COMP4

Norbert Podsadowski

Includes full analysis, design, specific implementation, testing and evaluation of a project management system for Top Builders

2015

Table of Contents

# Analysis

## Background to and identification of problem

Client name: Top Builders  
  
CEO: Peter Podsadowski  
  
Business address:

70 Lucerne Road  
Thornton Heath  
London, CR7 7BA

The main end user is my father, Peter, who currently owns a construction company called Top Builders. He conducts various types of construction work, including extensions, furniture fixing, painting & decorating, and more. The company is a small family-based business, and Peter tends to work alone, but sometimes he will hire workers to help him with larger-scale projects. After a few years of good trade, he is looking to expand his business by taking up more projects, hiring more employees and deviating away from a small-scale family business to something more structured. He needs a system to aid him in managing quotes, invoices, employees and his stock of materials in order to ensure a smooth running of the business and decrease the amount of time spent on bureaucracy and administration.

## Observation of the current system

# Physical observation of the current system

Being Peter’s son and living in the same house as him allowed me to observe the working of his current system almost every day. For example, very recently, a new client called him on a Friday enquiring about a job offer; I witnessed the entire conversation and observed how Peter begins his projects through exchanging various details with the client. He arranged an appointment to see the client on the next day (Saturday). I asked to accompany him in order to better understand his work pattern.

We left at 10:00am and made our way to the client’s house. Upon arrival, the client showed us to the kitchen where a whole refurbishment was needed. Peter spoke to him about exactly what needs to be done, which furniture needs to be replaced, the details of any plumbing/electrical work and the estimated time of completion. He used pen and paper to note these things down. At about 3:30pm, we left the client, and said that we would e-mail him with the quote for the required work. *(A quote is a breakdown of the entire project, listing individual jobs involved and estimating potential costs).*

We returned home at about 4:00pm and immediately began working on a quote in Excel. Peter often makes me operate the computer while he simply dictates the jobs, their details and prices as I input them into the spread sheet; this is due to my better IT and English skills. This makes me very knowledgeable in the exact workings of an integral part of the current system, and I have first-hand experience at the somewhat clumsy and time-consuming process of creating customer quotes.

Peter then e-mailed the quote to the client, and after some further negotiations and changes to the quote, work could begin. The details of the job were recorded in a “job book”. Peter needed a helping hand with this kind of project, so he hired an employee (which he has worked with before in the past). He did not need to record the employee on paper, as it was just one employee – but for other projects, where teams of employees are needed, an “employee book” would be used, where working hours and pay rates would be recorded. Both of these record-keeping books are held in the company cabinet at home.

# Interview of company CEO

In order to find out more details about the prospective end user, I carried out a live interview with Peter. This interview aimed to provide the basis for analysis of the current system limitations, setting targets for the new system, planning a feasible solution and ensuring that the finished product meets the requirements of the end user.

**Norbert**: To start us off, what do you currently do for a living?  
**Peter**: I own a building company called Top Builders. I do all kinds of construction work including extensions, kitchen and bathroom fitting, tiling, painting and decorating, etc.  
  
**Norbert**: Do you work on your own?  
**Peter**: Mostly, yes. I am self-employed under the Top Builders name and the company is entirely run by me. However, for some bigger jobs, I tend to hire a couple of workers to help me. I pay them a working wage but act as the project manager, while still of course working with them on-site.

**Norbert**: How do you co-ordinate your workers, keep track of how many hours they have worked, etc.?  
**Peter**: Well, I guess I just keep it in my head, or write it down in a little job book that I have. I haven’t used this much, but I expect that to change once I start hiring more workers. At the moment, I hand them the money after a week of work and just tell them the days when they should come in and where we’re going.

**Norbert**: That sounds a bit cumbersome?  
**Peter**: It is, but I don’t hire that much workers for it to be too much of an issue at the moment. However, if I carry on getting large-scale projects like I have been recently, it could get very disorganised.

**Norbert**: What about the details of all the jobs you need to complete?  
**Peter**: I record those inside a large “job book”, which records the details of every client, the jobs they need us to do, things like that. I use this information to later produce a quote for the client.

**Norbert**: Do you ever advertise, or is your business quite small?  
**Peter**: I give out business cards to any potential clients which may require my services. I am advertised on Yell and have a van with my business logo and contact details on it. I also had a simple website which listed all my services, but found it not useful enough in bringing in new clients for the price I was paying, so I got rid of it.  
  
**Norbert**: How do you find your work, and then how do you proceed to complete it?  
**Peter**: I receive work, very often through recommendation, from a potential client and immediately arrange a meeting to inspect the premises and discuss the details. Using this information I then produce a quote listing all of the specific jobs that need to be done, and the total cost. I then present this quote back to the client, discuss any amendments, and begin work immediately. Once complete, I hand the client an invoice, which is basically just the same as a quote, but with slight amendments to reflect the exact work that was completed.

**Norbert**: Could you tell me more about what you include in your quotes?  
**Peter**: All of my quotes tend to differ a lot due to the fact that I make them manually in Microsoft Excel. Usually I will have a table of 5-10 jobs for the project, with different materials listed for every job. I will then have a price for every job, and sometimes break this price up to indicate labour and material costs. I will also have a total at the bottom. I like to include my company logo and the name and address of the client, although sometimes forget to do this.

**Norbert**: And how long do these quotes take to produce?  
**Peter**: Well, I am nearly 50 years old, and although I don’t consider myself completely backwards in terms of technology, messing around in Excel can sometimes take me up to 3 hours to complete one quote. Most of the time I just ask you to do it, as you’re much quicker.

**Norbert**: What would you like the new system to do for you?  
**Peter**: I mainly want something easier than Excel produce my quotes for me quickly and efficiently. I want to be able to quickly launch an application, fill in some jobs details and figures and have it printed off and the client e-mailed within minutes. Of course the format still needs to be Excel, just not made using Excel, if that makes sense, as I find most clients are able to open such a format, and it can be made to look nice.

**Norbert**: Is that all you need the system to do, or do you have anything else in mind?  
**Peter**: Well, I’m open to suggestions. Anything to make my life easier.

**Norbert**: Well, what about the future? Where do you see your business going?  
**Peter**: Right now, everything is going pretty well and I would love to turn this into a more of a structured enterprise rather than a small family business. I will definitely be pursuing some higher forms of advertising in the future so that I can get a much larger volume of clients. Of course, this will mean hiring a proper team of employees. A proper website will eventually become a must.

**Norbert**: What would this website contain?  
**Peter**: A description of all the services I provide, a gallery, my contact details. It would also be nice if clients could request a quote online which I could somehow see and complete.

**Norbert**: That could definitely be something I could include in the system, allowing you to respond to quote requests submitted online. Would that be useful?  
**Peter**: Absolutely! It would be incredibly convenient to direct potential clients onto the future website and allow them to describe their problem without me having to take the time out to see them or call them.

**Norbert**: How do you currently get all of the materials you need to complete a project?  
**Peter**: After producing the final quote I take a trip to any builders’ warehouse such as Selco and pick out all the things I will need. This sometimes takes more than one trip and is quite tedious to do, but I have no other choice. If the business was large enough, I would probably store a stock of materials somewhere and pick them out from there.

**Norbert**: Surely you would need something to manage that stock electronically?  
**Peter**: I would indeed.

**Norbert**: That’s something worth looking into for the new system. Finally, do you have any final thoughts on how you would like your new system to function?  
**Peter**: I just want something that works well, is fast and isn’t overly complicated. Initially I just wanted a simple quote maker, but now that I think about it, the expansion of my business may come quite soon and I will most likely require other tools to help me. For example, that stock of materials manager you mentioned sounds like it could be very useful.

**Norbert**: I will definitely aim to produce something that satisfies your long-term requirements. Thanks for your time; it was very helpful in determining exactly what you need.  
**Peter**: No problem. I will be looking forward to closely watching over the design of this new system.

*Interview was conducted in Polish and later translated and adjusted for easier understanding.*

# Document analysis

  
In this section, I will analyse a typical Top Builders quote in order to increase my understanding of the needs of the end user. These are real, actual quotes of previous projects which the company has completed.

The quotes appear to differ in layout for every project. The proposed system should aim to avoid this and produce standardised and consistent quotes for every project. However, some projects will require quotes to occasionally differ. The proposed system needs to cater for this and allow toggling of various additional columns as necessary, such as “job description”, “material cost”, etc.

Top Builders company logo.

*Note: The word “quote” is interchangeable with “invoice”, as both are exactly the same in layout, and will be treated as the same thing in the proposed system.*

****

Company details need to be visible for every quote.

Every invoice displays the address of the project. *(Hidden for data protection reasons)*

Occasionally, jobs will require additional descriptions.

All quotes need to have a breakdown of all the individual jobs for every project.

## D:\Dev\Projects\BuildersCentral\BuildersCentral\COMP4 Project - Norbert Podsadowski\invoice2.jpg

Some quotes will display material and labour costs in one value. Other quotes may break this down to show material costs, labour costs and even time taken per job.

These columns and various settings for the customization of quotes should be easy to toggle in the quote creator of the proposed system.

A total for each column should be displayed at the bottom of a quote.

The biggest projects may need a categorical break-down of jobs, as shown on the left. The jobs for this project have been sorted in various groups. This capability needs to exist in the proposed system.

## D:\Dev\Projects\BuildersCentral\BuildersCentral\COMP4 Project - Norbert Podsadowski\invoice3.jpg

Occasionally, a “note” or final project description will be included at the end of a quote.

The table of materials in the database should store price per unit, amount in stock and type of unit (length, width, units, etc)

The following is an invoice from Selco Warehouse, one of Top Builders material suppliers. This is an example of materials that will have to be stored in the system.  
  


# 

# End user questionnaire

Although the interview and documents provided by Peter have provided me with a lot of information, this questionnaire was also given for him to complete. It deals with more quantitative information which will clear up concerns regarding data volume and other details which weren’t discussed previously.

1. How many hours per week do you spend on completing tasks related to the running of your business (creating invoices, speaking with customers, etc)?

*6-8*

*3-5*

*< 3*

*> 11*

*9-11*

1. How many of these hours are spent on tasks which require you to use a computer?

*6-8*

*3-5*

*< 3*

*> 11*

*9-11*

1. How many projects do you complete during an average quarter (3-month period)?

*> 11*

*9-11*

*6-8*

*3-5*

*< 3*

1. How many projects do you expect to complete during an average quarter in about 2 years?

*9-11*

*6-8*

*3-5*

*< 3*

*> 11*

1. How many projects does an average client provide after initially meeting them?

*3*

*2*

*1*

*> 4*

*4*

1. How many different types of materials does an average project require?

*< 4*

*> 21*

*16-21*

*10-15*

*4-9*

1. How many employees do you currently hire to accommodate for all your projects?

*2-3*

*1*

*> 7*

*6-7*

*4-5*

1. How many employees are you expecting to in the next 2 years?

*6-7*

*4-5*

*2-3*

*1*

*> 7*

1. How many individual jobs does an average project require?

*6-10*

*< 5*

*> 20*

*16-20*

*11-15*

1. How many times per week on average do you receive calls from potential clients?

*2*

*< 2*

*> 4*

*4*

*3*

## Description of the current system

After primary observation of Peter’s current system, document analysis, the interview and completion of the questionnaire, Peter’s current system may be summarised as follows:

1. Find a client, either through recommendation, advertising or otherwise
2. Consult with the client, assess the construction work to be completed
   1. Peter records a new client’s details and the job details in the company job book
3. Produce a quote (price estimate) of the work and present this to the client
   1. The quote should include a breakdown of all the jobs to be done for the whole construction project, the details of each job, the materials required and the prices for each individual job
   2. This is currently done manually, using an Excel spreadsheet
4. Further consultation with the client based around the produced quote
5. Employees are hired for the project, and assigned concrete tasks and working hours
6. Work then begins
7. After the construction project is complete, the client is handed an invoice which represents the actual work done, materials purchased, and overall cost
8. Employees are paid

## Identification of prospective users

The main user of the system will be Peter, as he is the CEO of the company and deals with all client-business communications, like initial consultation or quote creation. He is also responsible for managing employees, and in the future, when his company expands, there is a possibility of him hiring higher ranked employees to manage workers across different projects for him. Another prospective user is an employee of the company, who should be able to use the system to get information such as tasks assigned during a project, hours of work and wage rate. An easy way to contact the responsible supervisor, such as a manager or Peter himself, should also be provided in the form of e-mail.

Summary of users:

1. Peter (company CEO)
2. Any managers/supervisors
3. Regular employees

## Limitations of the current system

Based on the conducted interview and outline of the system, the following limitations can be concluded:

* Creation of invoices is cumbersome and requires tinkering with Excel spread sheets
* Invoices/quotes are not standardised, i.e. the layout and look of each quote differs
* Employees are managed on paper (sometimes not even formally recorded)
* No way of potential clients to request quotes without meeting Peter
* No way of organising tasks between employees, leading to disorganisation
* Materials must be found and purchased manually, and are not recorded on paper
* No way for employees to discuss current assignments out of the area of work

## Identification of user needs and acceptable limitations

Peter needs a system which solves the above limitations. The list of user needs and acceptable limitations may be summarised as follows:

* Manages any amount of current projects
* Automates the creation of quotes/invoices, with the capability of exporting to Excel
* Allows customers to request a quote using an interlinked online form. These “quote requests” will be placed in a queue and completed in the order received. The client will be e-mailed with the completed quote
* Provides an employee management system, with capability of assigning concrete tasks and different pay rates
* Provides an employee login system which lists assigned tasks, pay rate and easy contact with project manager through e-mail
* Manages the stock of materials, keeping track of the different types of material in stock and potential shortages based on the current projects to be completed
* Produces stock reports which list material lists/deficiencies
* Allows the exchange of messages between existing employeees

These are the main things which the new system should accomplish. Some other, less critical features, but ones which could be implemented as an extension to the completed system, are as follows:

* Allows new materials to be found through various online stores, through a search system which filters for best prices
* Allows account customisation through an adequate list of settings/preferences to suit user needs

## Data sources and destinations

|  |  |  |
| --- | --- | --- |
| Data | Source | Destination |
| Client details | Client (e-mail, phone, word of mouth) | Company job book |
| Job descriptions | Discussion between Peter and client | Company job book |
| Job costs | Peter | Client, company job book |
| Job materials | Peter | Client, company job book |
| Employee full name | Employee | Peter |
| Employee address | Employee | Peter |
| Employee contact number | Employee | Peter |
| Company business email | Peter | Client |

## 

## Data volumes

Data volumes will be analysed based on the information provided in the questionnaire and interview.

Top Builders works on about 3-5 projects during an average 3-month period. However, the business may soon expand, and Peter expects this to this number to increase to approximately 9-11 employees in about 2 years. It is therefore important to create a scalable system which is capable of managing the higher data volumes in the future. Taking 10 projects as an average volume of projects per quarter, this requires about 300 different individual jobs if the average project has 10 jobs. For these 10 projects, Peter indicated that he will hire more than 7 employees which will all need an account in the system. Peter indicated about 6-10 different types of materials are used for an average project, so 10 projects could mean about 60-100 different material requirement entries in the database. This doesn’t take into account the fact that a separate storage is needed for the entire material stock which could exceed about different 50 material types. Regarding quote requests, Top Builders receives less than 2 calls per week from a new customer. Therefore we can assume to receive about 1 online quote request per week, and this is likely to be higher as the future website increases its ranking on search engines and upon company expansion.

The database system to be used in this project will be MySQL. This is an industry-standard database, capable of executing queries at a very fast speed, and able to handle millions of rows of data. It is unlikely that the data stored in this project management system will ever exceed the capabilities or storage requirements of MySQL.

*Note: this project will not focus on creating an online website for Top Builders. The aim is to create a project management system, and the online quote request element will be a simple form designed to be “plugged in” to the (potential) future website, which I will not be creating. However, the database will be hosted online, to allow data to be stored in just one remote location and make it accessible from any machine. This is important, as employees will be logging into the system from their personal computers at home, and this would be impractical (or even impossible) without an online-hosted DB.*

## Analysis data dictionary

|  |  |  |
| --- | --- | --- |
| Field name | Data type | Details |
| Client full name | String | First and last name of a client |
| Project address | String | Street name, town, city and postcode |
| Project contact number | Integer | Preferably a mobile number on which the client of the project can always be reached |
| Project e-mail address | String | The main medium of contact between the client and the business, used to send quotes and other documents like construction plans, as well as general updates or enquires while the job is being completed |
| Job descriptions | String | Up to 150 words per job of a clear outline of what has to be done, e.g. “Fix new living room furniture” |
| Job costs | Real | The cost of an individual job |
| Job materials | String | The materials required to complete a particular job, e.g. for a kitchen refurbishment, a new counter top may be needed |
| Employee full name | String | First and last name of an employee |
| Employee address | String | Street name, town, city and postcode |
| Employee contact number | Integer | Preferably a mobile number on which the employee can always be reached |
| Employee e-mail address | String | Must be a valid e-mail address which is often checked, as it is used for a lot of business communications with the employee |
| Employee wage rate | Real | An hourly wage rate |
| Employee work hours | String | A specification of the concrete days on which the employee is expected to work as well as exact start/end times of shifts |

## 

## Data flow diagrams

The following are data flow diagrams, provided for both the existing and proposed systems, showing an overview of the context (level 0) as well as a detailed breakdown (level 1) of each system.

# Level 0 existing system DFDN:\git\BuildersCentral\BuildersCentral\COMP4 Project - Norbert Podsadowski\Level 0 DFD - Existing System.jpg **Level 1 existing system DFD**



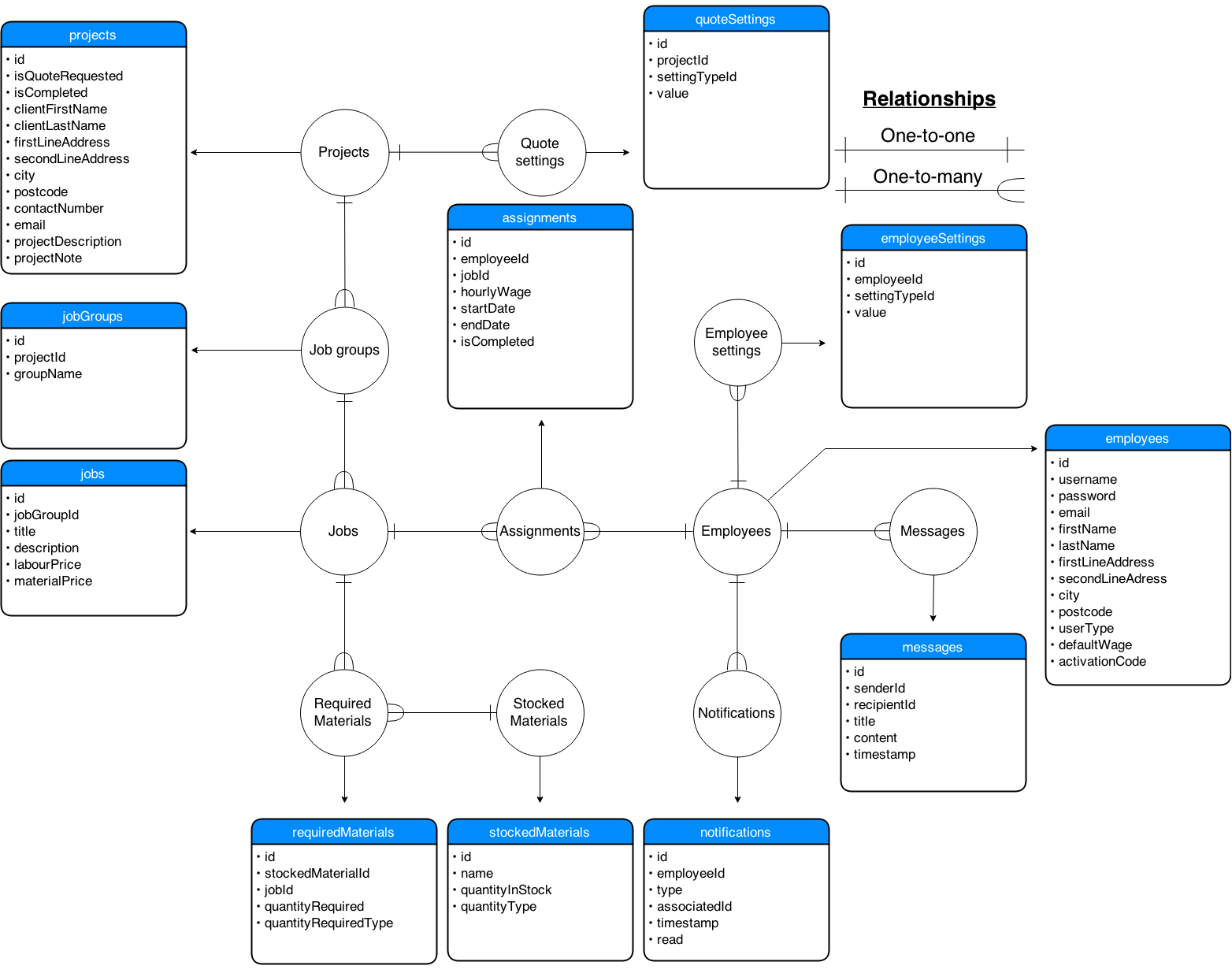
Level 0 proposed system DFD



Level 1 Proposed System DFD

Level 1 proposed system DFD 

Entity relationship diagram

The following ERD shows the planned database tables for the proposed system, displays their relationships to other tables and lists their individual fields. It is important to not confuse a “project” with a “job”. As an example, a kitchen refurbishment for a client is a *project* – the individual jobs belonging to this project could therefore be “paint walls” and “place tiles”, both with individual labour and material costs.

Notes regarding DB tables:

* Table *quoteSettings*: used to store Boolean values representing whether various columns in the quote should be displayed. The most simple, “generic” quote (with no additional settings enabled) will simply store a list of job titles and their corresponding prices. Peter may wish to enable other columns to suit the needs of a given project – for example, he might enable the materials column and list the materials required for each job. He could break this down further by enabling the materials price column for every job, or even the labour price column to distinguish the prices for both. He can also decide to group the jobs, as seen in the quote analysed earlier. If more detail is needed than just a job title, individual job descriptions may be also enabled.
* Field *isQuoteRequested* in table *projects*: used to indicate whether the given project is a quote request made online, or simply a new project created directly by Peter. This will be used to create a list of quote requests, distinct from a list of actual accepted projects. A quote request will supply most of the information that a project requires, and it is up to Peter to fill in individual jobs, costs and notes.
* Field *materialPrice* in table *jobs*: only used if the *splitTotalCost settings* is *TRUE.* If no listing of labour or materials is made and there is simply 1 type of cost per job (the total cost), then this cost is stored in the *labourCost* field. If both *splitTotalCost is TRUE,* then the total job cost will be calculated as *materialCost* + *labourCost*.

System objectives

**General objectives**

1. Allow customized quote/invoice creation and modification facility
2. Keep track of hired employees using a login system
3. Keep track of stored materials and their deficiencies
4. Provide an employee assignment facility
5. Provide an employee messaging facility
6. Provide a notifications system
7. Allow easy response to quote requests made by potential clients online

**Specific objectives**

1. Store user accounts for every employee, including their user type and personal details
2. Store projects, their corresponding jobs, groups, descriptions and material requirements
3. Store stock of materials and quantities, along with the type of quantity (meters, units, etc).
4. Store assignments of jobs across multiple employees, start and end dates, wages and completion status
5. Store messages sent between employees
6. Store any relevant user-specific settings or preferences
7. Process login details for employees, including password encryption
8. Process password change and recovery details
9. Process email changes
10. Process labour and material costs for quotes
11. Process estimated times of completion of projects based on assigned tasks and end dates
12. Process queue of quote requests, in chronological order
13. Process e-mails to be sent to employees
14. Process addition and deletion of quotes, employees, materials and quote requests
15. Process sending of messages
16. Process addresses and embed Google Maps imagery where appropriate
17. Process notifications of new messages, quote requests and assignments on employee home pages
18. Output formatted quotes in an Excel spread sheet, created programmatically, including content and styling
19. Output queue of quote requests received, displayed in a gallery-style with pictures (if attached)
20. Output messages received by other employees
21. Output stock reports
22. Output assigned tasks, including a list of co-workers with the same assignments
23. Output employee emails

Feasibility of potential solutions

**1. Improve existing system**

Storing clients and employees in designated, alphabetical files would greatly improve organisation, and prevent loss of data. Top Builders could invest in an index card organiser, allowing Peter to avoid memorising vital information (such as about his employees) which could be a risk to the business, especially after expansion. This improvement in organisation would make every entity in the system represented by a single expandable file, and allow quick searching for any given record.

Another, perhaps stronger improvement would be to use Peter’s familiarity with Excel and create a spreadsheet for his currently employed workers, their pay rates and contact information. This would allow quicker input, easier searching and a much lower risk of damage (as paper could be torn and lost) of data. Once the spreadsheet is created, its structure does require often changing, and provides a cost-effective way of greatly improving organisation. This spread sheet also be adapted to store the current stock of materials, although the effectiveness of this is questionable.

* Advantages
  + Very little associated cost
  + Can be accomplished very quickly
  + Reliability – Excel is a well-established and mature software package, unlike a potential bespoke solution
* Disadvantages
  + Storing materials on spreadsheets can be unreliable – constant modification of hundreds, potentially thousands of columns can lead to human error
  + All spread sheet and written solutions are subject to human error in both input (e.g. inputting a wrong value on a spreadsheet) and processing (e.g. later using a calculator and mistyping a digit).
  + Does not allow for any advanced processing (automatic e-mails, stock reports, employee messaging and assignment)
  + Difficult to scale – when the company gets large, having a lot records stored on paper greatly increases retrieval time, and potential loss/damage

**2. Purchase special-purpose software**

Various special-purpose software packages currently exist on the market. Top Builders could invest into one or more packages which could satisfy their user needs. A project management system, together with an employee tracking feature as well as a material stocking system would therefore be needed.

1. The Project Builder guides you through creation of the project
2. Create, edit or delete tasks and build phases with ease
3. Add sub-tasks to tasks and link related activities
4. Assign resources to tasks, sub-tasks and build phases
5. Add "Baselines" to track any amends to the project
6. Show planned vs actual progress on the job
7. Monitor allocation of resources, cash flow and revenue
8. Customise the look and feel of your charts
9. Export dates into iCalendar format to put into MS Outlook, Google Calendar etc
10. Print your chart, export to MS Excel or email to clients

One of these packages is called ProjectXpert by HBXL Building Software. It’s main features are quote from the website on the right.

* Advantages
  + Does not require a waiting period – the package can be bought and used instantly
  + Special-purpose software has been thoroughly bug-tested and matured over years, increasing their reliability
  + Often has a more complex feature set (can be a disadvantage, see below)
* Disadvantages
  + A single package does not meet all of the end user’s requirements
  + Multiple packages need to be bought which increases the overall cost
  + ProjectXpert alone costs £500 + VAT, and about 3 different packages are needed, adding a very heavy cost to a small family business with little capital
  + Most ready-made software will be bloated with features, most of which unnecessary to the end user
  + Due to these excessive lists of features, a non-technical CEO like Peter could have a hard time using the system, or take a very long time to adjust. Easy accessibility and a very friendly user interface are major requirements of the end user.
  + Many of these systems will not provide an remote, online database which multiple clients can connect to – rather, they are single packaged designed to be installed on one machine.

**3. Purchase a system from a bespoke software development company**

A bespoke system for Top Builders could be created, which aims to meet all of the end user’s requirements. Top Builders could hire a professional team of bespoke developers which would assess Peter’s needs again and design and implement a feasible solution. Later, this solution could be maintained, improved and tested – these services may come at an additional cost.

* Advantages
  + Can be made to fit the user needs very closely
  + The CEO can have a very large influence into the design of the software, personalising it to his needs
* Disadvantages
  + Very expensive
  + Some bespoke firms may go out of business or stop offering their services – leaving the system unmaintained in the long term
  + Extra charges for support, maintenance and features

**4. Create a bespoke system internally**

This would involve the same process as a software development company would go through. However, the system would be created entirely by myself.

* Advantages
  + Being Peter’s son, I will be able to customize the software more precisely to the user’s needs that in any of the other 3 possible solutions
  + No (or very minimal) costs involved
  + Can be supported and maintained in the long-term, as the software creator will always be around and capable of fixing bugs, adding features
* Disadvantages
  + Could take a significant amount of time to complete, given a 1 person team
  + Technical difficulties – although this would not be my first major programming project, I could sometimes run into problems which a professional bespoke software company would be able to resolve in a shorter amount of time
  + Depending on programming language/technologies used, could require learning of various new tools

Justification of chosen solution

Improving the current system simply does not satisfy enough needs of the end user. A major system requirement is the ability to easily created standardised, customisable quotes as well a managing stocks and quote requests. Without a computerised system, this is not possible. Furthermore, scalability is a major requirement of the new system – keeping all records on paper makes this very difficult for reasons outlined earlier.

While combinations of special-purpose software is likely to exist which meet the needs of the users, it is unlikely that there is a single package which satisfies all the precise needs listed. A combination of packages would therefore be needed to be purchased, and these packages can be very expensive. This is a major problem for a small firm like Top Builders which does not have a lot of operating capital.

Bespoke software allows the precise needs of the end user to be satisfied. It eliminates the need to deal with bloated, unneeded features which can confuse a non-technical audience (Peter and his employees). Peter wants a highly customised system and would like to play a major part in influencing its design, and this would be impossible without a bespoke system.

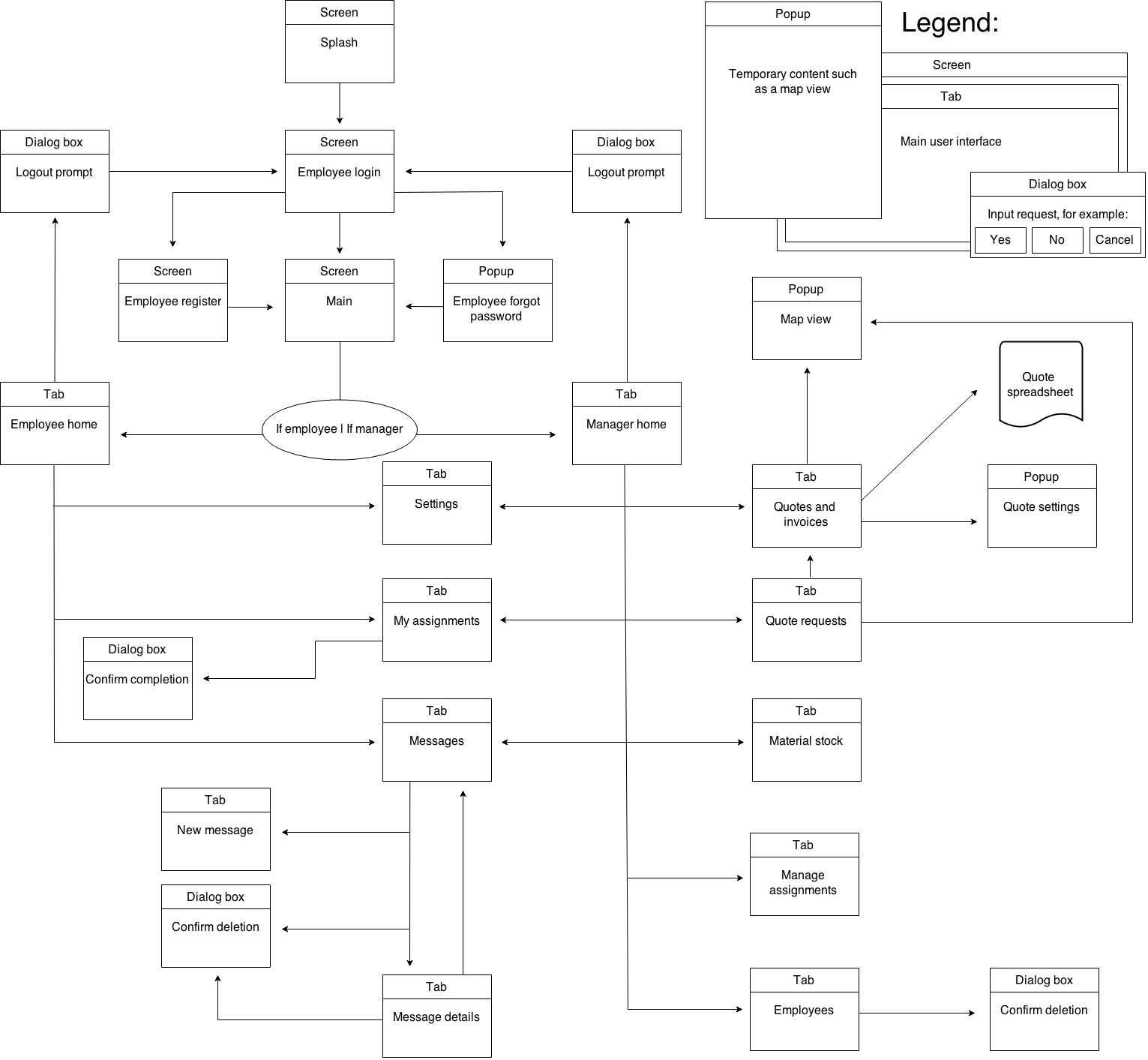
An internally created bespoke system is the most advantageous way of implementing the proposed system. It would be very cost-effective, as no professional software teams are hired, which can cost even more than a special-purpose software package “off the shelf”. A key advantage is the fact that the system is going to be created by the son of the main end user – through this, improvement and maintenance can be easily performed in the future, while a bespoke team would likely charge for such services. Although this solution would take the longest time to implement, this is not a major disadvantage as the company is only starting to expand – the current system is feasible for the current size of Top Builders, and a new system is not required right away. I have conducted various programming projects before, which were similar in size to the proposed system, and therefore I would be taking up this task with a good technical foundation.

In conclusion, it is decided that the best way for Top Builders to achieve a system capable of meeting their needs and objectives is an internally created piece of specialist software.

Design

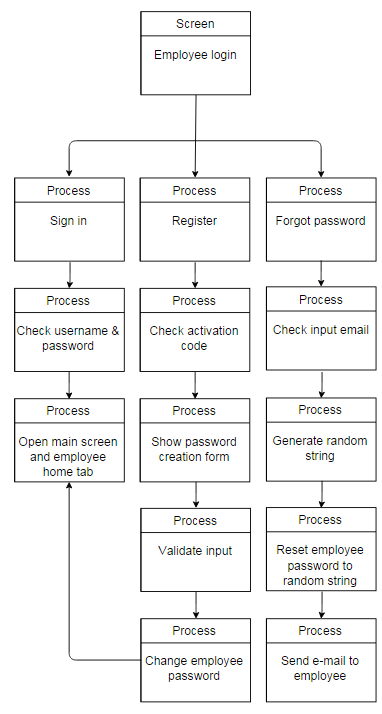
**Overall system design**

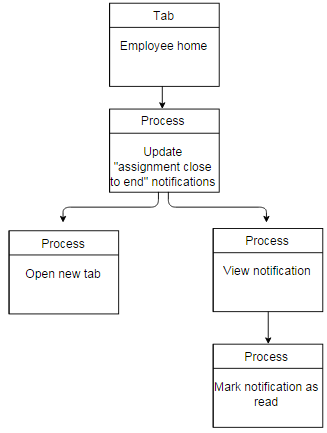
|  |  |  |
| --- | --- | --- |
| Inputs | Processes | Outputs |
| Log-in screen   * Login details * E-mail details (if forgot password)   Register screen   * Activation code   Projects tab   * Project details * Project settings   Employees tab   * Employee details * Employee settings   Material stock tab   * Material stock details   Manage assignments tab   * Assignment details   My assignments tab   * Assignment completion status   Messages tab   * Message details   Settings tab   * New password details (when changing) * New email details (when changing) | **Offline application processes**  **Log-in screen**   * Log in * Activate account * Reset password   **Projects tab**   * Add/delete project * Add/delete job group * Add/delete job * Add/delete required material * Export project to Excel spreadsheet * Save project * Change project settings   **Employees tab**   * Add/delete employee * Modify employee   **Material stock tab**   * Add/delete material * Modify material * Search for material   **Manage assignments tab**   * Browse assignment details * Search for assignments * Add/delete assignment * Modify assignment * Notify of assignment completion   **Messages tab**   * View messages * Read message * Add/delete message   **Settings tab**   * Change password * Change email * Change language * Save & apply settings   **Notification list**   * Add notification * Display notification * Delete all notifications for any given model * View notification   **Miscellaneous**   * View live map of any address * Send e-mail (for password reset and account activation purposes) * Log out   **Online processes**   * Submit new quote request * Upload quote request image | **Projects tab**   * Hierarchical display of all job groups, jobs and their required materials for any project * Material, labour & total costs * Client details * Project details * Project settings * Excel spreadsheet with project details   **Quote requests tab**   * Gallery-style list of requested quotes * Quote request details * Quote request image   **Employees tab**   * Table of employee details   **Material stock tab**   * Table of stocked material details * Material surplus/shortages based on current projects   **Manage assignments tab**   * Hierarchical display of all projects, job groups, jobs and finally their corresponding assignments * Assignment details for any job * Assignment details for any searched employee   **My assignments tab**   * Table of assignments for logged-in employee * Assignment completion status * Assignment co-worker details   **Messages tab**   * Table of messages send to logged-in employee * Details form for any message * New message form, including combo box of all possible recipients   **Settings tab**   * Current list of settings + language   **Notifications list**   * All notifications for any employee * Notification details based on the notification type, including appropriate notification icon * Computed days until assignment end date, for that type of notification   **Miscellaneous**   * Interactive Google map view * Any validation error lists * Any confirmation dialogs |

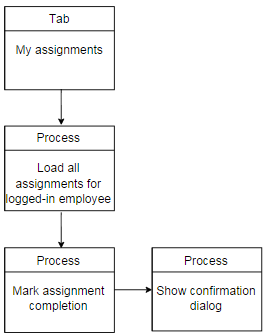
**Description of modular structure of system**

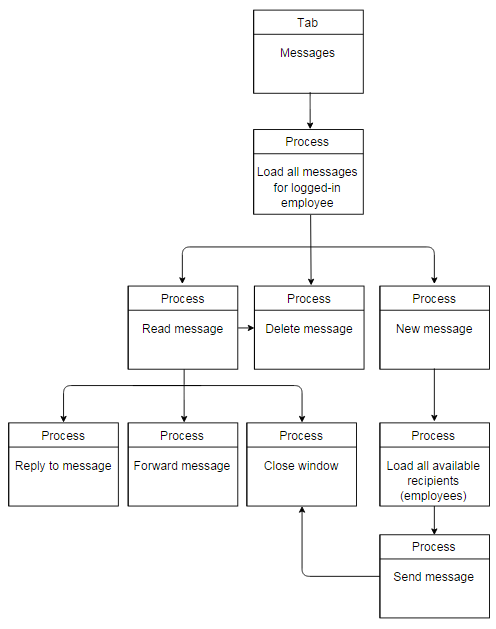
Below is a diagram of the overall planned modular structure of the entire system. Every planned screen, tab, popup window or dialog box is shown. Arrows represent possible transitions from one form to another.

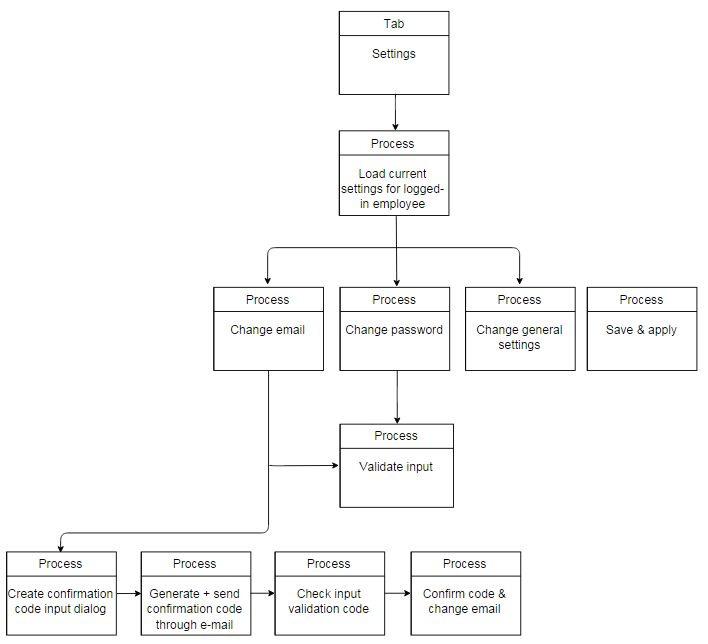
The list of diagrams below are individual, detailed modular diagrams for every single individual form (as shown above), displaying the processes involved with each form. Single arrows represent immediate transitions from one process to another on – arrows which split (at the same level) represent a choice of processes depending on user input.



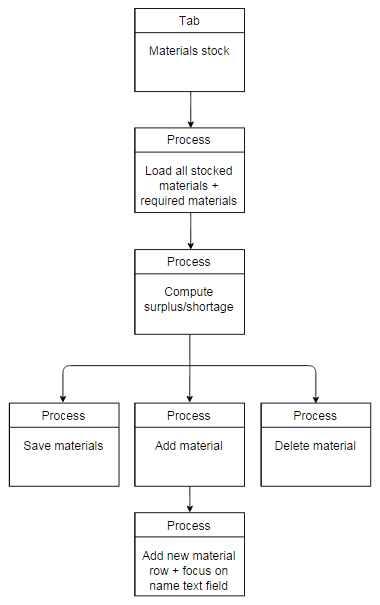


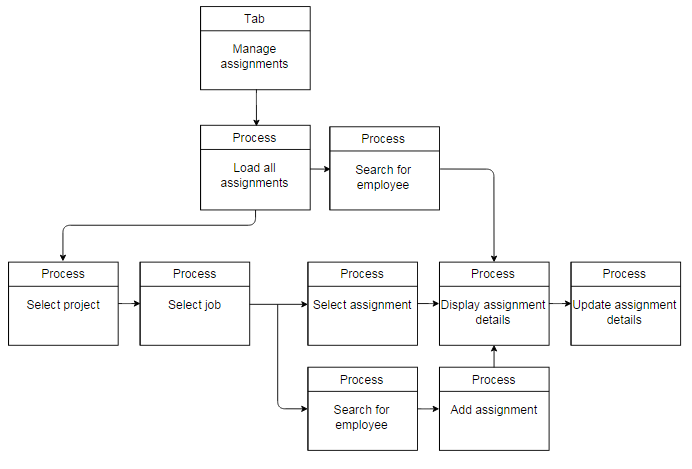


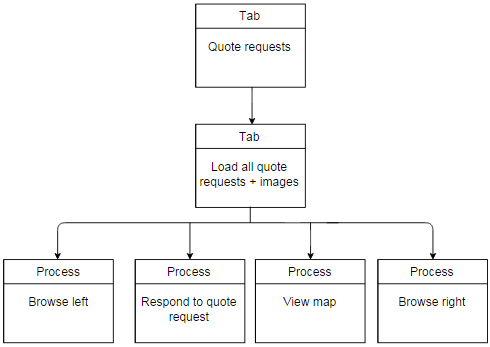


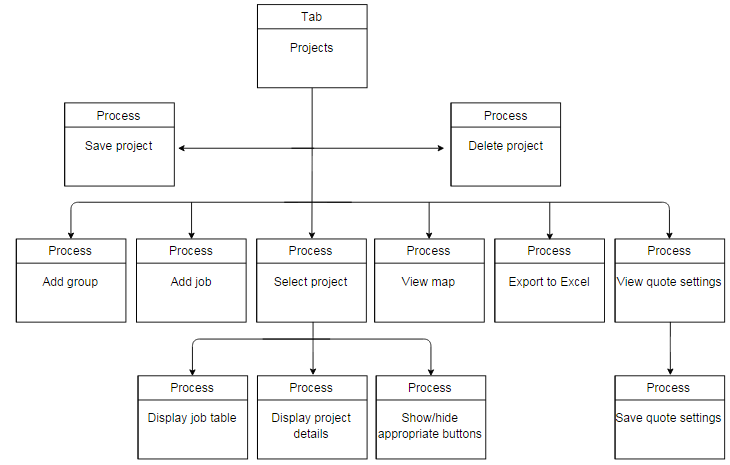












**Definition of data requirements (Design Data Dictionary)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: projects | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| quoteRequested | Bool | Yes | Presence check | Is present | No need, generated | True | Specifies whether this project was requested through the online form |
| completed | Bool | Yes | Presence check | Is present | No need, generated | False | Specifies whether this project has been completed |
| clientFirstName | String | No | Length check | Is 0-20 characters long | Please enter a shorter client first name | John | The first name of the person buying the services of Top Builders |
| clientLastName | String | No | Length check | Is 0-20 characters long | Please enter a shorter client last name | Smith | The second name of the person buying the services of Top Builders |
| firstLineAddress | String | No | Length check | - | - | 63 Green Road | The 1st line of the address where the project is to be completed |
| secondLineAddress | String | No | Length check | - | - | Thornton Heath | As above – 2nd line |
| city | String | No | Length check | - | - | London | The city of the address |
| postcode | String | No | Length check | Is 5-7 chars, irrelevant of spaces | Please enter a valid postcode | SW19 8HJ | The postcode of the address |
| contactNumber | String | No | Format check (regular expression) | Contains exactly 11 numbers | Please enter a valid contact number | 07545892134 | The contact number to be used when attempting to communicate with the client |
| email | String | No | Format check (regular expression) | Starts with any alphanumeric string, followed by ‘@’, followed by a similar string. Strings before and after the ‘@’ sign cannot start or end with a ‘.’ | Please enter a valid email address | email@domain.something.com | An alternative to the contact number |
| projectDescription | String | No | - | - | - | An entire bathroom refurbishment – plastering, tiling and painting, including the fixing of furniture | A description of the project – initially entered by the client (if requested online) and later modified by managers, if necessary. Not displayed on the quote – used to simply specify project objectives for the managers/employees |
| projectNote | String | No | - | - | - | Note: the total cost does not include the service charges of the asbestos company | A note entered by the managers, later to be displayed on the quote (unlike the description). Used to provide extra information about the quote or any other qualitative data |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: quoteSettings | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| projectId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – projects table. Indicates the project associated with this setting |
| settingTypeId | Int | Yes | Presence check | Is present | No need, generated | 0 - materialsEnabled  1 – splitTotalPrice  2 - jobDescriptionsEnabled  3 - groupsEnabled | Specifies the type of setting that this entry represents, as listed on the left |
| value | String | Yes | Presence check | Is present | No need, generated | True | The actual value of this setting. Example:  If the database has a quoteSetting entry with aprojectId of 2, settingTypeId of 2, and a value of false, that indicates:  “Project with the ID of 2 does not allow the entering of job descriptions” |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: jobGroups | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| projectId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – projects table. Indicates the project associated with this group of jobs |
| groupName | String | Yes | Presence check | Is present | Please enter a group name | Bathroom | Specifies the name of the group |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: jobs | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| jobGroupId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – jobGroups table. Indicates the group which this job is associated with. Has the value of 0 if “groupsenabled” is false (in the quotesettings table), in which case there are no groups |
| title | String | No | - | - | - | Strip out old tiles | A short title of what the job entails |
| description | String | No | - | - | - | May require filler | A more detailed description of the job, if necessary |
| labourPrice | Double | No | Type check | Is a numerical value | Please enter a valid labour price for job “[title]” | 120.50 | The cost of Top Builders labour to the client. If the labour price column and the material price column for this quote are disabled (as specified in the quoteSettings table), then this field is used as a combined “total” cost for this job |
| materialPrice | Double | No | Type check | Is a numerical value | Please enter a valid material price for “[title]” | 300 | The cost of materials to the client. If the material price column is disabled, this field is left blank |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: employees | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| username | String | Yes | Presence check, format check (regular expression), length check | Is present, is alphanumeric with the addition of “\_”, is 4-15 characters in length | Please enter a desired username for this employee. Usernames must be 4-15 characters long, contain only alphanumeric characters and the underscore. | npods | The user name used to log in to the system. Users will be created by Top Builders managers and therefore inputted into a form, so need to be validated |
| password | String | Yes | Presence check, format check (regular expression), length check | Is present, is in the range of 6-20 characters, contains at least one capital letter and at least one number. Must be alphanumeric with the addition of “\_” | Please enter a valid password. Passwords must be 6-20 characters in length, contain only alphanumeric characters and the underscore, and include at least one capital letter and at least one number. | Mystrongpassword\_2124 | The password user to log in to the system. Initially empty, and first created by employee upon the activation of their account. Can later be changed |
| email | String | Yes | Format check (regular expression) | Starts with any alphanumeric string, followed by ‘@’, followed by a similar string. Strings before and after the ‘@’ sign cannot start or and with a ‘.’ | Please enter a valid email address | email@domain.something.com | The e-mail address of this employee. Has to be valid and in use |
| firstName | String | Yes | Presence check | Is present | Please enter this employee’s first name | Harry | The first name of the employee |
| lastName | String | Yes | Presence check | Is present | Please enter this employee’s last name | Baker | The last name of the employee |
| firstLineAddress | String | No | - | - | - | 71 Some Road | The first line of address of the employee |
| secondLineAddress | String | No | - | - | - | Thornton Heath | The seond line of address of the employee |
| city | String | No | - | - | - | London | The city of the employee |
| postcode | String | No | - | - | - | SW19 8NB | The postcode of the employee |
| userTypeId | Int | Yes | Presence check, range check | Is present, is in the range 0-2 | No need, controlled by dropdown | 0, 1, 2 | The type of this employee’s user account.  0 = CEO, gives all abilities, including the creation of managers.  1 = manager, provides the ability to do everything but modify managers.  2 = regular employee, provides only capabilities of checking assignments and messages |
| defaultWage | Double | No | Type check | Is a numerical value | Please enter a valid default hourly wage | 8.50 | The default hourly wage received by this employee for the completion of assignments. This value will be the default value displayed upon the creation of assignments, but can be modified on a per-assignment basis |
| activationCode | String | No | Length check | Is 12 characters long | No need, generated | Ahj31k518scj | A 12-character code generated upon the creation of an employee (by a manager). This code is given to the employee, who can use the “activate” option on the login screen. Doing this will allow them to choose their own password upon activation. An account is activated if this field is empty – otherwise, it needs activation, and logging in is impossible |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: employeeSettings | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| employeeId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – projects table. Indicates the project associated with this setting |
| settingTypeId | Int | Yes | Presence check | Is present | No need, generated | 0 - fullscreenEnabled  1 - language | Specifies the type of setting that this entry represents, as listed on the left |
| value | String | Yes | Presence check | Is present | No need, generated | True | The actual value of this setting. Example:  If the database has a employeeSetting entry with an employeeId of 2, settingTypeId of 0, and a value of true, that indicates:  “Employee with the ID of 2 has fullscreen enabled” |

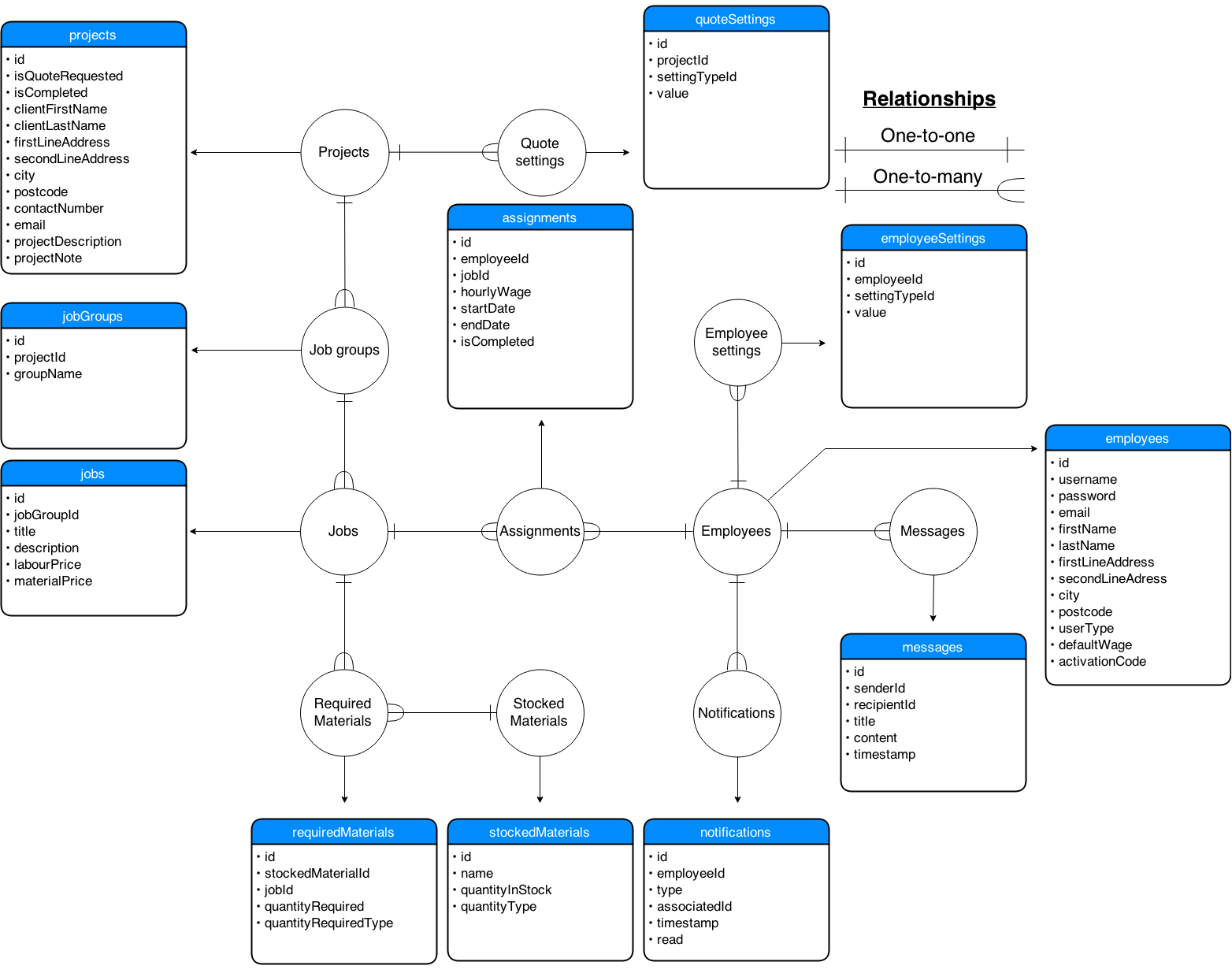
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: assignments | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| employeeId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – employees table. Indicates the employee which has been assigned with this task |
| jobId | Int | Yes | Presence check | Is present | No need, generated | 7 | Foreign key – jobs table. Indicates which job needs to be completed for this assignment |
| hourlyWage | Double | No | Type check | Is a numerical value | Please enter a valid wage for “[employee name]” on assignment “[job title]” | 6.50 | The hourly wage given to the employee while working on this assignment. The default value of this will be taken from the employee’s table (field defaultWage) |
| startTimestamp | BigInt | No | Format check | Must be a valid date in the format DD/MM/YYY | Please enter a valid start date | 1211235122 | The estimated date when work on this assignment should begin, saved as a Unix Timestamp. This timestamp will be generated based on the input of the user. Saving the timestamp as a completely numerical value allows for much more flexibility and easier computation of durations as well as handling of different time zones. |
| endTimestamp | BigInt | No | Format check | Must be a valid date in the format DD/MM/YYYY | Please enter a valid end date | 1267231124 | The estimated date when work on this assignment should be completed |
| isCompleted | Bool | Yes | Presence check | Is present | No need, controlled by tick box | False | Specified “true” by the employee once the assignment has been finished |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: messages | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| senderId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – employees table. Indicates the employee which has sent the message |
| recipientId | Int | Yes | Presence check | Is present | No need, generated | 7 | Foreign key – employees table. Indicates the employee which has received the message |
| title | String | Yes | Presence check | Is present | Please enter a message title | New construction plans | A brief title of the message |
| content | String | Yes | Presence check | Is present | Please enter some message content | Sample text | The main contents of the message |
| timestamp | Long | Yes | Presence check | Is present | No need, generated | 1416423221 – this corresponds to  Wed, 19 nov 2014 18:53:41 gmt | Indicates the time and date when this message was sent. This is in the format of a Unix timestamp, where the value represents the number of seconds that have elapsed since January 1, 1970, midnight GMT. This is easily convertible to a human date when necessary and allows for a wider range of acceptable date formats, as well as simplifies date and time storage in the database |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: notifications | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| employeeId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – employees table. Indicates the employee which has received this notification |
| typeId | Int | Yes | Presence check, range check | Is present, is in the range 0-2 (for normal employees) or 0-5 (for managers and CEO) | No need, generated | Valid for all users:  0 – new message  1 – new assignment  2 – assignment close to end date  Valid for managers:  3 – new quote request  4 – employee completed an assignment | This specifies the nature of the notification. This will be used to decide how the notification will be displayed in the notifications box on the home page of the employee. It will also determine which table the associatedId is referring to |
| associatedId | Int | Yes | Presence check | Is present | No need, generated | 7 | Foreign key – the table which this key points to will be determined by the “type” field above. If the type field is equal to the given value, then this field points to any valid ID in the given table.  0 – messages table  1 – assignments table  2 – assignments table  3 – projects table  4 – materialStock table  5 - employees table  For example, if “type” is 0, specifying that this notification is notifying an employee of a new message, then this field (associatedId) can be the id of any existing message in the messages table, which points to the new message received. Essentially, this field holds a foreign key to any of the messages, assignments, projects, materialsStrock or employees table based on what it is notifying to the employee |
| timestamp | Long | Yes | Presence check | Is present | No need, generated | 1416423221 – this corresponds to  Wed, 19 Nov 2014 18:53:41 GMT | Indicates the time and date when this notification was received. This has the format of a Unix timestamp, where the value represents the number of seconds that have elapsed since January 1, 1970, midnight GMT |
| seen | Bool | Yes | Presence check | Is present | No need, generated | True | Indicates whether or not the employee has seen this notification. A notification will be marked as “seen” whenever the employee opens the home page and sees the notification for the first time |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: stockedMaterials | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| name | String | Yes | Presence check | Is present | Please enter a name for this material | Cement | The name of this material |
| quantityInStock | Double | Yes | Presence check, type check | Is present, is a numerical value | Please enter a valid quantity for this material | 30.5 | The quantity in stock of this particular material. The unit of this is specified in the field below |
| quantityTypeId | Int | Yes | Presence check, range check | In the range 0-15 | No need, controlled by dropdown | 0 – units (e.g. 10 buckets)  1 – m  2 – m^2  3 – m^3  4 – cm  5 – cm^2  6 – cm^3  7 – mm  8 – mm^2  9 – mm^3  10 – l  11 – ml  12 – cl  13 – kg  14 – g  15 - mg | Specifies the type of quantity required – units, meters squared, etc. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table name: requiredMaterials | | | | | | | |
| Field name | Data type | Required | Validation type | Validation rules/details | Error message | Sample data | Description and key fields |
| id | Int | Yes | Uniqueness check, presence check | Is unique, is present | No need, generated | 3 | Primary key |
| stockedMaterialId | Int | Yes | Presence check | Is present | No need, generated | 5 | Foreign key – materialsStock table. Indicates the material required for the corresponding job (specified by jobId) |
| jobId | Int | Yes | Presence check | Is present | No need, generated | 7 | Foreign key – jobs table. Indicates the job for which the associated material (specified by materialId) is required |
| quantityRequired | Double | No | Type check | Is a numerical value | Please enter a valid quantity required of material “[material name]” for job “[job name]” | 10 | The required quantity of this material for the job to be completed. Is 0 when the quantity of this particular material has not been specified |

**Description of record structure**

The record structure mirrors the entity relationship diagram described in the analysis section. A relational database will be used to map every single entity onto an individual table. Foreign keys will be used to form relationships between them, all of which are shown below.

**Validation required**

**Validation types**

In order to ensure that the data saved into the database is valid at all times, there will be numerous validation processes used throughout the system. Validation will be performed in one of two ways:

* On-the-fly validation – validation performed while the user is inputting text
* Post-submission validation – validation performed after the user submits the data

The first type of validation will occur as soon as the user begins typing text into a text field. For example, when inputting the phone number during the creation of a new employee, the user will be prevented from entering any characters other than digits into the text field. This type of validation allows the user to instantly realise when wrong input is being provided, rather than informing them upon the submission of the input – this saves time and is much more intuitive than displaying an error message only once the form has been submitted.

Some types of validation, however, require tasks which are more computationally expensive than simple length or character checks, and/or may only be performed after form being filled is submitted. One example of this is format checks for e-mails – one may only check the format of an e-mail after a user inputs it in full, and therefore such a check will be done only once the user presses the appropriate submit button (for example, the button which adds a new employee once all of their details have been filled out). This type of validation displays a list of all validation errors which occurred after submission. This differs from on-the-fly validation which does not display any messages and merely prevents wrong data being inputted into a text field in the first place.

**Specific validation required**

The specific validation required for every data field was listed in the earlier Data Dictionary section. Below is a list of regular expressions which will be used to perform certain types of validation.

E-Mail expression

**[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?**

This regular expression is one of the many standard expressions designed to validate e-mail. It prevents the user from entering an e-mail without an at @ sign, missing out the domain, placing a dot in the wrong place, etc. It also includes checks for invalid domain names (domain names without a top-level extension), etc.

Password expression

**(?!^[0-9]\*$)(?!^[a-zA-Z]\*$)^([a-zA-Z0-9]{4,15})**

I designed this expression to only accept passwords which are between 4 and 15 characters long and contain at least 1 number, 1 letter and no special characters.

Numerical text-field algorithm

This is a type of on-the-fly validation which will be used on all text fields which require the user to input a numerical (real) value. The “key down” event will be captured and the character which was typed will be captured from the event and then tested. The following algorithm will only allow digits to be entered, as well as up to a maximum of one “.” character.

**procedure** onKeyTyped(event) {

character 🡨 event.character;

currentText 🡨 textField.text;

**//If the character is a dot, and one already exists, cancel the event**

**//Also cancel the event if there are no characters currently entered (the input cannot begin with a dot)**

**if** (character = “.”) {

**if** (currentText.contains(“.”) || currentText.length = 0) {

event.cancel;

}

}

**//Also, if the character isn't a dot but is anything but a number, cancel the event**

**else** **if** (!character.matchesRegex("\\d\*")) {

event.cancel;

}

}

**Database design including normalised relations**

**Un-normalised**

(projectId, isQuoteRequested, isCompleted, clientName, projectAddress, contactNumber, email, projectDescription, projectNote, quoteSettingId, projectId, settingTypeId, value, jobGroupId, groupName, jobId, title, description, labourPrice, materialPrice, employeeId, username, password, email, fullName, userType, defaultWage, activationCode, languageSetting, assignmentId, hourlyWage, startDate, endDate, isCompleted, messageId, senderId, recipientId, title, content, timestamp, notificationId, type, associatedId, timestamp, seen, stockedMaterialId, name, quantityInStock, quantityType, requiredMaterialId, jobId, quantityRequired, quantityRequiredType)

**First normal form**

1. **projects**

(projectId, isQuoteRequested, isCompleted, clientName, projectAddress, contactNumber, email, projectDescription, projectNote, projectId, quoteSettingId, projectId, settingTypeId, value)

1. **jobs**

(jobId, projectId, groupName, title, description, employeeId, labourPrice, materialPrice, employeeId, hourlyWage, startDate, endDate, isCompleted)

1. **notifications**

(notificationId, employeeId, type, associatedId, timestamp, seen, senderId, recipientId, title, content)

1. **materials**

(stockedMaterialId, name, quantityInStock, quantityType, requiredMaterialId, stockedMaterialId, jobId, quantityRequired, quantityRequiredType)

1. **employees**

(employeeId, username, password, email, fullName, userType, defaultWage, activationCode, languageSetting)

**Second normal form**

1. **projects**

(projectId, isQuoteRequested, isCompleted, clientName, projectAddress, contactNumber, email, projectDescription, projectNote, quoteSettingId, projectId, settingTypeId, value)

1. **jobs**

(jobId, jobGroupId, title, description, labourPrice, materialPrice)

1. **jobGroups**

(jobGroupId, projectId, groupName)

1. **employees**

(employeeId, username, password, email, fullName, userType, defaultWage, activationCode, languageSetting)

1. **assignments**

(assignmentId, employeeId, jobId, hourlyWage, startDate, endDate, isCompleted)

1. **notifications**

(notificationId, employeeId, type, associatedId, timestamp, seen, senderId, recipientId, title, content)

1. **materials**

(stockedMaterialId, name, quantityInStock, quantityType, requiredMaterialId, stockedMaterialId, jobId, quantityRequired, quantityRequiredType)

**Third normal form**

1. **projects**

(projectId, isQuoteRequested, isCompleted, **clientFirstName**, **clientSecondName**, **firstLineAddress**, **secondLineAddress**, **city**, **postcode**, contactNumber, email, projectDescription, projectNote)

1. **projectSettings**

(quoteSettingId, projectId, settingTypeId, value)

1. **jobs**

(jobId, jobGroupId, title, description, labourPrice, materialPrice)

1. **jobGroups**

(jobGroupId, projectId, groupName)

1. **employees**

(employeeId, username, password, email, **firstname**, **lastname**, userType, defaultWage, activationCode, languageSetting)

1. **assignments**

(assignmentId, employeeId, jobId, hourlyWage, startDate, endDate, isCompleted)

1. **messages**

(messageId, senderId, recipientId, title, content, timestamp)

1. **notifications**

(notificationId, employeeId, type, associatedId, timestamp, seen)

1. **stockedMaterials**

(stockedMaterialId, name, quantityInStock, quantityType)

1. **requiredMaterials**

(requiredMaterialId, stockedMaterialId, jobId, quantityRequired, quantityRequiredType)

**Sample of planned SQL queries**

This project will be a fully object-orientated solution, following a Model – View – Controller (MVC) pattern. This has many advantages, as will be explained later (in the object definitions section) – one of them includes the fact that SQL queries may be kept very simple and concise. The reason for this is when a model is loaded into a system (a model being an employee, a stocked material, a project, etc.) all of its data is loaded with it for future access.

Any access of foreign keys is done by looking up the model with the foreign ID (in the current loaded models list) and then accessing the required fields – the model is only loaded from the database if an instance of it isn’t currently loaded (unless we are forced to always load from the database, i.e. when updating quote requests, since they may be modified outside the system). Unlike in a non-MVC system, this prevents SQL queries from becoming excessively complex, which greatly increases performance and reduces the strain on the server hosting the database.

Another advantage resulting from the use of this design pattern is that SQL queries rarely have to be re-written throughout the source code of the project. In fact, the AbstractModel class will be responsible for every single query ran in the system. By implementing a very abstract, general query method, classes which are children of AbstractModel are able to easily run queries on their corresponding table in the database.

Below is a list of 6 general SQL queries which will be used throughout the system. The system will use prepared statements with parameterised queries, and the reason for this will be explained in the system security section later on. Question marks (?) represent values which will be replaced at the time when the query is executed.

**1. Load all models from a given table**

*table\_name*will be replaced by the child class deriving from the AbstractModel class by the name of the table which stores all the records for that model.

SELECT \* FROM *table\_name*

**2. Load all models from a table given certain conditions**

*field1, field2, field3…* will be replaced by the arguments passed to the query execution method in AbstractModel.

SELECT \* FROM *table\_name* WHERE *field*= ? AND *field2* = ? AND *field3* = ?...

**3. Load all models from a table, sorting records by a given field**

*asc\_or\_desc* will be replaced by ‘ASC’ or ‘DESC’ depending on the boolean value passed to the query execution method in AbstractModel.

SELECT \* FROM *table\_name* WHERE *field* = ? ORDER BY *asc\_or\_desc*

**4. Delete model with given ID**

DELETE FROM *table\_name* WHERE id = ?

**5. Update model**

Below is the specific update query which will be used for the assignment model. The following is an example of an update query which will be modified (particularly the italicised parts) on a per-model basis, covering all the required properties of a given model – each model will define this update query in its update method.

UPDATE *assignments* SET *employeeId*=?, *jobId*=?, *hourlyWage*=?, *startTimestamp*=?, *endTimestamp*=?, *isCompleted*=? WHERE id = ?

**6. Insert model**

Below is the specific insert query which will be used for the assignment model. The following is an example of an insert query which will be modified (particularly the italicised parts) on a per-model basis, covering all the required properties of a given model – each model will define this insert query in its insert method.

INSERT INTO *assignments* (*employeeId*,*jobId*,*hourlyWage*,*startTimestamp*,*endTimestamp*,*isCompleted*) VALUES (?,?,?,?,?,?)

**Identification of storage media**

The backbone of the system will be stored on a MySQL database. This database will be hosted on an online server, allowing the system to access stored data from any machine. I am currently the owner of a Virtual Private Server hosted on DigitalOcean – it uses a Solid State Drive which provides very fast read and write speeds compared to standard HDDs. I am currently on a $5/month plan, so the server specifications are not especially bold – but when the system is moved into a production environment, Peter may invest in a better server, if necessary. The current server includes:

* 1 core processor
* 512MB memory
* 20GB SSD storage
* 1TB bandwidth
* Ubuntu GNU/Linux operating system

The server is configured to run as little unnecessary applications as possible in order to maximise resource dedication to the MySQL database. This should be sufficient to allow about 10 users logged in to the system at any one time. The server also has a MySQL management tool installed called PHPMyAdmin which provides optimization and table modification features. It also ensures that the system is easily scalable in the future – if a change of server hosting is required, the database has to simply be exported using PHPMyAdmin and imported at the new location.

The 20GB of storage space is more than enough to accommodate for a huge amount of records stored in the system, which include mostly text, and perhaps a few images for the quote requests. If need be, old records can be purged once in a while if more storage is required.

**Identification of suitable algorithms for data transformation**

Notes on pseudo-code syntax used in this section:

* Code wrapped inside { and } represent the beginning and end of execution blocks, respectively
* Functions return a value, procedures do not return anything
* The dot . operator represents member field access (e.g. employee.password is accessing the password field of the employee object)
* Comments are prefixed with // (although more in-depth commentary is provided in the implementation section)
* All pseudo-code is written to be as language-independent as possible. It uses an imperative style which hides any language specific details (such as type declarations, exact method naming, database query syntax etc.)
* All pseudo-code concerns data transformation algorithms only. Common tasks like create new windows, layout out dialog boxes, saving models into the database etc. are not concerned with data transformation and will exists in static classes available across the entire application – the implementation of these will be listed in the implementation section.

**Employee register scene**

**//Called when the user inputs an activation code and clicks “next”**

**procedure** next() {

code 🡨 codeTextField.text;

**if** (code <> “”) {

valid 🡨 checkActivationCode(code);

**if** (valid) {

displayPasswordScreen;

**return**;

}

}

displayError(“Invalid activation code!”)

}

**//Called when the user picks a password (on the second screen) and clicks “activate”**

**procedure** activate() {

newPass 🡨 newPassTextField.text;

confirmPass 🡨 confirmPassTextField.text;

**if** (newPass > 0) {

**if** (newPass = confirmPass) {

set employee password to the MD5 hash of newPass;

remove activation code for employee;

save employee;

login employee;

**return**;

} **else** addError(“Passwords must match”);

} **else** addError(“You must input a password”);

}

**Employee login scene**

**function** login(inputUsername, inputPassword) {

execute SQL: get employee data where username = inputUsername

**if** (query returned result) {

create employee;

load employee from query data;

load employee corresponding settings;

**if** (employee.password = inputPassword) {

set current employee and resource bundle (global variables)

return employee;

} **else** **throw** invalid password exception;

}

}

**Update home screen**

**procedure** updateAll() {

employee 🡨 getCurrentEmployee();

**//Update all “close to end” assignments, so that the correct date may be shown**

**for** **each** assignment **in** employee.asssignments {

assignment.updateAssignmentCloseToEndNotification();

}

notificationsList.clear;

**for** **each** notification in employee.notifications {

notificationsList.add(notification);

}

**if** (notifications = 0)

{

Add label to notifications list, informing the use that there are no notifications;

}  
}



**Settings tab**

**//Note: when setting the new password, it must first be converted to the MD5-hash equivalent**

**procedure** changePassword() {

employee 🡨 getCurrentEmployee();

**//Check current password**

**if** (employee.password <> currentPasswordTextField.text) {

addError(“Invalid password”);

}

**//Check new password format**

**if** (invalid new password format using regex) {

addError(“Invalid format of new password”);

}

**//Check new password match**

**if** (newPasswordTextField1.text <> newPasswordTextField2.text) {

addError(“Passwords must match”);

}

**//If no errors occurred**

**if** (errors = 0) {

employee.password = newPasswordTextField1.text;

save employee;

showInfoDialog(“Your password has been changed.”);

}

}

**procedure** changeEmail() {

employee 🡨 getCurrentEmployee();

**//Check new email format**

**if** (invalid new email format using regex) {

addError(“Invalid email”);

}

**//Check new email match**

**if** (newEmailTextField1.text = newEmailTextField2.text) {

addError(“E-Mails must match”);

}

**//Check if new email exists**

**if** (newEmailTextField1.text already exists in database) {

addError(“That e-mail is already taken”);

}

//If no errors occured

**if** (errors == 0) {

**//Generate a random 10-character verification code**

generatedCode 🡨 *generateRandomString*(10);

**//Send this code to the user's e-mail address**

employee 🡨 getCurrentEmployee();

send email to employee with generatedCode;

**//Create a dialog window with an info label, text field and a confirm button**

create dialog window;

set behaviour upon clicking confirm button within dialog window {

**//If the inputted code equals the generated code**

**if** (confirmTextField.text = generatedCode) {

**//Change the email**

employee.email = newEmail1.text;

save employee;

**//Reset input boxes in settings tab**

newEmailTextField1.text 🡨 “”;

newEmailTextField2.text 🡨 “”;

currentEmailTextField.text 🡨 newEmail1TextField.text;

**//Close the dialog and open a confirmation dialog**

close current window;

open confirmation dialog, confirming the e-mail change;

} **else** {

**//Change the info label to an error message**

infoLabel.text 🡨 “Invalid activation code.”;

}

}

}

}

**Quotes tab**

**procedure** addGroup() {

project 🡨 getSelectedProject;

newGroup 🡨 create new jobGroup object;

newGroup.projectId 🡨 project.id;

newGroup.groupName 🡨 newGroupTextField.text;

project.jobGroups.add(newGroup);

}

**procedure** addJob() {

jobGroup 🡨 getSelectedJobGroup;

newJob 🡨 create new job object;

newJob.groupId 🡨 jobGroup.id;

newJob.title 🡨 “Job title”;

newJob.description 🡨 “Job description”;

jobGroup.add(newJob);

}

**procedure** saveProject() {

project 🡨 getSelectedProject;

**if** (all fields validated) {

**if** (project.newModel = **true**) {

add new project option to the project combo-box list;

save project;

}

}

}

**procedure** deleteProject() {

project 🡨 getSelectedProject;

display confirmation dialog;

set behaviour upon clicking the confirm button {

delete project;

}

}

**Employee model**

**function** checkPassword(inputPassword) {

if (password = md5hash(inputPassword)) {

return true;

} **else** {

return false;

}

}

**function** getFullName() {

fullName 🡨 firstName + “ “ + lastName;

return fullName;

}

**function** checkUsernameExists(String checkUsername) {

result 🡨 execute SQL: load all employees where username = checkUsername;

**if** (result contains at least 1 record) {

return true;

} **else** {

return false;

}

}

**function** checkEmailExists(String checkEmail) {

result 🡨 execute SQL: load all employees where email = checkEmail;

**if** (result contains at least 1 record) {

return true;

} **else** {

return false;

}

}

**function** checkActivationCode(String code) {

result 🡨 execute SQL: load all employees where activationCode = coe;

**if** (result contains one record) {

return true;

} **else** {

return false;

}

}

**Job model**

**//If the project is set to split total cost into material and labour cost, this returns the labour cost plus the material cost – otherwise it returns just the labour cost (which acts as the total cost)**

**function** getTotalCost() {

total 🡨 0;

**if** (settings.splitPrice = true) {

total 🡨 labourPrice + materialPrice;

} **else** {

total 🡨 labourPrice;

}

**return** total;

}

**//Returns a string of the required materials, comma-separated if the argument is true**

**function** getRequiredMaterialsString(commaSeperated) {

materials 🡨 “”;

**for each** material **in** requiredMaterials {

materials 🡨 materials + material;

**if** (commaSeperated = true) {

materials 🡨 materials + “, “;

} **else** {

materials 🡨 materials + “\n”;

}

}

**//To get rid of the last unnecessary comma and space or line-break**

trim off last 2 characters from materials string;

**return** materials;

}

**Assignment model**

**//Returns a string of employee names which have an assignment for the same job**

**function** getRequiredMaterialsString(commaSeperated) {

coWorkers 🡨 “”;

**for each** assignment **in** job.assignments {

**if** (assignment <> this assignment instance) {

coWorkers 🡨 coWorkers + assignment.employee.name + “\n”;

}

}

**//To get rid of the last unnecessary line-break**

trim off last 2 characters from co-workers string;

**return** materials;

}

**Project model**

**//Gets all the jobs associated with this project, across all job groups**

**function** getAllJobs() {

jobList 🡨 create new list object;

**for each** jobGroup **in** jobGroups {

jobList.add(jobGroup.getJobs);

}

**return** jobList;

}

**//If the project is set to split total cost into material and labour cost, this returns the labour cost – otherwise it returns 0**

**function** calculateTotalMaterialCost() {

total 🡨 0;

**if** (projectSettings.splitPrice = **true**) {

**for each** job **in** getAllJobs {

total 🡨 total + job.materialPrice;

}

}

**return** total;

}

**//If the project is set to split total cost into material and labour cost, this returns the material cost – otherwise it returns 0**

**function** calculateTotalLabourCost () {

total 🡨 0;

**if** (projectSettings.splitPrice = **true**) {

**for each** job **in** getAllJobs {

total 🡨 total + job.labourPrice;

}

}

**return** total;

}

**function** calculateTotalCost() {

total 🡨 0;

**for each** job **in** getAllJobs {

total 🡨 total + job.calculateTotalCost;

}

**return** total;

}

**Notification model**

**procedure** deleteCorrespondingNotifications() {

}

**Class definitions and details of behaviours**

**General system architecture**

The system will be implemented in the Java programming language. Java is a multi-platform, object-orientated, general-purpose language with a very large standard library and is one of the most used programming languages around the world. The system will be usable on any operating system (Windows/Mac/Linux/etc) which the Java Virtual Machine installed. This includes a vast majority of computers, given that Java is used by a wide range of applications. My previous experience in Java allows me to deliver a functional system which is not only bug-free but also scalable, maintainable, and comprehensible to other developers.

**Graphical User Interface library**

The system will need to make use of a GUI library to facilitate the creation, display, and management of the user interface. For this task, I have chosen the JavaFX library. Initially, it was a separate library, but since the latest release of Java 8, it has been merged into the standard library which ships with the Java Software Development Kit.

JavaFX provides a wide range of graphical components, and follows a very modern design pattern with an easy-to-use API. One feature of JavaFX is that components may either be created in code, or using a special flavour of XML (Extensible Markup Language) called FXML. FXML allows the developer to transfer layout code into a structured markup format which greatly speeds up development time and is very intuitive to use.

*Note: this system will not be created using a WYSIWYG (What You See Is What You Get) editor such the one provided by Borland Delphi or Visual Basic. No code generation will occur – the entire implementation, including the GUI components, will be created manually in code. The reason for this is that while WYSIWYG editors lower the barrier to entry to new developers, they greatly limit the full possibilities of the platform and almost always generate code which is not optimized or unreadable (“ugly code”). In real industries, WYSIWYG editors are rarely used.*

**JavaFX properties**

The JavaFX library provides a mechanism which encapsulates the primitive Java types (int, double, char, String) into classes which represent properties (IntegerProperty, DoubleProperty, StringProperty, etc). The components provided with the library are designed especially for this pattern and rely on the developer to create models without primitive types, but with the defined JavaFX property types. The advantages of this include:

* The capability to bind one property to another to ensure that they are kept synchronised at all times (when one changes, the other does too)
* As a result of the above, components may stay synchronised with their corresponding models and other components (e.g. changing a row in a table will update the property in the actual model object rather than just the table itself)

All of the models which I will define for this project will be defined in terms of JavaFX properties so that the above advantages may be exploited.

**Model-View-Controller pattern**

MVC is a software architectural pattern for implementing user interfaces. JavaFX loosely follows such a pattern.

* Models define objects stored in the database and their relevant behaviours
* Views define the actual user interface layout displayed to the user
* Controllers link models to views by deciding which components to update based on what model data

In the JavaFX context, models become classes which map the database structure on to JavaFX property classes, along with all their required access methods and various behaviours. Views become the actual GUI designs implemented in FXML code, and controllers become classes which have direct access to the components defined in FXML and decide how to manipulate their contents.

Furthermore, JavaFX adds a further component to this pattern, namely Cascading Style Sheets (CSS). CSS is normally used on the web to style the components of a web page once its layout has been defined in an HTML file. Similarly, JavaFX uses CSS to define how components of the GUI defined in an FXML file should be styled. JavaFX provides a wide range of properties which can be set on components to fully customize the look and feel of the application – colours, fonts, backgrounds, etc. This is very important in implementing the desired Top Builders colour scheme.

The separation of models from view layouts and styling allows for much easier maintenance and scalability in the future, a very important notion in software engineering.

**Class inheritance and model loading**

As I am making use the MVC pattern described above, it is clear that I will have to define a collection of model classes, one for each table in the database, including all of the relevant fields (refer to the ERD diagram) and behaviours. Each row in the database will be mapped to a separate class instance of the model corresponding to that row’s table. I therefore have to think about how the actual fields stored in the database will be loaded into the object and how all of the defined properties will initially be set.

I have opted to take advantage of Java’s inheritance features in order to create a general, AbstractModel which will serve as a base parent class for all other defined models. This class will define the basic operations which will be common to all model classes, such as save, load and delete. Every model class will also have the common “id” property, as every single database table has the common “id” primary key – hence AbstractModel will also define this. Finally, AbstractModel will have to manage a Boolean flag, such as “newModel”, to distinguish between models which have only just been created (and therefore have to be INSERTed upon saving) and models which have already been loaded from the database (and therefore have to be UPDATEed upon saving).

A further consideration which has to be made is the following: once a model is loaded from the database, it would be ideal if no other duplicate instance of this model exists anywhere during the runtime of the application. The reason for this is mainly to prevent the application from manipulating two instances of the same model, which would result in inconsistencies across different forms, for example, or the corruption of data. To illustrate this, imagine we are currently working inside the projects tab and add a required material to one of our jobs. If we then switch to the stocked materials tab and change the name of the same material which we added, that change will not be reflected until the project tab is closed and re-opened again (to allow the system to query the database and create the new instance of the updated model).

In order to prevent this, the model loading class will have to be implemented in a way which checks whether a model currently being loaded has already been loaded – if not, we create a new instance – otherwise, we simply return the existing instance. One way to implement this check is to iterate through the list of loaded models and check their IDs – if the ID of the model we are currently checking is equal to the ID of the model we are attempting to load then we have found an existing instance.

The specific implementation of the loading class is given below. The code uses reflection (an advanced programming technique which allows an object to examine objects at run-time rather than compile-time), in order to gain access into the static getModels() method inside the class which we are attempting to load (this class will be passed to the method itself). As the method is quite complex, comments are provided to explain various parts if it for clearer understanding.

1. //Returns a list of model instances based on the given result of a query. The class corresponding to the models must be supplied to make use of reflection
2. **protected** **static** <T **extends** AbstractModel> ObservableList<T> loadList(ResultSet result, Class<T> clazz) {
3. //Create a list to store loaded models which we will eventually return
4. ObservableList<T> models = FXCollections.observableArrayList();
6. **try** {
7. //Go through all loaded rows
8. **while** (result.next()) {
9. //Create a variable to hold the model object which we will eventually return, and initialise it to a new instance of the class
10. T model = clazz.newInstance();
12. //Create a variable to store the currently existing models
13. ObservableList<T> modelList = **null**;
15. **try** {
16. //Invoke the static method getModels() inside the desired class
17. modelList = (ObservableList<T>) clazz.getMethod("getModels").invoke(**null**);
18. } **catch** (IllegalArgumentException | InvocationTargetException | NoSuchMethodException | SecurityException e) {
19. //Print an error if one occurred
20. Log.error("Error loading list of models " + clazz, e);
21. }
23. //Try to find an existing model witin the model list
24. **for** (AbstractModel existingModel : modelList) {
25. //If a model ID matches the ID of the current row in the result set, then set the model variable which we are returning to the existing model object
26. **if** (existingModel.getId() == result.getInt("id")) {
27. model = (T) existingModel;
28. }
29. }
31. //If no existing model was found (i.e. if the model variable still has an ID of 0, which would be the case as we created a new instance of the class)
32. **if** (model.getId() == 0) {
33. //Load the row data into the new model instance
34. model.loadFromResult(result);
36. //Add the model to the list of existing models
37. modelList.add(model);
38. }
40. //Add the model to the list of models which we are returning
41. models.add(model);
42. }
43. } **catch** (SQLException | InstantiationException | IllegalAccessException e) {
44. //Print an error if one occurred
45. Log.error("Error loading list of models " + clazz, e);
46. }
48. //Return the models list consisting of existing instances of the required models or new instances based on the logic above
49. **return** models;
50. }

The above method is one of the most crucial parts of the entire application and will be called extensively through the runtime of the application. It is a static method, and will be accessible to all models which inherit from AbstractModel.

**Package structure**

A common Java practice is to organise class files into packages. These packages simply represent the directory structure of the source code. It is important to organise source code in a logical structure in order to ensure the system is and easy to maintain in the future.

The conventional package name starts with a reversed domain name of the organisation building the software (e.g. com.google). Freelance developers tend to use a name pointing to their online code repository on a version control service like GitHub. I have created an online repository for this system at <https://github.com/Norbo11/TopBuilders>. This will allow me to easily commit changes from any machine and work on the implementation of the system both at home and at school.

The package structure which will be used in this system is as follows:

* **com.github.norbo11.topbuilders** (top-level package)
  + **controllers** (package consisting of all controller classes)
    - **custom** (controllers for custom components)
    - **scenes** (controllers for scenes, forms, windows)
    - **tabs** (controllers for tabs)
  + **css** (package consisting all CSS files)
  + **fxml** (package consisting of all FXML files)
    - **scenes** (FXML files for scenes, forms, windows)
    - **tabs** (FXML files for tabs)
  + **models** (package consisting all models)
    - **[root]** (contains all models)
    - **enums** (enumerated types related to models)
    - **exceptions** (exceptions related to models)
  + **util** (package consisting of all utility classes)
    - **[root]** (contains miscellaneous classes)
    - **factories** (factory classes used to customize JavaFX components)
    - **helpers** (static functions for global use by the system)

**Class volumes**

It is difficult know exactly how many classes the system is going to use. However, I can make a fairly accurate prediction based on the descriptions of the design patterns made above.

* Every single scene and tab will require a separate controller class. There are currently 9 distinct tabs planned, and 8 distinct scenes. In addition to this, I will use an AbstractController and AbstractTab classes to allow these various controller classes to inherit common properties from these parent classes to avoid code duplication and take advantage of Java’s polymorphism. This gives a total of about 19 classes concerning controllers.
* Every model will also require a corresponding class. There are currently 11 planned database tables, resulting in about 11 distinct model classes, and potentially more if I decide to make abstract parent classes for certain models.
* Enumeration classes will be required to define elements of the system which may be “enumerated” into a logical group of items. For example, a quantity type enumeration will list all the possible quantity types which the system may accept, or a notification type enumeration will list the possible notification types. Based on the structure of the database, I predict about 5 classes will be necessary for this purpose.
* Various custom components will have to be created, and the corresponding controller classes for these components. The exact scenarios in which a custom component is most appropriate will have to be decided during development time – for now we I may predict about 3-5 custom component classes.
* One of JavaFX’s design patterns is the use of “factories” in conjunction with many of its components. Factories allow the developer to fully customize the behaviour of certain components, which will be very important in allowing the system to perform all of its required functions. For example, making a row display its text in a different colour based on a certain condition in a table will require a factory. I can predict about 7-10 factory classes will be necessary for this purpose.
* Utility and helper classes with static methods will also be required – these classes will contain functions which will need to be usable by multiple classes in the entire system. In order to keep the code structured and maintainable, I will separate these functions into logical groups, e.g. an EmailUtil class to provide e-mail sending methods or a HashUtil class to provide hash encryption methods. I predict about 15 classes will be necessary for this.
* Other classes, such as the Main entry class or a class to store global constants etc. will also be necessary, adding about 3-5 extra classes to our list.

This gives a total of about 63-70 distinct classes, all with their own properties and behaviours. Given such a large number of classes, it would be impractical to list all of them below. Not only that, but the exact behaviour of many of the classes, such as the utility or factory classes, may only be known during development time, and unable to be determined at the design stage – the reason for this being that these classes are highly dependent on the technical implementation of the system. For these reasons, the below class definitions are only a sample, showing only a list of classes where their behaviour can be relatively accurately predicted.

**Class defintions**

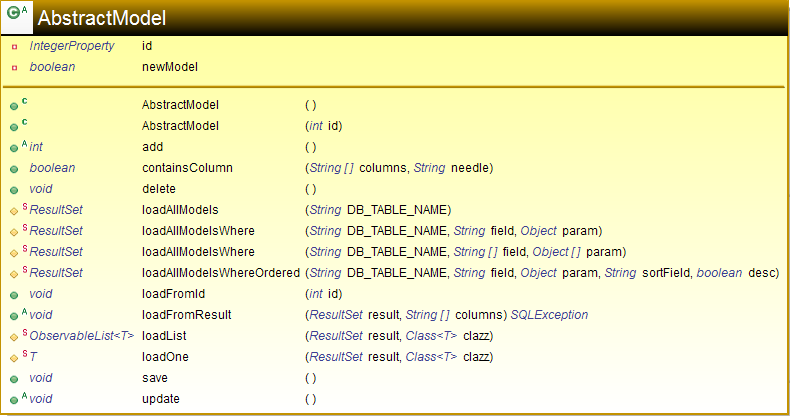
The class definitions are given as a list of properties, and then a list of methods which directly affect the behaviour of the object. Primitive access methods (getters and setters for the primitive values contained inside property objects), as well as property access methods (which return actual property objects themselves) are not included to avoid unnecessary clutter; for every property listed in the top section of the diagram, one can infer 3 extra methods which have not been listed (the getter, setter, and property access method for that property). Proposed arguments for methods as well as return values are also provided.

**AbstractModel**

Below is the proposed AbstractModel class. It contains a constructor which creates either a blank model or initializes it with an ID. It then has 3 primary abstract instance methods, covering 3 of the SQL queries mentioned earlier – add, update and delete. The add and update methods are marked as abstract, forcing all subclasses of AbstractModel to provide their own implementation – this implementation will include simply running the correct INSERT or UPDATE query based on the properties of the implementing model. The delete method is not marked as abstract as every model will usually have the same delete query. The add and update methods should never be called on their own – when a model needs to be saved, the save method should be called instead, which will decide which of the two are appropriate to execute based on whether the newModel boolean flag is true (if it is, add is called; otherwise, update is called).

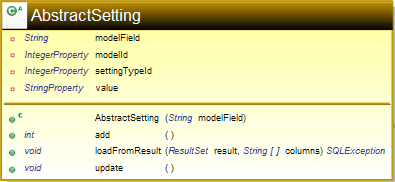
AbstractModel also provides 6 static methods – 4 of which concern the remaining 3 SQL queries which deal with SELECTing the actual models from the database (1 of those methods is just an overloaded method, allowing multiple fields and parameters to be passed in the form of an array). The other static method, loadList, returns a list of models loaded from the given SQL ResultSet – this implementation was described earlier. loadOne simply redirects to the loadList method, but instead returns only 1 model rather than a list (for purposes of cleaner code in the future).

Finally, the loadFromResult method is an abstract method responsible for populating properties with loaded data, which subclasses of AbstractModel will have to override as well.



**AbstractSetting**

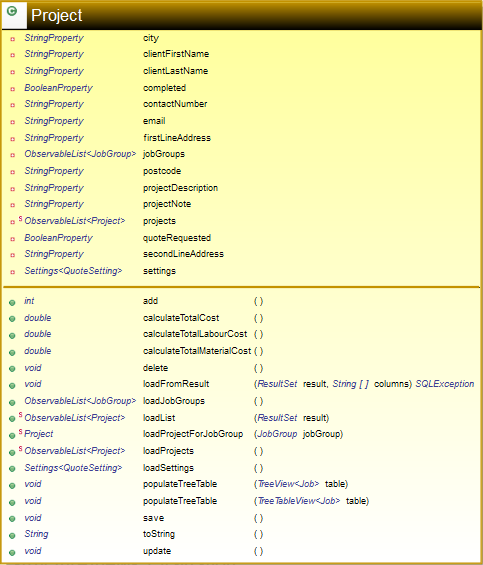
As the employeeSettings and quoteSettings follow a very similar structure, needless code duplication can be avoided by creating just one AbstractSetting class which extends AbstractModel and provides the behaviour for those two tables.

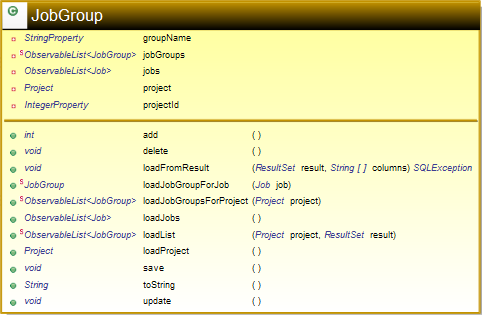


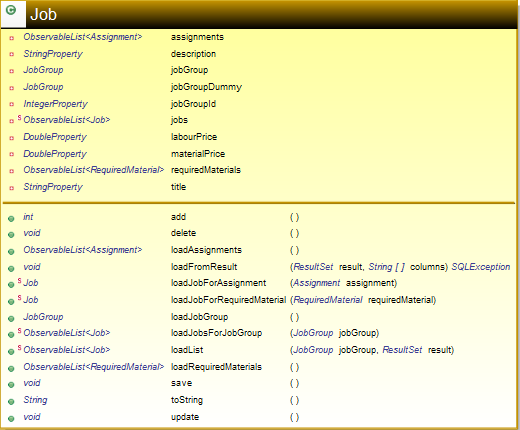
**All models**

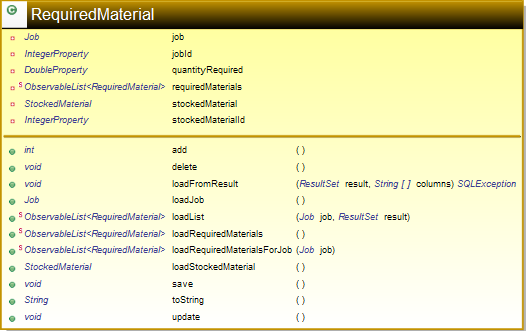
The following diagrams concern all of the other models in the system and list their proposed properties and methods. All of the following classes extend the AbstractModel class for reasons described earlier.

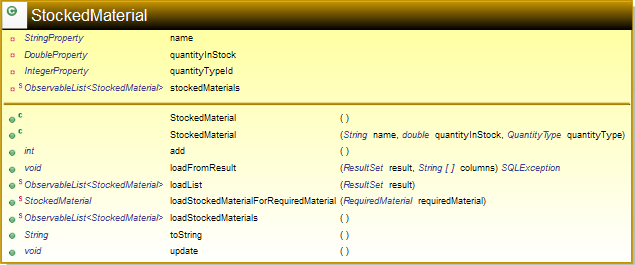


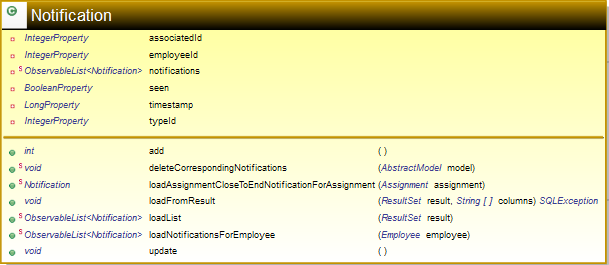


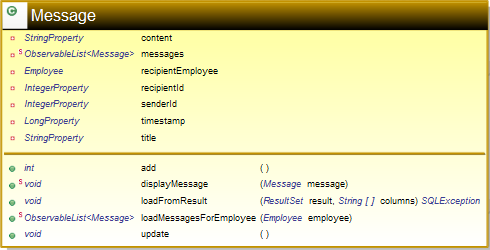


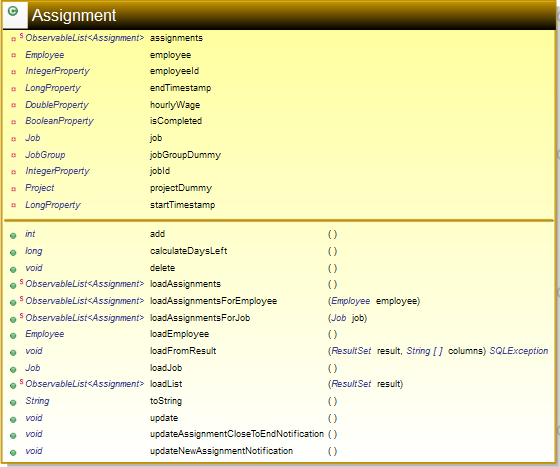


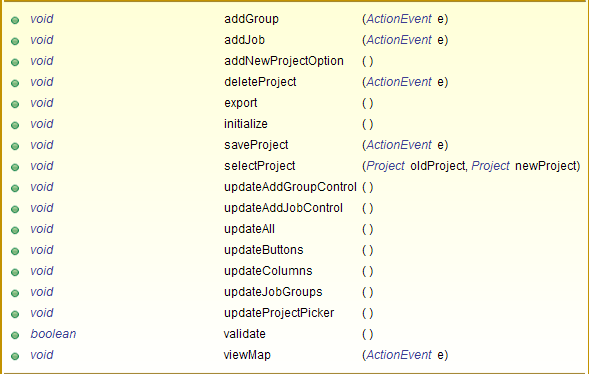
****

****



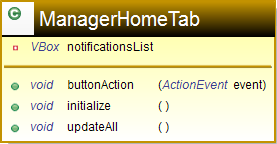


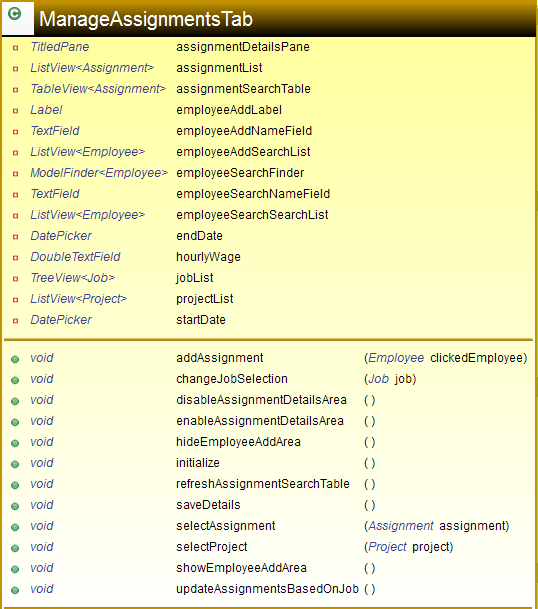
****

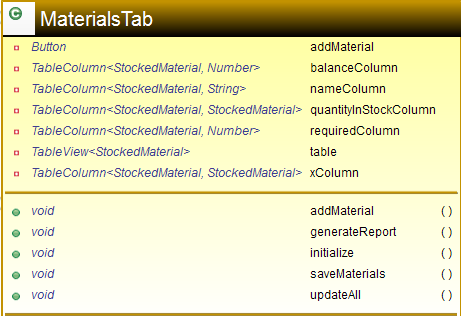




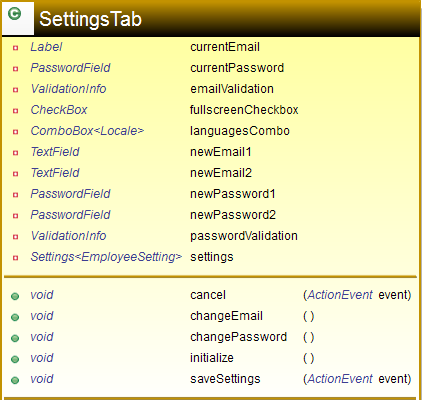


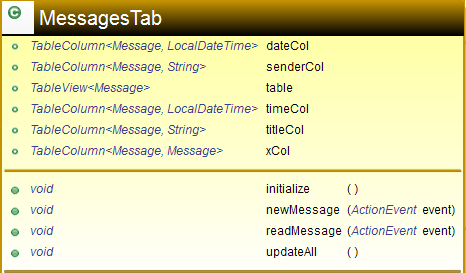


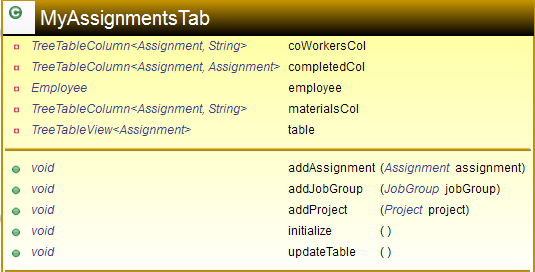












**User interface design (HCI) rationale**

**Layout**

The system will have a considerable amount of input/output forms to perform all of its various functions. It is likely that the end user will have to switch between these forms on a regular basis. As a result, I have opted for a tab-based design. The main user interface will be encapsulated in a large, main screen with a status bar on the bottom as well as a “welcome area” on the top. Between the top and bottom components will reside a tab pane, listing all the currently open tabs. The main navigation “home” tab will always be visible and never closable to the user, and act as an entry point to all other tabs in the system. The home tab will differ based on the current user type (manager or employee).

These design considerations allow very quick and easy access to multiple forms at the same time. Users do not have to close the form they are currently using in order to quickly use another part of the system, which is a huge boost to productivity.

Some items of interest, such as notifications and sections in the settings tab will be collapsible. Such interactivity allows the user to organise their current workspace by hiding items which are irrelevant, freeing up layout space and promoting greater focus on the things which are most useful to the user at the time.

The layout of the system will use a clear, logical order of controls. For example, the form which adds new employees will ask for their first and last name before asking for their address or contact details. This is important as users will expect to input information in the order which makes the most sense, and in the order which is standard across any input forms across the world, digital or otherwise.

**Logo**

The logo will be present inside the top component of the main form. It will also be present as a small window icon commonly present next to the close, minimise and maximise buttons provided by the operating system. Both of these considerations provide quick at-a-glance identification of the application.

**Colour**

The main colour theme present throughout the project will be based on the Top Builders base corporate style. From the logo and details provided during the interview we can infer that blue and red are the dominant colours used by the firm and this will be reflected in the design of the project management system.

The system will use a light shade of blue as the main background colour for all forms and tabs. This light blue will be contrasted with a slightly darker shade to make controls of interest like buttons and other clickable items stand out. To make table rows distinguishable, the system will use an alternating row colour, using a white background for odd table rows and a very light shade of blue for the even items.

**Font**

The main font family which will be used is “Segoe UI”. This is a very clear, easy-to-read font which looks very professional and elegant. All text will be of a reasonable size and important text will be made a bit bigger in order to stand out. For example, text on button will be bold.

**Icons**

Icons provide an incredibly fast visual aid which immediately tells the user the nature of the control or item of text which they are looking at. The primary use of icons in this system will be in the notifications panel – the three types of notification (quote request, assignment and message) will all have a unique icon which will be displayed inside the notification item.

Another use of icons will occur in the quote requests tab, with small arrow icons acting as buttons which allow the user to browse the list of current quote requests in a gallery-style tab.

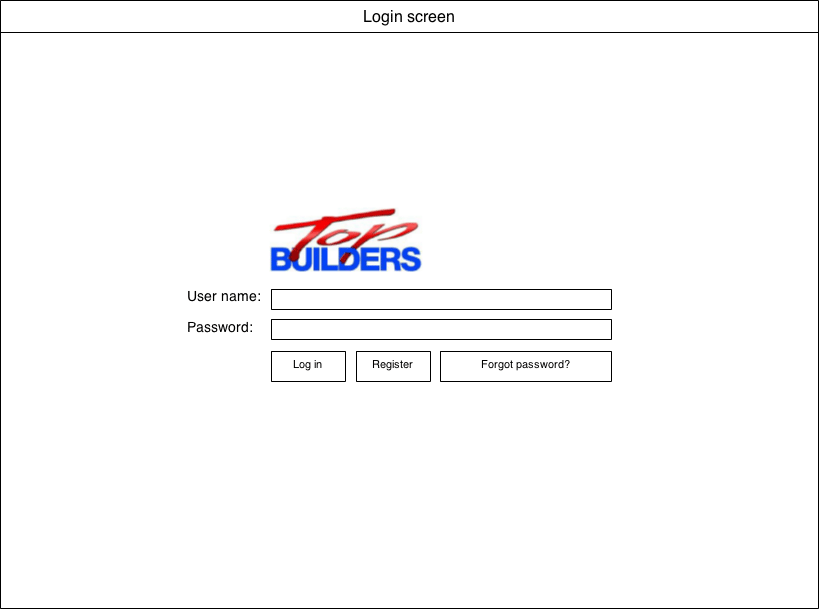
**UI sample of planned data capture and entry designs**

**Global forms (available to all user types)**



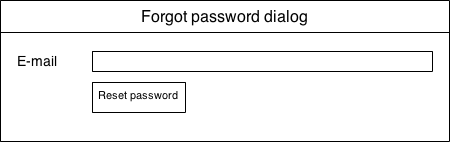
3 second splash screen

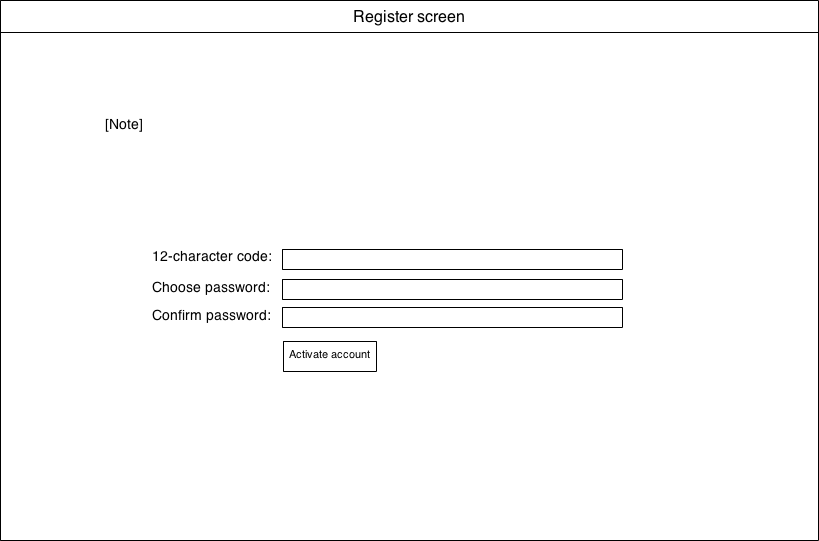
White background



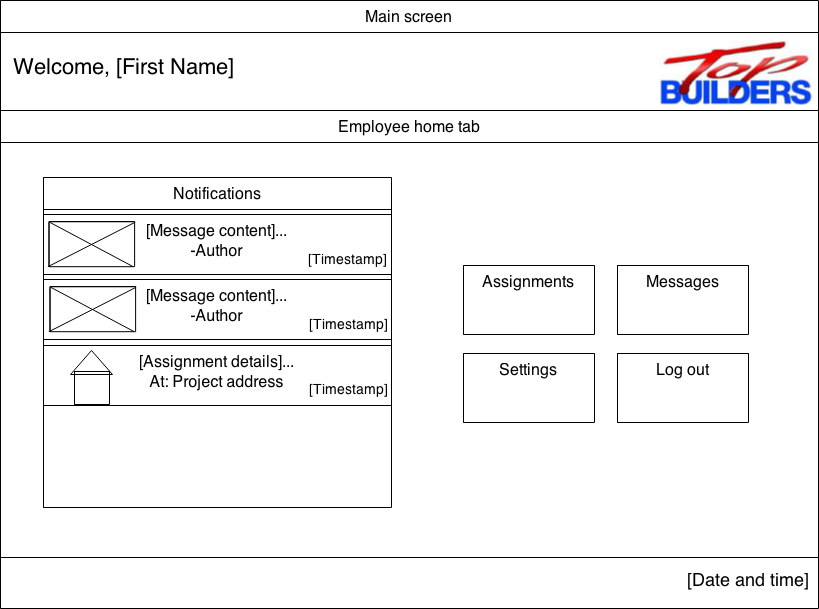
Buttons will have dark blue, bold text, “Segoe UI” font.

Blue button background, darker border. The same button colour scheme will be used throughout the whole system





A note explaining to the user how to use their activation code which they received from their employer



Same background colour

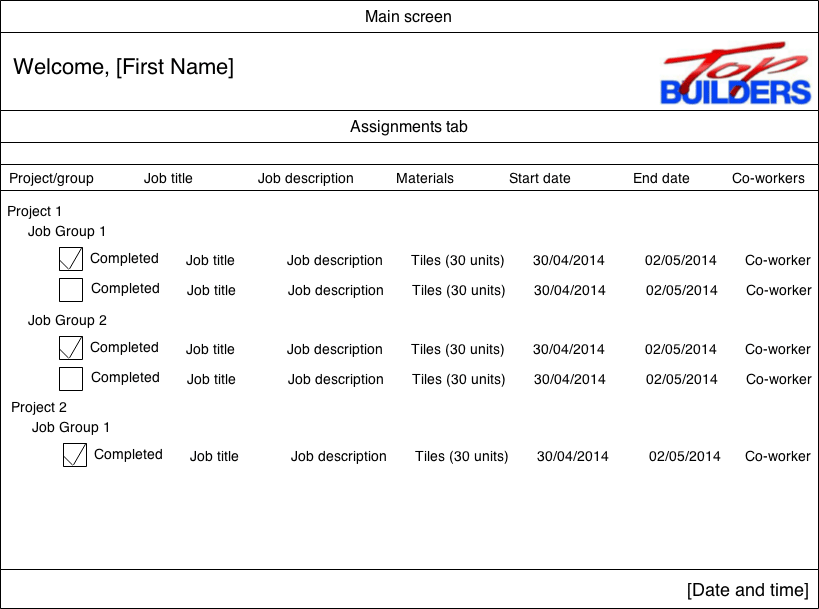
Bottom area. Blue background

Top heading/welcome area, white backgruond

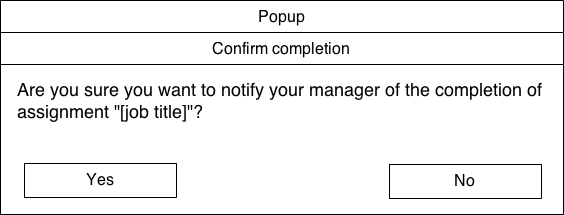
Main tab area. Very light blue background

List of open tabs displayed here

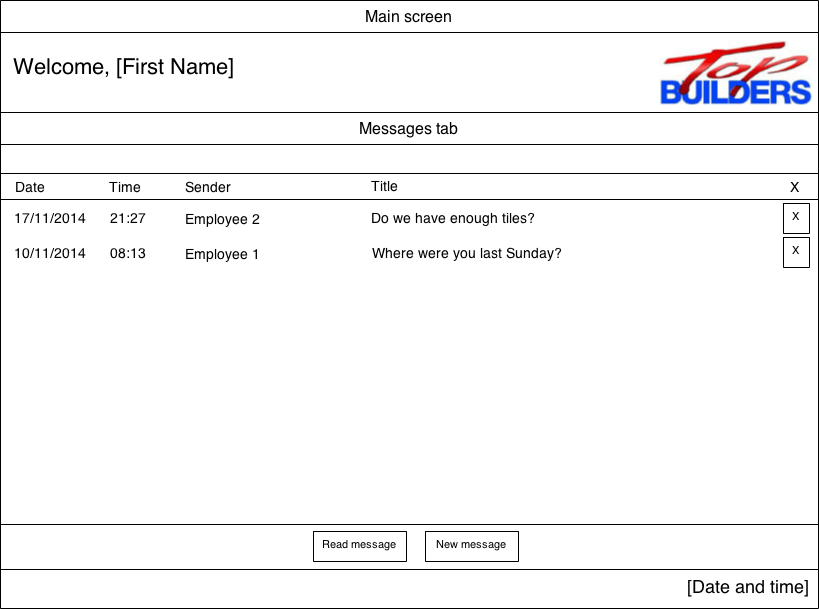
List of notifications (scrollable), with icons corresponding to the notification type



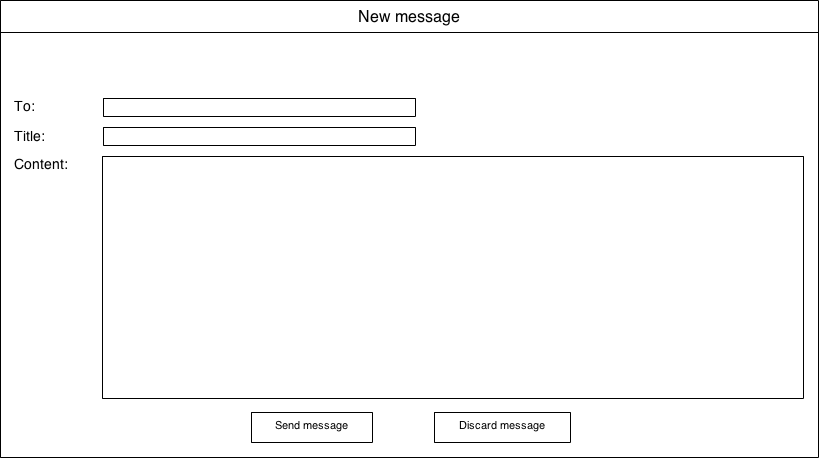
Hierarchal display of projects, job groups, and individual job assignments

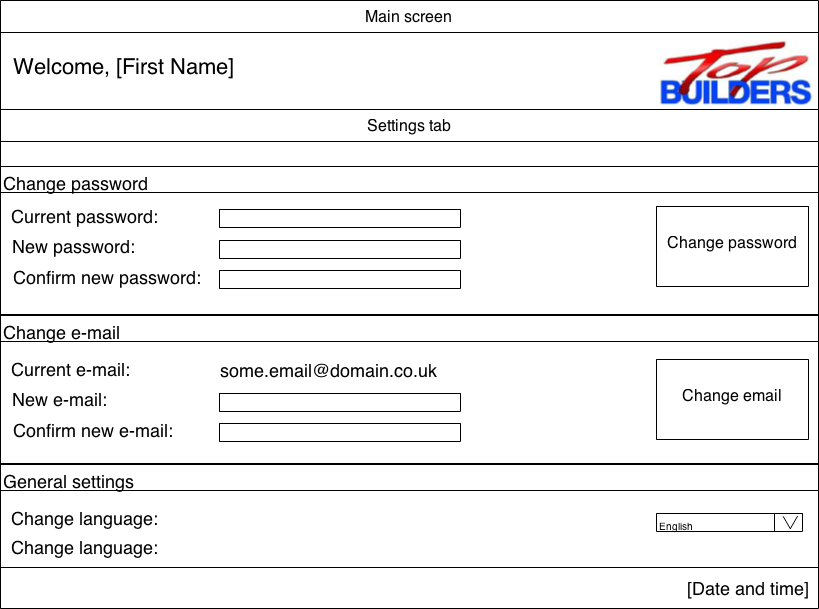
****

This popup will be displayed upon marking a project complete.

****

Combo-box listing all employees registered in the system

****

****

E-mail validation errors area

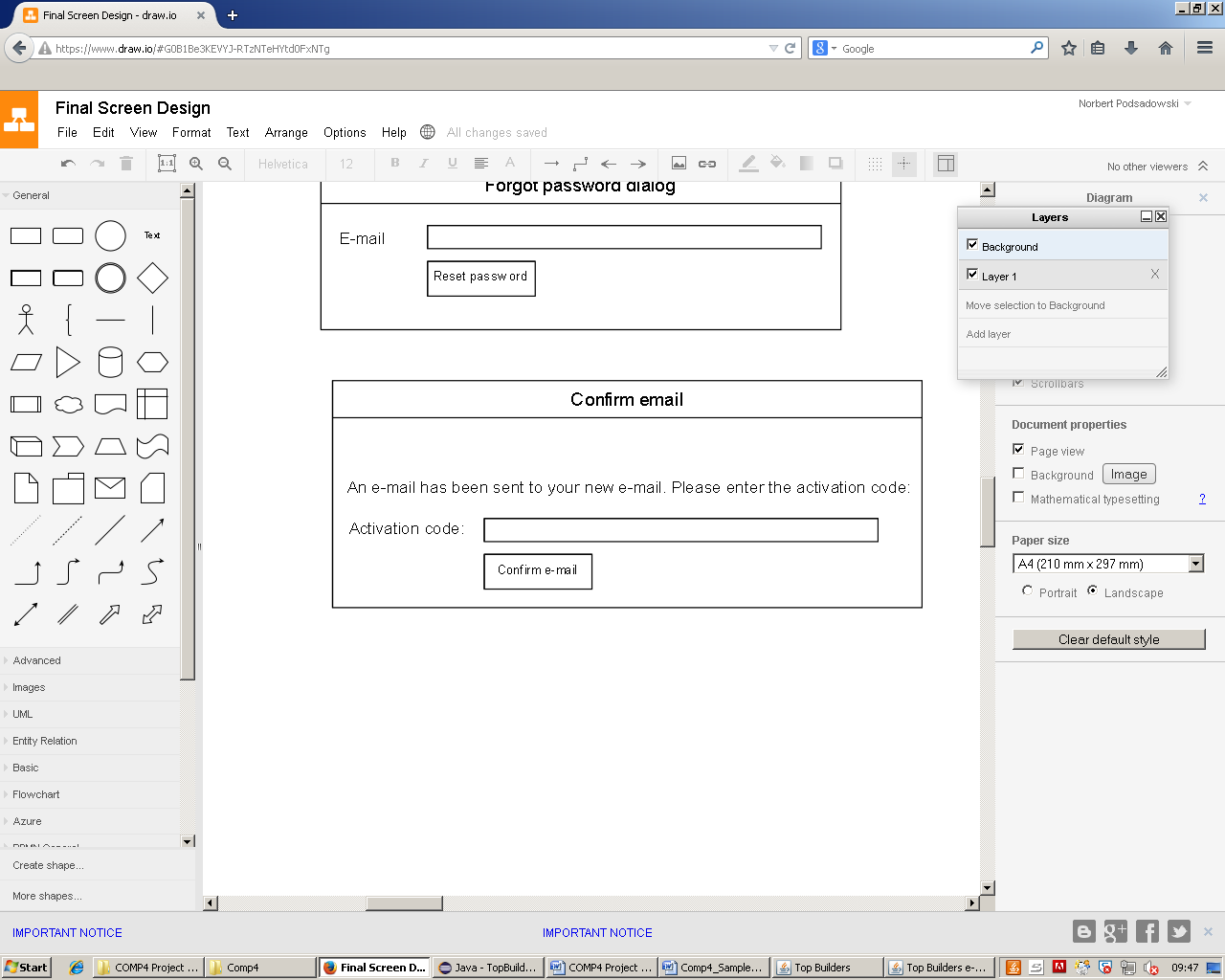
E-mail validation errors area

Discard changes

Save & Apply

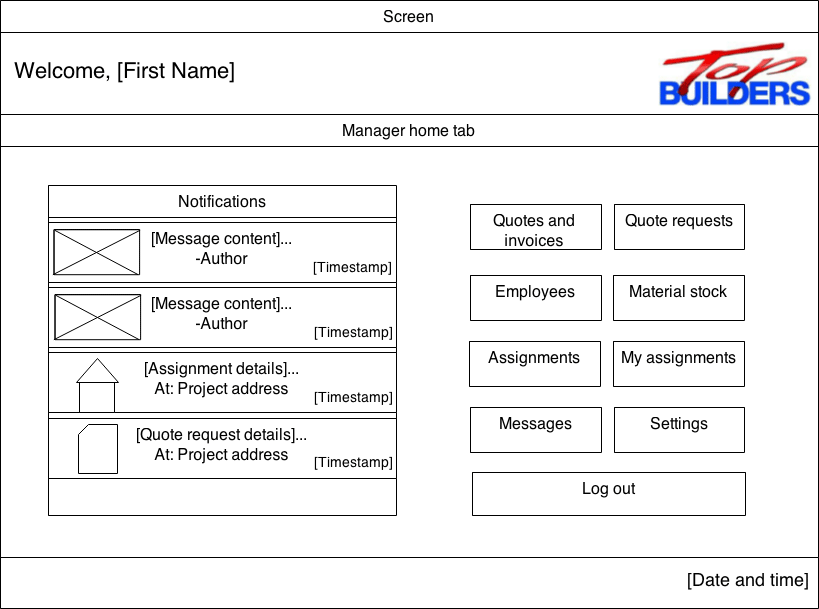
**Fullscreen mode:**

This pop-up is displayed whenever the user changes their e-mail.

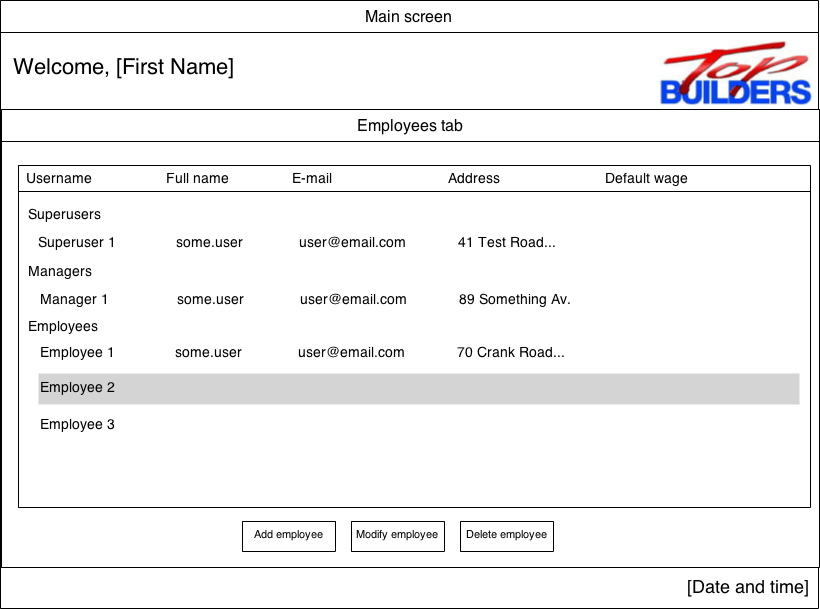


**Manager-specific forms**

A slightly modified version of the home tab is displayed for managers which enables full system management

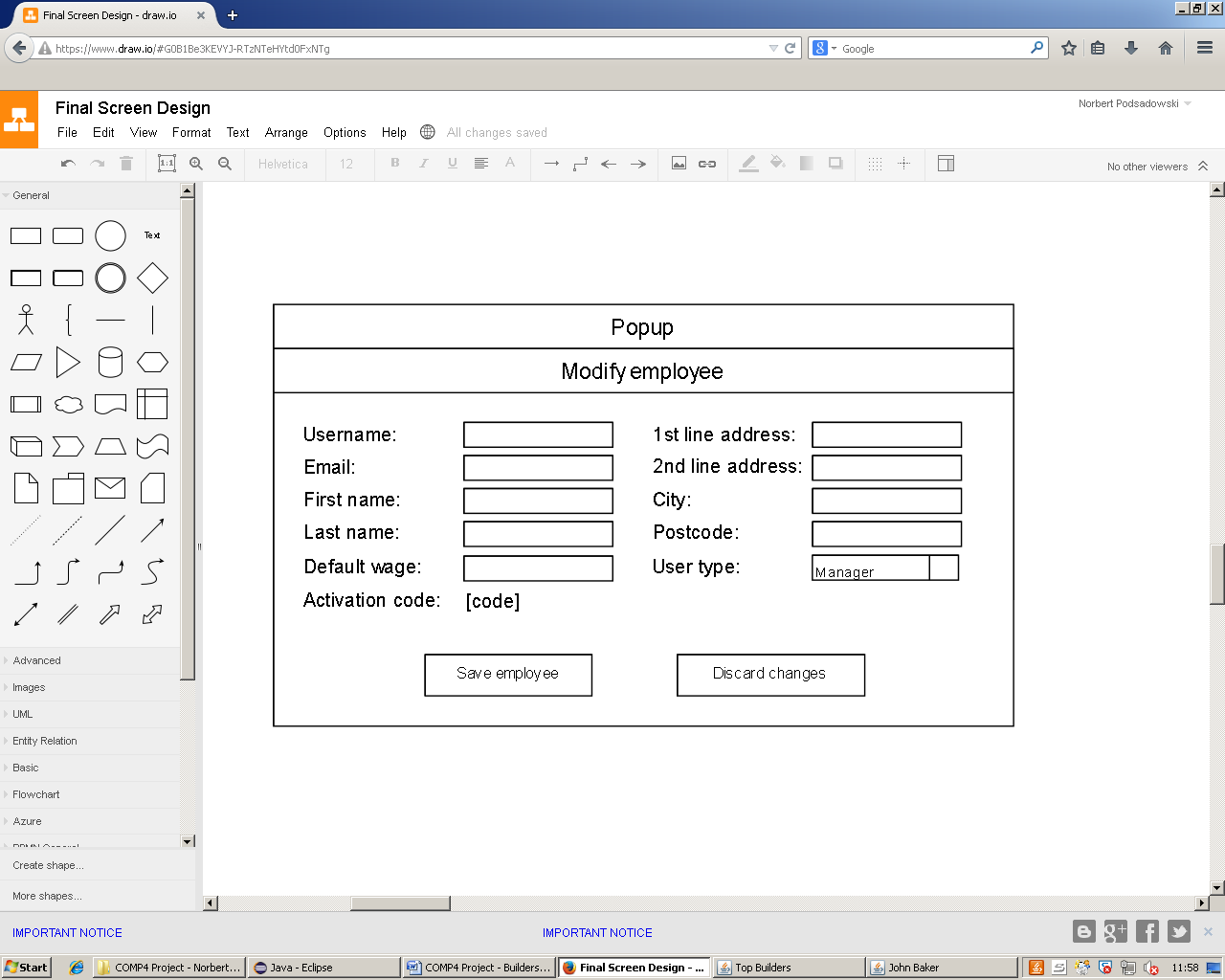
****

All managers will receive a “new quote request” notification upon the submission of a new quote request online as well as an “assignment completed” notification when an employee marks an assignment as complete

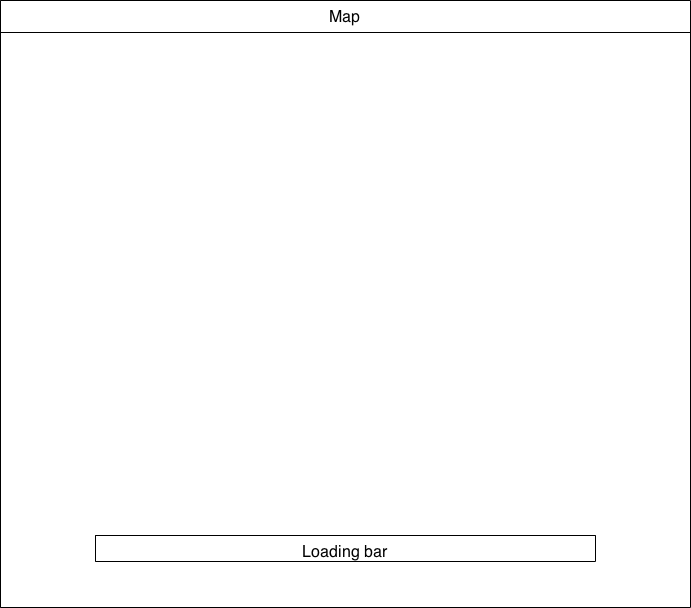
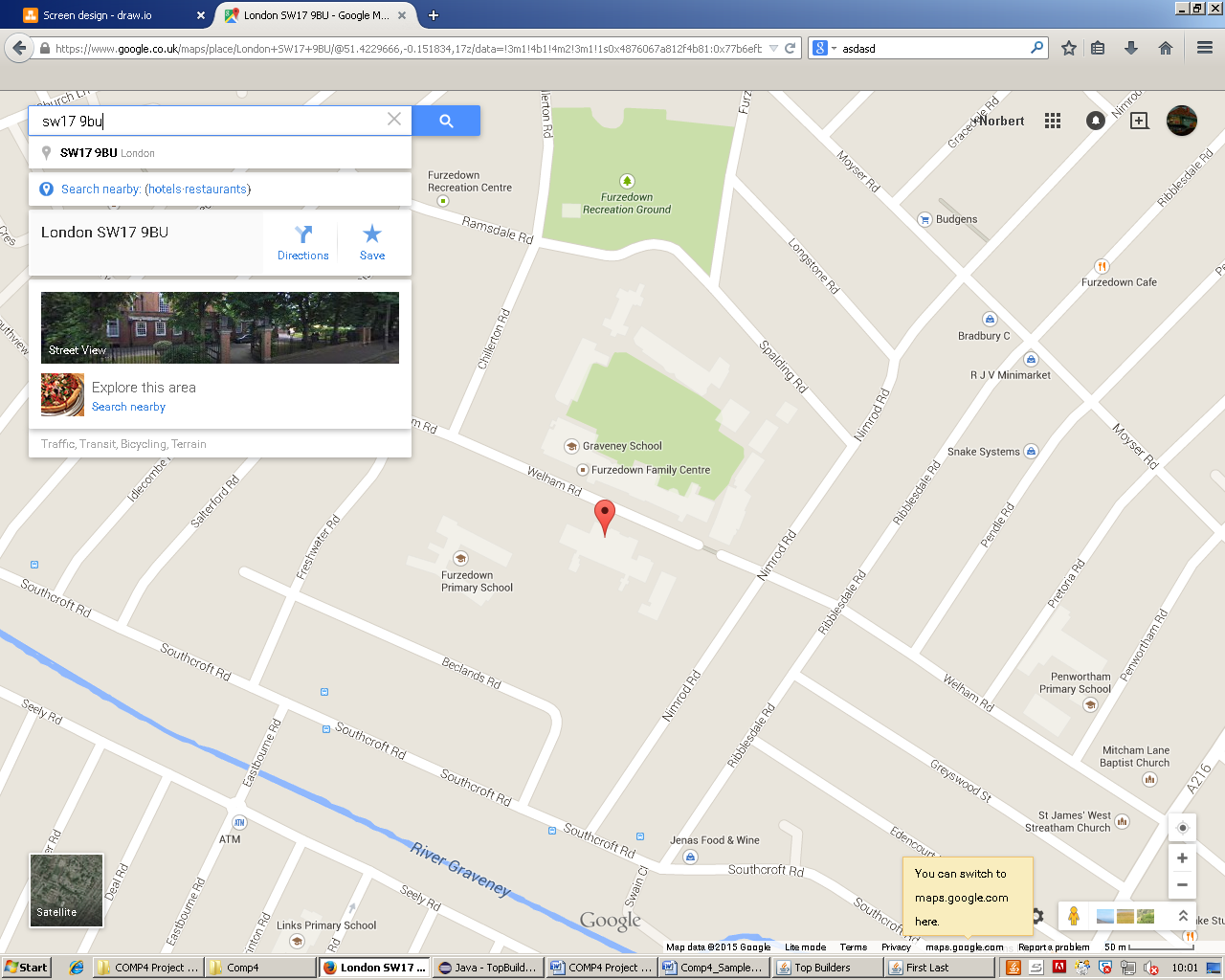


Clickable addresses which will display a popup with an interactive embedded Google Maps view.

Hierarchal display of all employees based on their type – super user/manager/employee

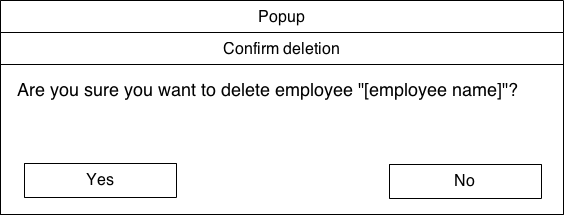
****

This dialog box will be displayed when the user wants to modify an employee

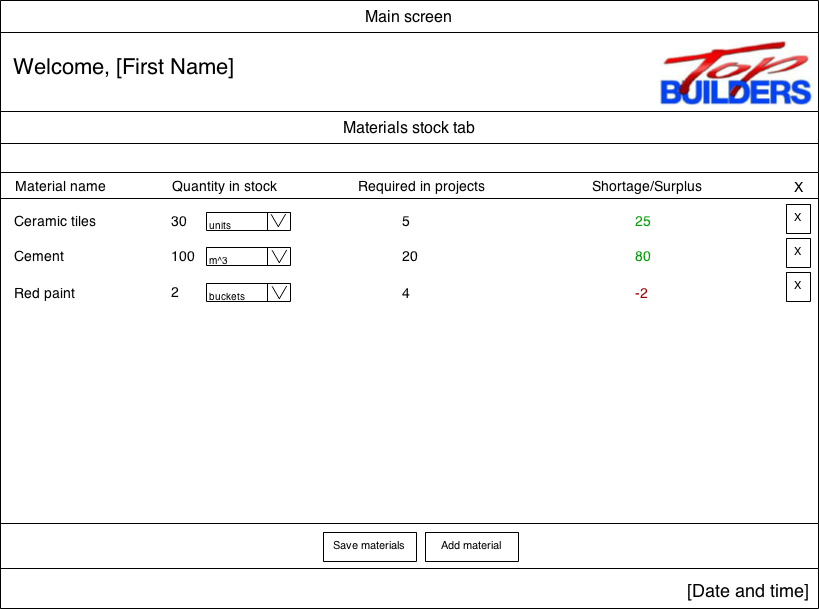
****

Loading bar representing the progress of map loading

Embedded Google Maps view.

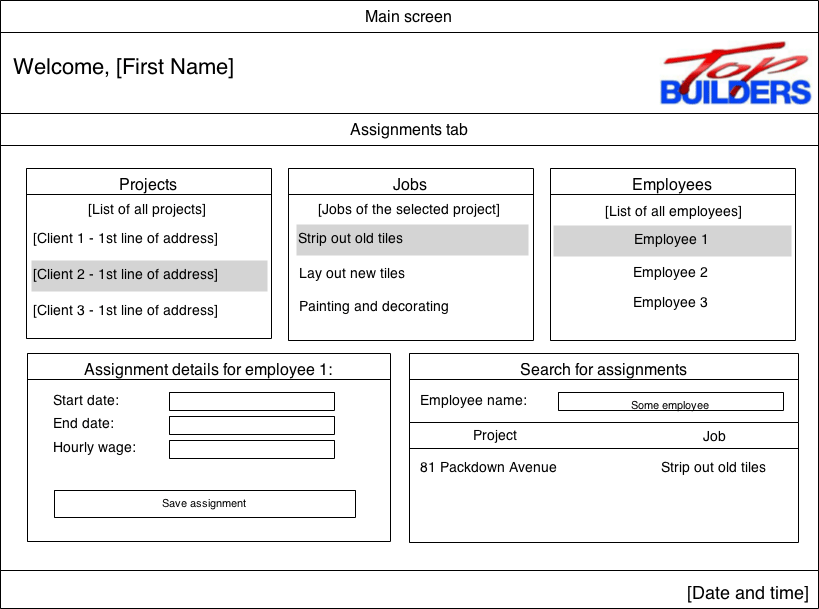


Confirmation dialog displayed when the user attempts to delete an employee



Colour-coded shortage/surplus, green if surplus, red if shortage

Selectable quantity type, like meters, liters, etc.

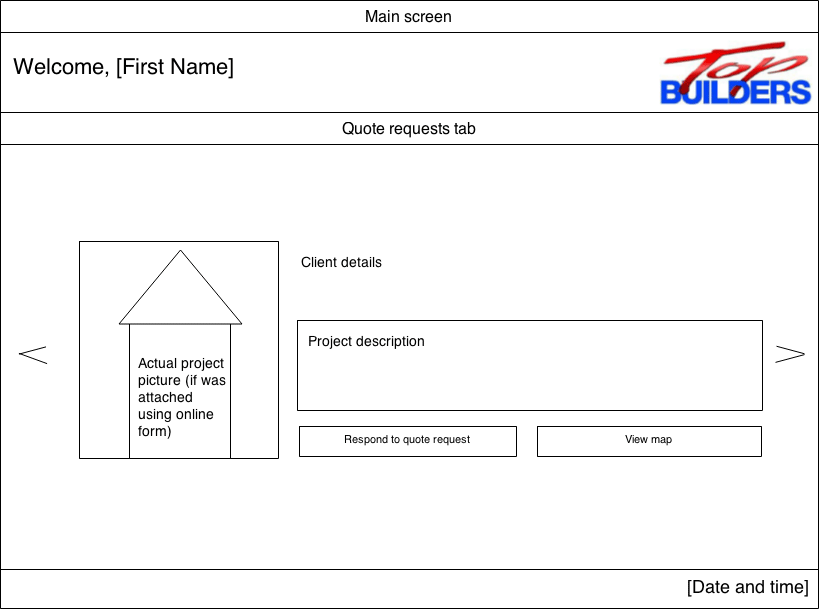


The contents of this area will change whenever an assignment is selected either from the search table on the right or the list of assignments on the top.

All job groups, and corresponding jobs for the selected project.

Table showing all assignments for the input employee

List of all employees assigned to the selected project

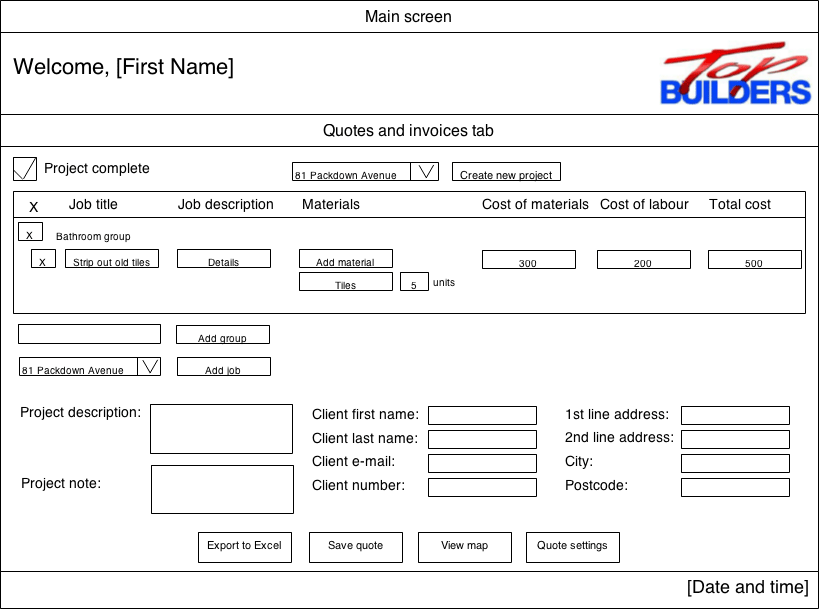
****

Opens up a Google Map view showing the location of the address supplied with the quote request.

This button will open up the projects tab and select the project associated with this quote request, allowing the manager to create a quote for the request.

Image attached with the quote request (if any) fetched from the server

Gallery-style list of quote requests, with left/right navigation arrows



Vertical list of all added materials and their quantities. Typing a material name will display a live search list.

Control used to add new groups

Hierarchal display of all projects, job groups and jobs

Combo-box listing all created job groups for this project

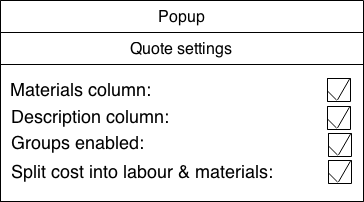
These two columns will only be displayed if the splitTotalCost setting is enabled for the project

Delete project

Combo-box listing all current projects. Selecting a project updates the entire form

This button will show a standard system file picker, and if a user chooses a valid file, the entire quote will be exported to an Excel spreadsheet, which the system will then attempt to open, if Excel is installed on the user’s machine.

Clicking this button will display a Google Map view of the current location of the project

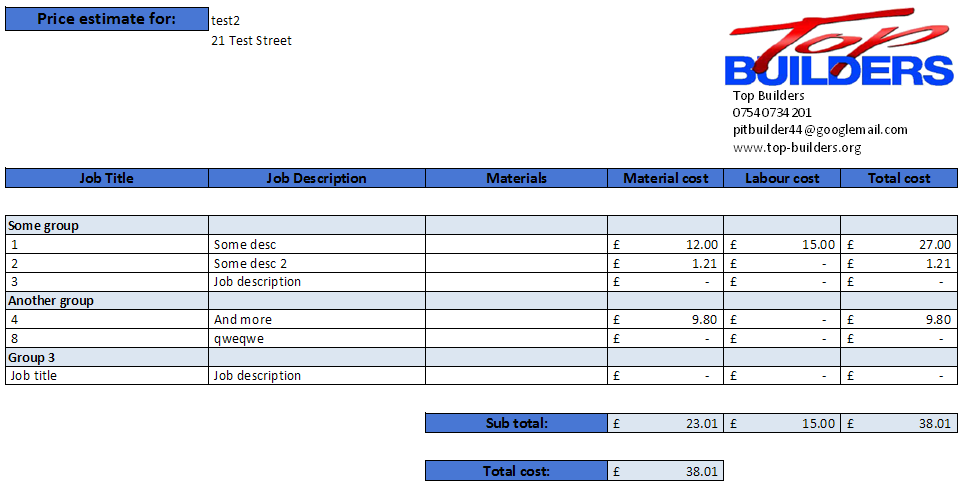


This pop-up will be displayed when the user clicks the quote settings button.

**UI sample of planned valid output designs**

**Excel spread sheet for quotes**

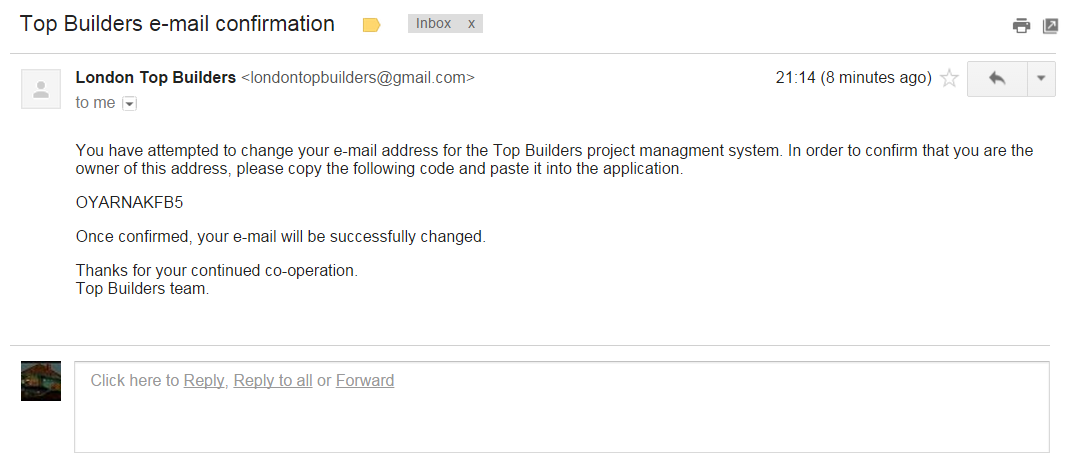
The following spread-sheet is a slightly re-designed version of the spread sheets in the document analysis section. Every project will be able to be exported into an Excel spread sheet file with the following design:



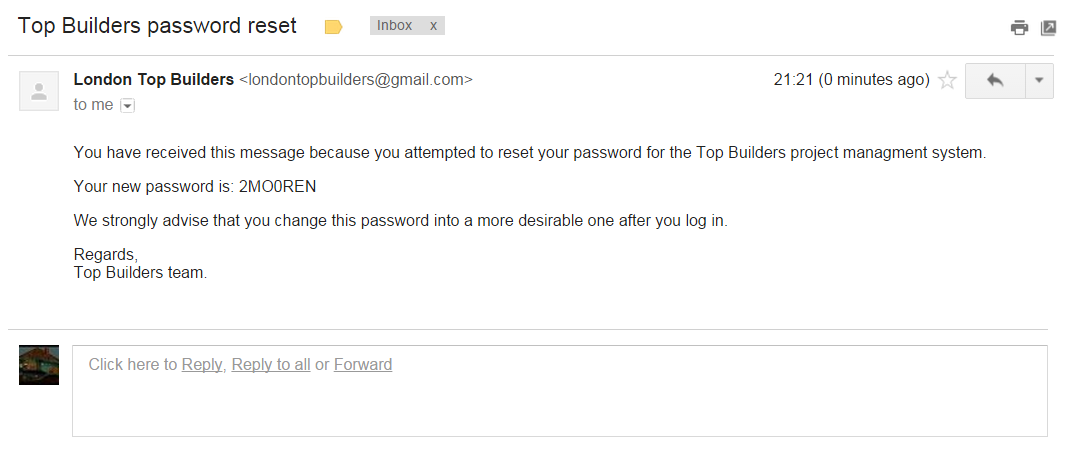
The design features a clear, crisp colour scheme which matches that of the company. Contact details and the logo will be included in the top right corner. Jobs will be split up into their corresponding job groups and their rows will be slightly tinted. There will be adequate spacing between column headings and the main quote area. Borders will be used around some cells, where appropriate. Finally, sub-total amounts will calculated using an Excel formula rather than pre-calculated by the system – this is to allow easy modification of the spread sheet outside of the system (if this is ever required) while maintaining the functionality of those cells.

**E-mails**

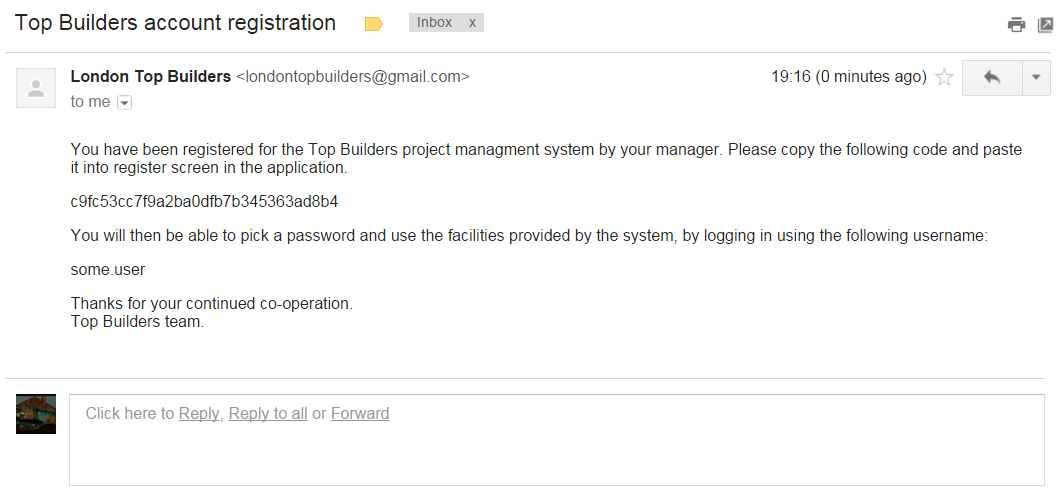
The following e-mail will be sent when a user changes their e-mail:



The following e-mail will be sent when a user submits a forgotten password request:



The following e-mail will be sent when a manager creates a new employee:



**Other**

Other output designs are already included in the previous section – most forms of the system will be designed to act as both input and output forms. The designs listed above are strictly output-only.

**Description of measures planned for security and integrity of data**

**Validation**

The validation techniques described earlier ensure that the data stored in the database is valid, sensible and correct at all times. For example, the database will never store data under the “email” field which is not in the format of an actual e-mail. This prevents data from being misinterpreted which may cause both run-time errors as well as logical errors preventing the system from working correctly.

**Backups**

It is strongly recommended that an appropriate database configuration is made when the system is moved out of development and into a production environment. As I am both the developer of this system and the database/server manager for Top Builders, I will ensure that regular database backups are made. A server-side script which automates these backups and performs them chronically may be of good aid.

*Regular* backups ensure that if the server or application itself ever break down (as a result of either software or hardware failure) then a backup may be used to restore the system to a past point, which will hopefully not be too far out-of-date.

Backups should be stored in a separate location (hardware wise) than the main database server, and made about once a week.

**Database access**

The system will feature a full login/register system which will prevent unauthorised users from accessing certain parts of the system. The access system will be separated into 3 user types, all with differing permissions:

* Super-users: can access/perform any action, including the creation of new managers
* Managers: can access/perform any action except the creation of new managers
* Employees: regular users which may only access/perform certain actions

Once the system is moved into a production environment, a super-user account will have to be set up manually through the database, and the details of which will need to be given to Peter, the CEO of Top Builders. He will then be able to add managers and employees as he pleases.

The login system adds a lot of overall scalability to the application, as even though currently Peter will be the main user of the system, later on he is likely to expand is company to the point where he will need to employ other managers to aid him in managing all of his projects. These managers will then easily be able to add any future employees which are hired.

**Integrity of data**

All primary indices in the database should have a UNIQUE constraint on them to prevent any more than one primary key being entered into any table. This ensures tables which reference other indices always access the correct record.

Measures which ensure referential integrity will also have to be considered. As the database will contain multiple tables, all related to each other and all accessing indices from foreign tables, I must make sure that at every point in time all foreign keys point to an index which actually exists.

For example, a job group will reference the project to which it belongs (by storing the project ID with that job group record). When that project is deleted, the job group will no longer reference a project which exists, invalidating the integrity of that job group record. I must therefore make sure to delete the job group as soon as the corresponding project is deleted – as well as delete every job belonging to that job group, and then delete every required material belonging to that job, etc.

**Description of measures planned for system security**

**Prepared SQL statements**

One major threat to database systems using SQL is an SQL injection attack. A system may be vulnerable to such an attack if its SQL statements are not “prepared”. Take this pseudo-code as an example:

**executeQuery(“SELECT \* FROM users WHERE username=’someUser’ AND password=‘” + passwordTextField.text + “’”);**

*Note: “” are quotes used to represent a string in the actual code, whereas ‘’ are quotes used to represent a string in the SQL statement.*

If the user types in a harmless input string, such as:

**mypassword**

Then the executed SQL becomes:

**SELECT \* FROM users WHERE username=’someUser’ AND password=‘mypassword’**

and our statement performs its expected function – if the input password is correct, our SQL statement returns a record, and we may log the user in.

However, if a malicious user types in a well-constructed input string, then they may easily trick the system into returning a record and gain unauthorised access into a restricted part of the system. One way of doing this would be to, instead of typing in an actual password into the text field, input a string such as:

**’ OR ‘1’=’1**

Then the SQL statement which gets executed becomes:

**SELECT \* FROM users WHERE username=’someUser’ AND password=‘’ OR ‘1’=’1’**

This is perfectly valid SQL syntax, and our database will do exactly as instructed – select all users from the users table, because the expression 1=1 will always evaluate to true. Our system, seeing that our SQL statement returned a result, will log the user in to another user’s account (possibly the first record in the users table, which is likely to be the super user).

This presents a huge security vulnerability which can be very easily exploited to cause havoc across the entire system (there are a lot more elaborate input strings which we may craft to do a plethora of other malicious tasks, like dropping all tables in the database).

One way to overcome this is to use prepared statements. Consider this revised version of our SQL execution pseudo-code:

**statement 🡨 prepareStatement(“SELECT \* FROM users WHERE username=’someUser’ AND password=?”);  
statement.bindParameter(passwordTextField.text);  
statement.execute;**

This code is completely SQL-injection proof. We are no longer inserting our user input directly into our query – we firstly prepare our SQL statement object, indicating any parameters with a question mark symbol (?). We then bind the user input to our defined parameter. Our statement object automatically inserts the user input in place of the parameter at the time of execution and even handles the escaping of reserved keywords, characters and appropriate quotation. Hence, any user input will be treated as that – purely an input string, as intended.

The system will use this kind of prepared statement for all communications between the application and the database to ensure security against SQL injection attacks.

**Password encryption**

If a hacker or unauthorised person ever manages to get access to the database, they would be able to see all usernames and their corresponding passwords as plaintext and be able to access any account they wish. If passwords are properly encrypted, this is no longer the case, as gaining access to the database does not reveal any information which can be used to log in to the system (the hacker will need the plain-text representation of the password, rather than its encrypted form). It is therefore important to ensure that no password in the system will be stored without appropriate encryption.

The particular encryption algorithm which I will use is the MD5 hash – this is an industry-standard, public algorithm commonly used to encrypt passwords as well perform a lot of other functions (i.e. digital signatures, etc.) The MD5 hash is a one-to-one mapping which means that every string has a unique representation and vice-versa. Also, there is no way to decode an encrypted hash into the original plaintext.

When a user picks their password, the plain-text password which they enter as input will be turned into an MD5 hash by the system, and then inserted into the database. Similarly, upon logging in, the input password will be converted into the MD5 hash and then compared to the one stored in the database – if they match, then the user is logged in.

**Overall test strategy**

**Testing strategies**

There are various test strategies one could use to ensure that the implemented system works as expected and meets all user requirements.

Functional testing

Functional testing involves identifying the functions which the system is expected to perform and then producing a suitable set of input data to be used with these functions. A set of expected output data is then created outside the system, to be later compared to the data produced by the system. The comparison will reveal how closely actual output data matches the expected output data and this is used to determine if the system meets the user requirements. The internal implementation of the system does not have to be known prior to carrying out the test cases, and this makes it a type of **black-box testing.**

In the context of the Top Builders project management system, a function test case could consist of exporting a project quote into an Excel spread sheet. The input data would include all the job groups, jobs, required materials and prices while the output would include the actual created Excel spread sheet. The generated spread sheet could then be compared to the expected spread sheet design decided earlier and a judgment would be made on how effective this function has performed its job.

System testing

System testing differs from functional testing in that it no longer evaluates individual functions and checks whether each one adequately performs its intended task. Instead, system testing can be seen as a level higher than functional testing and looks at how the system performs as a whole. In other words, it considers how well the individual functions of the system integrate with each other in order to deliver the overall end user requirements.

An example for Top Builders follows on from the earlier example of exporting a quote to Excel. Prior to such an export, a wide range of functions would have to be performed in order to put the system into a state which is ready to perform that single individual function (of exporting the quote into Excel). This includes firstly creating managers, which are able to firstly manage the material types and quantities stocked by Top Builders. This can then lead to creation of an individual project, then creation of all job groups, jobs and calculation of total costs. Only once these functions have been performed, the quote may be exported into Excel. This is an example of considering the system as a whole and judging its overall effectiveness in meeting the end user requirements rather than looking at a single particular function of the system as with functional testing.

End-user testing

End user testing is the third form type of testing strategy which will be used for the system. End user testing involves giving the finished system to the end user and allowing them to use it for a specified amount of time. They may either be provided with a set list of functions to test, or simply left to use the system on their own accord. This type of testing provides a very “natural” method of checking whether the specified system requirements have been met, as the end user was not only the person who directed these requirements, but also they do not have a technical awareness of the specific implementation of the system. We are able to then see whether the system is usable by someone who has not designed it and has not used it before, and produce a set of results which tends to be more “realistic”.

After finishing the entire implementation of the system, I will hand the system to Peter, and produce a list of functions performed by the system. I will then allow him to try out every aspect of the system in any order he likes, and mark the functions which he is satisfied with, and which functions do not work as expected.

Testing

**Design of test plan, sample of test data and expected results**

**Functional test plan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test No. | Purpose of Test | Test Data | | Expected Result |
| 1.1 | User-name text field accepts a maximum of 20 characters | Normal |  |  |
| Boundary |  |  |
| Inaccurate |  |  |
| 1.2 | Password text field accepts a maximum of 20 characters | Normal |  |  |
| Boundary |  |  |
| Inaccurate |  |  |
| 1.3 | Sign-in button authenticates the user only when the correct user and password are entered | Normal | Any correct account data | Show the main form and open the home tab based on the user type |
| Inaccurate | Any non-existent account data | Show on error message underneath the text fields, telling the user of if the given username does not exist or a wrong password was entered. |
| 2.1 | Forgotten password form is opened |  | |  |
|  |  | |  |
|  |  | |  |

**System test plan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test No. | Purpose of Test | Test Data | | Expected Result |
| 1.1 | User-name text field accepts a maximum of 20 characters | Normal |  |  |
| Boundary |  |  |
| Inaccurate |  |  |
| 1.2 | Password text field accepts a maximum of 20 characters |  |  |  |
| 1.3 | Sign-in button authenticates the user only when the correct user and password are entered | Normal | Any correct account data | Show the main form and open the home tab based on the user type |
| Inaccurate | Any non-existent account data | Show on error message underneath the text fields, telling the user of if the given username does not exist or a wrong password was entered. |
|  |  |  | |  |
|  |  | |  |
|  |  | |  |

**End-user test plan**

|  |  |  |
| --- | --- | --- |
| No. | Question | Answer |
| 1.1 | Does the log-in form allow you to easily sign-in to the system |  |
|  | Can you create a new project in less than 10 minutes |  |
|  | Does the system match your company house style |  |
|  | How intuitive is the system overall (1-10) |  |
|  | Is the layout easy to follow |  |
|  | Is it easy to navigate the system |  |
|  | Were the error messages helpful |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Test results & cross-referenced to test plan**

**Functional test results**

Login test (before and after)

**Systems test result**

**End-user test result**