g. with some responsibility at hand which should really be moved to the Contact class, or a class whose complexity could be greatly reduced with a Null-Object.

### 4.3 Convert Procedural Design to Objects

Section 2.3.2 introduced the PEP feature, allowing people to publish personal events about themselves. As it is expected that more different PEP types will be implemented in Gajim in the future, this section takes a closer look of how this can be a simplified.

#### 4.3.1 Current Design

The interesting part is the backend coding which provides the functionality to send and handle received PEP stanzas. For each PEP type, for example the current mood of a contact, a send and a receive function exist in the pep.py module. The functions are defined as module level function.

```python
1 def send_mood(account, mood, message=' '):
2     if not gajim.connections[account].pep_supported:
3         return
4     item = xmpp.Node('mood', {xmlns: xmpp.NS_MOOD})
5     item.addChild(mood)
6     if message:
7         i = item.addChild('text')
8         i.addData(message)
9     gajim.connections[account].send_pb_publish(' ', xmpp.NS_MOOD, item, '0')
```

Listing 4.2: Protocol Action to publish mood data to a user’s roster.
The send functions build XML nodes and send those over a corresponding connection, available via the connection global in the gajim module.

The receive functions use a similar scheme. They extract data from a received stanza and update the internal status of Gajim via some global instances in the gajim module.

```python
def handle_received_mood(items, account, jid):
    mood = None
    text = None
    # extract mood data from received stanza
    for item in items.getTag('item'):
        child = item.getTag('mood')
        if child:
            ...
    # update internal data structures
    for contact in gajim.contact_manager.get_contacts(account, jid):
        if 'mood' in contact.mood:
            del contact.mood['mood']
        if 'text' in contact.mood:
            del contact.mood['text']
        if mood:
            contact.mood['mood'] = mood
        if text:
            contact.mood['text'] = text
    ...
    # update UI
    if jid == gajim.get_jid_from_account(account):
        gajim.interface.roster_window.draw_account(account)
        gajim.interface.roster_window.draw_mood(jid, account)
        ctrl = gajim.interface.message_window_manager.get_control(jid, account)
        if ctrl:
            ctrl.update_mood()
```

Listing 4.3: Helper function extracting PEP mood data.

The different PEP handler functions are called from a Protocol Handler.

```python
def _pepEventCallback(self, con, msg):
    """ Called when we receive a <message/> with one or more pep events. """
    jid = helpers.get_full_jid_from_msg(msg)
    event = msg.getTag('event')
    # XEP-0107: User Mood
    items = event.getTag('items', {'node': common.xmpp.NS_MOOD})
    if items: pep.handle_received_mood(items, self.name, jid)
    # XEP-0118: User Tune
    items = event.getTag('items', {'node': common.xmpp.NS_TUNE})
    if items: pep.handle_received_tune(items, self.name, jid)
    ...
```

Listing 4.4: Protocol Handler delegating the extraction work to the different PEP specific helper functions.
The most obvious issues in the code are:

- The method `_pepEventCallback` needs to be adapted whenever a new PEP type is implemented.
- All different send and handler functions follow a similar pattern of how the data is extracted and propagated. Nevertheless, the different functions do not share code or abstractions.
- There are no abstractions that a client of the `pep.py` module can rely on. Especially lines 16 to 19 of listing 4.3 show that the UI must be pretty much aware which PEP types exist and how their information can be extracted and shown\(^3\) on the UI.

### 4.3.2 Object-Oriented Alternative

The proposed solution is not a sample of great object-oriented design.

The idea is to introduce a common supertype (see listing 4.5) and a subclass for each specific PEP type. It tries to avoid all issues as mentioned previously, at once. Though uniting the different responsibilities to extract information from stanzas and to render it in text or icon form in one class violates the SRP, the solution is still preferred. It seems good enough and does not add any yet unneeded complexity. The classes can be easily splitted later on, if needed.

Procedural code makes it hard to add new data structures as all functions must be changed to correctly handle a new type. In return, it is easy to add new functions operating on the existing data structures. Classes are the other way around. They make it complicated to add new methods, because those must be implemented in all existing types. However, it is easy to derive new types.

Considered that new PEP types will be added and not new different ways in which the existing types shall be represented, the switch to an object-oriented design seems valuable.

```python
class AbstractPEP(object):
    type = ''  # to be overwritten by subclasses
    namespace = ''  # to be overwritten by subclasses

    @classmethod
    def get_tag_as_PEP(cls, jid, account, event_tag):
        
        """
        Returns an instance of a PEP class which is encapsulating the information
        in the given XML event tag. Returns None if the event tag does not
        contain information of this PEP type. """
        
        items = event_tag.getTag('items', {'node': cls.namespace})
        if items:
            log.debug("Received PEP \'%s\' from \%s\" % (cls.type, jid))
            return cls(jid, account, items)
        else:
            return None
```

\(^3\)The UI needs to construct a text suitable for tooltips and to determine a suitable icon to show. The code to do this is duplicated and implemented in several different places: chat window, roster window and tooltip classes.\)
def __init__(self, jid, account, items):
    self._pep_specific_data, self._retracted = self._extract_info(items)
    self._update_contacts(jid, account)
    if jid == gajim.get_jid_from_account(account):
        self._update_account(account)

def _extract_info(self, items):
    """To be implemented by subclasses""
    raise NotImplementedError

def _update_contacts(self, jid, account):
    for contact in gajim.contact_manager.get_contacts(account, jid):
        if self._retracted:
            if self.type in contact.pep:
                del contact.pep[self.type]
        else:
            contact.pep[self.type] = self

def _update_account(self, account):
    ...

def asPixbufIcon(self):
    """To be implemented by subclasses""
    raise NotImplementedError

def asMarkupText(self):
    """To be implemented by subclasses""
    raise NotImplementedError

Listing 4.5: Proposed abstract PEP baseclass.

4.3.3 Transition from Old to New

No tests exist to safeguard the transition to the solution as listed above. It is also not intended to
write some, because those would come with the difficulty to correctly initialize and assert global
variables and the state of the UI. It will be easier to write tests afterwards.

The refactoring shall instead be backed by a local XMPP server and a test client configured to
receive a bunch of PEP stanzas at startup. Only doing small steps and running the test client
often should\footnote{The plan worked out just fine. Nevertheless it is still considered risky to refactor without appropriate test coverage.} be sufficient to half-automatically assure nothing broke.

While preserving interfaces as far as possible, the different methods for sending and handling
PEP data are moved over to a newly created ConnectionPEP class. The class itself is then inte-
grated into the existing organization of connection classes as shown in figure 4.4. Arguable this
design is not perfect, nevertheless it leverages consistency within the code base and is therefore
the next logical step.
Moving the methods is easy. All methods of `ConnectionPEP` are exposed on a globally accessible object and itself still refer to global variables.

Following the receive path of incoming PEP data, it is the next step to remove the hard coded coupling between the UI and the `ConnectionPEP` class (see listing 4.3 on page 19; lines 20 up to line 26): The connection must no longer update the UI directly. Instead, it should generate an event (`PEP_RECEIVED`) which can then be handled at the GUI level.

A common peculiarity of the connection classes in figure 4.4 is that the superclasses rely on functionality provided by the subclasses. They rely on the fact that due to multiple inheritance and duck typing some methods and attributes which are only defined in subclasses (and even other classes which the `Connection` inherits from) will be available at runtime. While this is not only difficult to understand for humans, also static analyzers fail to provide any useful output when run on these classes, due to too many ‘false’ positives.
A first step to approach this problem is to make all dependencies explicit by directly referencing the declaring class, or more specific, the implementing object. Figure 4.5 shows the general idea. Once this is done for all connection classes it becomes possible to break the class hierarchy apart. (Changing the hierarchy goes beyond the scope of this refactoring. It is a requirement of the plugin system to allow plugin developers to write own Protocol Handlers and Actions. It is important that those can be added at runtime; they must not be hardcoded.)

Due to duck typing it is not needed to define a specific interface for each reference. It would be possible to wrap and to restrict access to the wrapped class, but this additional complexity is generally not needed. The move is mainly motivated by the desire to be able to inject mock and stubs for testing reasons and to make static analyzers work better. Both goals can be achieved without wrappers.

With all dependencies being explicit, static analyzers can be used as another way to check for errors. This gives enough confidence to finally create subclasses for the proposed AbstractPEP class as introduced in listing 4.5.

```python
class ConnectionPEP(object):
    def __init__(self, account, dispatcher, base_connection, pubsub_connection):
        self._account = account
        self._event_dispatcher = dispatcher
        self._base_connection = base_connection
        self._pubsub_connection = pubsub_connection

    def _pepEventCallback(self, con, msg):
        """ Called when we receive a <message/> with one or more pep events. """
        jid = helpers.get_full_jid_from_msg(msg)
        event = msg.getTag('event')
        for pep_class in SUPPORTED_PERSONAL_USER_EVENTS:
            pep = pep_class.getTagAsPEP(jid, self._account, event_tag)
            if pep:
                self._event_dispatcher.dispatch('PEP_RECEIVED', (jid, pep))

    def send_mood(self, mood, message=None):
        if not self._base_connection.pep_supported:
            return
        item = xmpp.Node('mood', {xmlns: xmpp.NS_MOOD})
        item.addChild(mood)
        if message:
            i = item.addChild('text')
            i.addData(message)
            self._pubsub_connection.send_pb_publish('', xmpp.NS_MOOD, item, '0')
```

Listing 4.6: The final ConnectionPEP class with explicit dependencies. The _pepEventCallback is the PEP Protocol Handler; it creates the different PEP types. The send functions serve as Protocol Actions.
4.3.4 Evaluation

The presented refactoring changed the entire way how incoming PEP data is processed and stored. The diffstat of the pep related changes reads like 14 files changed, 523 insertions(+), 956 deletions(-). The effort to implement the new Geolocation PEP type also went down from 214 insertions in 5 files down to 76 insertions in 3 files.

There is still room for further improvements. In addition, the new code is not yet covered with automated unit tests. However, it is by now considered as good enough. Additional changes can follow if the need for it becomes apparent.
5 Lessons Learned and Future Work

This paper shared some experiences gained during refactoring of a legacy python program named Gajim.

The paper focused on three more or less successful refactorings meant to improve code quality and maintainability. Central contribution was the presented approach to legacy code. It distilled the experiences and the observation that refactorings can seem to be merely drops in the ocean, not making a substantial contribution to the improvement of the overall code quality, towards the idea that the context and the motivation behind refactorings matter.

The whole paper so far mostly concentrated on technical issues of reworking. A great lesson however is that the people involved matter a lot.

Maintainability of code is arguable a matter of subjectivity. Most find it far easier to maintain something they have written on their own, than to maintain someone else’s code. When starting a reworking or refactoring effort it is therefore important to make sure all developers share the same principles and a common goal. In an extreme case there are two kinds of developers involved: The original authors who do not see why there is a problem with their existing solution, and the others who want to refactor for some vague aesthetic reasons.

Program understanding constitutes the largest part of actually changing a program [Van Deursen et al., 2001]. The original authors might therefore not be keen on a new solution which is only slightly better, because they have to invest time to understand it.

Dubakov [2009] states that it becomes more and more difficult for new developers, who do not know all the fundamental decisions and original ideas behind an architecture, to master the code with confidence. Knowledge is lost when developers leave. But also fast growing teams are problematic. Less communication yields bad decisions and code duplication. If the code is too complicated, hacks are introduced to make something work without any deep understanding of the underlying conditions.

If the design is clear and simple and has less broken windows, it is easier to stay on course. Believing that one can encourage and enable other developers to adhere to good principles and patterns by setting a good example, the next required actions are clear.

There is still a lot of room for improvement in Gajim. However, the next actions must be to fix some flaws introduced by the refactorings themselves: E.g., on page 20 it was mentioned that the introduced AbstractPEP class can be splitted later on. In fact it must be splitted, because it violates a principle in Gajim not to have a dependency on GTK in certain packages.
Acronyms

**DVCS** Distributed Version Control System

**FMC** Fundamental Modeling Concepts

**GSOC** Google Summer of Code

**JID** JabberID

**MUC** Multi-User Chat

**PEP** Personal Eventing Protocol

**SRP** Single Responsibility Principle

**XEP** XMPP Extension Protocol

**XMPP** Extensible Messaging and Presence Protocol
List of Figures

2.1 A screenshot of the Gajim main window. ................................................. 2
2.2 The decentralized architecture of the XMPP Network. Clients connect to servers from different domains, which in turn connect to each other. ......................... 3
2.3 FMC Block diagram of the basic fundamental architecture of Gajim. ............. 4
2.4 Tooltip showing an Activity published via PEP. ....................................... 6

4.1 FMC Block diagram of Gajim as known from figure 2.3. All components to be extended by plugins are dyed dark gray. ................................................. 14
4.2 FMC Block diagram showing the compositional system structure of the Interceptor pattern. ................................................................. 16
4.3 Class diagram depicting the design of contacts and how they are managed. .... 18
4.4 Design sketch of the connection classes. .................................................. 22
4.5 Design sketch of the connection classes. Dependencies made explicit. .......... 22
## Listings

4.1 *Event Handler* method responsible for the `ROSTER_INFO` event. (This version is slightly modified to be easier to understand.) ......................................... 15

4.2 *Protocol Action* to publish mood data to a user’s roster. ................................. 18

4.3 Helper function extracting PEP mood data. ...................................................... 19

4.4 *Protocol Handler* delegating the extraction work to the different PEP specific helper functions. ................................................................. 19

4.5 Proposed abstract PEP baseclass. ................................................................. 20

4.6 The final `ConnectionPEP` class with explicit dependencies. The `_pepEventCallback` is the PEP *Protocol Handler*; it creates the different PEP types. The send functions serve as *Protocol Actions*. .................................................. 23
Bibliography


