University of Southampton

Electronics and Computer Science

A Group Design Project report submitted for the award of MEng Computer Science

by:

Aleksandar Botev

Kim Svensson

Dionisio Perez-Mavrogenis

Liam de Valmency

Project Supervisor: Mark Weal

Second Examiner: Mike Wald

Controlled Trials in LifeGuide Interventions

December 11, 2013

Abstract

The following report outlines the undertaking of a group design project which aimed to create a self-contained code module for LifeGuide, an existing piece of software. LifeGuide provides researchers with the tools to set up and run online interventions, for use in randomised control trials. Participants in these trials must be allocated into one of a number of interventions using a randomised algorithm, but in the current system researchers have to manually code these algorithms in a proprietary, complex format. The project's goal was the creation of a module which takes a trial's setup and allocation parameters as input, and which can subsequently perform participant allocation, replacing the need for the manual coding of this process. The report outlines the design and creation of this module, detailing how it was set up to be easy to integrate into LifeGuide and quick to extend if necessary. The module provides several of the most common allocation algorithms, each of which has been thoroughly tested and analysed to ensure correctness. The project also resulted in the creation of an offline application, which will allow researchers to perform randomised control trial allocations outside of the LifeGuide system.

Contents

1 Introduction

1.1 Problem Description

LifeGuide is a piece of software developed by the Electronics and Computer Science (ECS) department at the University of Southampton, intended for use by researchers in the psychology faculty. The software allows researchers to set up clinical trials for their research, specifically randomised control trials, which are considered the gold standard for clinical trials. These trials are carried out to test the effectiveness of a set of different interventions. An intervention is considered to be any process which tries to enforce a positive change in participants in the trial with regard to the factor being studied. Interventions can be anything from medical treatments or prescriptions to web-based guidance or counselling. Randomised control trials allow researchers to test out new interventions for a given problem factor, by comparing and contrasting the results of each intervention in the trial, when applied to the trial's participants.

The primary use of LifeGuide in the University of Southampton is to study the effect of online behavioural interventions (e.g. trying to encourage weight loss through the use of web-based guidance sessions). To set up the trials, researchers must be able to easily create the web resources required, as well as specify how the trial is set up. Specifically, it must be possible to define the various interventions in the trial, the method in which participants are allocated to a specific intervention for the trial's duration, and any additional parameters which define how the trial operates (for example, a maximum participant limit for one of the interventions). The system currently provides an authoring tool to allow creation of the web-based resources researchers may need, as well as a method of specifying the trial's setup. However, this specification is currently in the form of a proprietary scripting language. For each trial, the researcher must write code which sets up the trial, defines its parameters, and performs the allocation of participants into the various treatment arms. The code required to do this is extremely long, difficult to understand and maintain, and requires the manual coding of a participant allocation algorithm for each trial. This presents a significant overhead in setting up a trial, in terms of both time and complexity. The setup also becomes more error prone, as the participant allocation algorithm is difficult to verify without spending large amounts of time allocating participants by hand. Currently, this format is restrictive enough that the University of Southampton researchers only consider running trials using the simpler allocation methods.

1.2 Project Specification

The aim of the project is to address the difficulties which currently arise from LifeGuide's trial setup format. Specifically, it is our goal to remove the need for researchers to manually code the participant allocation algorithm for each trial, and to simplify the specification of the trial's various parameters. To this end, the group is producing a self-contained, fully-featured randomisation component which can be integrated into the LifeGuide system at a later date. This module must provide the ability to allocate participants into a trial's treatment arms one at a time, using a range of both randomisation or minimisation techniques. The module must allow a researcher to easily construct a trial, specifying any parameters which may influence the allocation process. This means designing the module's input in such a way that it is possible for a researcher to provide values such as:

- Allocation method and parameters.
- Treatment groups into which participants may be assigned.
- Allocation ratios or limits, for non-balanced allocation.
- Relevant participant attributes to take into account.
- Trial specific parameters, such as attribute weights for minimisation (see section 2.2.4).

Given some or all of these parameters (further discussed in section 3.2.2), the module must be able to set up the trial, then accept participant data through some generic API, returning the intervention group into which the participant has been allocated.

Due to the limitation of LifeGuide's current trial specification format, researchers in the university do not use very complex allocation methods. It is therefore a necessary part of the project to carry out research into which methods exist, which are most frequently used, and which are desired by the end users. This will inform the choice of allocation methods included in the module's design.

Thorough testing is a vital component of the project. It must be rigorously checked to ensure that allocation is performed completely correctly in all circumstances and for all allocation techniques, as the allocation results will be used in clinical trials, the results of which will be published. If the allocation cannot be relied on, then the statistical significance of the results becomes more suspect to error or bias, which damages the credibility of the study.

LifeGuide is a Java application, and so to allow integration at a later date, the module must also be written in Java. Due to the online nature of the system, trial participants will sign up and be allocated over the course of weeks or even months, so the module will be required to store intermediate data relating to the allocation process. To allow the data storage mechanisms to be more easily combined with those of the existing system, the module will provide built-in support for MySQL database interaction, as this is the database management system currently employed by LifeGuide.

Due to the limited time frame of the project, integration with LifeGuide is not be included in the project aims. However, several extensions over the baseline module requirements have been incorporated into the project's implementation, as the initial requirements could be completed with sufficient time to spare. These extensions include the ability for researchers to monitor the participant sign-ups and allocations over the course of a trial, as well as the provision of a standalone application which will allow researchers to perform offline participant allocation outside of the LifeGuide system.

1.3 Report Outline

This report has been organised in such a way that it is most useful to read each section sequentially. However, each section has been written with as little reliance on other sections as possible, to allow for readers who only have an interest in specific aspects of the project. References to additional information contained in other sections are included where necessary.

Section 2 provides more detail on the problem domain, discussing the purpose and problems with randomised control trials, and giving a description of the various algorithms used to allocate participants in these trials. It also contains a look into the advantages and disadvantages provided by use of the LifeGuide system, along with a comparison with other existing systems with a similar function.

Section 3 is a full overview of the project's deliverables, including their design, implementation and testing. This section contains a high-level architecture of the created module, a discussion of the extension work, as well as low-level details on the code structure and database.

Section 4 discusses the team's process, specifying how they went about the project by breaking down the requirements into individual goals, and how the group members organised to work effectively as a team, ensuring all client requirements were met. It also contains a detailed breakdown of the project work undertaken.

Section 5 is an overview of the user testing undertaken as part of the project. This took the form of a two-phased test: a questionnaire for initial data gathering, followed by a user study in which researchers were tasked with using the system to evaluate the intuitiveness of the module's input design.

The report wraps up in section 6 with an appraisal of the project's success, considering whether the requirements were met, how effectively this was done, how improvements could have been made and what future work remains.

2 Background

2.1 Randomised Control Trials

What are Randomised Control Trials?

Randomised Control Trials (RCT) are a type of scientific experiment in which the participant allocation to a given group is non deterministic. Currently these are the most accepted in the scientific community and are the de facto standard. The reason for this is that RCTs allow the removal of any selection biases from the researchers performing the experiment, and increase the statistical significance of the trial results. The main aim of a study is usually to compare two or more interventions to assess their effect on the general population, but the fact that such experiments are impossible to be carried out on everyone leads to a much smaller number of participants being selected. The goal is to generalise the results on these participants to the whole population, which creates a major challenge since the smaller the selected patients sample size is, the larger the standard deviation of this result from the "true" results, where by "true" we mean the hypothetical result if the study was carried out on every single individual of the population.

Selection Bias

One of the main reasons for using RCTs in clinical trials is their power to produce unbiased results of the study done. Without randomising, an allocation bias can be introduced in the groups being allocated, which could lead to less reliable or even false results. Consider the following example:
A study of treatment 'A' against a well known and established treatment 'C' for discovering coronary problems is carried out in some small town, and the participants selected are either programmers or medical personnel. We assume that it would be impossible to use both treatments on the same patient due to side effects caused by the other treatment. It has been concluded from previous studies on the whole population in the country that, on average, the distribution of people with coronary problems is the same between different employment areas, so the researches decide to split the groups by their job type, so that programmers use the new treatment 'A' and medical personnel use treatment 'C'. As it turns out, in this town there has been a campaign inside the programming community to majorly increase their physical activity, so that on average they perform three times more physical exercise than the medical personnel. As a result, treatment 'A' finds only 2% of the programmers to have coronary problems, where the treatment 'C' finds 10% of the medical personnel to have such problems. By this the researchers conclude that treatment 'A' is not sufficiently accurate. The unintentional bias introduced in this study is due to the fact that in this sample of the general population, the job area had high positive correlation with physical activity which has a major effect on the coronary condition

of a patient.
From this example we can see the importance of removing any selections bias that could be introduced in a study. We want to emphasise that the example given is very simple and probably unrealistic, but it represents the problem in a very clear way. In reality, such dependencies and correlations can be much more complex and unforeseeable. Randomising the allocations of patients can be seen essentially as a coin flipping technique, which removes any biases within groups as it gives equal chance of each patient to be in any group.

Balance of Groups

Sometimes in clinical trials the problem of balancing groups arises due to biased selection, or even in randomisation when the study sample is relatively small. This can be seen as having an unintentional but significant difference in the number of patients assigned to one treatment compared to those assigned to another. This could make the results of the study incorrect, as this introduces higher standard deviation from the "true" results in the under-sampled group, and makes the comparison of the treatments uneven. With randomisation there are several techniques used to tackle this problem. It generally arises only when the trial is limited to small samples, since as the size of the participant pool increases, the chance of receiving incorrect results decreases.

2.2 Algorithms

2.2.1 Simple Randomisation

As the name indicates, this is the simplest randomisation technique which is used for RCTs. The randomisation mimics coin tossing for deciding into which treatment group the next participant should be allocated as described in [1]. It is fully unbiased in regards to any of the factors of each patient, as it does not take them into account at all. As a randomisation algorithm the only requirement for it to work is a random number generator. In the simple binary case where we have to assign a patient to one of two groups with equal probability, the method is equivalent to flipping a coin. Of course, this can be extended easily to allow for more than two groups, or for non-uniform probability distribution among each group. After receiving feedback from university researchers, it was found that a limit on a specific intervention group is often required in clinical trials (see 5.2.1). For this reason, we also include the capability to specify a limit on the number of participants per treatment group. Here we present pseudocode of the algorithm for a single user allocation, where we consider allocating into k different treatments:

```
Probabilities: $p_1, p_2, \ldots, p_k$ for each treatment, i.e. $\sum_{i=1}^{k} p_i = 1$
Limits: $l_1, l_2, \ldots, l_k$ for each treatment.
begin
  $S = \{i : 1 \leq i \leq k \cap l_i$ have not been reached$\}$
  for $i \in S$
      $p'_i := \frac{p_i}{\sum_{i \in S} p_i}$
  end
  $r := random()$
  for $t := 1$ to $\|S\|$
      if $r < \sum_1^t p'_{i_t}$
          $allocation := i_t$
          $break$
      end
  end
end
```

One of the major limitations of simple randomisation occurs with small group sizes, where it fails to balance groups well. This can be seen very easily through a simple statistical analysis of the algorithm. If we ignore the limits it is easy to see that the random allocations produced, $a = (a_1, a_2, \ldots a_k)$ have a multinomial distribution with parameters N and $p = (p_1, p_2 \ldots, p_k)$. Now, for a given group we consider the random variable X_i, representing the number of people assigned to that group and define $Y_i = \frac{X_i}{N}$. One can easily see that $X_i \sim Binom(N, p_i)$ if we consider all other groups to collapse to a single one, and define success to be a patient being assigned to the i^{th} group and failure otherwise. From this, it follows that $\mu_x = p_i N$ and

$\sigma_x^2 = p_i(1-p_i)N$, which are the standard binomial mean and variance. Using the properties of mean and variance we can find that $\mu_y = p_i$ and $\sigma_y^2 = \frac{p_i(1-p_i)}{N}$. In fact, the random variable Y_i is the proportion of people assigned to group i. This statistical analysis shows that on average Y_i will be exactly p_i as desired. On the other hand, the variance σ_y^2 is different, and depends on both the sample size and the probability p_i. If we assume a uniform probability - $p_i = k^{-1}$ this reduces to $\sigma_y^2 = \frac{k-1}{k^2N}$. When working with random variables, we're usually interested in confidence intervals, and one of the most widely used is the the 95% confidence interval, which covers ~ 2 standard deviations from the actual mean. We can use this derivation and a normal approximation to the binomial distribution to show how many participants would be needed in order to have α(0.95 for 95%) confidence that the sample mean would lie within s tenths from p_i, or in mathematical terms:

$$\Phi(x) \text{ - standard normal cumulative distribution function}$$

$$\alpha \text{ confidence interval, of } m\sigma \text{ when } 2\Phi(m) - 1 = \alpha$$

$$\text{Every mean within } \alpha \text{ confidence interval - } 2\Phi(m) - 1 = \alpha^{\frac{1}{k-1}}$$

$$\Rightarrow m = \Phi^{-1}\left(\frac{1+\alpha^{\frac{1}{k-1}}}{2}\right)$$

$$\text{We want that } m\sigma_y \text{ to be less than s tenths}$$

$$m\sigma_y \leq 0.1s \Rightarrow m^2\sigma_y^2 \leq 0.01s$$

$$\frac{m^2p_i(1-p_i)}{N} \leq 0.01s$$

$$\frac{100m^2p_i(1-p_i)}{s} \leq N$$

$$N \geq \frac{100m^2p_i(1-p_i)}{s}$$

$$N \geq \frac{100m^2(k-1)}{k^2s}, \text{ when } p_i = k^{-1}$$

In order to illustrate these results in action, assume that $k = 3$, and $\alpha = 0.5$. From the formula derivations we can calculate $m = 1.052$. Given that $p_i = \frac{1}{3}$, we have the inequality $N \geq \frac{24.5934}{s}$. For $N = 30$ we have that the 50% confidence interval is within $s = 0.8198$ tenths of the expected mean - [0.25140.4153]. Below we show results for several different population sizes, N which are compared with the empirical results in section 3.3.3:

N	30	150	600	1200	6000	12000
25th percentile	0.2514	0.3169	0.3292	0.3313	0.3329	0.3331
75th percentile	0.4153	0.3497	0.3374	0.3354	0.3337	0.3335

The main advantages of this technique is its simplicity and the fact that it is unbiased. The only disadvantage is that when the sampled size is relatively small, the method might assign imbalanced groups to the treatments, which could reduce the statistical significance of the study. When the sample size is big enough, the method guarantees well balanced allocations.

2.2.2 Blocked Randomisation

The Blocked Randomisation method is very closely related to the Simple Randomisation. Its main purpose is to overcome the problem of unbalanced groups within small samples and it's benefits have been studied in [2]. To do so, instead of randomly assigning each patient to a different treatment we define a block of treatment allocations. Each block is perfectly balanced between the different treatments, for instance if we have 3 treatments and we want to have twice as many participants in the third treatment than either of the other two, and they in turn have same number of participants, then any block size which is divisible by 4 will suffice. We will assume for this example the block size selected is 12. We then look at the string created by this block - "112311231123". Of course this introduces a bias, because the order of assignment is known prior to the experiment. For this reason, instead of the original ordering we choose a random shuffling of the string. A critique of using only this procedure of shuffling is that if the block size is known, and all patients allocations until the last one are tracked, then the last one is uniquely determined. For this reason an extra step of randomisation is introduced - instead of using a fixed block size, select several block sizes which are feasible to perfectly balance the treatments within, and each time a new block is created randomly select the block size. In the example above for instance, if we select block sizes 8, 12, and 16, then whenever one block is filled and a new one is created, we randomly select one of the three sizes. This guarantees that the allocation is fully unpredictable for any observer, even one who knows all of the initial parameters of the strategy. Below we present pseudocode for the algorithm, considering the we have selected m number of block sizes $B_1, B_2, \ldots B_m$:

```
Probabilities: $p_1, p_2, \ldots, p_k$ for each treatment, i.e. $\sum_{i=1}^{k} p_i = 1$
Limits: $l_1, l_2, \ldots, l_k$ for each treatment.
Block sizes: $B_1, B_2, \ldots, B_m$ and $\forall i, j\ p_i B_j \in \mathbb{N}$
Current block: $C_r$
counter (0 based)
begin
  $S = \{i : 1 \leq i \leq k \cap l_i$ have not been reached$\}$
  for $i \in S$
      $p_i' := \frac{p_i}{\sum_{i \in S} p_i}$
  end
  for $j := 1$ to $m$
```

```
        $B'_j := B_j - B_j(\sum_{i \notin S} p_i)$
    end
    if $counter == C_R$.size
        $r := random();$
        $C_r$.size $:= B'_r$
        Fill block with allocations indexes
        $C_R$.shuffle
        $counter := 0$
    end
    $allocation := C_R[counter]$
    $counter := counter + 1$
end
```

Blocked Randomisation extends the ideas of Simple Randomisation and addresses the problems which may arise when the given sample size is small. When the selection of the block size is random, it is guaranteed to be unpredictable and unbiased towards any of the treatments. It also assures that within a single block the number of allocation between treatments is perfectly balanced.

2.2.3 Stratified Randomisation

Both of the techniques presented so far are unbiased and try to balance the number of people assigned to each treatment in order to increase the statistical significance of a study. Very often researchers will have some expert knowledge about several of the prognostic factors of a participant which shows that difference in those factors can have a significant impact on the results. Imbalance in the numbers of participants with different factors can significantly reduce the credibility of a study. As an example we could consider a drug which is very effective on male participant, but has no effect, or even deleterious effects on females. If, during the study, we assign them to the treatment and control group as a whole and three times more females sign up for the trial, it is possible to be concluded that the drug has no beneficial effects. Even with techniques such as randomisation this effect cannot be overcome, as it is a problem arising from the sampled population, rather than the allocation technique used. To address this problem, Stratified Randomisation is introduced. The method splits the participants into subpopulations by each prognostic factor. Then it uses some method to randomise participants within each subpopulation. After the study is finished, researchers are able to analyse the results for each factor separately, by aggregating all the data matching it. In fact, Stratified Randomisation together with Blocked Randomisation is one of the most widely accepted methods for proving the credibility of an RCT. The only disadvantage of this technique is that the number of subpopulations is a product of the possible outcomes of each prognostic factors. If the number of these groups is large, it would mean that the number of participants assigned to each subpopulation may be relatively small. In such cases, selecting Blocked Randomisation instead of Simple Randomisation as the method for allocation is essential in order to have balance within each subpopulation.

2.2.4 Minimisation

Minimisation is an adaptive sampling technique first described by Taves in 1974 [3] and shortly after by Pocock and Simon in 1975 [4]. Its goal is to minimise the imbalance between groups based on several prognostic factors. Given a new participant which must be allocated, it will look at the participant's prognostic factors, and for each group it calculates a sum based on these factors. It will then allocate the participant to the group which has the lowest sum of them all. The sum calculated for each group can be regarded as, for each factor, how many participants in the group have the same factor. Thus, allocating to the group with the least sum indicates that the group has the lowest number of individuals who are similar to the participant to be allocated with regard to the prognostic factors.

Prognostic factor	Intervention	Control
Sex		
Male	3	5
Female	5	3
Age band		
21-30	4	4
31-40	2	3
41-50	2	1
Risk factor		
High	4	5
Low	4	3

Table 1: Typical tabel for storing participant information when using minimisation.

To illustrate further exactly how minimisation works as shown by Scott et. al. [5], table 1 illustrates a trail where 16 participants have been allocated to the intervention and control groups. Assume we must allocate a 17th participant who is a male, aged between 31-40 and has a high risk factor. To figure out which group this participant would be allocated into, we use Taves type of minimisation for its simplicity, which consists of summing up the number of males (3), people aged between 31-40 (2) and people with a high risk factor (4) in the intervention group, giving us the sum of 3+2+4=9. We carry out the same operation for the control group and arrive at a sum of 5+3+5=13. Therefore, as the sum for the intervention is the lowest, the 17th participant would be placed in the intervention group, and each prognostic factor in the intervention group matching the 17th participant would be incremented by one to indicate the new status of the trial. If however the sums of the intervention and control groups were equal (the groups are balanced in terms of the new participant), then the participant would be randomly allocated to one of them.

The main advantage of minimisation is that no matter the sample size and the distribution of participants, minimisation will always yield greatly balanced groups both in terms of participants per group, and also in terms of the representation of each prognostic factor in each group. This can be contrasted with blocked randomisation, where if a malicious practitioner who approaches potential participants knows the block layout, he may select participants which would make the trial unbalanced, resulting in null and void outcomes for the study. This is due to the predefined block layout, which is not adaptive in regards to the participant to be allocated. With minimisation however, all the malicious practitioner can do when given a participant is deduce where the participant will be allocated, assuming they also know all of the previous allocations. The practitioner may try to unbalance the groups, but as minimisation is aware of the participant being allocated, unlike blocked randomisation, minimisation would in the end try to balance the groups.

As mentioned, if a malicious practitioner has knowledge of all the previous allocations and its participants, the practitioner may deduct where a new participant will be allocated, given the groups are unbalanced in regards to the new participant. This deterministic behaviour has been raised as a criticism of minimisation in regards to blinding. To counteract this, the technique can employ a probabilistic logic in the last stage of the algorithm. What it does is, given a probability of P, it has a P chance that minimisation will allocate the participant to the most suitable group, otherwise it will allocate the participant to one of the non-optimal remaining groups. During testing in section 3.3, using the implemented Taves form of minimisation with a probabilistic value of 0.8, the result showed that minimisation is clearly the superior choice when aiming for balanced groups.

2.2.5 Other Techniques

Other than the three randomisation methods described above, there are several others which were not implemented into the module.

Response-adaptive
Response-adaptive randomisation is known as one of the more moral randomisation methods, as it increases the probability of allocating participants into treatment groups which have been proven to show positive impact on the participants. E.g. increasing the probability of an intervention group over the control group, if it has been shown that the intervention has had a net positive outcome.

Adaptive biased-coin
The adaptive biased-coin randomisation method devised by Bradley [6] in 1971 revolves around the notion of trying to balance uneven number of participants between groups. It works by calculating a value C, which is the number of subjects in treatment group A minus the number of subjects in group B. If $C = 0$, the groups are

balanced and each group get equal probability of being chosen ($\frac{1}{2}$). If we have an excess of subjects in group A ($C > 0$) we assign group B a probability p, and vice versa if there is an excess in group B ($C < 0$). The value of p is selected so that $p \geq \frac{1}{2}$, which will give a higher precedence to the group which has fewer subjects.

Wei's urn design
This randomisation method does an adaptive biased-coin randomisation, but uses the concept of balls in a urn instead of a weighted coin. It was put forward by Wei in 1977 [7] and developed in response to the criticism of adaptive biased-coin's high predictability during certain circumstances, and its use of a fixed probability parameter. The algorithm starts out with an urn which contains X balls for each treatment group. When a participant must be allocated, you simply allocate the participant to the ball which is picked from the urn. You then replace the ball in the urn with a ball from the group of smallest size. In this way, the groups will be fairly balanced whilst maintaining a high degree of randomisation. The reason it is a better method than adaptive biased-coin is due to its scaling with the number of balls in the urn. Larger numbers of balls will yield relatively more unbalanced groups but reduces the predictability of each allocation, while decreasing the amount of initial balls will make for more balanced groups but with increased predictability.

These are just a few of the various randomisation methods which have been developed over the course of time. However, due to time constraints and end user feedback which indicated the desire for only the most common allocation techniques, these additional methods were not implemented. However, each of them could be trivially implemented by extending the module via the strategy interface (see 3.2.2).

2.3 LifeGuide

2.3.1 Overview

LifeGuide is a piece of software which the school of psychology in the University of Southampton use to run internet-based behavioural interventions, enabling the researchers to create, deploy and evaluate these online interventions. These interventions work in the following way: first, the researcher creates the layout and pages for the intervention and uploads it to their LifeGuide server. When users wish to participate in the trial, they go to the page which the researcher has set up and start filling in the data requested by the researcher. If this trial has distinct treatment groups, during the course of the trial every participant would be allocated to one of these treatment groups by the LifeGuide software, which could be based on the information the participant previously provided. A trial may progress for several months, where the users regularly update the system with new information. After a trial has finished, the researchers may export all of the data from LifeGuide in order to study it and perform statistical analyses.

The software itself can be subdivided into three parts: page layout, back end server and intervention logic. The page layout is done using a relatively simple *What You See Is What You Get* (WYSIWYG) editor, in which researchers can drag and drop objects they wish to display, such as text boxes and radio buttons, which have various options to set attributes, such as naming these objects. This is done on a page-by-page basis to construct the full trial design as seen by an participant. The researchers also have to construct any pages which may differ between the different treatment groups, if these are needed. After the pages have been constructed using the LifeGuide software, they will be exported into a well formed HTML page which will in the future be served by the LifeGuide server, and be presented to all the future online participants of the trial. The back end server will store the trial itself, how many participants are in each treatment group, the participant responses, as well as the participants' session data, where a session is defined as all actions generated from the time a participant logs in until the time they log off, consisting of various statistics such as how long they viewed each page.

To tie the distinct treatment groups to the respective subset of pages specific to each group, LifeGuide has to be able to distinguish between participants, and allocate them to different groups. This is where the logic code of LifeGuide act as a middleman between the data in the database and the serving of specific web pages. The logic code used in LifeGuide by the researchers designing the trial is a Boolean-like scripting language with a highly limited arithmetic capability. As there are no built in functions to do randomisation in LifeGuide, the code to control the allocation process has to be written by the researchers themselves.

2.3.2 Problems

One of the major issues with LifeGuide is its need for the researchers to write the allocation process themselves, which is why this project aims to alleviate that process. To give an overview of how convoluted the logic code is, we will use an example provided by the project client, which uses stratified blocked randomisation with a block size of 45. The trial shown has three treatment arms (usualcare, web and nurse1), which stratify on clinic and the participants' waist measurement.

```
#### practice 30+1 & HIGH WAIST #####
show q_patientenable_1w
savevalue (usernam, "enablecomplete", ...
after q_patientenable_1w if (group = "usualcare") goto groupuc
after q_patientenable_1w if (or (group = "web", group = "nurse1") )
                             goto q_att_int
stats increase "random_pra31w" by 1
after q_patientenable_1w if and (or(
getstat("random_pra31w") = 3, getstat("random_pra31w") = 4,
getstat("random_pra31w") = 8, getstat("random_pra31w") = 12,
...
...
...
getstat("random_pra31w") = 41, getstat("random_pra31w") = 43 )
,  savevalue (usernam, "group", "usualcare")
```

This shortened code sample shows the blocked randomisation of one treatment group (usualcare). The database variable *random_pra31w* is a counting variable which shows how many participants have been allocated in this stratified arm so far (the sum of all participants in the treatment groups usualcare, web and nurse1 in the stratified arm for practice 30+1 and HIGH WAIST). The statement *getstat("random_pra31w") = 3* compares the variable *random_pra31w* to the number 3 and if any of these Boolean statements is true the participant will be allocated to the treatment group usualcare. Knowing this, one can start to dissect the source code. What has been done here is that the researchers have generated a block of 45 entries (as per blocked randomisation) outside of LifeGuide, which would look like the list {nurse1,web,usualcare,usualcare,...,usualcare,web,nurse1} and then from that, generated a list of all the indexes each group is found, i.e. usualcare is found at indexes 3,4,8,12,...,42 and 43. They have to manually input these numbers in a long Boolean statement (each index represented as a statement of *getstat("random_pra31w") = index*) for every treatment group (e.g. usualcare). So in the example received, they have 30 practises with two stratified arms each, resulting in a total of 60 stratified arms, meaning they have to generate 60 random blocks of size 45, and then manually input these numbers for each treatment group for each stratified arm. As a result, in their logic file, this allocation process takes up roughly 2000 lines of static code, which in contrast, is roughly the same length as this report.

This generates major concern in several ways. Firstly, as this allocation process has to be manually written, it could be subjected to input errors, leading to unbalanced groups. Secondly, as each trial has to be unique, one can not use a previously generated block in a new trial, adding more time spent generating blocks. Thirdly, as the code is static, it hinders code reuse, as on such occasions a user would have to rename the variables associated, yet still have to input the new blocks. Fourthly, and one of the most important notes, this logic file describes the blocks in plain text, which would mean a malicious practitioner with the file could severely affect the trial by fully predicting participant allocations.

Due to the logic code's limited capabilities (especially arithmetic), and the inherent obfuscation of the code itself, using any form of adept randomisation method such as minimisation is nearly impossible. Due to this, researchers have only been able to use simple randomisation and blocked randomisation, potentially affecting trial outcomes as a result of unbalanced groups in terms of prognostic factors. When a more advanced technique is used such as stratified blocked randomisation, as it is in this case, the overhead of writing it is a major inhibiting factor, and may put researchers off of using a better method due to time constraints.

To conclude, the LifeGuide system's largest flaw is its use of its proprietary logic code, reducing available time for the researchers to focus on designing the trial, since a large portion of their time has to be spent coding obfuscated logic. This project's goal is to replace the 2000-odd lines of logic code, with more or less a simple function call, enabling more advanced randomisation methods and trial configuration options, and at the same time saving invaluable time for the researcher which an be spend focusing on the trial itself.

2.3.3 Comparison With Other Systems

There are a plethora of available solutions which could be used in conjunction with LifeGuide, though there are several obstacles which makes these options unfeasible. To begin with, as LifeGuide would require some sort of integration into their software, as it is undesired to redirect users to a new web page, none of the reputable companies running RCT-as-service provide any API which would enable it. With this in mind, no seamless integration is possible, introducing a major barrier for these systems.

Secondly, pricing is a large factor to take into consideration, as many solutions charge a set-up fee, with the most reputable company, sealedenvelope.com, (used by e.g. University of Cambridge and Imperial College London) charging up to £1100 as a setup fee if researchers wish to use the more advanced randomisation techniques such as minimisation. They will also charge a fixed cost for each participant allocated, charging from £3.10 per participant. Currently, as LifeGuide is used to run anything from small trials with tens of participants to large trials with thousands, such pricing would severely affect the number of trials able to run. This would have a negative effect upon undergraduate and postgraduate students wanting to run smaller trials using, for example, minimisation as a randomisation method, with such pricing making it an unfeasible option under limited funds.

Thirdly, as you use a third party to run the trial, you are limited to what methods and configuration options the third party provides, meaning any esoteric randomisation method cannot be used. In contrast, in the project's delivered module, a new randomisation method can trivially be implemented, giving the researchers more breathing room in regards to the available allocation options.

As such, the current state of randomisation services are either too costly, or do not provide any type of integrability, providing a sound motivation for the undertaking of this project.

2.4 Summary

The main goal of the project presented in this report is to produce a software module to be integrated with the current LifeGuide system. Its main goal is to assist researchers in setting up Randomised Control Trials, which are currently the standard in medical research. The main goal of using randomisation in such studies is to remove any bias which may potentially be introduced, and thus to support the credibility and statistical significance of the results. The LifeGuide system is currently very hard to maintain, using a very convoluted logic, which makes it highly challenging for anyone without a good background knowledge in programming and allocation strategies to set up trials. The developed module implements some of the most widely used algorithms currently - Simple Randomisation, Blocked Randomisation and Minimisation. This section presented a very detailed explanation of these algorithms, together with an in depth analysis and short discussion of any proposed alternative techniques. The following section will discuss the technical details of the exact implementation of the module and its design.

3 Technical Overview

3.1 Scope

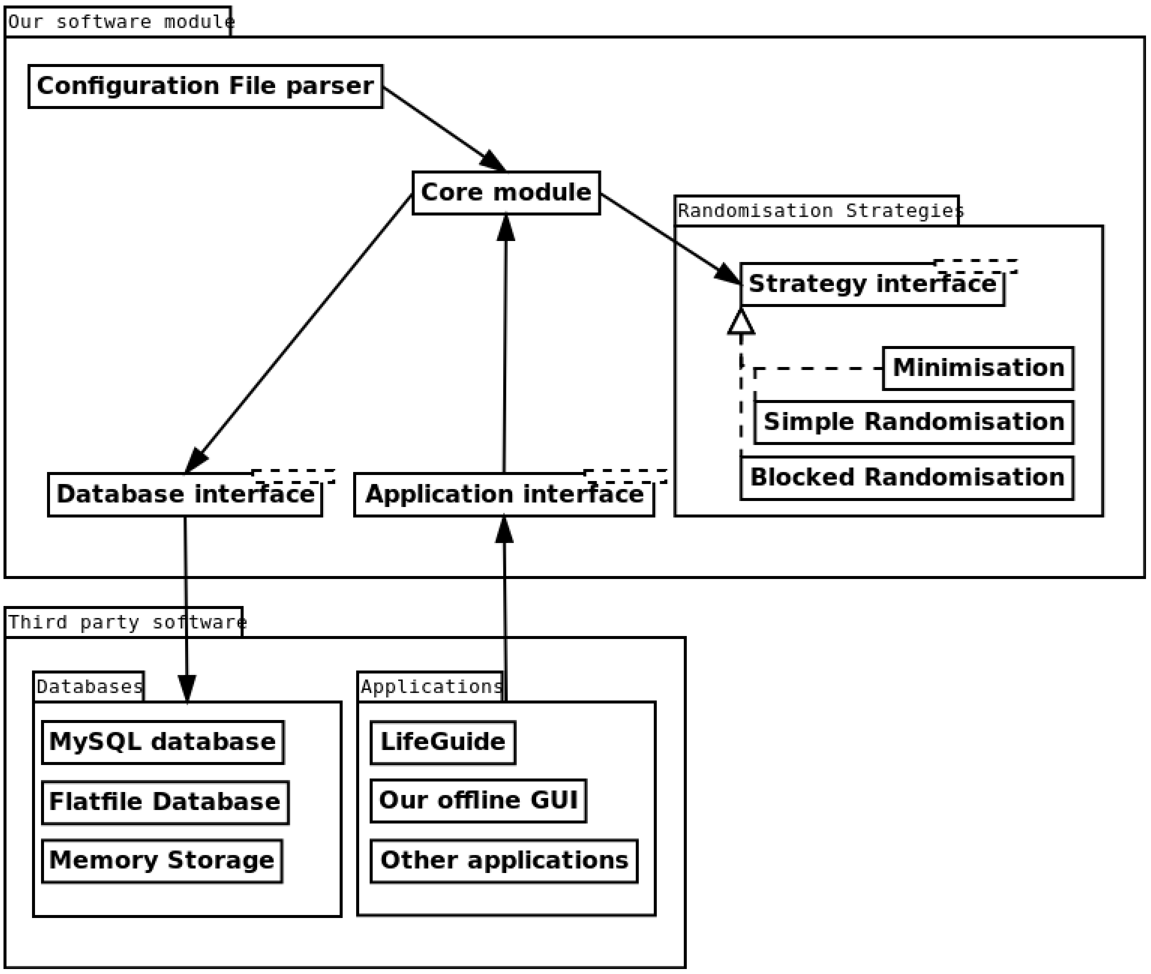


Figure 1: The scope of the software module.

The main goal of this project was to implement randomisation methods for running randomised control trails in conjunction with the LifeGuide software. However, as LifeGuide is considered overly complex to be modified in such short time span, it was decided that the software module would be a standalone package, with a high focus on extensibility and modularity. Figure 1 illustrates a simplified block diagram of how the module is self contained, showing that its only external interaction is through predefined interfaces.

As the main objective was to assign participants to a treatment group using various randomisation strategies, the definition of a strategy interface which standardises the communication between the core module and the allocation algorithms was drafted. This also allows for far easier extensibility in the future, if a need arises to add further strategies. There was also a requirement for various randomisation strategies to be implemented.

The application interface shown in figure 1 is what communicates between the software module and any third party applications wishing to integrate with the module. It consists of various methods with which the application may send participant data, receiving in return which treatment group the participant should be placed in. This can be sent either through the XML parser which was designed and implemented for the purpose of processing data from LifeGuide's existing database (see 3.2.2), or it can use a native Java function, supplying information using the internally defined classes. To understand why this is one of the most important parts of the project, one has to remember the 2000 lines of logic described in section 2.3.2. When the developer of LifeGuide integrates the module, these 2000 lines of logic (which researchers with little or no programming experience have to manually write out) would virtually be replaced by one function call.

As LifeGuide itself, along with its database, is overly complex and wasn't available for the team's use, it was required to use some type of persistent storage. The database interface was therefore designed in order to serve the needs of storage for the software module and the randomisation strategies. It consists of a few methods which handle serialisable data to be stored and read back from a database. As modularity and extensibility was a high priority in this project, it was decided not to constrain the system to one database management system, as a result the methods defined were kept as generic as possible whilst still allowing full functionality. This allows for easy integration with all kinds of databases, which has been tested using both a relational database (MySQL) and pure binary storage.

In all, for an application to be able to use the module, all that is required is to either adopt the project's custom MySQL database schema and implementation, or implement the requirements of the database interface. It also has to implement the requirements of one of the functions in the application interface which communicate between their application and the software module.

To identify what tasks were out of scope for this project, it is important to look at the existing functionality of LifeGuide. The first part which is considered relatively easy to use is the page creation in LifeGuide. As described in section 2.3, it serves its purpose to create web pages for the participants to view during the coarse of the trial. Secondly, as LifeGuide at the moment stores all of the participants' data in its database in the form of session data and response data, this is currently tightly integrated within the LifeGuide software, and would take far too long to either integrate with or modify to be possible within the time frame of the project.

3.2 Architecture Design

3.2.1 Module Design

The software design initially took the form of a high-level overview of the operations the module would be required to perform, taking into account the necessary input/output required for it to perform as desired. This resulted in figure 2.

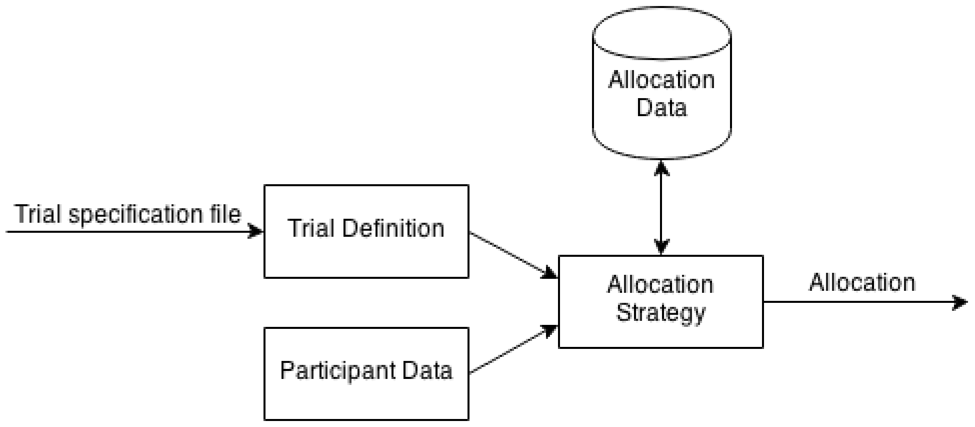


Figure 2: A high-level overview of the module's design.

Module Input
The module requires two key inputs: the trial specification file, and the data of the participant to be allocated.

The project specification required the implementation of an interface through which the user would be able to specify the setup of a trial, including its various parameters and attributes. It was decided that this would take the form a plain text input file. The main alternative was the implementation of a graphical user interface (GUI) which would allow a researcher to specify a trial by completing a form. It was decided that although this alternative would be slightly more intuitive than writing a text file trial specification, it would not be the best choice, as it would take much longer to implement. This GUI would also be have to be implemented into LifeGuide at a later date, increasing the difficulty of integration. Furthermore, a text-based specification would allow researchers to store the trial's set up locally, in a human readable format which could be easily modified should changes be required, or shared between researchers.

To derive a format for the participant data, it was necessary to investigate how LifeGuide currently provides participant information. It was found that a participant's data comprises of a number of attributes, which can be of several types: strings, integers or floating point values. LifeGuide's authoring tool allows gathering of these values through various form elements such as text input fields and drop-down menus. It was therefore necessary to consider two cases for data values. The first is the input of values which identify a specific group. For example, a user may be provided with a question asking if they smoke, with available options in a drop-down menu being "yes", "no" or "no, but used to". The system takes their response as a numeric value of 0, 1 or 2 depending on selection, which can be easily used by the module to group participants for stratification purposes. This differs greatly from the second case in which a participant is asked for a raw value in a text input field - their age, for example. In these cases, researchers may wish to put the participants into groups based on bands of values. For example, a researcher may group people who are less than 20 years old, who are between 20 and 50, and who are above 50 years of age. In these cases, the system must be able to take participant data as a raw value, then determine the group into which the participants are placed, to allow for stratification on these attributes. It was decided that the trial specification file would include some method of telling the module which attributes would be provided as group numbers, and which would be raw values (and in these cases, what the group ranges are).

Module Storage
Many of the allocation methods require a permanent data store to allow strategy-specific information to be retained between allocations, as participants will sign up independently over time. As an example, a trial using the blocked randomisation method must keep track of its position within the current allocation block, as well as the group assignments within the block. Two-way communication with the data store, whichever format it may take, allows the module to use the data during allocation, and update it once an allocation has been made. Further discussion of the specific kinds of data stored by the module can be found in section 3.2.3.

3.2.2 Module Implementation

The finalisation of the module's high-level design allowed for a subsequent low-level design phase in which the module's implementation details could be decided (classes, interfaces, database design etc.).

User Input Implementation
The first system element to be defined was the input format, specifically the plain text file which would be provided by researchers to set up a trial. This decision could then go on to inform the model for data in the system by defining the data which would be received. From preliminary discussions with researchers, as well as feedback from questionnaires (see section 5.2.1), the data requirements for the input file were found to be the following:

- Allocation method: the chosen algorithm to use to allocate participants.
- Allocation parameters (optional): values specific to the allocation method (e.g. block size for the blocked randomisation method, see section 2.2.2).
- Treatment arms: each of the interventions into which participants may be allocated.
- Allocation ratio (optional): the ratio of participants to allocate into each intervention (in cases where completely balanced groups are not desired).
- Participant limits (optional): a cap on the number of participants in a given treatment arm.
- Default treatment arm (optional): the intervention to allocate participants into once all others have reached their participant limit.
- Participant attributes: the attributes which must be recorded about a participant (e.g. their age, gender, exercise habits etc.), as well as the number of groups for the attribute (e.g. two for a boolean attribute), or value ranges for raw-value attributes (e.g. the ranges for an age attribute might be <18, 18-40, and >40).
- Stratification factors (optional): demographic factors on which to stratify to avoid the introduction of confounding variables (see section 2.2.3).
- Attribute weights (optional): the weights to give to each attribute for the minimisation allocation algorithm (see section 2.2.4).

The specification file format to input these details went through several iterations. The earliest were evaluated informally with researchers, asking for feedback on their design during requirement gathering meetings. Once the design was finalised, the group set up a user study to evaluate how intuitive the text file was to write for researchers. From this, it was possible to gain an insight into several issues with the design which had not been picked up on, and could then rectify these (see section 5.2.2 for details). The final format results in specification files with the following syntax. Note that this is not a particularly sensible trial setup, but is shown to illustrate each of the possible syntax elements of the text file.

```
Method: BlockedRandomisation
Block Size: 20

# Comments allow researchers to document their trial
Stratify: gender, age

Arms: pill1, pill2, placebo
Weight: pill1 2
Weight: pill2 2
Weight: placebo 1 # Comments may also be inline
Default: pill1

Limit: pill2 200
Limit: placebo 150

Priority: gender 10
Priority: age 5

Group: gender 3     # Male/female/unspecified
Group: smokes 2     # Yes/no
Group: smokeFreq 4  # Radio button groups

Group: age
<20
20 to 50
>50

Group: bmi
<27.5
>27.5
```

As displayed above, the researcher is able to input an allocation method (blocked randomisation in this case), any parameters (block size), and the factors on which to stratify the participants. This is followed by a listing of the various treatments in the trial, as well as the ratio for allocation. For example, in this instance there will be twice as many participants allocated to the ”pill1” and ”pill2” treatment arms than to the placebo treatment. Limits can also be specified. In this instance, there is a limit of 200 participants in the ”pill2” group, and 150 in the placebo group. Once these limits have been reached, the participants will be placed in the default group, specified to be the ”pill1” group.

The lower half of the file lists the attributes which will be recorded for each participant. This is where the distinction between group values and raw values mentioned earlier becomes relevant. For those groups where a participant will simply select one of a number of values, the researcher only needs to specify the attribute name and number of groups (as shown in this example for gender, smokes and smoking frequency). For those values which must be banded into ranges, the researcher simply uses a basic format which gives either the lower bound, upper bound, or both for the range.

Also shown in the above example are priority elements. These set weights for attributes, for use in the minimisation allocation method (note this example uses blocked randomisation, and these priorities are for demonstrating syntax).

Data Model Design

The finalisation of the trial input design informed the structure of the system's data model. Each of the necessary trial elements were set up in the following logical format:

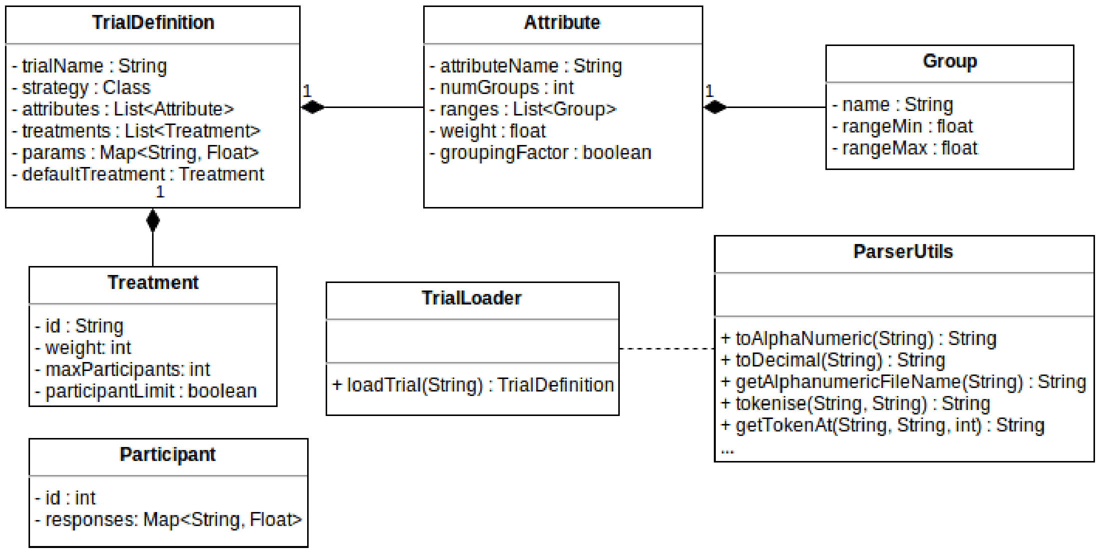

TrialDefinition
- trialName : String
- strategy : Class
- attributes : List<Attribute>
- treatments : List<Treatment>
- params : Map<String, Float>
- defaultTreatment : Treatment
1
1
Attribute
- attributeName : String
- numGroups : int
- ranges : List<Group>
- weight : float
- groupingFactor : boolean
1
Group
- name : String
- rangeMin : float
- rangeMax : float
Treatment
- id : String
- weight: int
- maxParticipants: int
- participantLimit : boolean
TrialLoader
+ loadTrial(String) : TrialDefinition
ParserUtils
+ toAlphaNumeric(String) : String
+ toDecimal(String) : String
+ getAlphanumericFileName(String) : String
+ tokenise(String, String) : String
+ getTokenAt(String, String, int) : String
...
Participant
- id : int
- responses: Map<String, Float>

Figure 3: The module's data model design.

This setup is mostly self-explanatory. Each loaded trial has an associated TrialDefinition, which stores basic details about the trial, such as its name, attributes and treatments. A reference to the Strategy class which will be used to allocate participants is also stored to allow client objects to perform allocation using only the trial definition object. The final element for a trial is a map of parameters which are specific to that trial. These are the allocation-specific parameters written into the trial specification by the researcher, such as block size, stored as floating point values to allow for the possibility of non-integer parameter values.

Attributes are stored with a name, a number of groups, and a weight (for use in the minimisation algorithm). A flag is also present which specifies whether or not this attribute should be used to group participants during stratification. If this flag is set, participants with differing values for this attribute will be put into separate groups during allocation. The attribute may also store a number of Group objects, which represent value ranges. If this list is empty, it is assumed that the value stored for this attribute per participant directly identifies the group which they are placed in (e.g. 0 for male, 1 for female). If elements are present in the list it is assumed that participants provide a raw value for this attribute, and so the group into which a participant is placed for this attribute must be derived by finding the appropriate range band. These ranges are stored in each Group object, which hold the minimum and maximum values specified by the researcher in the trial specification file.

Treatments are much simpler than attributes. All the system stores for each treatment is the string identifying it (for output/debugging purposes), a weight, the participant limit, and a flag specifying whether or not this limit should be used. In the context of treatments, the weight determines the allocation ratio. For example, a treatment with a weight of 2 would receive twice as many allocations during a trial as one which had a ratio of 1.

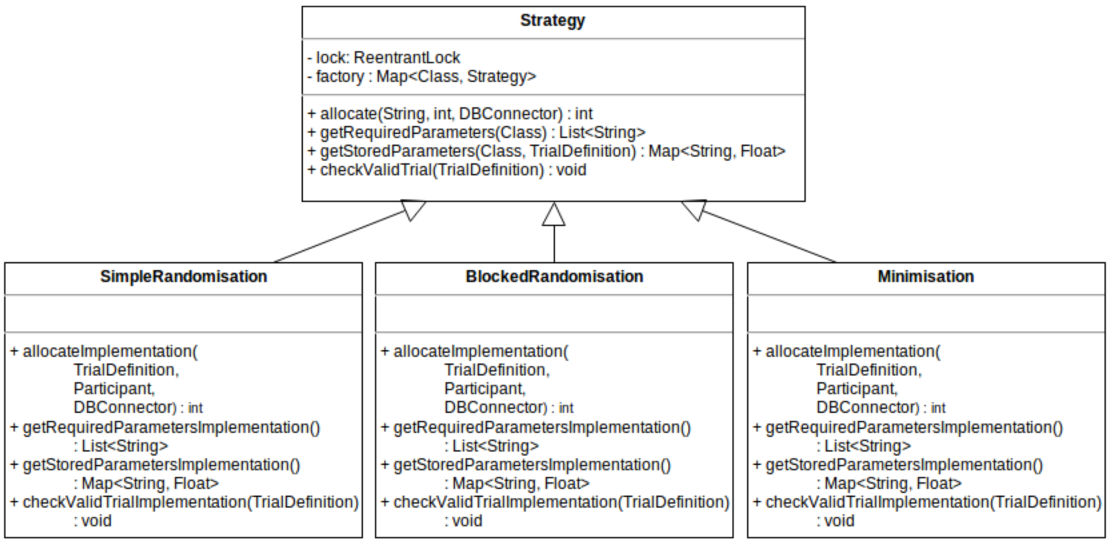
Each participant is treated internally as a combination of an integer ID and the participant's attribute values. Attribute values are stored as a map from the attribute's name to the value provided by the participant. During allocation these values can be used for stratification purposes.

The final element in the data model is a simple static function stored in the TrialLoader class which, upon being called with the file path for a trial specification, attempts to parse and interpret the file to create and return a TrialDefinition object. A utility class, ParserUtils, provides functionality to assist in the file's parsing, such as tokenisation, string manipulation and type conversion.

Allocation Class Design

In the initial stages of designing the module, it was decided that high importance should be given to ensuring the module is as extensible as possible, allowing for future additions in the form of extra allocation methods. Additionally, the module classes should require no modifications, allowing all extra functionality to plug directly into the system without altering the provided code. The allocation class designs were heavily influenced by this principle, with the result that it is possible for the module to be extended with new allocation methods through the implementation of a single extra class file.

The key to this design is the abstract Strategy class, which any new allocation methods should extend. This class provides an entry point for client objects wishing to


Strategy
- lock: ReentrantLock
- factory : Map<Class, Strategy>
+ allocate(String, int, DBConnector) : int
+ getRequiredParameters(Class) : List<String>
+ getStoredParameters(Class, TrialDefinition) : Map<String, Float>
+ checkValidTrial(TrialDefinition) : void
SimpleRandomisation
+ allocateImplementation(TrialDefinition, Participant, DBConnector) : int
+ getRequiredParametersImplementation() : List<String>
+ getStoredParametersImplementation() : Map<String, Float>
+ checkValidTrialImplementation(TrialDefinition) : void
BlockedRandomisation
+ allocateImplementation(TrialDefinition, Participant, DBConnector) : int
+ getRequiredParametersImplementation() : List<String>
+ getStoredParametersImplementation() : Map<String, Float>
+ checkValidTrialImplementation(TrialDefinition) : void
Minimisation
+ allocateImplementation(TrialDefinition, Participant, DBConnector) : int
+ getRequiredParametersImplementation() : List<String>
+ getStoredParametersImplementation() : Map<String, Float>
+ checkValidTrialImplementation(TrialDefinition) : void
Figure 4: The allocation strategy class structure.

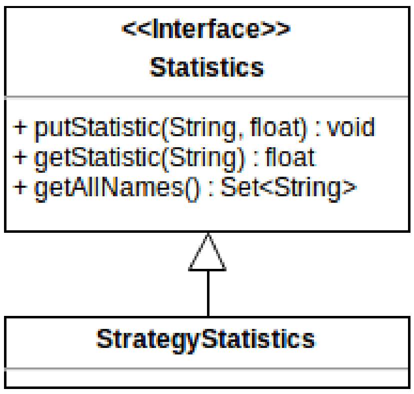
call the allocation methods, and also imposes the requirement of implementing certain necessary methods on its subclasses. The primary function of the Strategy class is the allocate method, called statically with a trial name, participant identifier and database connector (discussed later). This function uses the database connection to retrieve the trial details and participant information, determines the subclass which has been requested for the loaded trial, and calls the concrete allocate implementation from that subclass with the trial definition and participant information. For memory saving purposes, the Strategy class stores an instantiation of each Strategy subclass the first time it is requested, using this in any further calls to allocate which require the same allocation method. An important detail of the allocation call is that the Strategy class uses a lock to ensure that only one allocation is performed at any given time. The necessity of this can be shown by example. If a call to the allocate method is made for one participant using the blocked randomisation algorithm, the concrete implementation will be called. Within the algorithm's execution, it must check its position in the current allocation block, and assign the participant accordingly. If the algorithm retrieves the next allocation group, but is interrupted by another thread wishing to allocate a second participant with the same method before it can write the first, both participants could take the same slot from the block, thereby defeating the algorithm's purpose of striving for evenly balanced groups. The use of a lock around the allocation call prevents these issues.

The additional methods within Strategy operate in a similar fashion to allocate. Each method gets the class of the allocation algorithm required, and calls the concrete implementation within that subclass. In this way, additional algorithms are easy to define by simply extending the Strategy class and implementing the required concrete implementation methods shown in the three allocation strategies shown in figure 4.

The allocate implementation for a given strategy must use the details within a TrialDefinition and a Participant to choose the appropriate group. It is usually necessary to store intermediate data between allocations, so the function is also provided with a database connector. The result of a call to this function is an integer representing the index of the treatment group which the participant has been assigned to. A return value of negative 1 represents that allocation could not take place, whether this is because the participant is missing attributes, or because the trial is full.

Different allocation techniques may require different parameters. While simple randomisation requires none, block randomisation needs a specification of block size and a delta (how much block size can vary between blocks). A strategy therefore contains a method which returns a list of required parameters, allowing the TrialLoader to check for the existence of any of the parameters in this list within the specification file (flagging an exception if provided parameters are not required for the requested strategy). Similarly, strategies contain a method which returns a map of stored parameters, mapping parameter names to default values. Upon a trial being registered with the database, this information can be combined with a trial definition to permanently store all the parameters needed for the algorithm to run, using the user-specified parameter values if they were present in the specification file, or falling back to the default values from the provided map if the researcher did not specify the value themselves.

A final function is required for all strategy subclasses, which performs any strategy-specific checks on a trial to ensure it is valid for the given allocation method. For example, blocked randomisation requires that the allocation ratio across the treatment groups divides evenly into the block size. In this design, the parser is decoupled from the strategies. Instead of including a strategy-specific check within the parser, it can simply call the check method for the strategy, which must throw an InvalidTrialException containing an informative description of the problem. This avoids the need for modification of the parser code if further strategies are implemented, making integration and extension much easier.



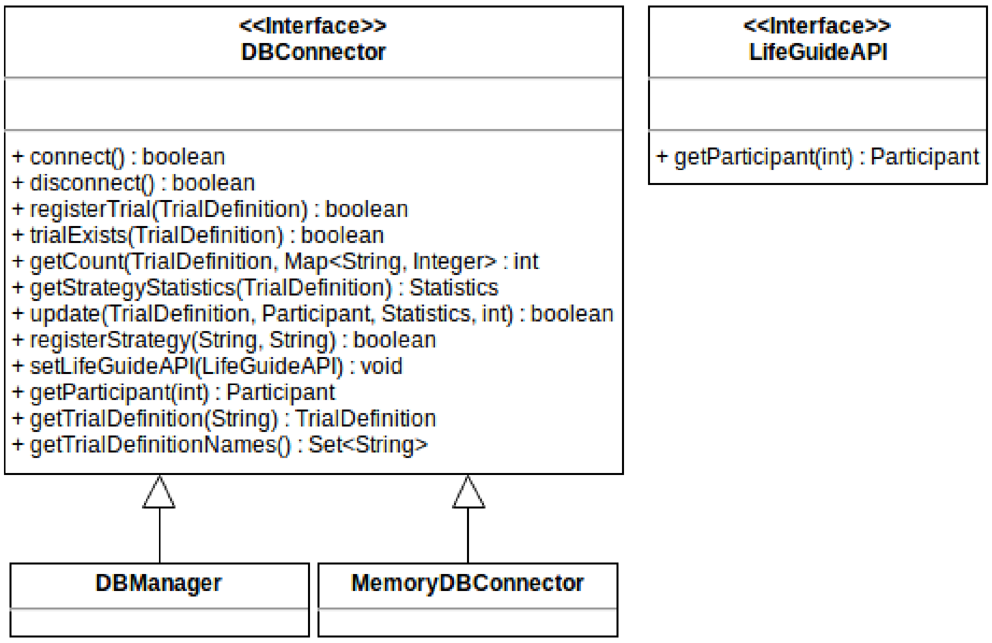
<<Interface>>
Statistics
+ putStatistic(String, float) : void
+ getStatistic(String) : float
+ getAllNames() : Set<String>
StrategyStatistics

Figure 5: Allocation statistics storage.

As mentioned above, strategies may need to store, load and update statistics to keep track of technique-specific values such as block position or running attribute weight totals. These are used internally as Statistics objects. In the currently implemented allocation strategies these are derived objects of the StrategyStatistics type. This stores parameters as a map from parameter names to floating point values. However, Statistics itself is simply an interface which provides certain methods, allowing the system to be extended to allow storage of strategy statistics in an alternate format should the need arise.

Database Interface Design

For the purposes of the project it was necessary to set up a database for storing trial, participant and allocation details. The database design and setup itself is discussed in detail in section 3.2.3. The module also required a wrapper around the database to perform the various operations it requires to extract, add and update data. It was decided that in this functionality, it was important to preserve the module's extensibility, decoupling and ease of integration. This led to a design which defines a single database interface, which may be implemented differently depending on the database which the module is integrated with.



<<Interface>>
DBConnector
+ connect() : boolean
+ disconnect() : boolean
+ registerTrial(TrialDefinition) : boolean
+ trialExists(TrialDefinition) : boolean
+ getCount(TrialDefinition, Map<String, Integer> : int
+ getStrategyStatistics(TrialDefinition) : Statistics
+ update(TrialDefinition, Participant, Statistics, int) : boolean
+ registerStrategy(String, String) : boolean
+ setLifeGuideAPI(LifeGuideAPI) : void
+ getParticipant(int) : Participant
+ getTrialDefinition(String) : TrialDefinition
+ getTrialDefinitionNames() : Set<String>
<<Interface>>
LifeGuideAPI
+ getParticipant(int) : Participant
DBManager
MemoryDBConnector

Figure 6: Database wrapper design.

The connect and disconnect methods required by the interface allow the database connection to be established or closed, and any setup/clean-up performed properly, provided that client objects use the interface correctly. Basic functions are available for checking existence of a trial, getting a list of all available trials, and getting a trial definition by name, as well as any associated strategy statistics. Methods are also specified for counting the number of participants in a trial which match a map of response values. This allows the allocation strategies such functionality as performing a single call to find the number of participants of a given stratification group.

The wrapper allows two types of data to be pushed: trials and strategies. Registering a trial should be performed on the first instance of a trial being set up, to ensure that the details are stored in the database and can therefore be retrieved on subsequent uses (i.e. when allocating additional participants at a later time). Strategies may also be registered with the database by providing the strategy name and class name. This is necessary each time a new strategy is added to the module, as it allows the database to reconstruct the strategy Class object stored in each TrialDefinition when a trial is requested.

As discussed in section 3.1, integration with LifeGuide is not within the bounds of the project's goals. However, LifeGuide's database already stores participant data, and so it is necessary for the module to make use of some generic API which can return a participant's data given the ID. This is the purpose of the LifeGuideAPI interface, which should be implemented during integration with the main system, and provided to the database manager for use when retrieving participants.

Two main implementations of the database connector have been included in the module. The first is the DBManager, a MySQL wrapper. This is the likely candidate for use after integration has taken place, as it successfully implements all of the data storage functionality required for online participant allocation using the DBMS currently used by the LifeGuide system (see section 3.2.3 for more details). However, for testing purposes the MemoryDBConnector was also created. This shows the flexibility of the DBConnector interface, as it does not connect to a literal database, instead holding all data in the program's memory as lists and maps. Data is lost after the program ends, but this class proved extremely useful for testing the various allocation strategies without regularly flooding the MySQL database with test data.

Utility Classes

Two additional utility classes were implemented which don't fit into any of the previous categories.

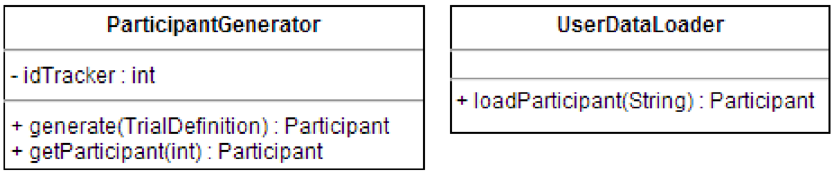

ParticipantGenerator
- idTracker : int
+ generate(TrialDefinition) : Participant
+ getParticipant(int) : Participant
UserDataLoader
+ loadParticipant(String) : Participant

Figure 7: Utility classes.

One of the key goals arising from the requirements gathering stages of the project was that the module should be made as easy to integrate into the existing LifeGuide system as possible. LifeGuide's database already stored participant data, so for easier integration, the module provides code which takes user data in the form of XML (the current storage method for participant information, see section 3.2.3), and parses it to produce a Participant object which can be handled by the module. This should speed up the integration process, as it removes the need for input modification on the part of the LifeGuide system, making the module more of a "black box" system.

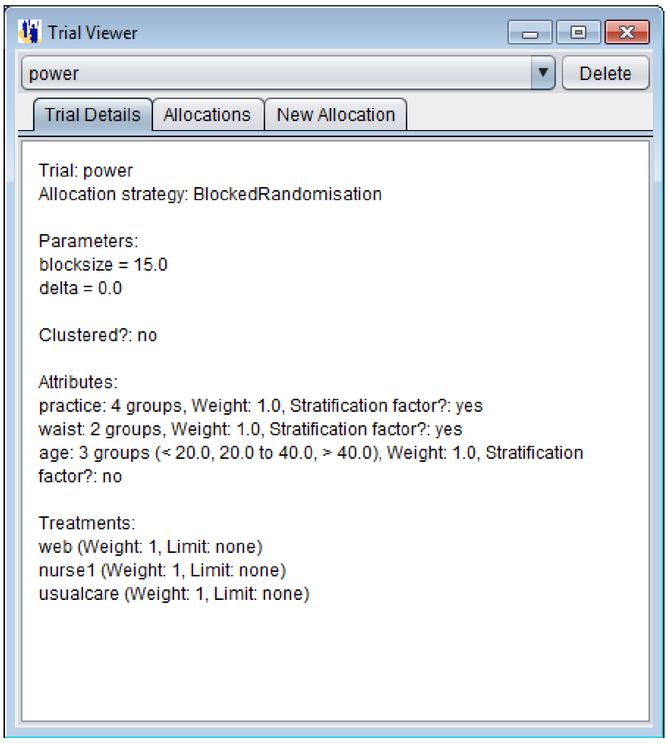
The final implemented class is a participant generator. For the purposes of testing, it was necessary to be able to generate large quantities of participants with random attribute values, to ensure the correct functionality of the allocation algorithms. The class returns participants with incrementing identification numbers, where for each attribute the participant either has a random group, or a random value, depending on whether the attribute uses raw banded data or not.

Extension Design
Two potential extensions to the project were proposed in the requirements gathering stages. The first was the possibility of providing a means of monitoring the participant sign-ups and allocations over the course of a trial. However, given that the project's output is a module to be integrated into the system, it can't be determined what format the final merged participant storage will take, limiting the ability to effectively implement such a system.

The alternative proposal for additional work was the creation of a standalone, offline application which could be used outside of LifeGuide to allocate participants into a trial, for cases which do not require LifeGuide's web authoring tools. Sufficient time was available following the implementation of the core requirements to develop this application.

Offline Application Functionality
The application's primary goal is to allow participants to be allocated either one at a time, or in batches, given a trial specification file. The user interface is small and simple, to ensure that it is as intuitive as possible to use. Users are presented with a drop-down list from which to select the trial they wish to work with, as well as a delete button to remove trials from the system when no longer needed. The main area of the user interface contains three tabs, each providing a different function. The first (figure 8) shows the details of the currently selected trial in a human readable format, to allow researchers to verify that the trial specification file they have input has been interpreted correctly by the system. In order to load a new trial, the user must open the drop-down menu, and select the "Load a new trial" option, which will bring up a standard file browser to allow the user to select the trial specification file to read in.

The application also provides a detailed breakdown of the allocations which have



Trial: power
Allocation strategy: BlockedRandomisation

Parameters:
blocksize = 15.0
delta = 0.0

Clustered?: no

Attributes:
practice: 4 groups, Weight: 1.0, Stratification factor?: yes
waist: 2 groups, Weight: 1.0, Stratification factor?: yes
age: 3 groups (< 20.0, 20.0 to 40.0, > 40.0), Weight: 1.0, Stratification factor?: no

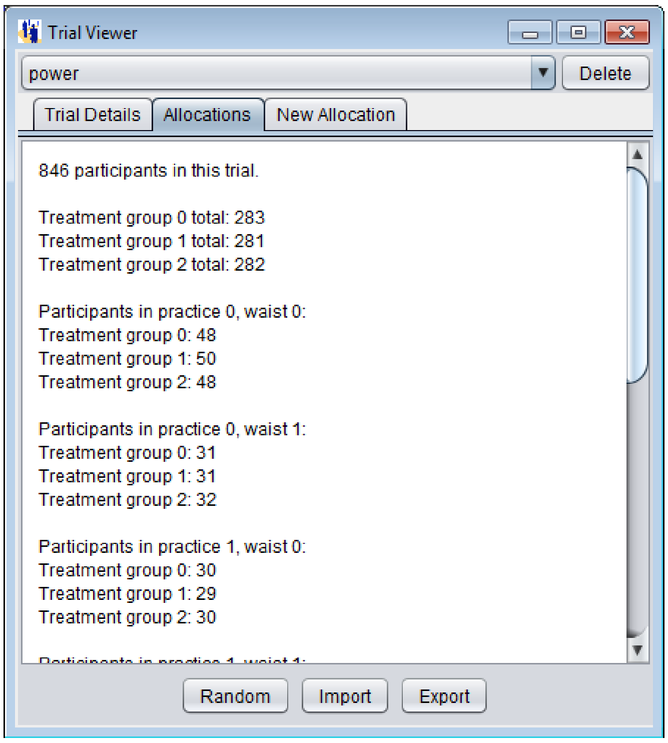
Treatments:
web (Weight: 1, Limit: none)
nurse1 (Weight: 1, Limit: none)
usualcare (Weight: 1, Limit: none)

Figure 8: The standalone allocation application's first tab.

already been made (figure 9), showing not only how many participants are on the trial, but also how many have been allocated to each treatment group, as well as the treatment group distribution within each stratified group. Also featured in this application tab, as well as in the third, is a set of buttons which provide additional functionality. The researcher is able to import participants from a comma-separated variable (CSV) file, allowing participants which have already registered to take part in the trial to be allocated in bulk. A researcher can also export a CSV file containing the treatment group allocation for each participant, for further use in running the trial. Also present is a "random" button, which allocated a number of random participants using the participant generator. This is included for demonstration and testing purposes, but if the LifeGuide team wish to distribute the application post-integration, it would likely be necessary to remove this.

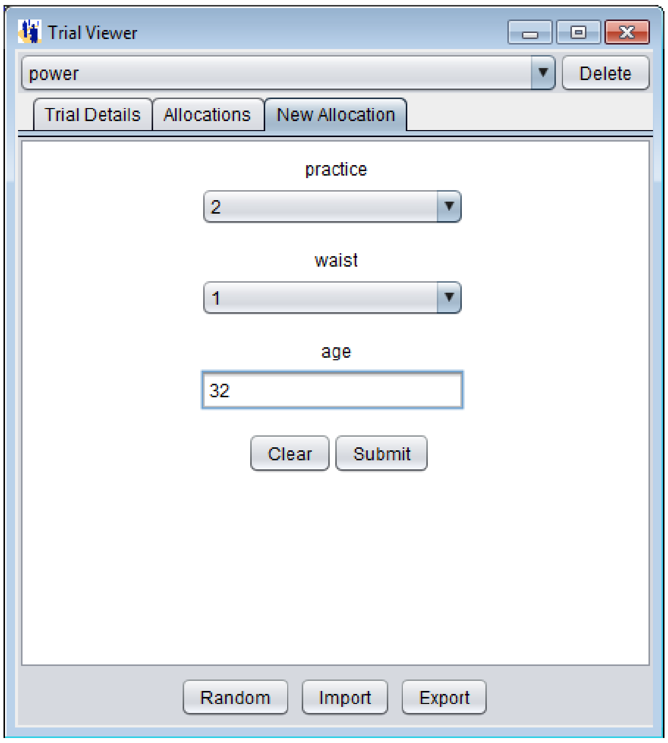
The final tab contains a dynamically generated form, allowing participants to be registered one at a time by entering the specific attribute values for that participant. Upon submission, the participant is allocated, and a message is presented to the user showing the allocation result.

Any errors which occur during the application's execution are displayed in a panel which is added to the bottom of the window, and can be dismissed by clicking the error. This provides the user with feedback on the system's success or failure, allowing them to better comprehend the effect their actions are having on the system's internal state.



```
846 participants in this trial.

Treatment group 0 total: 283
Treatment group 1 total: 281
Treatment group 2 total: 282

Participants in practice 0, waist 0:
Treatment group 0: 48
Treatment group 1: 50
Treatment group 2: 48

Participants in practice 0, waist 1:
Treatment group 0: 31
Treatment group 1: 31
Treatment group 2: 32

Participants in practice 1, waist 0:
Treatment group 0: 30
Treatment group 1: 29
Treatment group 2: 30
```

Figure 9: The standalone allocation application's second tab.



practice: 2

waist: 1

age: 32

Figure 10: The standalone allocation application's third tab.

The application manages data using an extension of the DBConnector interface mentioned earlier (figure 6). This made it possible to set up the system without any modification of the existing database, allocation or parsing code, showing that the project deliverable achieves its goal of being as self-contained as possible, and easy to integrate into existing applications. By implementing the various methods required by this interface, as well as adding several specific to the application, it was possible to set up a database which stores and manages data in the form of local text files. These files are loaded by the application on start up, and written to when it is closed. To avoid the possibility of overwriting data in the event that two application instances are open at once, the application creates a lock file, which can only be held by one instance of the program, ensuring that only one application instance may run at any given time.

3.2.3 Database Design

LifeGuide currently uses a relational database to store all information that is needed in order to carry out randomised control trials.

Since the aim of this project was to extend LifeGuide, it was decided that it would be wise to store the data required by the developed module using a relational database as well. This was also a core part of the functional requirements.

The Database Management System (DBMS) that LifeGuide currently uses is MySQL[1], and as a result it was decided to make the created software work with MySQL as well. The software drivers needed to allow a Java application to query and manipulate a MySQL database already exist, so this was not an issue.

Data currently stored by LifeGuide

The data that the LifeGuide software currently stores is of a varied nature, and serves more purposes than just allocating a participant to a trial. A subset of the data currently stored by LifeGuide includes participants' answers to all of the questions of an intervention, session data that describes user interactions with the web page of a given intervention, and the logic for allocating a participant in a given intervention. The allocation logic is hard-coded into a file, whereas the session data and participant answers are stored in XML format in the database.

Not all of the database's stored data is used for allocating participants into the treatment arms of an intervention. The only information needed for allocation is the answers of the participants, which are input into the allocation logic. The rest of the data is used purely for tracking and recording participants' behaviour, for post-study analysis.

With its existing schema and mode of operation, LifeGuide's database size is approximately 75 gigabytes, which makes it difficult to maintain, store and manipulate.

Data needed by the implemented algorithms

The currently implemented algorithms (blocked randomisation, minimisation and simple randomisation) require a number of different parameters in order to work. For example, blocked randomisation requires the following variables:

- Block size
- Delta - maximum block size variation
- Actual size - size of the current block
- Block seed - random number seed for generating block contents
- Counter - current position in the block

[1]MySQL website : http://www.mysql.com/

Simple randomisation requires no stored parameters, and the parameters that are required for minimisation to perform an allocation are arbitrary in number and are specified by the researcher who defines the intervention. An additional parameter that can be specified for the minimisation algorithm is the determinism factor (whether or not to use a probabilistic component, see 2.2.4).

Furthermore, each intervention has its own set of parameters specified for these algorithms, otherwise they would be of little practical use.

Database considerations
Coming up with a concrete database design was not trivial. Firstly, it was necessary to identify the data to be stored in the database for the allocation module to be useful, and then to come up with a database schema which would be efficient and extensible, while conforming to existing relational database design principles.

Identifying the information to be stored in the database was relatively straightforward. The database needed to store everything that would be helpful in allocating participants for a given intervention. The information stored can be roughly split in two groups, namely participant answers, and intervention information, where intervention information includes a variety of data. Intervention information includes the questions that constitute an intervention, details on the treatment arms of an intervention, which treatment each participant has been allocated into, information on the currently allocated participants, and the various variables which will be used in order to determine the allocation of a new patient, as well as parameters required by the algorithms, if any.

The database schema itself was also a challenge. It was necessary to create a schema that was coherent, yet easily extensible should someone wish to add new algorithms in the future. This meant that it must account for algorithms with an arbitrary number of parameters and interventions with a variable number of questions, amongst others. Various schemes were proposed, including having a separate table for each intervention's answers and questions, accompanied by a configuration table to store shared information. Although this approach seems quite natural and straightforward from an organisational point of view, it violates several rules of the relational model and could create a very fragile system. Unless special precautions are taken, frequent or total collapse of the system is very realistic (e.g. users might input MySQL reserved keywords as table names) with an non-standard database schema.

Even though a good designed was not critical or mandatory for the project, it was deicded as a group that a well designed database would be beneficial for everyone involved with LifeGuide. A well designed database would make debugging and inspection much easier, would allow for less obscure code and would allow other developers to easily continue development since the only required knowledge on their part would be the database schema. The final schema implementation is presented in Figure 11, with explanations of what each table contains, followed by a more detailed explanation.

Database design
It is evident from Figure 11 that the key table in the design is the `Intervention` table. The `Intervention` table will store information on an intervention, namely its auto-generated numerical identifier, its name, the name of the randomisation method to be used and a set of cluster factors which are a set of numeric identifiers referring to the list of this intervention's parameters, allowing one to perform clustering.

The questions asked by each intervention are stored in the `Attribute` table. Each row in this table represents a different question of an intervention, and has its own unique identifier, a foreign key to its intervention, a name, a weight (represents the importance of the attribute), a boolean to indicate whether it is a grouping factor (stored as a character array) and the number of groups of this attribute, in case the answer is picked from a list. This schema allows for interventions with an arbitrary number of questions. In contrast, if the schema had a table whose columns represented each of the questions for any given intervention, this would impose a global limit, which might not suit every researchers' needs.

Closely associated with the `Attribute` table is the `Groups` table, which stores a given option from the (potential) list of options for a given attribute. Each row in the `Groups` table represents a different answer option and each option has a unique numeric identifier, a foreign key indicating which attribute it belongs to, a name, a minimum and a maximum range.

The `Intervention_Params` table is a collection of additional variables used for allocation purposes. In contrast with the values stored in the `Attributes` table, these quantities represent scalar variables. Each row in the table represents an new variable and has a unique numeric identifier, a foreign key that refers to its parent intervention, a name for this variable and a float, which would be the numeric value for the response.

The `Strategy` table serves mostly for configuration purposes, as it contains a list of each randomisation algorithm's code name and fully qualified class name. The reason for this is that using a code name would allow the underlying implementation to change, while still using the same code. The class name needs to be fully qualified because the created module loads each class dynamically when an allocation needs to be done. This also allows for different allocation strategies with the same experiment, should one wish to do so in order to compare results.

The `Strategy_Statistics` contains information on the variables used by the algorithms that are currently in use for implementation, for example the current seed for the current block in a blocked randomisation scenario. Each row in the table represents a different parameter for the randomisation algorithm and is represented by a unique numeric identifier, the name of the variable, a value for the variable and a foreign key referring to the parent intervention of this variable.

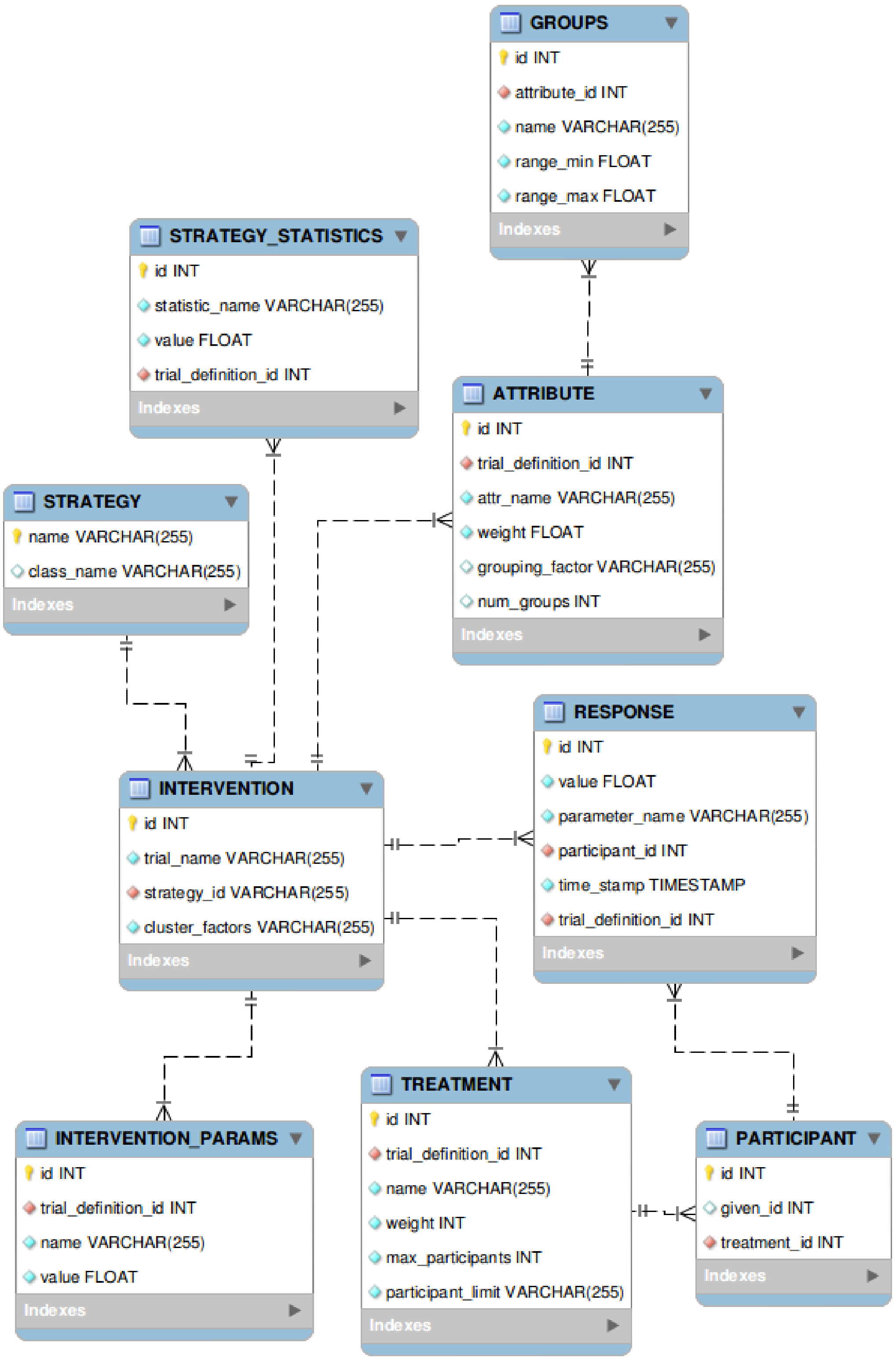


Figure 11: Current database schema for randomisation module.

The `Treatment` table contains all the treatment arms for a given intervention. Each row in the table contains a unique numeric identifier, a foreign key that refers to the id of the intervention it belongs to, a name for that treatment, a weight parameter that specifies how important this treatment arm is in relation to the other arms of the intervention, a maximum population limit and a boolean value (identified by the `participant_limit` field) that indicates whether there is a maximum participant limit for this treatment arm.

The `Participant` table is simply a mapping between a user id and a treatment id, which is a foreign key referencing the `Treatment` table. Records in this table are being added after an allocation has been performed. The role of this table is to give information on participant allocation to treatments without having to run the participants against the allocation algorithms each time. While in trials with a small number of participants this might not present an issue, larger trials will have a performance benefit due to the smaller number of database requests and less computational overhead.

Finally, the `Response` table contains the responses submitted by the trials' participants. Every row in the table represents the response to a different question and contains a unique numeric identifier, the value of the response, the name of the response's correspondent parameter, a foreign key referencing the participant id that submitted this response, a time-stamp of the submission date and time and a foreign key referencing the intervention id that this response belongs to.

Database performance

Trials can be of varying size in terms of participants taking part. During the group's meeting with the developer, it was found that experiments with as many as 1500 participants are carried out in LifeGuide. In order to see how the software module copes with these numbers, it was necessary to run some tests.

The group set up a MySQL database in the `linuxproj` server and had three machines in the Zepler undergraduate labs simulating running interventions, each having 2000 participants. Each intervention had 5 attributes, in an attempt to simulate an average intervention.

The results were satisfactory, as the queries did not interfere with each other and the latencies were not noticeable. This experiment was repeated two more times, achieving similar results on each run. The database size after these trials was just under 5 megabytes.

3.3 Functional Testing

3.3.1 Methodology

In order to assure correctness and stability of the system, different rigorous test were employed. During the process of development the team had the goal of developing unit tests alongside any class being produced. This methodology minimised the number of bugs occurring later in the development cycle, and allowed each member to produce and verify their code during the development stage. All the tests were automated using Maven's built in test cycles. Before a member of the team was to submit anything new, he had to run all tests on all classes to guarantee consistency and correctness of the code. After passing the unit tests and basic functionality, stage two was harness testing and extreme case testing. In both instances, members of the team would come up with very unlikely scenarios, where potential problems might occur, and would test the working of the system in these cases, or would test the system's performance under high usage.

3.3.2 Test Execution

Unit Testing
Unit testing was essential to guarantee faster development and reduce potential risks in the future. Each of the major functionality classes in the repository has unit tests in the test folder, to assure that the code was working as expected. Most unit testing covered the algorithms and the database. The algorithms were tested based on the statistics produced at the end of each iteration, to ensure that the effect on the allocation groups was as desired. The DBManager class was also tested for every single interaction with the database to assure the storage that the system created would not produce any unexpected results or side effects.

Algorithm Testing
After the unit tests each algorithm was tested rigorously, checking the results produced on different types of population. For this purpose, a helper class for auto-generation of synthetic participant data was created, which allowed generated participants to be passed to the algorithms for allocation. Each of the tests investigated different aspects of the results of the algorithms. The randomisation part of the algorithms was guaranteed to be working correctly because it was implemented using the native java Random class for generating random numbers, which have been extensively used in many applications. The second very important aspect of each of the algorithms is their capability of retaining balance between groups. This is one of the hardest thing to assess due to the inherited randomness in the allocations.

For simple randomisation we selected a margin of error or unpredictability of the algorithm in the expected number of people being assigned to each group. This margin was based on the sample size, as we knew from the theory in section 2.2.1 that the error decreases with the size of the sample. For further reassurance of the results each test was run 100 times. Blocked randomisation inherently guarantees that there is a balance between the groups. Instead the algorithm was tested to ensure that it uniformly produces different sizes of the block. Another test performed was to check that the shuffling of the blocks does not repeat (i.e. duplicate blocks), which could lead to increased predictability.

The minimisation algorithm guarantees the balance between groups based on the prognostic factors to be achieved. We designed several automated tests which would check both the deterministic variant of the algorithm and the probabilistic. When the deterministic one is used, one can achieve perfect balancing, while the probabilistic alternative makes it impossible to guarantee at any point the exact distribution of the participants, so it is possible only to check the results up to some threshold of randomness. The last tests were run 100 times in order to be certain that there aren't any cases in which the algorithm fails to balance the groups.

DBManager Testing
The DBManager class was tested for several different very important aspects. The first one was assuring that the interaction with the database is correct, and there isn't any inconsistent or irrelevant data being stored. The second aspect included verifying that all the data stored after calling the algorithms produces the expected statistics and that they update correctly. The final test was a harness test that the system can take high load of requests to the allocation method and does not fail either on the module or database side of execution.

Manual Testing
Several manual tests were performed in order to verify aspects of the system and the expectations of the results. These tests included testing the file parser for different specification files for setting up trials, deploying the module on a new machine with a new database to assure it creates any required tables without manual intervention, and testing that the offline application communicates correctly with the module.

3.3.3 Results

When testing the final system, all of the unit and second-stage tests are passed. During implementation phases, the automated testing provided the team with a rapid way of debugging and analysing new development code, which was very productive. The second stage testing of the algorithms is summarised in figure 12. The boxes represent 50% variance spread between the 25th and 75th percentile. We can see that the theory matches the test results almost exactly. The produced figure shows the results for a trial with three treatment arms which are equally weighted, for every algorithm on different population sizes, displaying the distribution of one of the arm's allocation in proportion to the whole population. As is evident, Simple Randomisation has quite significant variance on small numbers of participants, but converges very accurately when the size of the population exceeds 1000, and the empirical results match the calculation in section 2.2.1. Blocked Randomisation shows faster convergence and much less variance. It is important to note in figure 12 that for population sizes of 60 and 150, it looks like the randomisation algorithm has a bias towards under sampling this treatment arm, but this is an artifact of the discretisation, and the fact that the box rectangle shows exactly 50% of the variance. To explain it, we note that in Blocked Randomisation the only possible values of the proportion close to the mean, for the given sizes of population and algorithmic parameters, are 0.3,0.33 and 0.36. Given that the three of these values cover more than 80% of the variance, the visualising software was forced to select only one of the near values to 0.33 as the spread. As expected from the theory, minimisation has the best convergence rate, working perfectly even in very small samples when using a probability of 0.8 for the algorithm. This graphic can be a very useful guide for any user of the system when selecting an appropriate algorithm. The tests were generated by running each algorithm 1000 times for one fixed population size in order to guarantee no outlier effects.

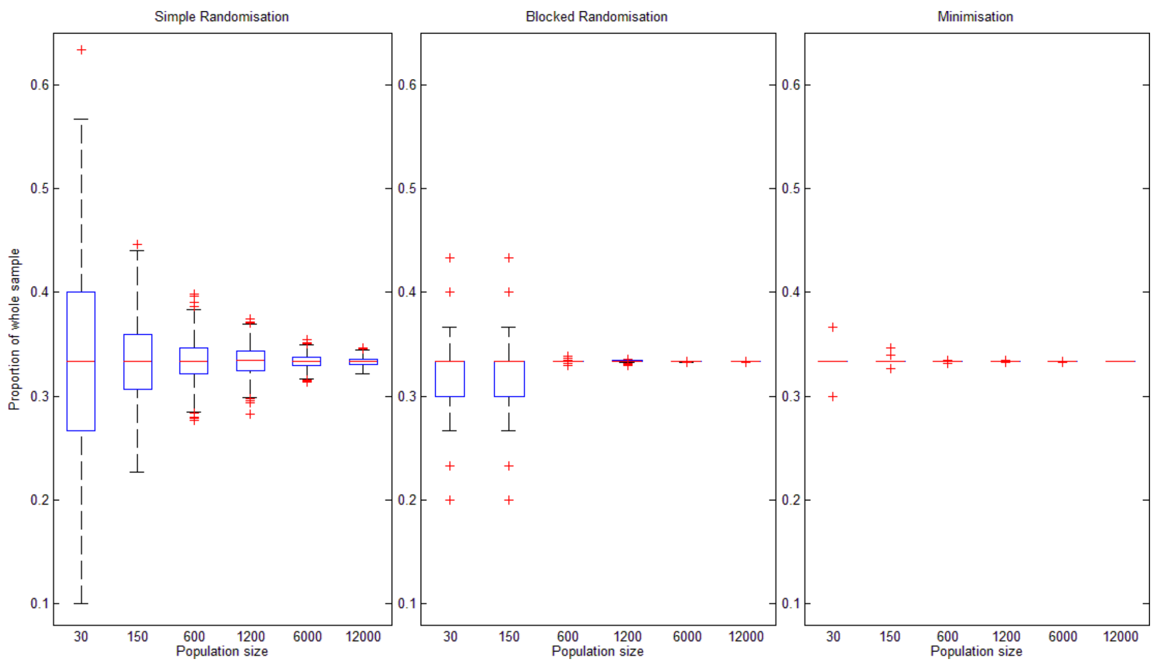


Figure 12: Comparison of the algorithms for three equally weighted treatments. Blocked randomisation uses block sizes of 12, 18, and 24. Minimisation uses a probability of 0.8.

4 Management

4.1 Planning

4.1.1 Initial Plan

After reading the project brief, it was decided that it was necessary needed to investigate the existing software (LifeGuide Authoring Tool) first, in order to gain better insight into what had to be done in order to set up an intervention. In addition to looking at existing software, the first couple of weeks were also invested in identifying and reading the relevant literature, as discussed in Section 2.1.

Having understood what is involved in the process of creating an intervention, and the requirements for performing an allocation of an individual, the group was able to start decomposing the model of an intervention into sub-components, in order to create a class diagram and an accompanying database schema. This decomposition helped identify relevant tasks to be carried out, and it was then possible to create our Gantt chart (Figure 13) in order to set a well defined schedule for carrying out the work.

After this initial phase and the creation of the appropriate UML diagrams, namely a class diagram and a database schema, responsibilities were assigned to the team's members. Group members were given development responsibilities by preference, and these were organised in such a way so as to maximise the group's utilisation and the number of tasks being carried out simultaneously. This is evident in the Gantt chart, which shows many of the tasks overlapping, allowing the group to complete the project faster.

The created Gantt chart overestimates the time requirements for most of the tasks, and introduces some contingency time, in an effort to allow the group to handle any unexpected failures or disruptions to the development schedule which might threaten the quality of the final delivered product, or the possibility of meeting the project's deadlines.

4.1.2 Progress

Although Figure 13 represents the initial Gantt chart, the group self-organised and worked in such a manner that it was not necessary to change the Gantt chart. The initial Gantt chart corresponded to our work schedule until approximately the end of the semester, where it was necessary to modify it in order to reflect some extra work carried out, described in the next section.

The yellow line in Figure 13 represents the project progress at the time of the first progress seminar. As shown, the group was fairly ahead of schedule. This was achieved without decreasing the the quality of the written code (i.e. introducing difficult to read, maintain or extend code) or the features that the module offered, due to the group's development practises (discussed in section 4.2.1), as well as having slightly overestimated the time requirements for some parts of the project.

Another aspect of development that helped maintain quality from the early stages through to the final deliverable was continuous code iteration. When a change was made to the code base, due to a bug fix or requirement implementation, the group iterated through the affected code base to ensure that the changes in the code did not affect the existing functionality. This iteration included reviewing source code that was affected and doing regression testing by exercising the code through the test suite. Through this process of continuous iteration over the code base, the team members were able to familiarise themselves with the code written by others in the team, and as a result decrease development times.

At the time of the first progress seminar, it was clear that the group would be able to meet the project deadlines, and perhaps expand the scope of our project or otherwise incorporate additional features that the clients would like, providing a strong motivating factor.

4.1.3 Discussion

Figure 14 shows the final Gantt chart, revisited the week before the second progress seminar. The main changes in the Gantt chart include the addition of a finalised time slot for writing the final report, and research into new features to implement.

The final Gannt chart shows that at the time of the second progress seminar, the project was still ahead of schedule (indicated by the yellow line), but not by as large a margin this time. At the time, the project's focus was on client meetings and making sure the clients agreed that the project is going in their desired direction. Furthermore, the team was heavily focused on coming up with extensions which could be implemented on top of the original scope, mainly by looking at what other software on the market offers and by conducting some interviews with the clients. For these interviews to happen, it was necessary to produce a short questionnaire and a follow-up study, as well as conduct the study in person with the volunteers. Development was not the central concern at this time, since coding of all the requirements was complete, and the resulting module was passing all of the verification tests.

Another activity that was in the centre of the group's attention was testing. It was necessary to be absolutely sure and have evidence that the module was functioning properly, since it would be used in an existing system that people expect not to fail.

Even though it might be reasonable to state that overestimating the time requirements for components would mean that the group was limited in terms of features of the created module, in actuality, by allowing extra time, it was possible to be sure of the quality of the code fulfilling the core requirements. Furthermore, because the core requirements were completed ahead of schedule, it was possible to further extend the scope of the project with the creation of an offline allocation tool, as explained in section 3.2.2.

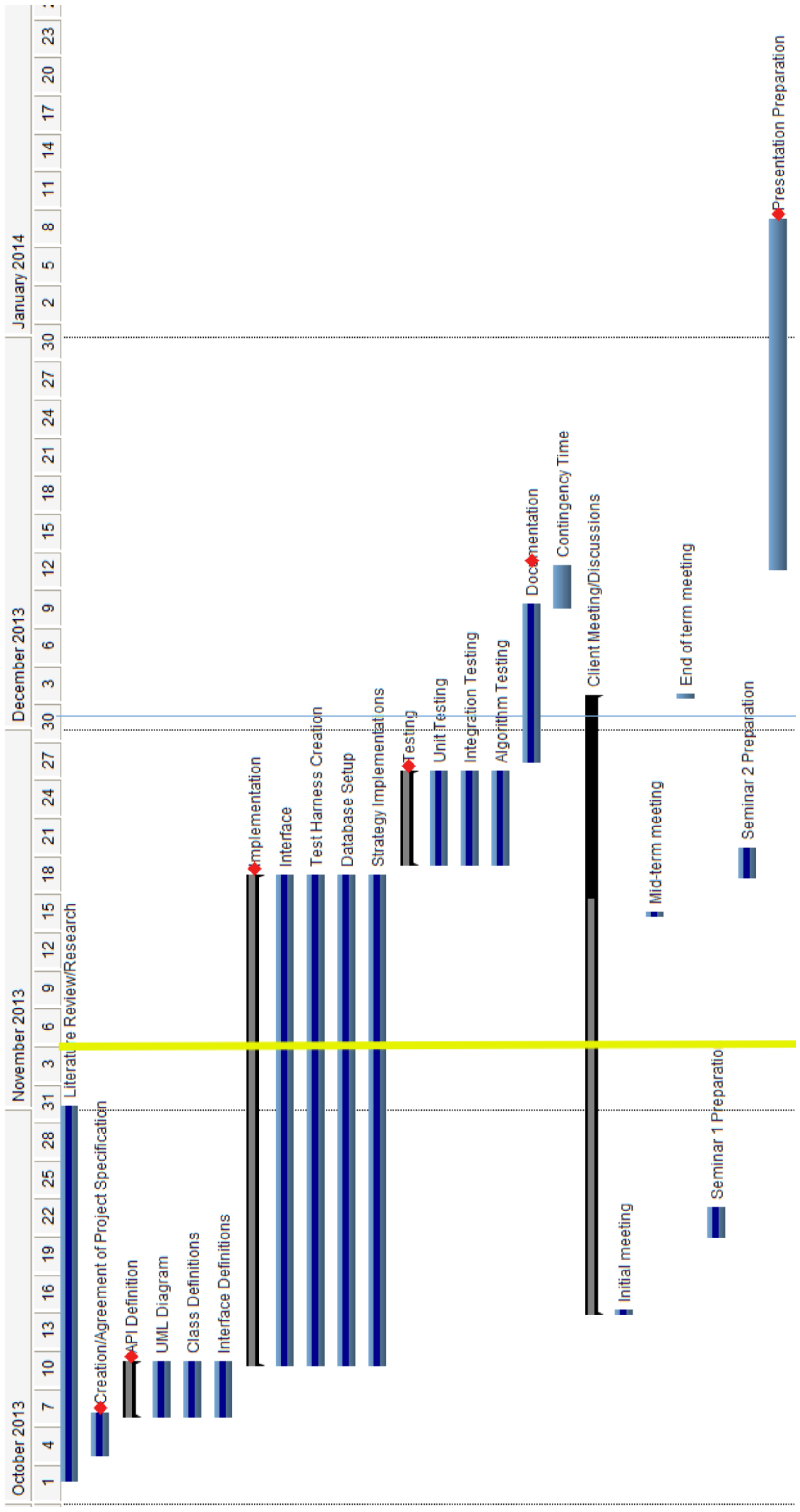


Figure 13: The initial Gantt chart.

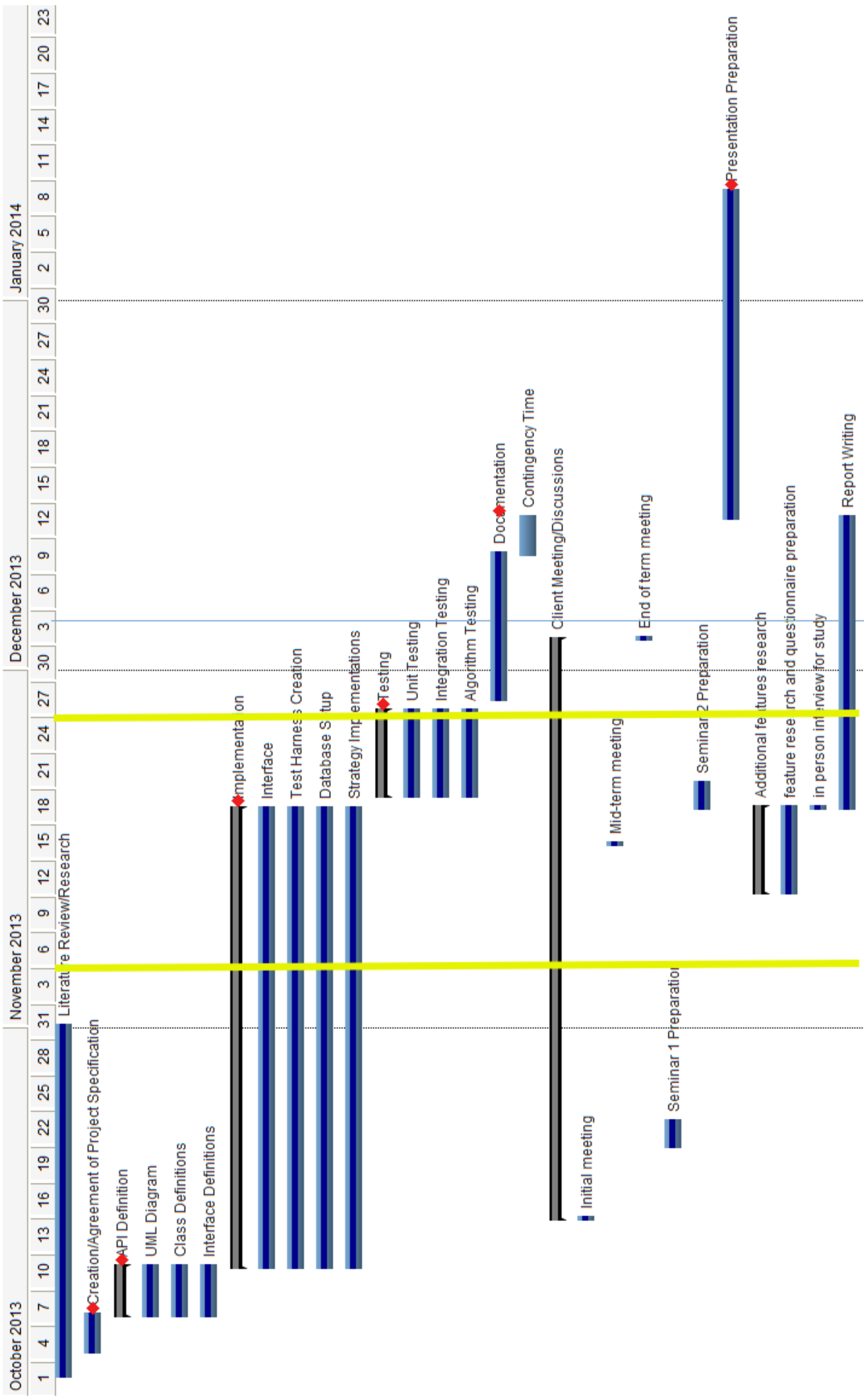


Figure 14: The final Gantt chart.

4.2 Project Strategy

This section describes the development and general working practises of the group, how the team dealt with the clients, and certain security practises that were followed in order to reduce the risk of damage to the project.

4.2.1 Group Meetings

One of the most important things which aided in the seamless completion of the project was good team communication and dedication to the project. In the beginning of the project, it was agreed that the team would meet on a specific day of the week in order to work together.

The standard workday was Wednesday where all four team members got together in the Zepler undergraduate labs and worked on the project as a group. As might be expected, a workday on its own is not much. Most days of the week, the entire team were in the labs carrying out their respective assigned tasks, and therefore in reality the group would meet at least three to four times per week and work together for a number of hours in each session. The reason for picking a standard day dedicated solely to the project was because it was convenient for everybody and, in the worst case scenario, everyone would attend at least one weekly session. Additionally, team members worked on the project at other times in the week as well, with each group member choosing their working hours or location to their liking. A good estimate of the time spent working together in the labs would be upwards of fifteen hours per week.

This working practice was extremely helpful, as it encouraged a lot of team communication and helped bring forward many ideas that ended up improving the project. For example, when designing the UML and then the database structure, the group went through a lot of designs and ideas before settling for the ones found currently in the module. Talking face to face with other developers in the team helped each one explain their reasoning behind their idea or design quicker and more effectively.

This strategy proved to be even more valuable during the coding phase of the project. Because some of the concepts involved in randomised control trials and the algorithms in use are not always self explanatory, developers tasked with a particular algorithm could explain it better to the others and more rapidly move forward with solving a bug or implementing a new feature. This form of immediate communication made it a lot easier for the database developers to understand and accommodate the needs of the developers implementing the randomisation algorithms and overall model logic, as well as designing and implementing the database in an algorithm agnostic way such that implementing a new algorithm would be easier and more straightforward. It also made propagating changes to parts of the code easier, as everyone was on the same page regarding the current task being undertaken.

Another benefit of this working practice is team bonding. Coding and working together was not a mundane or bothersome task, as the atmosphere within the team was quite relaxed and motivating. Furthermore, as time passed and the team got to know each other better, communication itself improved, making brainstorming easier.

These group meetings were used to complete all tasks necessary for the project. A non-exhaustive list of these would include research, implementation and presentation preparation amongst others. Team meetings were organised and communicated with each team member through a private Facebook group which was created for this project.

4.2.2 Supervisor and Client Meetings

In order to avoid miscommunication and inconsistencies in what the team thought the clients wanted and what they actually wanted, weekly meeting were organised, both with the clients and the project supervisor. One such scenario did arise, but was worked around without too much trouble, due to the team's constant communication with the clients. The term "cluster" appeared in the literature quite frequently, as well as in discussions with the clients. However, the team assumed the literature's definition of a cluster, whereas the psychologists were using the term in a slightly different manner for the context of randomised control trials. The resolution to this confusion came when the matter was brought to the client's attention, allowing them to explain precisely what they meant, or how clustering should function.

The project supervisor's suggestions and guidance were also very helpful. In addition to suggestions regarding features or other project related work, he helped a lot by putting the team in contact with both the school of psychology and the LifeGuide developer. It was then up to the team to set up meetings with the clients at their discretion.

Meeting with the LifeGuide developer helped in providing a better insight into the inner workings of LifeGuide and its importance to the university. It also increased the team's awareness about what currently works and what doesn't, where LifeGuide fails and what could be improved. Furthermore, the team was able to discuss specific technical details with the developer, such as what sort of database back-end LifeGuide is running on, as well as discuss different integration ideas for the project.

Communication between the team and the clients, the project supervisor and the LifeGuide developer was carried out through e-mail, in which all team members and clients were the recipients. This way it was possible to keep everyone up to date with what must be done and the meeting schedules. The group managed to meet with the project supervisor at least once a week. The meetings with the clients were also weekly, many of which the project supervisor would attend as well. The team met only once with the LifeGuide developer, as no further information or assistance were required from him following the meeting.

4.2.3 Source Code and File Management

The project required the tracking of many different files, not just source code files. The team had to keep up to date on administrative files, such as Gantt charts and presentation files, formal consent forms for the user study and other study related material, as well as the project report and source code documentation.

Source code management
The most important files were by far the source code files, as they were what would be delivered to the client. The team used the well known distributed version control system `git` and `BitBucket` as a hosting server. `BitBucket` offers both private and public source code repositories with the option to allow collaboration of different developers on the same code base for free, and this is why it was chosen over the most well known service, `GitHub`. This allows the client to have the flexibility of either keeping the final system's source code private, or share it with the general public as an open source project. Currently we have set up the project to be public under the BSD licence as advised by our supervisor and the repository can be found at `https://bitbucket.org/abotev/gdp-lifeguide`.

Documentation
A very important subject area closely related to source code management is the documentation of the code base. It is very important that the code has corresponding documentation, and as an extra requirement that the documentation matches the code at each committed snapshot.

That was as easy issue to address. Documentation was auto-generated by writing in-line comments with a specific structure, to let the documentation generator know that it should include that information in the generated files. These comments were implicitly under version control because the file they were embedded in was under version control and thus were always up to date. The documentation generator used for the project is `javadoc`, a well established tool that comes bundled with the Java SDK.

File management
Other than source code files, it was necessary to keep track of lot of other files. These files include Gantt chart files, formal documents relating to the questionnaire and the follow up study, presentation slides, images and diagrams used for the report and source code documentation files just to name a few.

It was decided to add these files to the project's version control tree with `git` as well, since it was easier to keep track only of one, slightly larger, set of documents than two separate ones. Furthermore the overhead of keeping a larger collection of (mostly) heterogeneous documents was less than the hassle of making and maintaining two different repositories.

The final report was written using an on-line collaboration tool called `ShareLatex`, which is an on-line LaTeXeditor offering version management, compiling and exporting the report in PDF format as well as allowing multiple authors to work on the same document.

Redundancy
Another issue which had to be taken into account was redundancy and risk of file loss. It was very unlikely that BitBucket would suffer a non-recoverable file loss, leaving the team much further behind on the project, but the team put safety measures in place nonetheless. Every team member had a local copy of the project tree and commit history on their machine, as well as a copy in the ECS file store. Additionally, Dropbox was used to keep an up to date archive of the project files. This increased confidence that the risk of a permanent file loss, or one which would take too much time or effort to recover from, was reasonably minimised.

4.2.4 Discussion

It is fair to say that the group's strategy overall was successful, given that the project met all of its deadlines, with enough additional time to implement an extension.

There are always two sides to every story though, and it is necessary to touch on some of the aspects of this strategy that were negative. Firstly, the initial time plan overestimated most of the tasks that had to be undertaken in order for the project to complete. This might have deprived the team of the chance to implement any extra functionality that the client might have desired, although a desire for more extra features has not been expressed at the time of writing.

Another non-critical aspect of organisation that could have been improved is the use of version control. It may have been beneficial to maintain two different document repositories, one for the project's code and one for working documents or other needed files. The purpose of this is twofold. Firstly it would cleanly separate the two document types according to their purpose, which is more coherent from an organisational point of view. Secondly, it would make submission of the full git history to the client a lot cleaner, since they would only receive their code and documentation. The clients should not, and need not, interact or possess the group's internal documents since they are of little use to them, in addition to avoiding confusion and the clients having to look at a set of unknown documents.

The benefits of the group's working strategy were more than enough to outweigh the few negative practises. Team spirit and communication was always present within the group and increased as time went by. The group communicated appropriately with the clients and project supervisor, and listened and responded to their needs with the appropriate level of care and dedication. The group demonstrated their commitment to the project with weekly meetings, as well as going the extra mile by meeting much more frequently. Both the project supervisor and the clients knew the direction development was heading towards and were regularly informed of the project's progress as well.

Another very critical issue to which the group dedicated a lot of attention was the testing of the module's code, where testing, refers to a verification that the code, under deterministic conditions, would always give the same results. This was very important because this module will be used in a live system that deals with people and patients in the real world. An error in the module could potentially damage someone or affect them in a negative way. The group did not want to be held liable or make the department accountable (as well as have the moral burden), therefore it was ensured that the code passes all tests before moving on with the implementation, as well as making new tests as the code base was expanded.

Another aspect of the development routine which worked well was the use of a build framework to automate the build, verification and final deployment processes. Specifically, the `Maven` build framework was used in order to achieve this. Maven allows easy addition of dependencies, while keeping them up to date at the same time and provides an easy way to standardise a workspace for a development team so that each developer's workspace is identical to the rest. Furthermore, when submitting deliverable files, the LifeGuide developers will not have to sift through numerous build files, but will instead use a standard procedure that is well documented in order to build, test and deploy the randomisation module.

In retrospect, the things which could be done differently if the project were to be restarted would be to separate code and administration-related files in two different repositories, and only commit to the master branch code that is passing all of the test cases it is meant to pass. Furthermore, a more realistic estimate of the time requirements of the tasks in the project might also be helpful.

4.3 Division of Labour

4.3.1 Code Tasks

Division of labour is one of the most important aspects of a project, as a poorly thought out task assignment could lead to a project's failure for various reasons. This section will describe the group's process of dividing up the project, detailing which parts were individual contributions, as well as outlining those parts which were required to be developed by all of the group members.

As this project focused on delivering a product to a client, which was substantially different to other projects the team had been faced with, it was important to acknowledge that fact while dividing up the work. Further, the time span of this project was relatively short, yet there was much to achieve, thus the group had to put a great emphasis on rapid progress of core functionality, while maintaining a concise integrability between the respective code tasks, in order to avoid late-project design changes which would require major code changes. Therefore, the team reviewed a multitude of development techniques and division of labour techniques; traditional pair programming was one of the initial plans, where two people together focus on one part of the project, collaborating in-person. This approach could have worked well if the project was undertaken in a company environment, where each group member had the same working hours, but this approach would have resulted in tying up two people with one task, and restricting individual contributions on those occasions where the pair was not in the same location. Another downside would be that uninformed initial design decisions would be discovered at a later stage, as progress on each part of the project would not be equal, leading to major revisions in the design, and resulting in wasted time. As an alternative, it was decided that each individual would be given a relatively modular task to implement, in order to achieve maximum productivity for each part. As each task was relatively modular, the requirement of relying on other parts of the system was reduced, allowing early unit testing. It was also decided that work would often be carried out in the same environment in order to work out any API changes between modules which would make integration at later stage difficult. Working together in the same environment also allowed the group to collaboratively implement any common code needed for the cohesion of each individual part.

After the initial structure of the system had been designed, the strategy for allocating areas for each of us to work on was on the basis of group members' past experience and preferences for tasks. As a result, Dionisio was assigned the task of implementing the database, its schema, and the database interface code, as he had extensive previous experience with databases. As trial information is submitted to the module using raw text files, it was necessary to implement a parser specific to the specification format. Liam worked on this, and was also responsible for some of the testing functionality such as the participant generator, and the XML parser which is able to convert the LifeGuide user data format to the module's internal for-

Member	Report Sections
Alex	2.1, 2.2.1-2.2.3, 2.4, 3.3, 5
Dio	3.2.3, 4.1, 4.2, 6
Kim	2.2.4, 2.2.5, 2.3, 3.1, 4.3
Liam	Abstract, 1, 3.2.1, 3.2.2, Final draft editing

Table 2: Table associating report sections with authors.

mat. Aleksandar was interested in implementing the simple randomisation strategy and the blocked randomisation strategy, while Kim was assigned to implement the minimisation strategy of his preference. The task of writing unit test was assigned to everyone, where each group member would implement tests for their respective module. After the core functionality had been completed, and integration of each of the module areas was underway, Liam developed an offline application which would enable researchers to load in trial definitions and allocate participants into a trial outside of the LifeGuide system.

One of the important parts of a client-provided project is meeting with the client (and, in this case, the group's supervisor too) in order to gather an accurate set of requirements. For these meetings, Alexandar acted as the chairperson, keeping the agenda on the key topics of discussion, whilst Kim took minutes for each meeting in order to minimise any confusion regarding the meetings, and to keep a consistent record on what had been decided.

For the user evaluation testing it was decided that only two people were needed for it to run, and as such Liam and Alexandar ran the user study.

4.3.2 Report Sections

In writing the report, each group member was tasked with writing those sections which they themselves had implemented, whilst additional sections which everyone had participated in would be divided evenly, depending on how long each section was estimated to turn out in terms of the word count. As Liam was the only group member with English as a native language, he was assigned with editing the final draft of the report. Each section is attributed to its author in table 2.

5 User Acceptance Testing

5.1 Methodology

In week 6, the group performed a User Acceptance Test (UAT) within the School of Psychology at University of Southampton. It was designed to be performed in two stages - a questionnaire and a follow-up user study. The main goal of the testing was to gather functional requirements from potential users of the system, including any desired core functionality or extensions. The goal of the second phase of the study was to allow the users to interact with the system, and to evaluate their ability do so. The UAT was scheduled following the finalisation of the core functionality of the module, in order to allow further development to align directly with user requirements. Results from stage one were collected in week 6, and stage two of the testing was performed in week 7. The questionnaires were distributed among the researchers at the School of Psychology by collaborating with Dr Judith Joseph and Dr Leanne Morrison, who acted as the team's main representatives on the client side of the project. The group also arranged the location and time of the second stage of the study in collaboration with them.

5.1.1 Questionnaire

The main goal of the first stage of the UAT was to find any functional or non-functional requirements which the team hadn't yet identified, and to verify that any expectations of what the module should do aligned with that of the potential users. The first part of the questionnaire (questions one to three) asked for previous experiences of people performing Randomised Control Trials - what allocation strategies have they used in the past, what statistics they gathered, and what participant sample sizes they have previously encountered. The second part of the questionnaire was intended to assess how familiar users are with the LifeGuide system and to highlight what the most challenging aspects of its use are at present. The full text of the questionnaire can be found in appendix A.

5.1.2 User Study

The second stage of the UAT was designed with the aim of evaluating the currently developed system, following any modifications produced as a result of the first stage. Each participant in this phase was given a guidance sheet, which included examples and instructions on how to set up a trial specification file for the created module. They were then tasked with using this help sheet to attempt to write a specification file for three randomised control trials that were designed and written in an explanatory text format. The guidance sheet and the actual tasks can be seen in appendices B and C. After completion of the tasks, each of the participants was provided with a questionnaire to fill out, containing questions on what they think about the current method of setting up trials, how easy they find it, and any recommendations which they could provide which could make for a better user experience - see appendix D. Any questions or comments the study participants made during the study were also recorded, as these could provide a good indication of areas of confusion or potential improvement for the specification file format.

5.2 Results

5.2.1 Questionnaire Results

The results of the questionnaire showed that all of the participating researchers have so far only used Simple and Blocked Randomisation, where the latter was found to have been used by less than 50%. This showed that in the current LifeGuide system it is most likely hard to implement any of the more complicated techniques. The fact that none of the researchers had ever have used any different technique than the ones listed was an indication that the module should not implement any other algorithms than what were currently developed, which was already suspected within the team. From the questionnaire it was inferred that most of the studies performed by the researchers at the School of Psychology have participants in the range of up to a 1000 participants, and very rarely more. This result gave us a good idea of what kind of participant sample sizes the system should expect, and was important in deciding what to be the highest number of participants to test the system - 100,000. The final part of the test showed that a significant portion of the participants find the current LifeGuide system quite hard if they need to implement some of the more complicated allocation strategies which was consolidated by the answers received on the first question. The main focus of development after receiving the results was to make the creation of a trial within the system as easy and intuitive as possible.

5.2.2 User Study Results

Results from the user study were very useful in improving the trial specification syntax. This led to several changes in the initial proposed format. The first minor change was to exchange the keyword 'Strata' with 'Stratify' which was proposed by one of the participants. Another important remark was that the syntax should not use the keyword 'weight' for both weighting between treatments (i.e. the allocation ratio), and prognostic factors when using minimisation, which was changed to 'priority'. Some of the recommendations suggested exchanging the word 'value' with 'group', and to allow the input of commas to separate input values, as this is the method currently employed in LifeGuide's logic code, and thus would make the transition easier. However, in the final specification syntax, commas and colons are optional (although may improve clarity), so a specification file with or without commas would work seamlessly. It was indicated during the study that a more in-depth guide should be provided for the end users, with more complex examples of specification files given in order to fully understand how to set up harder problems without any doubts. This led the team to expand on the help sheet, which will be provided alongside the code to be added to LifeGuide's Wiki system, which already contains guidance on how to use the system. All of the participants in the study found the system much easier than the current LifeGuide specification set up, which was a strong confirmation that the goal of increasing ease of setup had been reached.

6 Conclusion

Requirements Fulfilment and Extensions

The requirements specified in the project brief were few, but not as specific as one would expect. In particular it was necessary to implement the following (as copied from the project brief):

1. Definition and implementation of an API for the online randomisation, minimisation and stratification of custom data, for the purposes of effectively allocating end users to branches of an online intervention. The API should allow the LifeGuide server to provide an intervention participant, and receive an allocation in return.
2. Definition and implementation of a database wrapper which allows data manipulation within a MySQL database for the purposes of intervention creation and end user allocation.
3. Implementation of a primitive interface to allow interventions to be created with selected allocation strategies, based on specified participant attributes.
4. Delivery of the module in a form which allows it to be easily integrated into the existing LifeGuide system.
5. Use of rigorous testing strategies and verification methods to ensure that the system performs correctly.
6. Documentation of the software module to allow future integration with the core LifeGuide system.

These requirements seem quite straightforward at first, but if examined more closely, some of them seem to need further discussion. We shall consider each one of these in turn.

Requirement 1 was fairly open ended in terms of how the task could be accomplished, but specific in terms of functionality. This requirement was completed as described in Section 3.

Requirement 2 was again fairly open ended in its specific requirements. The team assumed that since the database was only going to be working with the delivered module, it was not necessary to account for external access to it. The final solution to this is described in Section 3.

Requirement 3 was difficult to interpret due to its subjective nature. What is a primitive interface? How complicated should it be? All these questions were answered by communicating with the client, and the solution is described in Section 3.2.2.

Requirement 4 was relatively easy to interpret. The application was packaged into a `JAR` file using a standard build process, described in Section 4.2.4.

Requirement 5 was satisfied by the use of an automated test suite which tested the project's code, with all tests reporting success. The group's approach was discussed in section 3.3.

Requirement 6 was easily satisfiable and was the most straightforward requirement of all. The project's approach was described in 4.2.3.

The team met all of the above requirements during the project, and completed them with a reasonable level professionalism, following a well documented and well organised procedure, ensuring that our clients should be satisfied with the group's performance.

Improvements in the project strategy and working habits could still be performed, something that is discussed in Section 4.2.4. These changes are not critical in nature and did not impact the project's quality, as they were mostly of internal organisational importance.

Future work on the project will include integration of the developed software module module into the LifeGuide system, most likely to be carried out by the developers of LifeGuide. Another aspect of future development that might take place would be the implementation of more allocation strategies or the expansion of the intervention specification file to accommodate any future needs that the group did not become aware of, or were not requested at the time of writing.

References

[1] Kenneth F Schulz and David A Grimes. Generation of allocation sequences in randomised trials: chance, not choice. *The Lancet*, 359(9305):515 – 519, 2002.

[2] Jimmy Efird. Blocked randomization with randomly selected block sizes. *International Journal of Environmental Research and Public Health*, 8(1):15–20, 2010.

[3] Donald R Taves. Minimization: a new method of assigning patients to treatment and control groups. *Clinical pharmacology and therapeutics*, 15(5):443, 1974.

[4] Stuart J Pocock and Richard Simon. Sequential treatment assignment with balancing for prognostic factors in the controlled clinical trial. *Biometrics*, pages 103–115, 1975.

[5] Neil W Scott, Gladys C McPherson, Craig R Ramsay, and Marion K Campbell. The method of minimization for allocation to clinical trials: a review. *Controlled Clinical Trials*, 23(6):662 – 674, 2002.

[6] Bradley Efron. Forcing a sequential experiment to be balanced. *Biometrika*, 58(3):403–417, 1971.

[7] Lee-Jen Wei. A class of designs for sequential clinical trials. *Journal of the American Statistical Association*, 72(358):pp. 382–386, 1977.

A UAT Questionnaire

LifeGuide - Randomised Control Trials
Questionnaire ERGO/FoPSE/8175

1. What methods of allocation have you previously used during randomised control trials?

Simple randomisation ('coin flip' to decide group)?	Yes	No
Block randomisation?	Yes	No
Minimisation	Yes	No

Other:

___

___

___

2. What statistics about the allocation process have you previously needed to access while running a trial?

Total allocations	Yes	No
Allocations per group	Yes	No
Sign-ups over time	Yes	No

Other:

___

___

___

3. How often do you run trials with the following number of participants?

<100	Frequently	Sometimes	Rarely	Never
<500	Frequently	Sometimes	Rarely	Never
<1000	Frequently	Sometimes	Rarely	Never
>1000	Frequently	Sometimes	Rarely	Never

4. Have you used the LifeGuide system previously to allocate participants into groups?

Yes No

5. If your answer to (4) is yes, would you say you are confident in setting up the allocation process with LifeGuide?

Yes No N/A

6. If your answer to (5) is no, what do you find most challenging about the existing LifeGuide method for setting up trial allocations?

____

____

____

7. Would you be willing to participate in a short (less than 30 minute) experiment, in which you would be asked to write some basic trial descriptions? If so, please leave your email address below.

____

B UAT Guidance Sheet

Study Help Sheet

To set up a trial in LifeGuide, you'll need to submit a file which specifies how you want the trial to run. This trial must be in a certain format, as described in the next section. It contains a number of parts, some of which are optional. Although the order of these parts can be changed, it's recommended that you use the following order to avoid problems:

- Allocation method name
- Allocation method parameters (optional)
- Stratification (optional)
- Treatment arms
- Treatment arm weights/limits (optional)
- Default treatment group (optional)
- Attribute values
- Attribute weights (optional)

Each of these is described in more detail below.

Allocation method name

Currently the system supports three main methods of allocation:

- SimpleRandomisation
- BlockedRandomisation
- Minimisation

The choice of allocation method must be specified in the following way:

```
Method: <name>
```

For example:

```
Method: BlockedRandomisation
```

Allocation parameters (optional)

Some allocation methods need additional parameters (for example, blocked randomisation requires a block size). If the parameters used by a method aren't specified, some default values are used instead. These are specified in the following way:

```
<Parameter Name>: <Parameter Value>
```

For example:

```
Block size: 30
```

Stratification (optional)

If the trial needs to be stratified on any of the participant attributes, this needs to be specified in the following way:

```
Strata: <attribute 1> <attribute 2> …
```

For example:

```
Strata: gender age
```

Treatment arms

To set up the various groups the participant could be allocated to, you must specify the following:

```
Arms: <treatment1> <treatment2> ...
```

For example:

```
Arms: pill1 pill2 placebo
```

Treatment arm weights/limits (optional)

You may want participants to be allocated to treatment groups with a specific ratio. To do this, you must set weights on each arm (if you don't specify a treatment arm's weight, it defaults to 1). This is specified in the following way:

```
Weight <treatment name> <weight>
```

For example, to ensure three times as many participants are assigned to a treatment:

```
Weight pill1 3
```

You may also wish to limit the participants in a group. To do this, you use the following format:

```
Limit <treatment> <limit>
```

For example, to put a cap of 200 participants for a given treatment arm:

```
Limit placebo 200
```

Default treatment group (optional)

If you want a default treatment group, into which participants are allocated once all groups have reached a limit, you specify:

```
Default <treatment name>
```

Attribute values

The system needs to know how to categorise a participant's answers to the trial's data gathering questions. If the user has to select from a number of options, the group should be specified in the file as:

```
Group: <attribute name> <number of options>
```

For example, a yes/no question about whether a user smokes may be defined as:

```
Group: smokes 2
```

If a user will be entering a raw value as a response, you will need to specify how to group these responses. To do this, you specify:

```
Value: <value name>
<list of ranges>
```

Each range must be either:

```
<[value]
>[value]
[value] to [value]
```

For example, if you wanted to use a participant's age, you might specify:

```
Value: age
<20
20 to 40
40 to 60
>60
```

Attribute weights (optional)

If you're using minimisation for allocation, you may want the process to give priority to balancing some attributes over others. To do this, you assign a greater weight to the attributes which should influence the balancing process more, using:

```
Weight <attribute name> <weight>
```

For example:

```
Weight age 2
```

A Complete Example

This is an example of what a full trial specification might look like for a basic trial:

```
Method: SimpleRandomisation

Strata: bmi gender

Arms: exercise diet control
Limit control 200

Group: gender 2

Value: bmi
<27.5
>27.5
```

C UAT Task Sheet

Task Sheet

1. This trial will evaluate the effectiveness of the presented medication. Participants would be split into three groups – one following a diet, one doing specified exercises and one using prescribed medication. Since the medication are limited a maximum of 1000 participants using drugs would be allowed. Also we want to have twice as many participants assigned to the diet treatment compared with the other two. One of the major prognostic factors which might affect the effectiveness of the medication is the age of the participants, so they should be stratified in 4 groups: 18-25, 26-35, 35-50 and 50+. Furthermore for the allocation process the method of blocked randomisation should be used with block size of 123.

2. This trial will evaluate two newly presented treatments of asthma – a revolutionary pill and an exposure to light amounts of radiation. They are going to be tested against the standard usage of inhalers. Because of limited access to the linear accelerator at the Southampton General Hospital no more than 250 people could be put under this treatment group. The pills provided by the pharmaceutics company would be enough for no more than 500 treatments. If those two treatments are full all incoming patients should be assigned to the inhalers group. It is desirable to have twice as many participants given the pills than are treated with radiation, and twice as many as that treated with standard inhalers. The two main prognostic factor which might affect the results of the study are the gender and whether their parents have asthma too, so the allocation method should stratify them based on this factors. The methods used for allocation should be simple randomisation (simple coin tossing).

3. This trial will assess the effects of different nutrients used predominantly in the training community – OKLAS and VBO, on personal BMI compared with standard diet. The OKLAS nutrients are limited to up to 200 participants. It is desirable to have two times less people assigned to OKLAS group compared with any of the other two groups. The method that should be used for allocation is the newly implemented minimisation technique. The prognostic factors on which it should be applied are gender and age, where the age groups are 18-30, 30-50 and 50+.

D UAT Questionnaire 2

LifeGuide - Randomised Control Trials
Questionnaire, ERGO/FoPSE/8175

1. After completing the tasks, would you feel confident in setting up a new randomised control trial?

Yes No

2. What did you find most challenging about setting up the trials in the manner specified by the help sheet?

____

____

____

3. If you could make one change to the specification format to make it easier to use, what would it be?

____

____

____

4. If you have used LifeGuide previously to set up participant allocation, which method would you prefer to use?

LifeGuide The experiment's method

Why?

____

____

____

5. Did you find the help sheet covered everything that you required to complete the tasks?

Yes No

6. Is there anything else you feel that the help sheet could have included to make the tasks easier to complete?

____

____

____

7. Do you have any further comments on the experiment (e.g. about its difficulty, suggestions for improvements etc.)?

____

____

____

Thank you very much for participating in the study!