UCL
Computer Science
COMP103P

Pathways

Group 56* Project Report

George Pîrlea
george.pirlea.15@ucl.ac.uk

Shivam Shah
shivam.shah.15@ucl.ac.uk

April 25, 2016

*With special thanks to Sarah Payne, Nick Maisey and Alex Robson from Guy's and St Thomas' NHS Foundation Trust, as well as to Peter Coates, Alan Fish and David Jobling from NHS England Code4Health.

Contents

1 Introduction

1.1 What is Pathways?

Pathways is a web application that aims to make it easier for patients to access information regarding their treatment history, help them make informed decisions about available treatment options, develop their understanding of their medical condition and reduce their anxiety and confusion about the future.

At Guy's and St Thomas' Trust, clinicians would often pass patients information meant to assist their decision making, in the form of photocopied handouts. The problem with this system, as the trust discovered, is that clinicians would sometimes not have the required documents on-hand, and even when they did, patients would very often misplace the handouts they received. The result? Patients were not informed well enough, resulting in stress and confusion for the patient and low satisfaction scores for the trust.

Pathways is meant to solve this problem. For a patient, it is a simple and intuitive way to access and navigate their treatment history and related documentation – in a secure fashion. For a doctor or nurse, it is a communication tool: using Pathways, clinicians can make sure patients have access to all the information they need to make educated decisions regarding their treatment.

Its development is a collaboration between UCL Computer Science and Guy's and St Thomas' NHS Trust.

1.2 Development team

1.2.1 Previous team

Development was started in 2014 by Alex Gamble, Henry Scott-Green and Iustin Sibiescu, second year Computer Science students at UCL, as part of their COMP2013 and COMP2014 modules. They worked on Pathways for two terms in the 2014-2015 academic year, built the app from scratch and took it to a fully-functional state, but one which still needed improvements.

1.2.2 Current team

During the second term of the 2015-2016 academic year, we continued and built upon their work, adding new features, fixing old bugs, and creating better and more extensive documentation and adding automated deployment processes. This was part of the COMP103P module in our first year.

Team roles:

- **George Pîrlea**: Team Lead, Back-end Developer, Deployment Administrator
- **Shivam Shah**: UI Designer, Front-end Developer

Although we had different roles, with George focusing on the back-end and Shivam focusing on the front-end, it is important to note that we did a significant amount of *pair programming*: both of us working on the same computer, developing the UI and the back-end code simultaneously.

A notable side-effect of this is that the number of commits each of us made to our code repository does not accurately reflect the amount of work each of us put into the project — George made most of the commits, as we tended to work on his computer.

We go into further detail about how we split up the work in section about development.

1.3 Client

Our client was Sarah Payne, a specialist in Medical Oncology from Guy's and St Thomas' NHS Foundation Trust (GSST). She was our main point of contact and *product owner*, but we also had meetings with and gave demonstrations to other stakeholders from within GSST.

Of great help, especially when it comes to technical infrastructure, were Peter Coates, Alan Fish and David Jobling from the NHS England Code4Health.

2 Requirements and research

2.1 Client interview

2.1.1 Preparation

During the winter vacation, we researched knowledge elicitation [1] and negotiation techniques [2] so we could have a more productive professional relationship with our client. We came up with a few points that were immensely helpful:

- Plan interviews in some detail, but appear relaxed to clients so that they can speak freely.
- Use open, not leading questions!
- Check your interpretations with the users.
- Show users how the benefit from the process.
- Experience life as a user / observe users at work.

On January 21st, 2016, one day before our first (virtual) meeting with the client, we had a team meeting to review the documentation from last year and plan our interview structure.

2.1.2 Meeting the client

We had our first virtual meeting with Sarah Payne, Alan Fish and Peter Coates (over Skype) on January 22nd, 2016. We discussed the general purpose of the app (to support oncology patients with information personal to them) and our aim for this project (get the app in a more functional state for testing it with patients, with the intention of eventually handing it over to a professional software development house).

After setting-up the application on Azure and going through an initial phase of testing against the previous requirements, we had a face-to-face meeting with Sarah Payne on February 11th. Together, we walked through the app and came up with a set of requirements in MoSCoW format. Sarah formally confirmed the written requirements in an e-mail sent on February 12th.

2.2 Requirements

2.2.1 Writing requirements

As part of our requirements process, we first went through the requirements list produced by the previous development team. We noticed that almost all of the requirements were phrased in an impersonal and very hard to read fashion (*"Pathways must include patient account functionality to view all treatment information assigned by medical professionals."*).

After doing some research on how to write better requirements [3], we discovered the anatomy of a well-written requirement:

1. **User type**: *Patients*
2. **Result type** (verb): *must be able to view*
3. **Object**: *all their appointments*
4. **Qualifier** (adverbial phrase): *on a single timeline/calendar.*

We rewrote the previous requirements in this fashion (and, in the process, discovered that some were redundant), and also produced a list of 11 *new* or *updated* requirements (highlighted), for a grand total of 22 requirements.

2.2.2 Requirements list

Superuser functionality

1. Superusers must be able to create (patient and doctor) accounts.
2. Superusers should be able to delete (patient and doctor) accounts, with confirmation messages.
3. Superusers must be able to assign certain patients to certain doctors.
4. Superusers must be able to lock certain documents so only superusers (not doctors) can edit/delete them.
5. Superusers must be able to do everything doctors can do.

Doctor functionality

6. Doctors must be able to create patient accounts.
7. Doctors must be able to assign documents to a patient, including for dates in the future, only for patients assigned to them, with confirmation messages.
8. Doctors must be able to add to, update or delete the unlocked documents, both the title and the content, with confirmation messages.

9. Doctors must be able to embed pictures and videos in documents.
10. Doctors should be able to access a master document listing all existing documents, when they were created, when they need revising and who updated them.
11. Doctors should be able to send notes along with treatment information.

Doctor constraints

12. Doctors must not be able to delete patient accounts.
13. Doctors must not "see" or be able to modify patient accounts not assigned to them.

Patient functionality

14. Patients must be able to view all their appointments on a single timeline/calendar (irrespective of category).
15. Patients must sign in with their NHS numbers, rather than e-mail.
16. Patients must be able to see contact details for clinicians in the team they are assigned to.
17. Patients would be able to access their account offline.

System constraints

18. Users (patients, doctors, superusers) must be authenticated to be able to interact with the system.
19. All patient data must remain confidential.
20. Updated information must be immediately accessible to every patient.
21. System must be scalable for a potential roll out to other hospitals and NHS trusts.
22. Doctors and patients must be grouped in "teams" and access controls be strictly enforced.

2.2.3 MoSCoW analysis

All requirements in the list above have been analysed in terms of their priority. This is reflected in the way each requirement is phrased ("must", "should", "could", "would"). For ease of reference, we reproduce the new or updated requirements, grouped by MoSCoW priority, below.

Must have:

(R4) Superusers must be able to lock certain documents so only superusers (not doctors) can edit/delete them.

(R7) Doctors must be able to assign documents to a patient, including for dates in the future, only for patients assigned to them, with confirmation messages.

(R8) Doctors must be able to add to, update or delete the unlocked documents, both the title and the content, with confirmation messages.

(R9) Doctors must be able to embed pictures and videos in documents.

(R14) Patients must be able to view all their appointments on a single timeline/calendar (irrespective of category).

(R15) Patients must sign in with their NHS numbers, rather than e-mail.

(R16) Patients must be able to see contact details for clinicians in the team they are assigned to.

Should have:

(R2) Superusers should be able to delete (patient and doctor) accounts, with confirmation messages.

(R10) Doctors should be able to access a master document listing all existing documents, when they were created, when they need revising and who updated them.

(R11) Doctors should be able to send notes along with treatment information.

Could have:

None.

Would have:

(R17) Patients would be able to access their account offline.

2.3 Technology research

As we had to work on a project that was already partly implemented, our research was mostly concentrated on understanding the existing technology stack, rather than comparing competing technologies and choosing the most appropriate one. We did, however, need to compare solutions when it came to replacing the existing rich text editor inside the app.

2.3.1 Ruby on Rails

Pathways is built using the Ruby on Rails [4] web application framework. While Rails is a model-view-controller (MVC) framework at heart, it subscribes to the "*everything you need*" [5] philosophy, coming with default solutions for everything, from object-relational mapping (ActiveRecord) and view templates (ERB) to user login (Devise) and testing (MiniTest).

Rails guiding principles [6][7]:

- **Rails is Opinionated**: one *true* "Rails way" of solving any given problem
- **Rails is *Omakase***: Rails is a full stack; every framework has already been picked
- **Convention over Configuration**: instead of relying on extensive configuration files, Rails makes assumptions
- **Don't Repeat Yourself**: take advantage of language features to write as little code as possible

Having little experience with web frameworks and no experience with Ruby, we found it difficult in the beginning to understand how the Pathways code-base works, especially since a lot of behaviour in Rails is *implied* — there is no code and no configuration for that behaviour.

The learning curve is quite steep. It takes a while to understand the *magic* Rails carries out behind the scenes and it can be frustrating when it seems to be actively working against you — as we have seen with a nasty JQuery bug that was triggered by Turbolinks which prevented our confirmation messages from working correctly.

In the end, Rails is a good framework, but we are not entirely happy with it. Were we in the position of choosing a web framework, we would have chosen the Django [8] framework for Python, since it is more explicit and a lot less magical than Rails. This, in our opinion, makes it easier for people not familiar with the framework to understand an existing code-base and be able to extend it.

2.3.2 HTML and Slim

Being a web application, HTML is at the centre of Pathways. However, we wrote almost no HTML directly because we used Slim [9], a templating language for Ruby. Slim, unlike HTML, has constructs similar to those of a programming language — variables, conditionals and loops. These allow you to define the user interface in a more flexible way, and crucially, in the MVC pattern, allow you to keep the view code completely separate from the controller code.

2.3.3 CSS and Sass

For styling our application, we used Cascading Style Sheets (CSS) and a bit of Sass (Syntactically Awesome Style Sheets) [10], a CSS extension language that includes useful features like variables, nesting, imports, mix-ins, inheritance and operators.

2.3.4 Javascript and JQuery

Front-end interactivity (dynamic modals, "tab" switching, interactive date picker) was created using Javascript and the JQuery [11] library, which makes things like DOM manipulation, event handling, animation and AJAX much simpler to use, in a cross-browser manner.

2.3.5 Bootstrap

We used the Bootstrap [12] HTML, CSS and Javascript framework extensively, both to define the skeleton of our front-end (with the added benefit of having a responsive user interface) and also for its encompassing library of reusable interface components like toolbars, tab bars, bread crumbs, buttons, modals and date pickers.

2.3.6 Git

Thankfully, last year's development team kept their code in version control and on GitHub, so the code transfer between teams was easy — we simply forked the project on GitHub. The simple guide to Git [13] and the official Git documentation [14] were useful learning resources.

2.3.7 Automatic provisioning

While the code transfer was easy, the same cannot be said about getting the code to run. While there *was* a `Gemfile` which described which Ruby packages (called "gems") are needed, there was no documentation whatsoever describing what kind of system packages need to be installed, or indeed, what needs to be done after the required system dependencies are installed. We had to discover this on our own.

As we were going through this process of discovery, which took us two days, we documented our steps. Installing all dependencies, however, was tedious and took a long time. If at all possible, we wanted to solve the problem in such a way that future development teams would find it effortless to run Pathways, even if they have no experience with Rails.

We first thought about providing a pre-configured virtual machine, but we would have nowhere to host it. After some research [15], we discovered Vagrant [16]. With Vagrant, instead of providing a full-fledged virtual machine, you provide a *recipe* (called a `Vagrantfile`) for creating a virtual machine, which Vagrant can interpret.

The end result is the same, but the process is much more flexible: if your app has a new dependency, you don't need to re-create the virtual machine yourself — you just add a line to the `Vagrantfile` to install the new dependency.

In addition to Vagrant, which we added ourselves, we made use of Rails' provisioning tools, which were used by the previous development team. The `bundle` utility automatically installs all Ruby gems the project needs (stored in the `Gemfile`), while `rake` creates the database structure and populates it with mock data.

All these tools, when put together, make getting a running instance of Pathways on your own computer as simple as cloning the GitHub repository and running `vagrant up`.

2.3.8 Rich text editor

One of the requirements we had to complete (R9) related to how clinicians create and edit documents and what kinds of content can be embedded in documents. More specifically, our client wanted to be able to embed images and videos within the documents that patients would see.

Pathways already used Quill [17] as a text editor, but it had no way to embed videos directly (clinicians would have had to manually edit HTML code), lacked certain features (inserting tables or special characters) and did not function correctly in all browsers (it failed to load in some versions of Firefox).

Rather than try to fix Quill's problems, we decided we should look at other rich text editors and select one that would fit our requirements. We looked at TinyMCE [18], CKEditor [19] and NicEdit [20].

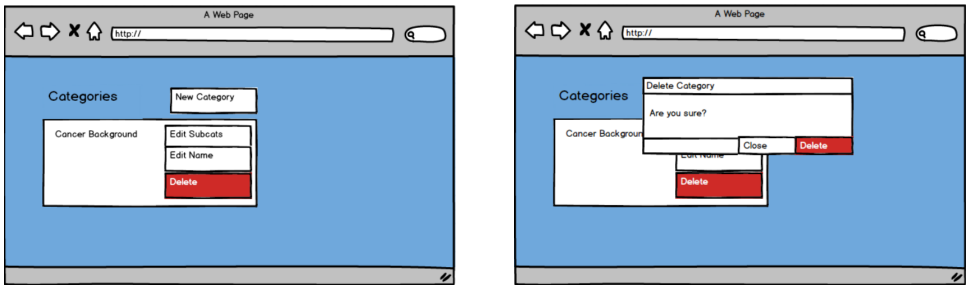
We quickly discarded NicEdit, as it was too simplistic, but had a hard time choosing between TinyMCE and CKEditor, as they were both very similar in terms of features, documentation and flexibility.

In the end, we chose TinyMCE because it displays a menu bar similar to Microsoft Word (which clinicians would find familiar and makes the interface less cluttered) and also because it seemed easier to integrate it with the already existing image upload functionality.

3 Design and structure

3.1 Interface mock-ups

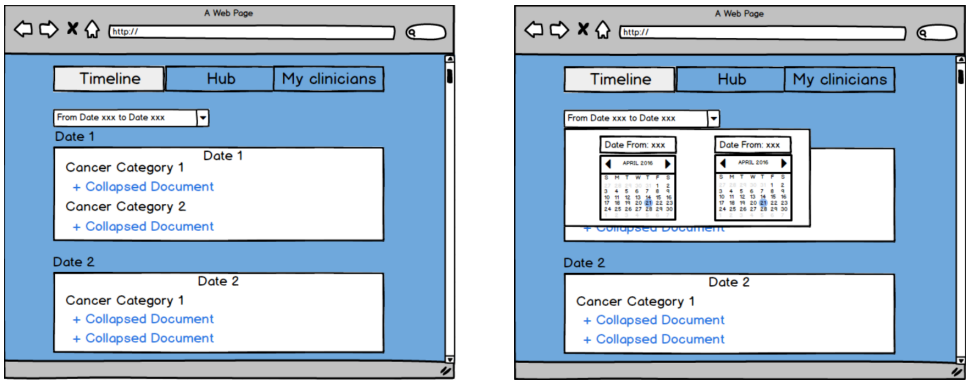
3.1.1 Confirmation modals



(a) Before clicking important button (b) After click, confirm modal

Figure 3.1: Confirmation dialogs for irreversible actions

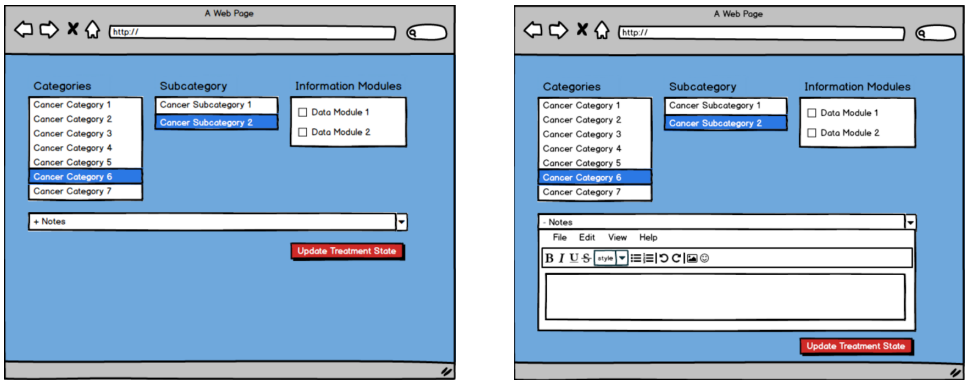
3.1.2 Patient timeline



(a) Show appointments in vertical timeline (b) Filter appointments by date

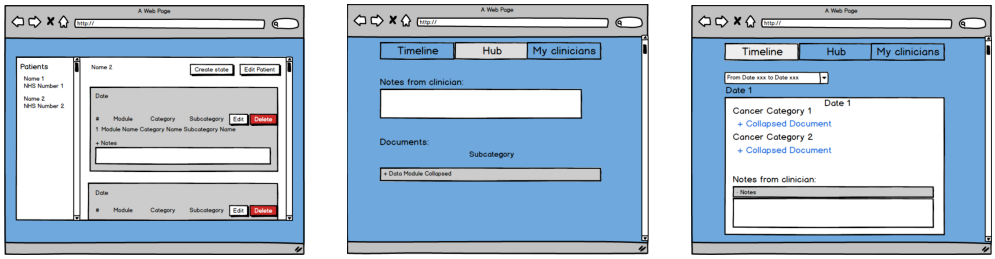
Figure 3.2: Timeline interface

3.1.3 Treatment state notes



(a) Editor is hidden by default (b) Click Notes to reveal

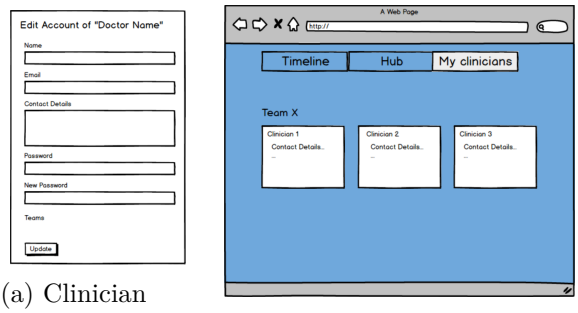
Figure 3.3: Clinician assigning a treatment state and attaching a note



(a) Clinician's view (b) Patient state view (c) Patient timeline view

Figure 3.4: Various location where notes are displayed

3.1.4 Clinician contact details



(a) Clinician profile (b) My clinicians view

Figure 3.5: Patients can see contact details of all their clinicians

3.2 Structural overview

Pathways respects the MVC design pattern. Application logic resides within the Models, but due to Rails' philosophy, most of the behaviour is implied, rather than explicitly programmed for. Presentation is handled using the Slim templating language, inside the application Views. Controllers connect the views with the models.

3.2.1 Models

The Model terminology chosen by last year's development team *is* confusing, and it took us a fair amount of time to navigate it.

User permission models:

- `Patient`: user type for patients
- `Doctor`: user type for clinicians; superusers are just a decorated doctor
- `Team`: construct for enforcing access permissions; a team has many doctors and many patients; doctors can only see information for patients in their teams

Document models:

- `Data module`: a rich text document containing treatment information; these are not specific to any patient, they form the database of content clinicians can assign
- `Image`: used for uploading images; returns the URL of the uploaded image, which can then be edited into a data module or note
- `Subcategory`: each data module belongs to exactly one subcategory
- `Category`: each subcategory belongs to exactly one category

Treatment information models:

- `Treatment state`: belongs to a patient; consists of an entire encounter or appointment with a clinician on a particular date; has an (optional) associated rich text note written by the clinicians, and zero or more treatment modules;
- `Treatment module`: belongs to a treatment state, and refers to a data module; it is the patient's view of a document (data module) assigned to them by a clinician
- `Pathway`: belongs to a patient; has many treatment states; it's the way treatment states are grouped into categories in the patient hub view

Not all of the models have corresponding controllers, as not all of them are supposed to be seen by the end user of our application.

For those that do have controller, however, the controller connects to multiple views, which are stored in a folder with the same name as the controller.

3.2.2 User login

The user login and page permission system is provided by a Rails package called Devise. Because of Rails' "magic", there is very little explicit code in Pathways that handles user logic or page access permissions.

`Doctor`s and `Patient`s overwrite a method called `find_for_database_authentication` which checks whether a given user exists when logging in.

`config/initializers/devise.rb` specifies a list of valid authentication keys. All of the logic is handled implicitly by Devise.

3.3 Design patterns

3.3.1 Model View Controller

The entire architecture of any Rails web application (including Pathways) is created around the MVC design pattern. Application logic resides within Models, presentation is handled by Views, and Controllers handle the logic of connecting the views with the models. This clear separation of concerns makes the code significantly easier to understand and work with.

3.3.2 Facade

Rails' Active Record system makes use of the Facade pattern for Object-Relational-Mapping. By modifying a configuration file, we can change the database we use for storing data (SQLite, PostgreSQL, MongoDB) without needing to change any line of code. Translating from object notation into appropriate database API calls is handled automatically by the Facade.

3.3.3 Proxy

Also in relation to the database, we use the Proxy pattern for transparently caching database requests. As such, accessing an object attribute does not necessarily trigger a database request; the result might be served from a cache. The proxy implements the object's interface, but changes the behaviour of its methods accordingly. In Rails, this is done automatically.

3.3.4 Decorator

One of requirements is that superusers need to be able to perform all actions that doctors can perform. Rather than implement a separate `Superuser` model, we use the Decorator pattern to extend the behaviour of the `User` model. This helps us avoid unnecessary code duplication.

3.3.5 State

We use the State design pattern to implement document locking and unlocking. The state is stored in the document object and the appropriate action is carried out depending on its state.

4 Development and testing

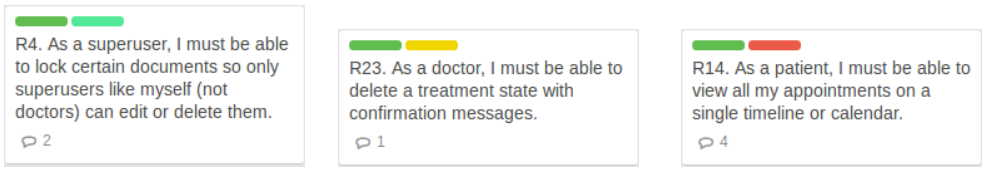
4.1 Agile methodology

Throughout the project, we used the Scrum development methodology to organise and manage our workload, and we used Trello boards for keeping track of the state of our project.

4.1.1 User stories

We documented all of the requirements as user stories, estimated the difficulty of implementing each of them and assigned color codes based on difficulty. Along with the color coding, we agreed on the estimated time needed to complete a requirement based on its difficulty:

- **Quick**: 1 day or less
- **Average**: 1 day to 3 days
- **Hard**: longer than 3 days



(a) Quick (b) Average (c) Hard

Figure 4.1: User stories with different estimated difficulties

4.1.2 Development stages

Once we had all the user stories, we created "buckets" to help organise our work and keep track of progress. Each bucket corresponded to a development stage a particular user story could be in, from left to right:

1. **Product backlog**: consisted of all the user stories that were not worked on yet
2. **Sprint backlog**: included the user stories that we agreed to be in the *current* sprint

3. **In progress**: user stories that were being actively worked on
4. **Ready to verify**: after implementing the user story, we moved it into this bucket to await approval from our client
5. **Done**: after a user story was approved, it was moved to this category to signify its completion

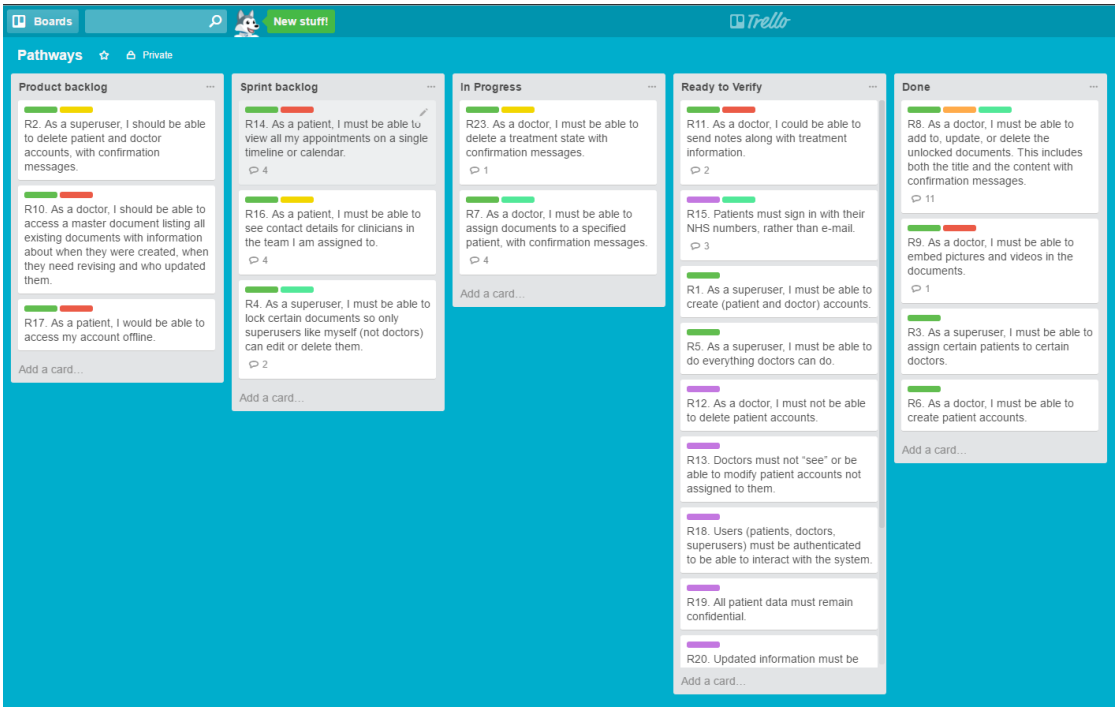


Figure 4.2: Trello board we used for project management

4.1.3 Sprints

We organised sprints that were 14 days long, to align with our bi-weekly report schedule. Estimating difficulties for each task came in useful when we had to decide which user stories we would be working on during each sprint.

We chose requirements based on their difficulty, using the following rules:

- **Hard**: no more than 2 in a single sprint
- **Average**: at least 1, but no more than 3 in a single sprint
- **Quick**: at least 1, but no more than 4 in a single sprint

These rules helped us arrange enough work for each sprint so that we would make use of the time available productively, but without overwhelming ourselves with work.

4.1.4 Team communication

Keeping in tone with the Agile philosophy, we communicated daily via either Slack or Skype, besides our face-to-face meetings during our two weekly lab sessions (we both attended each other's lab session) and in other contexts.

Furthermore, we implemented the concept of a regular "scrum", where each of us talked about:

- What we worked on the day before
- What we will work on today
- What obstacles we overcame
- What obstacles are in our way

By doing this, we were able to keep each other up to date with our progress and help one another in case of need.

4.2 Features

4.2.1 Login with NHS number

When we first began working with the Pathways code-base, we specifically chose easier requirements to tackle so that we could get familiar with the structure of the application. This was the first feature we implemented.

We met on March 2nd to discuss the high level structure of the application and try to understand how the login system works. To our surprise, there was no obvious authentication code. After looking in the `Gemfile` and looking up every package name to see what it did, we discovered Devise [21], which is what Pathways uses for user authentication.

We tried modifying the login logic in the `Patient` models, but saw that had no effect. Using StackOverflow, we figured out that we also needed to change the Devise configuration file [22] to reflect the new authentication key. We then followed the Devise guide for changing the authentication key [23] to make the necessary changes.

4.2.2 Clinician contact details

This was tackled in two parts. First, we added a contact details field for Doctor objects and created the necessary user interface for updating contact details. After that, we created the patient interface for displaying the contact details of all the clinicians in the patient's teams.

For the first part, we had to create a database migration [24] to add the new column to the `Doctor` model. While adding the ability for doctors to update their own contact details, we redesigned the password reset functionality as well — with the previous

version of the form, a doctor would have had to also update their password every time they wanted to change their details.

For the second part, we had to create a new patient view and research the Active Record Query Interface [25] to learn how to write a `JOIN` query using the Rails ORM.

4.2.3 Confirmation dialogs

We assumed this requirement would be quite simple, as it only involved changes to the front-end, but it actually turned out to be very extensive due to bugs — not only in the Pathways codebase, but also in some of the libraries we were using.

Initially, we implemented the confirmation dialogs using basic browser alerts [26]. However, these looked unpleasant, so we decided to replace them with Bootstrap modals [27][28], which looked nicer and could be customised to our liking.

After drawing some mock-ups and implementing the modals for the majority of non-reversible actions, we came across a problem where forms would not activate modals correctly. The issue was that the form was submitted without the modal being triggered for confirmation. This turned out to be due to a bug in the Javascript code.

Amending the code solved the problem relating to forms, but introduced another. It seemed that the modals would not pop up initially, but they did work after we refreshed the page. This was perplexing. We thought that the problem was in our code, but after a lot of searching and debugging, we still could not find why it was happening. Eventually, we discovered that our problem was created by a weird interaction between Turbolinks [29], one of the libraries installed by the previous development team, and JQuery [30][31]. The solution was to install a patched version of Turbolinks, called `jquery_turbolinks`.

4.2.4 Doctors add/edit/delete documents

Implementing this turned out to be easier than anticipated. Since this behaviour was already implemented for superusers, the only thing we needed to do was update the doctor's view code to show the appropriate buttons.

4.2.5 Document locking

Implementing document locking and unlocking was fairly straightforward. We created a database migration to add the new column to the `Data module` model and created a toggle button for locking/unlocking. After adding the appropriate route [32] for the POST request to lock or unlock a document, the functionality was working as intended.

4.2.6 Treatment notes

As a prerequisite for fulfilling this requirement, we replaced the Quill editor with TinyMCE so clinicians would be able to embed pictures and videos in their documents. As there was an existing TinyMCE package for Rails [33], this was not very difficult: we had to

install the package and configure it. Additionally, we had to install a TinyMCE plugin for image upload, configure it [34] and integrate it with the existing image upload solution.

After integrating the new editor, we added it to the "Add Treatment State" page and created the necessary model and controller code for attaching a note to a treatment state. Furthermore, we modified the clinician's view of a patient, as well as the patient's pathways view to display the newly added notes.

4.2.7 Patient timeline

For the timeline view, we had to create new presentation and controller code and connect it to the new methods in the `Treatment states` model. However, this was not very difficult — we were using already existing data and just presenting it in a different way.

After finishing implementing a static timeline (showing all treatment states for a given patient), our next task was to implement date filtering. We used the Date Ranger Picker plugin for Bootstrap [35] as a ready-made solution for the front-end side, but had to modify the controller code to return the patient timeline entries for a specific date range.

Initially, we wanted to use AJAX for dynamically updating the displayed entries, but this would have meant having to add a sizeable amount of new functionality. In the end, we decided that using URL parameters was a reasonable compromise.

4.3 Development schedule

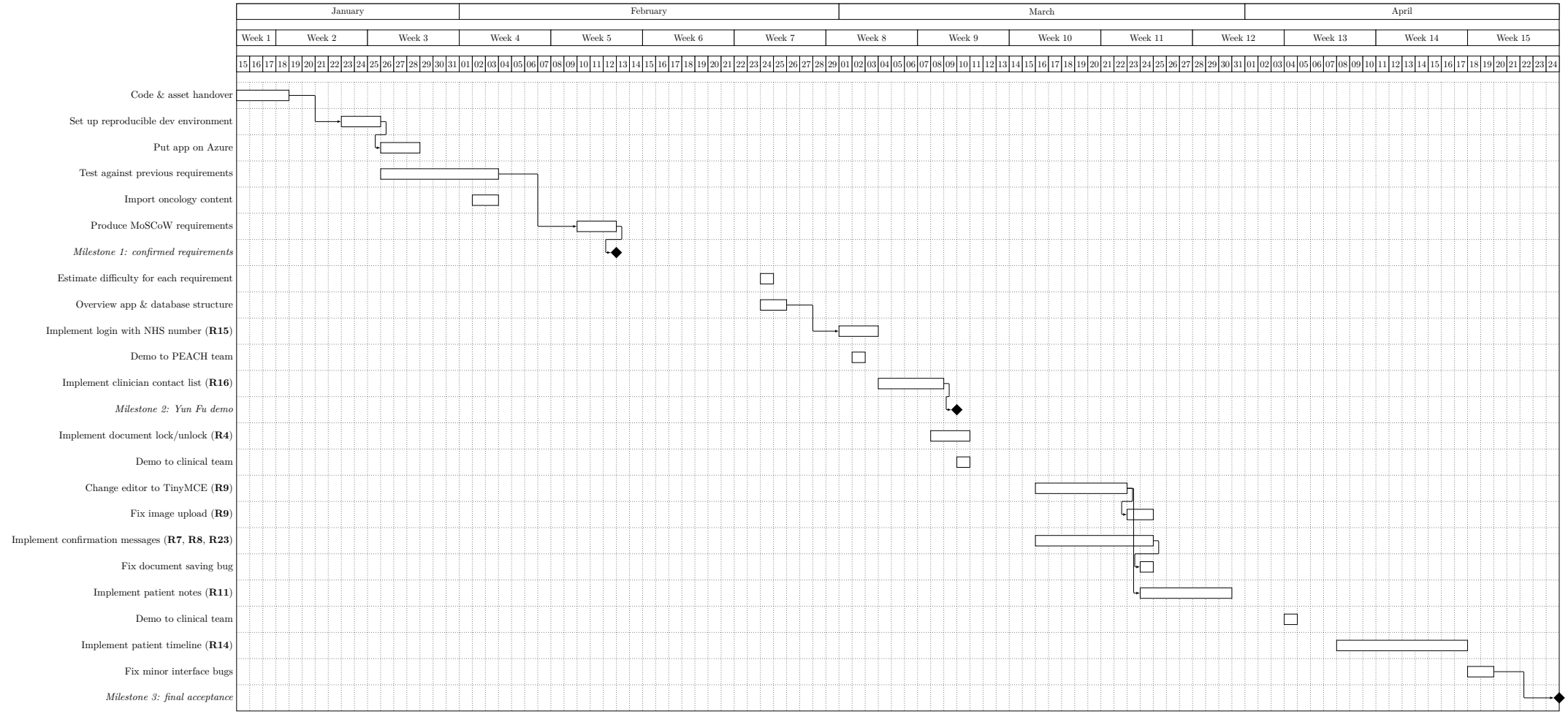


Figure 4.3: Gantt chart showing tasks undertaken, as well as their duration and dependencies.

4.4 Testing

4.4.1 Initial testing

As part of the initial handover process, we went through every requirement and manually verified whether it was implemented correctly. We discovered that some of the requirements that the previous development team marked as completed were not present in the implementation — for example, being able to delete user accounts. We marked these as incomplete and added them to our requirements document.

4.4.2 Issue tracking

While performing our manual tests, we noted any issues we encountered and kept track of them on our GitHub project's Issues page. This made it easier to reference any bugs or potential enhancements we discovered.

As we implemented features, we would test all desired behaviour before merging the code into our `master` branch. As an added precaution, we made sure to review and test each other's code, so as not to increase our chances of spotting potential mistakes.

4.4.3 Automated testing

Unfortunately, at the moment, all of our tests are manual. The previous development team seems to have set up a framework for automated testing (RSpec), but did not actually write any unit tests. Due to lack of time, and partly because we have very little model logic (most of it is implicitly defined by Rails), we chose to concentrate our efforts on implementing new features, rather than writing tests for the existing code-base.

4.4.4 Acceptance testing

We had regular meetings with our client and part of the clinical team at Guy's and St Thomas' Trust, who tested the features we added to the application and validated the functionality.

At the end of the project, we went through another round of testing, using multiple web browsers, and recorded outstanding issues on our GitHub page, so future development teams can reference them easily.

5 Evaluation, conclusion and future works

5.1 Evaluation

In hindsight, we did a lot of things right. Perhaps the most significant was the fact that we definitely did not neglect the project management side of things. We followed a development methodology, we used Trello boards to track our progress, we had an issue tracker for bugs, and most importantly, we established frequent communication with our client: e-mail whenever necessary, written reports on even weeks, Skype calls on odd weeks, and monthly face-to-face meetings.

We also placed a lot of importance on preparation. We really did our research. Before our first interview with the client, we read about how to carry out productive interviews [1], how to negotiate requirements [2] and explain that it is in everybody's interest to agree upon a set of requirements that we can actually deliver given the time we have, and also about how to write requirements [3] so it is easy to verify whether they have been completed or not.

Thankfully, our client was incredibly gracious in their demands, so collaborating with them was a pleasure. Nevertheless, the research we did was of tremendous help.

On the development side of things, perhaps we could have done better, but given the fact that we inherited a codebase that was far from perfect (as any code is) in a language we did not previously know or study, we still managed to significantly improve the app.

Given more time, we could have added more features and cleaned up some of the code. However, within the given circumstances, we did leave the codebase in a slightly better state than it was in previously. We have added some documentation and, very importantly, we made it easy for a future development team to get a running instance of Pathways on their own machines.

5.2 Conclusion

Our project has been a success. We have implemented all of our must-have requirements, as well as one of the should-haves, and our client is satisfied with the outcome.

5.3 Future works

A lot of improvements can be made.

On the technical side, the code needs a bit of re-factoring (there's some cross-over between views and controllers that shouldn't happen) and needs to have automated testing. For some reason, the previous development team set up a testing framework

(RSpec), but did not actually write any tests. Documenting the manual tests that need to be carried out would also be very helpful.

In terms of features, there are many possibilities. A few salient ones:

- Make the timeline in the category view vertical, rather than horizontal; essentially, turn the hub into a way of sorting timeline entries by category
- Add a search feature to the timeline and perhaps think of a better way to present the existing information
- Create a patient diary functionality; let patients attach their own notes to treatment states (and maybe create treatment states of their own?)
- Let new users start with some default information already assigned to them
- Integrate Pathways with openEHR

5.3.1 OpenEHR integration

One of the goals of Pathways is to eventually integrate it with openEHR [36], an open standard in terms of electronic health records and representing health-related information. However, we did not tackle this in our work.

Our client's priority at this stage of the project has been to extend the functionality of the application by adding new features. Nonetheless, making Pathways compatible with other e-health systems remains a goal and we have done some research on it, even though we have not been able to start making it a reality.

There are two different ways openEHR integration might be accomplished:

1. Replace the current PostgreSQL database with an openEHR back-end; store all information in openEHR format
2. Keep using PostgreSQL, but create a translation layer from our internal representation to openEHR format (export only)

The first option would most likely involve tremendous amounts of work, as current open-source solutions [37][38] seem immature. In any case, having to pull all of openEHR's complexity into Pathways and trying to fit all existing data around openEHR might not be the best decision.

The second option seems more plausible. Conveniently, a Ruby implementation of openEHR already exists [39], although we are not sure how complete it is. If you choose to opt for this route, you would need to find a set of openEHR archetypes to represent Pathway's data (superficially, a series of `Encounter`s with free-text `Note`s might do the job, but further research is needed), design an openEHR template for these archetypes and implement the translation layer between our object representation and ADL.

Integrating Pathways with openEHR is going to be at least a few months worth of work, in our estimation.

References

[1] M. Firlej and D. Hellens. *Knowledge elicitation: a practical handbook*. Prentice Hall, 1991.

[2] J. Mattock and J. Ehrenborg. *How to be a better negotiator*. Kogan Page, 1996.

[3] I. Alexander and R. Stevens. *Writing Better Requirements*. Addison-Wesley, 2002.

[4] *Ruby on Rails*. URL: `http://rubyonrails.org/`.

[5] *Rails has everything you need*. URL: `http://rubyonrails.org/everything-you-need/`.

[6] *What is Ruby on Rails*. URL: `https://railsapps.github.io/what-is-ruby-rails.html`.

[7] D. Heinemeier Hansson. *Ruby on Rails: Doctrine*. Jan. 2016. URL: `http://rubyonrails.org/doctrine/`.

[8] *Django: The Web framework for perfectionists with deadlines*. URL: `https://www.djangoproject.com/`.

[9] *Slim – A Fast, Lightweight Template Engine for Ruby*. URL: `http://slim-lang.com/`.

[10] *Sass – Syntactically Awesome Style Sheets*. URL: `http://sass-lang.com/`.

[11] *jQuery – The Write Less, Do More, JavaScript Library*. URL: `https://jquery.com/`.

[12] *Bootstrap – The world's most popular mobile-first and responsive front-end framework*. URL: `https://getbootstrap.com/`.

[13] *Git – the simple guide*. URL: `https://rogerdudler.github.io/git-guide/`.

[14] *Git Documentaion*. URL: `https://git-scm.com/doc`.

[15] V. Viallet. "Vagrant, Docker and Ansible". In: (2013). URL: `http://devo.ps/blog/vagrant-docker-and-ansible-wtf/`.

[16] *Vagrant by Hashicorp*. URL: `https://www.vagrantup.com/`.

[17] *Quill – A Rich Text WYSIWYG Editor with an API*. URL: `http://quilljs.com/`.

[18] *TinyMCE – The Most Advanced WYSIWYG HTML Editor*. URL: `https://www.tinymce.com/`.

[19] *CKEditor – The best web text editor for everyone*. URL: `http://ckeditor.com/`.

[20] *NicEdit - WYSIWYG Content Editor, Inline Rich Text Application*. URL: `http://nicedit.com/`.

[21] *Devise – Flexible authentication solution for Rails with Warden.* URL: https://github.com/plataformatec/devise.

[22] *Devise authenticating with username instead of email.* URL: https://stackoverflow.com/questions/10866667/devise-authenticating-with-username-instead-of-email.

[23] *How To: Allow users to sign in using their username or email address.* URL: https://github.com/plataformatec/devise/wiki/How-To:-Allow-users-to-sign-in-using-their-username-or-email-address.

[24] *Active Record Migrations.* URL: http://guides.rubyonrails.org/active_record_migrations.html.

[25] *Active Record Query Interface.* URL: http://guides.rubyonrails.org/active_record_querying.html.

[26] *Rails confirm before delete.* URL: https://stackoverflow.com/questions/19588058/rails-confirm-before-delete.

[27] *Bootstrap Modal Plugin.* URL: http://www.w3schools.com/bootstrap/bootstrap_modal.asp.

[28] *Boostrap Modal in Slim.* URL: https://gist.github.com/StanBoyet/448f04f44d88c21522fa.

[29] *Rails 4: A Look at Turbolinks.* URL: http://blog.teamtreehouse.com/rails-4-a-look-at-turbolinks.

[30] *Form Submit button only works after reload.* URL: https://stackoverflow.com/questions/19365809/form-submit-button-only-works-after-reload.

[31] *jQuery Modal Dialog works fine on the first page after refresh, but not on any other pages after that unless refreshed.* URL: https://stackoverflow.com/questions/18646889/jquery-modal-dialog-works-fine-on-the-first-page-after-refresh-but-not-on-any-o.

[32] *How to add custom routes to resource route.* URL: https://stackoverflow.com/questions/16693185/how-to-add-custom-routes-to-resource-route.

[33] *Integration of TinyMCE with the Rails asset pipeline.* URL: https://github.com/spohlenz/tinymce-rails.

[34] *Rails - tinymce-rails-imageupload configuration.* URL: https://stackoverflow.com/questions/22939043/rails-tinymce-rails-imageupload-configuration.

[35] *Date Range Picker for Bootstrap.* URL: http://www.daterangepicker.com/.

[36] *OpenEHR – An open domain-driven platform for developing flexible e-health systems.* URL: http://openehr.org/.

[37] *LiU EEE – A REST based Educational EHR Environment based on openEHR.* URL: https://github.com/LiU-IMT/EEE.

[38] *EHRflex – An archetype-driven EHR system.* URL: http://ehrflex.sourceforge.net/.

[39] *Ruby implementation project of openEHR specification.* URL: http://openehr.jp/projects/ref-impl-ruby.

Accepting requirements

Subject: Re: (Team 56) Milestone 1: confirming requirements + bi-weekly report
From: Sarah Payne <sarahpayne75@yahoo.co.uk>
Date: 16.02.2016 06:55
To: "Pirlea, George" <george.pirlea.15@ucl.ac.uk>
CC: Aisling O'Kane <a.okane@cs.ucl.ac.uk>, Dean Mohamedally <d.mohamedally@cs.ucl.ac.uk>

Dear George

Just to confirm the requirements are correct

Best wishes

Sarah

Sent from my iPhone

On 12 Feb 2016, at 14:54, Pirlea, George <george.pirlea.15@ucl.ac.uk> wrote:

> Dear Sarah,
>
> As we've previously discussed, we need written confirmation (e-mail is fine)
>of the requirements before we can proceed with development.
>
> I've attached two documents:
>
> * MoSCoW analysis.pdf - the requirements document you need to confirm; the
>first page contains the requirements for this stage of the project; the second
>page contains all requirements (including those from last year), most of which
>are already met
>
> * Bi-weekly Report 1.pdf - the first bi-weekly report, containing an overview
>of what we've accomplished so far, a summary of all meetings we've had and our
>plan for the next two weeks
>
> On behalf of team 56,
> George Pîrlea

Client feedback

Subject: Pathways update for report From: Payne Sarah <Sarah.Payne@gstt.nhs.uk> Date: 27.04.2016 13:45 To: Dean Mohamedally <d.mohamedally@cs.ucl.ac.uk>, Robson Alex <Alex.Robson@gstt.nhs.uk>, Ngan Sarah <Sarah.Ngan@gstt.nhs.uk>, Maisey Nick <Nick.Maisey@gstt.nhs.uk>, Chowdhury Saira <Saira.Chowdhury@gstt.nhs.uk>, "alan.fish@nhs.net" <alan.fish@nhs.net>, "george.pirlea.15@ucl.ac.uk" <george.pirlea.15@ucl.ac.uk>

Dear Dean

Please see below for comments regarding the client side of Pathways

Best wishes

Sarah

PATHWAYS - Workflow report - SP and Alan Fish

Comments

Both students were engaged with the project from the start. They worked quickly at the start to get access to previous project information and coordinate discussions with previous students for efficient handover. They were able to assess for glitches in the previous system and fix them, as well as to add features and streamline the prototype in line with the new brief. Both students have successfully liaised with the clinical team and the team from NHS England Code4Health.

Communication

The team were involved in regular dialogue by Skype or face to face to discuss progress. They coherently and efficiently presented updates and were confident in asking questions for clarification of the client brief. Documentation and guidance produced by the team was clear and easy to understand.

Delivery

The team have developed an enhanced prototype of pathways which will support discussions with external stakeholders for further development. They have kept to defined timelines and submitted timely reports regularly to give updates on progress. The team also migrated the solution to the NHS Code4Health Azure server with no issue.

Overall

The team have delivered an excellent project which they have embraced from the start. They have demonstrated excellent communication skills and have come up to speed in relation to the needs of a clinical team very quickly. They have created a prototype which has been presented to the whole clinical team and will be an asset for ongoing discussions regarding development for real world use.

PATHWAYS: Final Report - from presentation 25.4.16 - Nick Maisey, Sarah Ngan, Alex Robson, Saira Chowdhury

Description

Pathways is a web-based application designed to support the patients' journey through their cancer treatment, allowing easily retrievable treatment information which can also be accessible to other health professionals such as the patients' General Practitioner.

App Development

The team inherited Pathways at a more advanced stage of development than APPetite. They managed to make significant positive changes to the product and make it more user-friendly. They also incorporated a number of changes suggested by the clinical team.

Customer Liaison

There was a good interaction with the clinical team. Although not initially obvious, they quickly understood the relevance of suggested changes and how these adaptations could benefit the patients. They worked quietly and efficiently and developed a good raport with the clinical team.

Strengths

The team developed an excellent 'customer' relationship and adapted the

product quickly to suggested changes. The final Customer ‘pitch’ was professional and showed a high level of maturity.

Weaknesses

The ‘front-end’ of the product needs some improvement in terms of how intuitive the time line is. However development time was short and these are changes that can be addressed in future.

Overall Impression

The Pathways team have worked very well as a team and have been professional in their approach. They have added significant value to an already workable prototype.