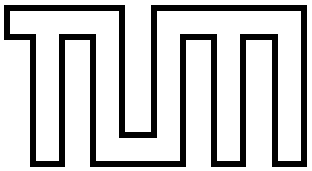


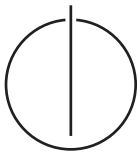
FAKULTÄT FÜR INFORMATIK

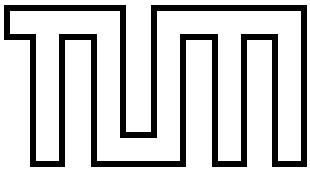
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Myriad – a mailmerge tool for massive parallel, yet individual email conversations

Ludwig Schubert





FAKULTÄT FÜR INFORMATIK

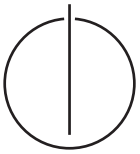
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Myriad – a mailmerge tool for massive parallel, yet individual email conversations

Myriad – ein Serienbrief Email-Tool für hochgradig parallele, jedoch individualisierte Emailkonversationen

Author:	Ludwig Schubert
Supervisor:	Prof. Dr. Johann Schlichter
Advisor:	Dr. Wolfgang Wörndl
Date:	September 30, 2013



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 30. September 2013 Ludwig Schubert

Abstract

This thesis introduces the myriad system for email mass communication.

Despite its age, Email has remained the prevalent form of electronic communication, it's usage being wildly different from how it was imagined. The tools to handle it, however, are still stuck in their original UI metaphors.

Myriad aims at producing personalized communication on a comparable level to manually composed messages, while reducing user effort. A cross-over of mailmerge and customer support/helpdesk software, it enables managing big volumes of bidirectional email-based communication. It is based on a self-developed framework for separating information extraction, decision making, and personalization steps in communication.

The myriad system consists of a server component that handles interfacing with email servers, a core logic system and a web frontend for users.

It can be tested at `http://myriad.ludwigschubert.de`.

Keywords

Email, Workflow, Helpdesk, Mailmerge, Crowdsourcing, Personalized Communication, Assisted Templating, Generated Documents, Web-Application

Acknowledgements

This thesis is based on research conducted from October 2012 through April 2013 together with Christian Ikas, Barış Öztop and Nicolas Kokkalis under the guidance of *Prof. Michael S. Bernstein* and *Prof. Scott R. Klemmer* during a stay at Stanford University's Human Computer Interaction Department.

The research stay was partly financed by Elitenetzwerk Bayern through a grant to the study program *Technology Management* at the Center for Digital Technology and Management.

Stanford
University

Elitenetzwerk
Bayern

Contents

Main Matter

1. Introduction

This chapter gives an overview of the work, arguments for the relevance of new Email tools, as well as the goals pursued by this thesis. Concluding, the structure of the work is explained.

1.1. Motivation

The total number of worldwide email accounts – 3.3 billion as of 2012 [1] – indicates how email still is the primary means of asynchronous online communication, despite continued efforts from corporations for successor technologies.
While specialized tools, e.g. Facebook's Events or Doodle, can provide a better experience, email is still the ubiquitous fall-back medium that everybody can access. It's non-centralized approach, relying only on deeply embedded infrastructure like the DNS system, has allowed email to become bigger than any proprietary tool.

Thus it is unsurprising that email is used a lot as an organizational tool instead of the mere letter-like two person ocmmunication medium that it was set out to be. In fact, use of email as an organizational tool is primed to become even more prominent, with most of the growth and traffic in email usage coming from the corporate world. [1, p. 3] A person might use email to figure out a good time for a meeting, soliciting comments on his work, or even just organizing a party.

In all of the abovementioned scenarios, the user has to make a trade-off between effort and personalization – writing individual emails versus sending out a mass mail.

Now, there are lots of benefits of personalized emails. A higher response rate is one of the basic ones. Some scenarios require personalized information as part of the email in order to be effective, e.g. a grade report. Sometimes a personally addressed email is just

a lot more likely to produce the desired result [2, p. 1375, 1380], as it increases perceived social pressure.

In observing one instance of such a usage of email, a recruitment campaign for a scientific study, we observed a lot of duplicated efforts, inconsistent communication and overall potential for tool assistance.[1] We set out to develop a mailmerge system that didn't require complicated desktop software, yet was more powerful and workflow oriented than previously available mailmerge software.[2]

1.2. Goals and Background

We developed a web-based email client with mailmerging capabilities for massive parallel, yet personalized email conversations, Myriad. The focus of Myriad lies not on traditional, individual email based replies. Instead, it aims to provide tools that allow managing big amounts of similar emails efficiently. To fullfill this goal, Myriad provides a high level organizational abstraction called `Campaigns`, a visualization of individual conversation state and actionability, filtering and actions on groups of results, easy reusability and automation of replies (`Messages`), as well as support for delegation to assistants and generally usable integration into existing email and inforation management infrastructure. Those simple, combinable features are supposed to enable users' email workflows to be peersonalized and well-organized, while reducing manual or duplicated effort. To define a simple to understand goal for ourselves we created Figure 1.1, which plots user effort against personalization for an email campaign. The lower, left-hand corner is exemplified by a single mass mail: barely any personalization at minimal effort. The upper, right-hand corner shows writing every email by hand: maximum personalization, but at a high effort. Myriad's goal is to end up below the drawn line; achieving the desired degree of personalization at sublinear effort.

This thesis is based on research I conducted together with Christian Ikas, Barış Öztop and Nicolas Kokkalis. To avoid overlap between the researchers, I will focus on the system design process, architecture and implementation details, for which I was mainly responsible.

For a more in-depth look at the motivation for the project, the comparison with existing

[1] See section 4.1.
[2] See section 3.1 for a more in-depth comparison with pre-existing systems.

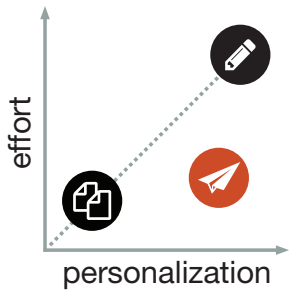


Figure 1.1.: Myriad's goal is to achieve a desired degree of personalization at sublinear effort.

systems and the relevenace for survey research, see Barış Oztop's master thesis.
For a more business-process focussed look at email workflow tools, see Christian Ikas' Master thesis.

1.3. Outline

This chapter gives an overview of the work, and arguments for the relevance of new Email tools, as well as the goals pursued by this thesis. Concluding, the structure of the work is explained.

In **chapter 2**, **Technical Backgrounds**, relevant technical background information is provided. The decision to build a web app is motivated. Standard-compliant email systems are explained and relevant standards are introduced. Lastly, an introduction to workflow systems and historic examples is given.

In **chapter 3**, **Comparison with similar systems**, two major categories of software – dedicated mailmerge systems, and customer support systems – are introduced, which overlap in functionality with Myriad.

In **chapter 4**, **Concept**, the architecture of the proposed system is motivated and its development process is highlighted.

In **chapter 5**, **Implementation**, is concerned with technical details of how Myriad was implemented. The collaborative development process is described, as are actual system component and their functionality.

In **chapter 6**, **Evaluation**, Myriad's initial goals will be reiterated and contrasted with real world observed usage.

In **chapter 7**, **Conclusion**, the relevance of this system and the proposed framework is discussed; future work is outlined and possible directions proposed.

This is followed by a **Bibliography** of cited works and a **List of Figures**.

The **Appendix** contains selected source code examples, the development of the system design over time in UML Diagrams, and a **Colophon**.

2. Technical Backgrounds

In this chapter, relevant technical background information is provided. The decision to build a web app is motivated. Standard-compliant email systems are explained and relevant standards are introduced. Lastly, an introduction to workflow systems and historic examples is given.

2.1. Web Applications

2.1.1. Why build a web application instead of a native one?

2.1.2. Main Components of a Web Application

2.1.3. Rails and Ruby

2.2. Email Systems

2.2.1. Working with RFC 2822

2.2.2. IMAP and SMTP

2.3. Workflow Systems

2.3.1. History of Workflow Systems

2.3.2. Famous Examples

3. Comparison with similar systems

In this chapter two major categories of software – dedicated mailmerge systems, and customer support systems – are introduced, which overlap in functionality with Myriad.

3.1. Mailmerge Systems

3.1.1. CRM Systems

3.1.2. Dedicated Mailmerge Systems

3.1.3. Backend Services

Amazon SES

SendGrid

3.2. Customer Support Systems

4. Concept

In this chapter the architecture of the proposed system is motivated and its development process is highlighted.

4.1. Functional Analysis

4.1.1. The Myriad Model of ABMC

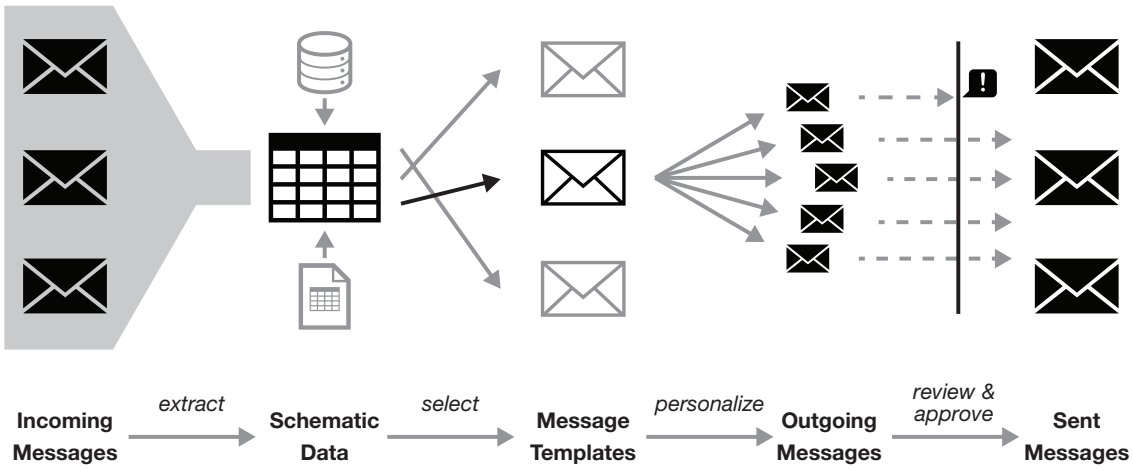


Figure 4.1.: The Myriad Model decomposes ABMC into distinct substeps to allocate them to different actors.

4.2. Product Functions

4.3. User Interface

4.3.1. Prototyping Approaches

4.4. Technical Analysis

4.4.1. Runtime Environment

4.4.2. Server Software Stack

4.4.3. Client Side

4.4.4. Backend Service Connections

Google Mail

Google Docs

4.5. System Design

4.5.1. Database Schema

Unusual Patterns

4.5.2. Distribution of System Components

5. Implementation

This chapter is concerned with technical details of how Myriad was implemented. The collaborative development process is described, as are actual system component and their functionality.

5.1. Preparation and Tools

5.1.1. Development Environment

5.1.2. Collaborative Development

`git`, `gitflow` and Github

5.1.3. Deployment

Deployment Tool `Capistrano`

5.2. Server Component

5.2.1. Workers and their Jobs

5.2.2. Maintenance and Rake Tasks

5.3. Core Classes

Myriad is built around the interaction of 5 model level concepts, or classes, and a set of assisting subsystems. `Campaigns`, which have a set of `Email`s and a bunch of `Contact`s as recipients. The relation between the campaign and the contacts is called `Conversation`; the relation between campaign and the emails is called `Template`.

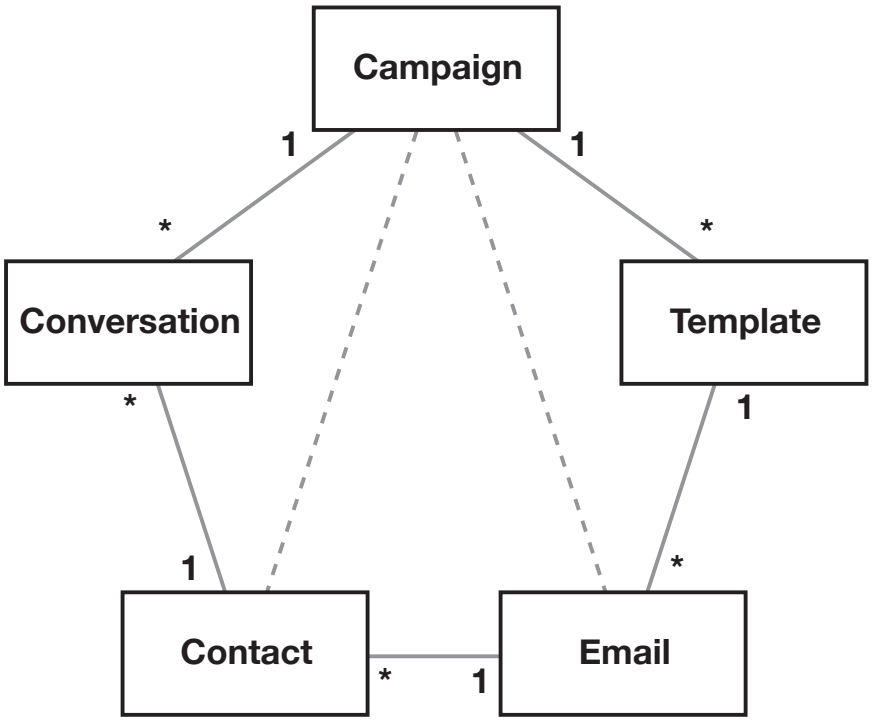


Figure 5.1.: Myriad is built around the interaction of 5 model level classes.

5.3.1. Contact

Contacts are the names and email addresses of the people you want to email. They are created when importing from spreadsheets, when typing an email address into a To: field, or when importing existing emails by adding a Gmail label to them. Contacts belong to users. This means that there is only one fred@gmail.com for each user and this fred@gmail.com may be linked to multiple campaigns of that user.

5.3.2. Template

Templates are prototypes of emails. They consist of body text that can contain placeholders, a subject, actions that are triggered when they are sent, and rules that can send them automatically.

5.3.3. Email

Emails are instantiations of Templates. Their body doesn't contain placeholders anymore, but only the merged email body. They have a delivery status, for example, and keep lots of information on their external representations, like a `message_id`, UIDs for IMAP folders, thread IDs (THRIDs) for Gmail, etc.

FetchedEmail

`FetchedEmail` represents Emails that were fetched as raw text from an IMAP server. `FetchedEmail`s encapsulate their respective `RawMail`s.

IncomingFetchedEmail

`IncomingFetchedEmail` represents an email from someone else than the user that was fetched from an IMAP server. E.g.: responses from recipients.

OutgoingFetchedEmail

`OutgoingFetchedEmail` represents an email the user wrote themselves, but that was fetched from an IMAP server. E.g.: Emails the user wrote in their personal email client, but still in response to recipients of a current campaign.

CreatedEmail

`CreatedEmail` represents Emails that were created by Myriad. They don't have a Rawmail, but `CreatedEmail` provides functionality for serializing to a raw text representation.

OutgoingCreatedEmail

`OutgoingCreatedEmail` represents an email the user wrote as a message template and was later generated by Myriad.

IncomingCreatedEmail

This class is not implemented yet. While, at first sight, it might seem like an *incoming* email could never be *created* by the system, this could be used for a multitude of plausible scenarios, such as sending persistent messages to the user (instead of the more ephemeral on-screen notifications), importing emails from a non `IMAP` compatible email server, or even integrating a chat system, making incoming chat messages appear as emails to the system.

5.3.4. Campaign

Campaigns are a collection of templates and meta information like a connected Google Drive spreadsheet. They additionally contain a list of Keys. You can think of keys as column headers in a spreadsheet.

5.3.5. Conversation

Conversations are like email threads in gmail, but scoped to a specific campaign and contact. They contain all the emails a contact wrote or received within a campaign.

5.3.6. Key & Value

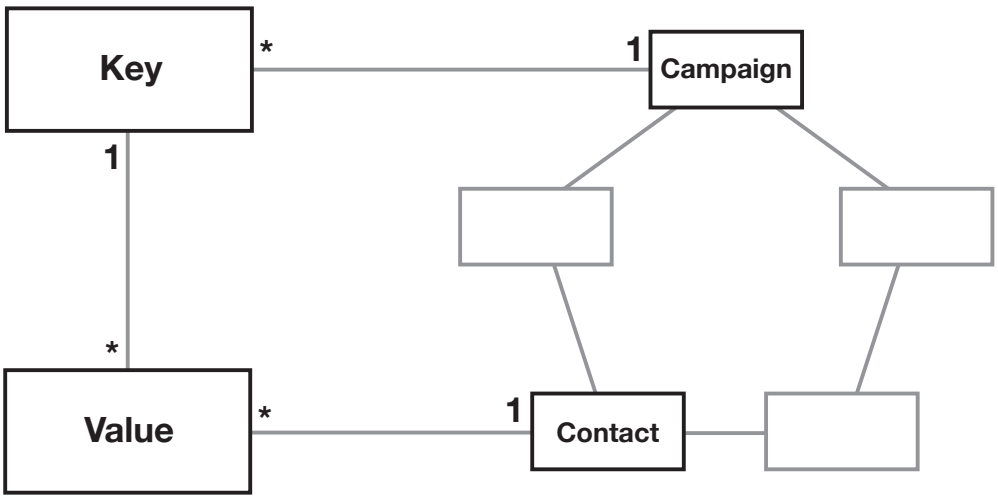


Figure 5.2.: Key and Value together allow for storing arbitrary information about a contact in a campaign.

One goal of a campaign can be the collection of information. For that, users create a data schema made up of Keys, that they (or somebody they share the campaign with) fill with Values. The Keys consist of a name – the header of the spreadsheet column – while values have a content – the content of a spreadsheet cell in the row of the contact they belong to.

5.3.7. Rules & KeyBinding

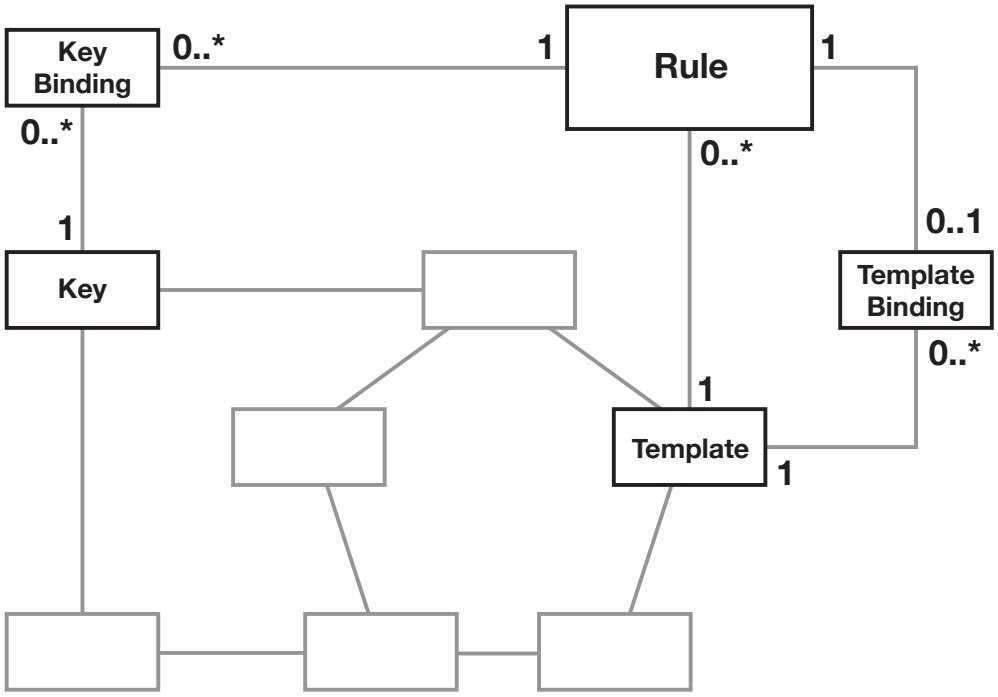


Figure 5.3.: Rules rely on different “bindings” to define their matching criteria.

Searches allow users to constrain the set of all conversations to a subset of them by defining criteria those conversations match. Each search consists of zero or more KeyBindings, zero or more TemplateBindings, as well as an optional conversation status constraint. A KeyBinding is a value that a key must have to satisfy a specific search. Similarly, a TemplateBinding tb of search s specifies a Template t such that a conversation c matches s if c contains an email e, and e is an instantiation of t. For example, a search might specify that a conversation should be unread to be part of the search’s results. But we also want to allow users to specify constraints on their own Keys, e.g. coming? = “yes”. For that, we needed a flexible way to specify those constraints. So we came up with the concept of a KeyBinding; it “binds the free Key variable to a specific value”. So a KeyBinding associates a search with a Key that contains a specific value (todo: add comparison operators. e.g. age ¿ 21). Note that Searches are saved to the database. If they specify a template to be sent, we consider them to be “Rules” that can be automatically triggered. Otherwise they just clutter up the database and should eventually be purged by a cronjob. This persisting of searches could be used to remember recent searches.

5.3.8. ValueSettingActions

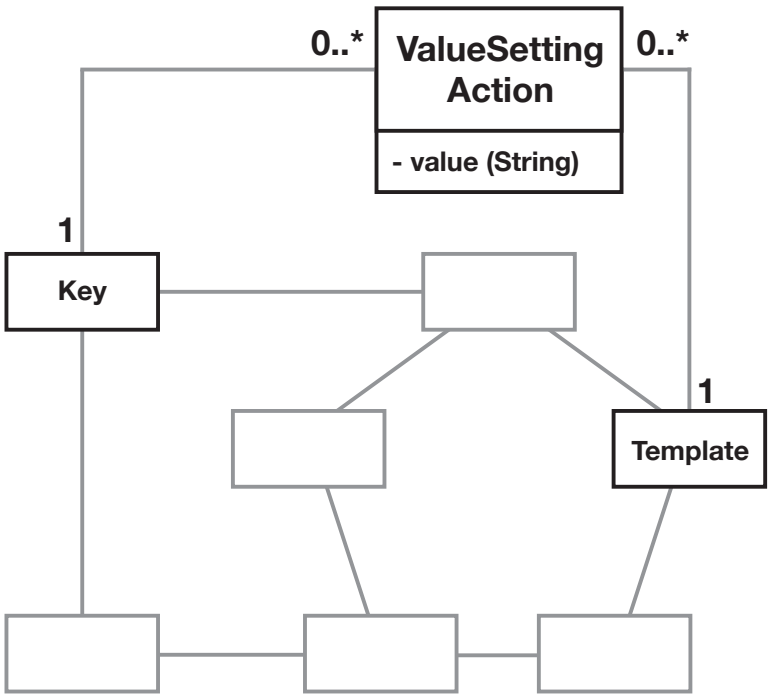


Figure 5.4.: ValueSettingActions allow a template to set a certain Key's value to a predefined value. Together with Rules, this allows for custom, state-machine-like behavior specification.

5.3.9. Notification

Notifications are used to "notify" but not only. They also hold state (e.g. invalid user credentials) that can be used only internally by the system (e.g. to avoid syncing emails from users with invalid credentials). They are like "state" descriptors of other objects and in some cases these are visible to the users (e.g. we are syncing your spreadsheet), in other cases only the backend uses them. So maybe they should be called differently. (any ideas?)

They're an abstract class that we might have taken a little too far. They basically contain a polymorphic Type field, and an ID. Then, madness ensues as we use this information to

attach Notifications to any ActiveRecord::Base object in our database. There is a whole class hierarchy beneath Notification, containing classes like ErrorNotification, which uses aforementioned feature to attach Errors or warnings to basically anything. There are also very specific subclasses like TemplateSentToSearchResultsActivityNotification, which are used to notify users. Another example would be a SpreadsheetSyncingNotification, which is also used to inform users. You can define groups of Notification Subclasses that are displayed in different parts of the UI, for example on the Admin Dashboard or on top of the screen for the respective user, as a kind of persistent flash message.

ActivityNotification

These are used to display the activity stream, which is currently woefully underdeveloped. They could also be used to record Assistant actions in the future. Maybe they could even be tied to resources to enable tracking of the last spreadsheet sync, etc.

UnexpectedStateNotification

We used these to make sure that all of those little instances where you're writing a case statement without a default, expecting values to be of a certain type, etc *actually* never occur. And in reality, of course, they do. So if you're almost certain a certain state should not be reached in a program, put an UnexpectedStateNotification there and be greatful when, two months later, your admin dashboard informs you that what you had deemed impossible happened 6,000 times last night.

ExceptionNotification

A simple way to "save" an exception for review from the Admin Dashboard. (Todo: Also save the stacktrace with these, so they might be even more useful.)

ErrorNotification

These might be named a little badly, since they can actually mean anything that is attached to a certain resource and might be resolved later on. For example, we could use these to implement an InvalidCredentialsErrorNotification. When, at a later point, the credentials work again, we'll be able to call InvalidCredentialsErrorNotification.resolve and pass it the user object, deleting the InvalidCredentialsErrorNotification. An example where these are used to notify not about an error, but simply about a state that's attached to a resource is the SpreadsheetSyncingNotification. A possibly better future name might be PersistentNotification or ResourceNotification.

5.4. Backend Services Connection

5.4.1. Email Fetching

As we spent a lot of time figuring out the intricacies of the Gmail IMAP implementation and the manifold cases that can occur when normal (or heavy) users go about their daily email business without considering the restrictions of poor Myriad, we want to spare you repeating the same. We hope to have brought the fetching and email creation logic to a relatively stable state, yet we are certain there remain a bunch of holes that could cause failures in edge cases. If you do not need to touch the code, you might find the following explanation too detailed, so you shouldn't waste your time with it (bar this overview section). For anybody working with the fetching code, however, the following information should prove very useful - especially if you are unfamiliar with email headers, IMAP protocol details and the like. In brief, our fetching process works as follows: 1. Identifying emails on the Gmail server: First, we connect to the user's Gmail account and identify the UIDs (one of many types of email identifiers, see below for a more detailed explanation) of all emails we want to fetch. We use multiple different queries for that, which will later be explained in more detail. Before we move on to the next step, we filter out emails in our database whose UIDs we already know, i.e. have fetched before. 2. Fetching and processing email data: For the remaining UIDs, we fetch the corresponding email from Gmail, create a Mail object (from the mail gem) for it and determine its type – IncomingFetchedEmail, OutgoingFetchedEmail, etc. 3. Email creation in Myriad: As a last (but not trivial) step we try to use the information from the Mail object to create an Email object in Myriad.

IMAP IDLEing

5.4.2. Google Docs Syncing

5.5. “Best Practices”

During the development process we discovered a set of “Best Practices” that didn’t seem to fit into any specific software development methodology.

5.5.1. A fixed set of ruby core extensions

A dynamic language like ruby makes it trivially easy to reopen core classes. [citation needed] While initially the convenience seems awesome, it’s easy to lose track of already implemented extensions, you risk colliding method names[1], and reusability is higher if one just packages those extensions as a new class.

However we did decide to keep a couple of our extensions, which we felt were essential. Here’s a list of them, together with explanations why they seem like good ideas:

String

remove_all_whitespace A common task when interfacing with APIs, especially when interfacing with IMAP where one can’t use a proper parser. This came up multiple times during development, and a commonly suggested solution is to use regular expressions, like so: `gsub /\s+/, ''`. While working perfectly fine, we wanted a more expressive name. This was left as an extension because similar whitespace altering functions like `.strip` or `.squish` are also declared directly on `String`.

```
1  "  This is a string  !   ".remove_all_whitespace
2   => "Thisisastring!"
```

[1] Ruby 2.0 finally fixed this by introducing *refinements*, a way to extend classes namespaced to the current module.

similar_to? A custom equivalence class on `String`s, which are similar if one only considers numbers and downcased letters. This is used to match placeholders with spreadsheet column headers, for example. In our experience users often used “First Name” instead of “first_name” and we wanted to offer an equivalence class that was sufficiently big, while avoiding being totally fuzzy.

```
1  "first_name".similar_to? "First Name"
2   => true
```

ellipsisize Takes a string and information on how how much of it to keep. It adds an ellipsis where the string was cut off. We used it in many places throughout the interface, for example for only displaying the first line of a collapsed email.

```
1  "Supercalifragilisticexpialidocious".ellipsisize
2  => "Sup...ous"
```

to_bool Once again an indication of working with badly specified protocols. This was needed after some Javascript libraries apparently disagree about how to encode `true` in a URL parameter.

```
1  ['true', 't', 'yes', 'y', '1'].map(&:to_bool).all?
2  => true
3  ['false', 'f', 'no', 'n', '0'].map(&:to_bool).none?
4  => true
```

Hash

deep_find This does a deep traversal of a Hash, looking for any key that matches the argument of this method. When working with APIs that return structured data the format is usually fixed, but we experienced slight inconsistencies for Google’s different versions of its OAuth APIs. Of course this method will run into trouble if a key occurs multiple times, but when used in the right places, it makes parsing structured responses incredibly solid.

```
1  nested_hash
```

```
2   => {
3        "a key"=>"a value",
4        "a nested hash" =>
5        {
6          "another key"=>"another value",
7          "the key"=>"the value"
8        }
9       }
10
11  simple_hash
12  => {
13       "another key"=>"another value",
14       "the key"=>"the value"}
15      }
16
17  simple_hash.deep_find "the key"
18  => "the value"
19
20  nested_hash.deep_find "the key"
21  => "the value"
```

Array

uniq? Checks whether all elements inside the receiving array are unique. This was not used to abuse `Array`s as `Set`s, but rather to check user input.

contains_duplicates? Semantic alias of `not uniq?`, this was the method that was actually used in code.

```
1   ['word', 'thing', 'word'].contains_duplicates?
2   => true
```

5.5.2. Lean Workers with AbstractWorker

We've built an AbstractWorker Superclass for our Resque Workers. Why? Basically I think Resque is awesome, but very, very basic. When faced with the task to build a DRY function to start worker threads for all our workers, I found no simple way. So now, AbstractWorker

takes care of generating queue names form the Class names of it’s subclasses, and I can use the result of the descendants method (returns all subclasses) to iterate over and start a worker thread with the correct queue. It also allowed us to extract the ResqueDirector plugin, which starts those workers in our Development environment.

5.5.3. Monitoring Services

Monit is an awesome tool that you configure via a DSL that ensures all parts of your appliction keep running. The most common way for monit to monitor an application is that this application writes its Unix process ID into a specific file. This way monit can check if this process is still alive. Big applications like nginx or mysql already do this for us. For things like workers or the above mentioned rake idle task we need a custom solution. Imho it would be best to use a deamonizer wrapper appor script, that does the starting and PID-into-a-file-writing generically. I wasted a few hours on several approaches and couldn’t get this to work, unfortunately. So now our rake idle task writes it’s own PID file and is monit’d. Semi-awesome.

6. Evaluation

In this chapter Myriad's initial goals will be reiterated and contrasted with real world observed usage.

6.1. Comparison with Initial Goals

6.2. Observed Use Cases

6.2.1. Recruiting Exchange Students for one of the Most Desired Universities of the World

6.2.2. Requesting paper Reviews for a Journal

6.2.3. Managing Incoming Class Administration Emails

7. Conclusion

Herein the relevance of this system and the proposed framework is discussed; future work is outlined and possible directions proposed.

7.1. Conclusion of this work

7.2. Discussion of results

7.3. Future Work

Bibliography

[1] Sara Radicati and Quoc Hoang. Email statistics report, 2012 - 2016. 2012.

[2] Adam N. Joinson and Ulf-Dietrich Reips. Personalized salutation, power of sender and response rates to web-based surveys. *Computers in Human Behavior*, 23(3):1372 – 1383, 2007.

[3] Leslie Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

List of Figures

Appendix

A. Selected Source Code Examples

```
1  class Notification < ActiveRecord::Base
2    belongs_to :user
3    attr_accessible :user, :message, :resource_id, :resource_type
4
5    def resource
6      @resource ||= resource_class.find resource_id
7    rescue ActiveRecord::RecordNotFound
8      nil
9    end
10
11   def resource_class
12     resource_type.classify.constantize
13   end
14
15   def has_resource?
16     should_have_resource? and resource.present?
17   end
18
19   def should_have_resource?
20     resource_type.present? and resource_id.present?
21   end
22
23 end
```

B. UML Diagrams over time

Test esetnetcyiegslcsgeycagfjegsf

B.1. Usage of UML Diagrams for internal communication

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

B.2. Stability of UML Diagrams over time

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

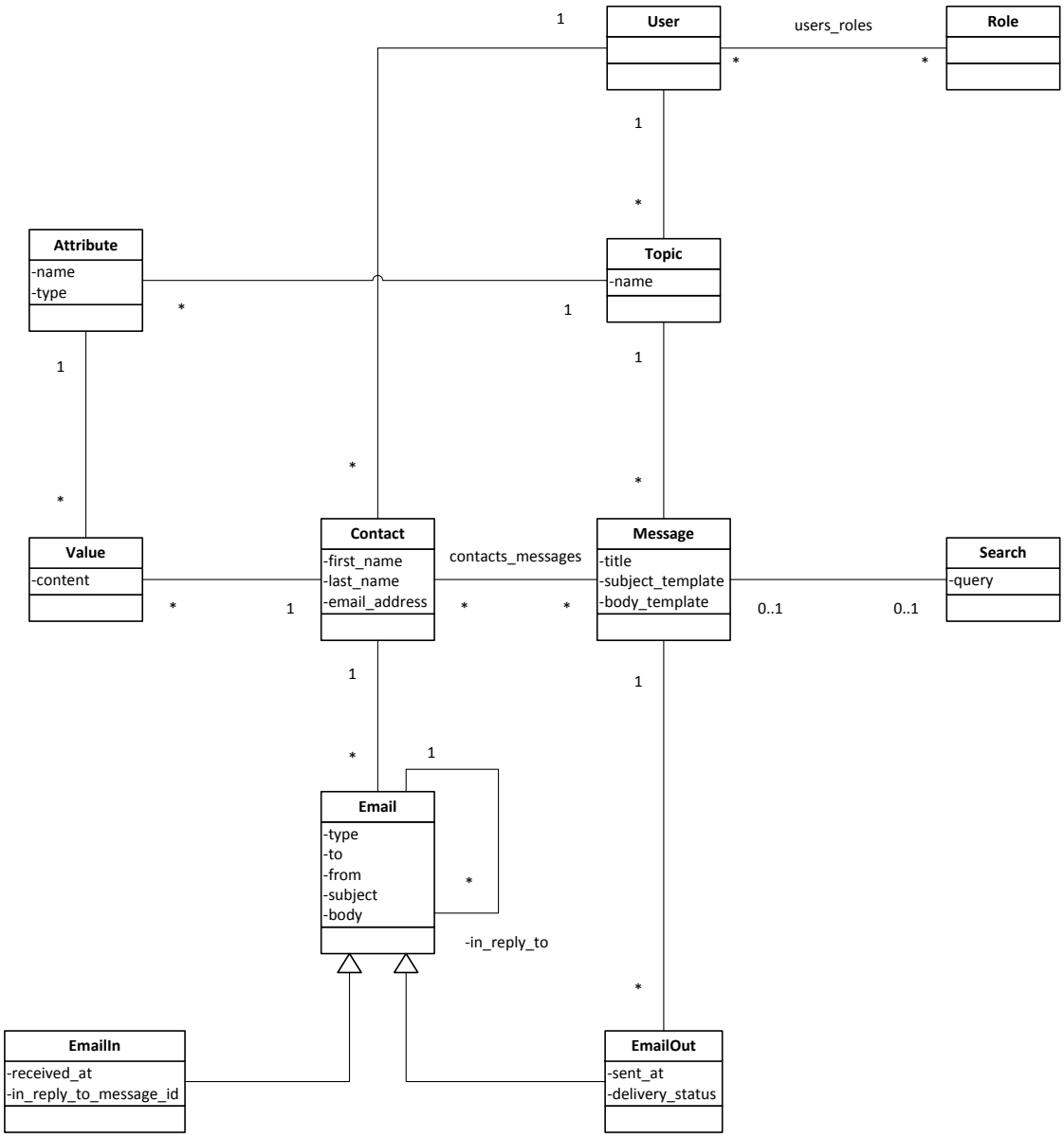


Figure B.1.: The first stable UML Diagram from January 24th still had separate `Email` and `contacts_messages` tables, no rule automation and a semantically incomplete model of Emails which didn't discern between user generated emails and system generated emails.

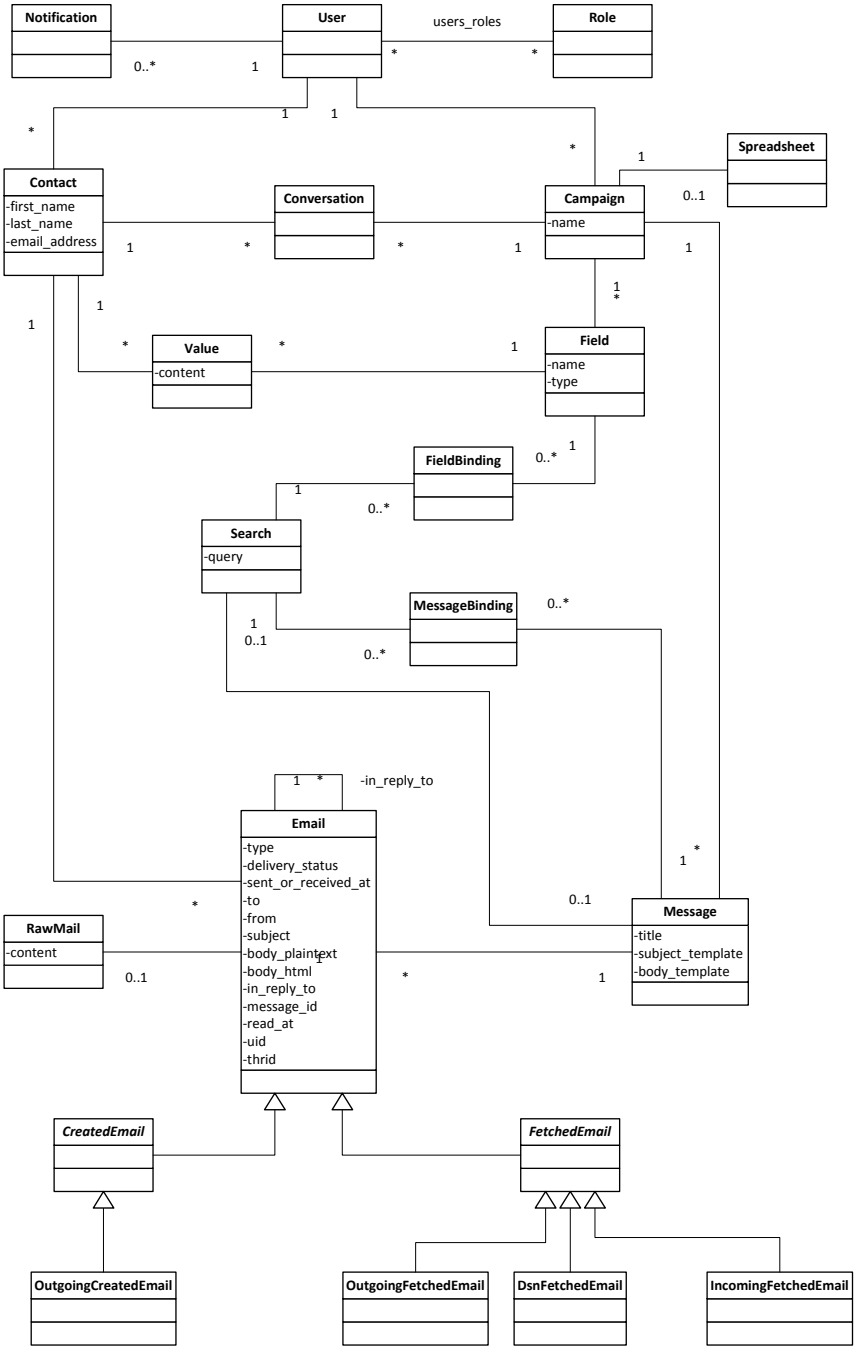


Figure B.2.: The second UML Diagram from May 28th fixed most of the aforementioned problems and already introduced optimizations such as extracting the RawMail content to a different table. Those had become necessary when support for attachments was introduced.

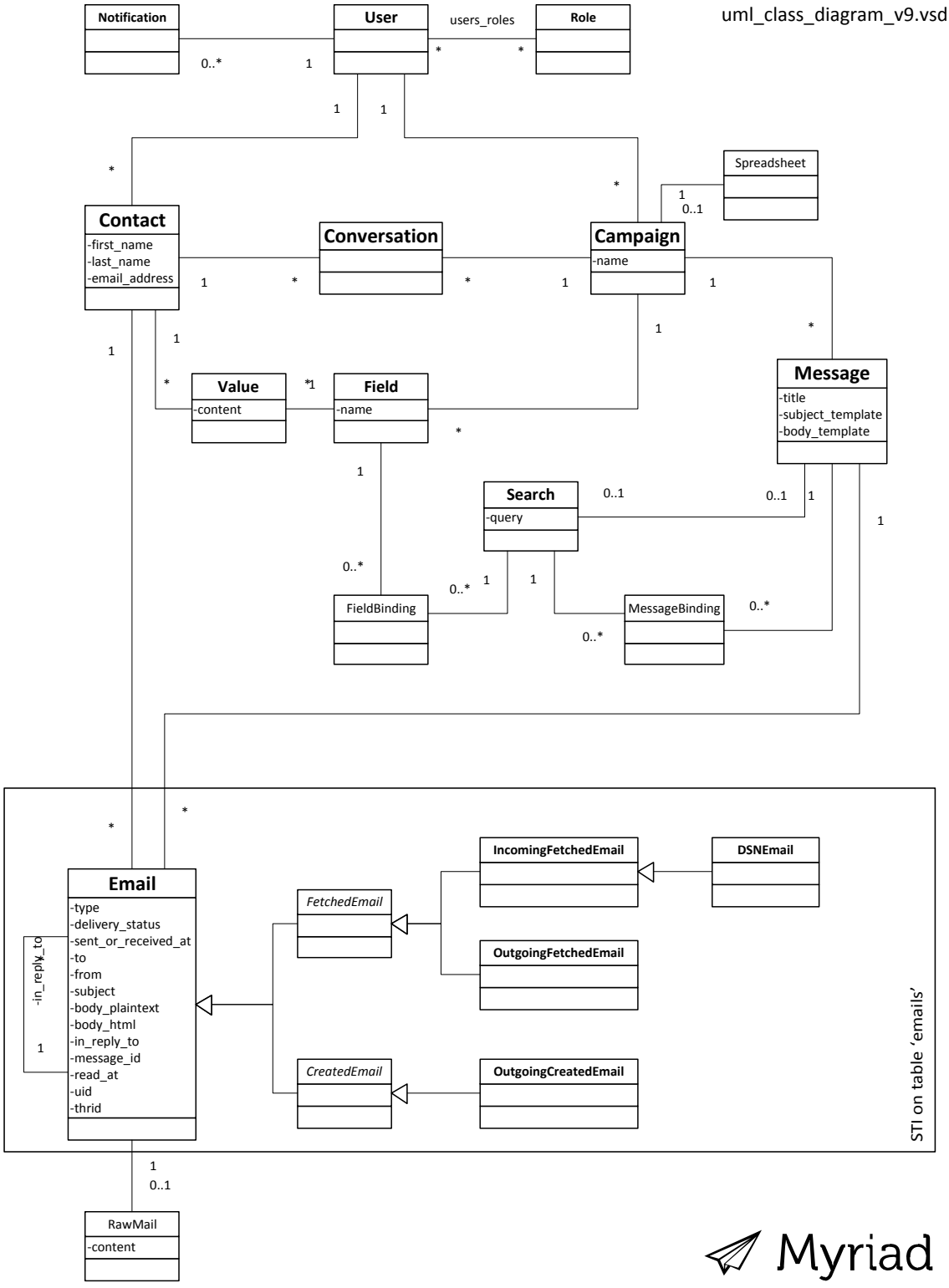


Figure B.3.: This third UML Diagram from May 29th finally tamed the `Email` inheritance tree, and also introduced improvements to legibility for the first time.

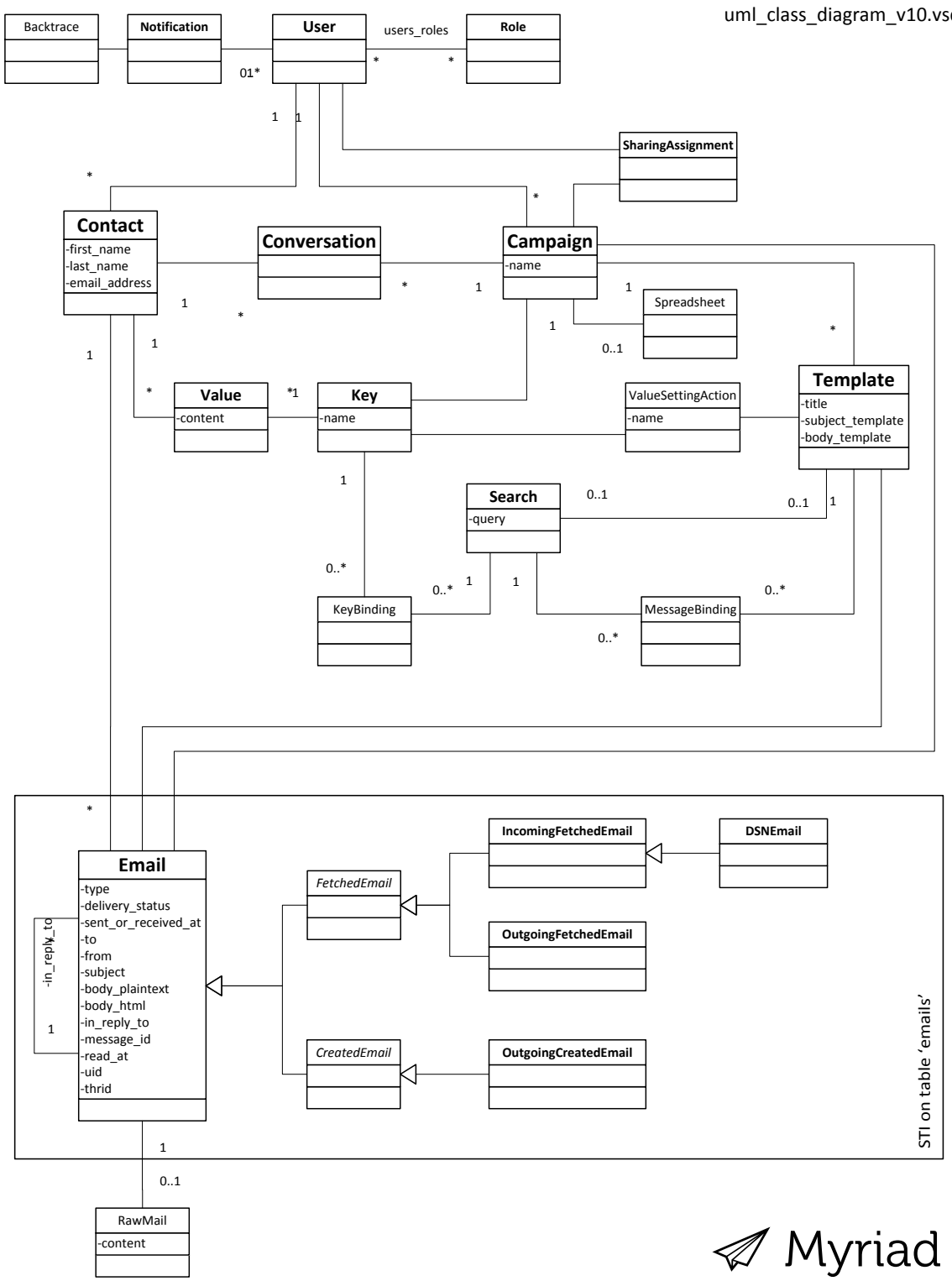


Figure B.4.: By the time of this fourth UML Diagram from April 18th only minor improvements were still made to the system design, such as adding `ValueSettingAction`s and supporting assistants via `SharingAssignment`s.

C. Colophon

This thesis is set in LaTeX[3]. The template used is based on the official TUM Computer Science template. The sources are hosted publicly on GitHub, while the actual PDF file is built by the continuous integration server Travis.