Reengineering the TCL Job Compiler

SEM49060
Final Report

Ben Tagger
Department of Computer Science
University of Wales, Aberystwyth
Ceredigion, SY23 3DB
7th May 2003

Prifysgol Cymru
Aberystwyth
The University of Wales

Department of Computer Science

Module code: SEM49060 .....

Declaration of Originality

This submission is my own work, except where clearly indicated.

I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.

I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
I understand and agree to abide by the University's regulations governing these issues.

Signature: .....

Name (please print): BEN TAGGER .....

Date: 06/05/2003 .....

MGR 21/10/98

Acknowledgements

I would like to thank Dr. Ross King for his supervision and guidance throughout the project. I would also like to thank Ken Whelan for his technical assistence with respect to the current TCL Job Compileras well as other areas of the project. I would also like to thank my friends and family who have continued to offer their support even when they hadn't the faintest idea what I was talking about.

Abstract

With the completion of the mapping of the human genome, the incentive to provide methods for understanding the function of genomes has never been greater. It was predicted that the mapping of the human genome would herald a new era for the life sciences. Unfortunately, little (by comparison) has come to fruition due to lack of understanding of the genome. The aim of functional genomics is to establish methods of deriving gene functionality, given the information from structural genomics.

The robot scientist aims to automate part of a laboratory function in the establishment of metabolic pathways. The purpose of the robot scientist is to provide a learning system that can discriminate between competing experiments, select the 'best' ones, perform the experiments, analyse the results and then repeat the whole process.

The role of the TCL Job Compiler is to convert the experiments (those that have been chosen within the robot scientist) into machine operations that can be used by the Biomek Workstation (the robot that physically performs the experiments). This project aims to completely reengineer the current TCL Job Compiler, providing greater readability, portability, verstality and maintainability.

Chapter 1

1 Introduction

1.1 An Introduction to Functional Genomics

The area of genomics can be divided into 2 distinct categories; structural genomics and functional genomics [13]. Structural genomics is based largely on preliminary genome analysis and the creation of genetic maps for an organism. The aim of structural genomics is to establish a complete map of an organism's DNA [13]. The aim of functional genomics is to establish methods of deriving gene functionality, given the information from structural genomics.

A cell can be viewed as a biochemical machine. It consumes simple molecules in order to manufacture more complex ones. It achieves this by chaining together biochemical reactions into long sequences, which are known as metabolic pathways [24]. Functional genomics functions by analysing these pathways and determining the components.

1.1.1 The importance of functional genomics

Genomic data has various uses, including the following:

- Discovering new vaccine or drug candidates.
- The identification of the causes of diseases in genetically similar systems.
- Improving the understanding of genome/chromosome organisation, structure and evolution.
- The analysis of metabolite production (methane, etc.).
- Establishing of common metabolic pathways.

Functional genomics can help identify the genes responsible for various human diseases, which result from afflictions of a single gene. For example, a single nucleotide change can result in sickle cell disease, cystic fibrosis or breast cancer. Such single nucleotide changes have been linked to other hereditary variations; differences in height, brain development, facial structure, pigmentation, etc.

1.1.2 Methods in functional genomics

A considerable challenge exists in developing methods that will aid in the identification and definition of the role of genes that cannot be identified using other bioinformatic analyses. Data mining and other bioinformatic methods can be employed to predict the possible function of many genes, especially those similar to other genes with known functions. The Pairwise alignment techniques are such a technique that uses database search algorithms for searching for relationships based on sequence properties [2].

The following section describes three relatively common methods of gene identification.

1.1.2.1 Expression Profiling

An expression profile can be defined as the characteristic range of genes expressed at different stages of a cell's development and functioning [2]. Expression profiling involves defining the genetic sequence and then proving the functionality in a regulatory network. This method consists of micro array analyses. Thousands of genes are placed as arrays on

slides and observed under varying physiological conditions. The alterations in gene expression due to the physiological changes can be observed and studied with a view to establishing gene function.

1.1.2.2 Proteomics

Proteomics is a process used to understand the function of proteins that cannot be identified through similarity to databases. A protein is a chain of one or more amino acids, linked in a specific order [2]. The order of this chain is specified by the base sequence of nucleotides in the genetic code for the protein [2]. Proteomics is the study of protein expression of biological systems, measuring values such as the abundance, stability and fluctuations of various proteins and from this, establish their probable function.

1.1.2.3 Single Gene Deletion

Single gene deletion techniques are used to identify the function of an unknown gene. Mutations of the microorganism are grown without the gene that has been selected for functional identification. The mutant strains are grown with known differing growth solutions and the growth of these can be compared to the growth of a wild strain of the organism. Using this technique and by feeding the organism differing growth media with varying chemical compositions and then measuring the resultant growth, you can deduce which enzyme (and thus, gene) is responsible for a particular pathway.

Auxotrophic growth experiments are a technique for determining the metabolic pathways exhibited by a particular organism. An *auxotrophic mutant* is a strain of organism that has been grown with a particular gene missing or mutated [24]. The organism is grown so that the particular gene is defective and so cannot play its normal part in the pathway. Consequently, the organism cannot synthesise a particular component of the metabolic pathway and so will not grow. However, if the product that is created by the missing gene is manually added, the pathway can be restored [24].

"The deletion of any genes essential in the synthesis of these molecules will prevent growth and replication. As a result, auxotrophic experiments can be used to infer their biochemical function." [24]

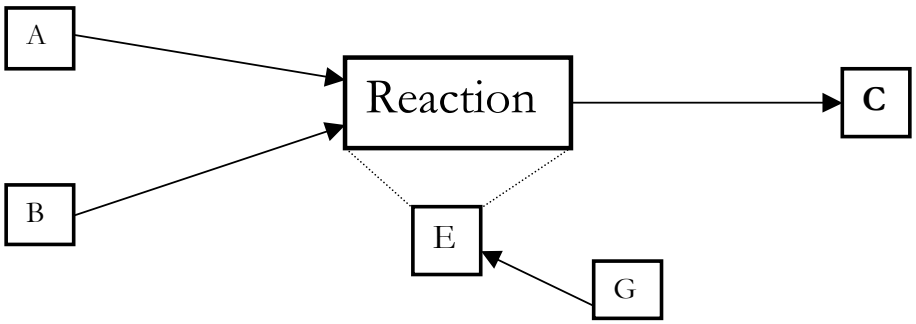


Figure 1 - *An illustration of the single-gene deletion process*

As an illustration of the single-gene deletion process, consider the diagram above. A, B and C are all metabolites. E_1 is the enzyme that catalyses the reaction that uses the reactants A and B and produces the product C, and is controlled by the gene G. If a mutant organism is grown with the gene G missing, then the reaction will be unable to occur, as the enzyme needed will no longer be present. Hence, the metabolites A and B will no longer be able to be used to produce C and other reactions that use the metabolite C will be unable to function as well. If this mutant organism is given various growth media, one of which has the metabolite C added, then the pathways depending on C will be able to function again and the organism will continue to grow. From this, it can be concluded that the gene G is responsible for the production of E_1.

The technique of single-gene deletion is widely used and has proved extremely successful in many applications, especially single-gene disorders [10][37]. However, there is a cost attributed to the trials and so, the fewer trials that can be done, the better.

1.1.3 The use of Yeast in Functional Genomics

Functional genomics research is usually conducted through the use of model organisms such as mice and yeast. The use of model organisms provides a cost-effective way of conducting genetic research as many generations can be studied over a relatively short period of time.

1.1.3.1 Why use Yeast?

There are several reasons why yeast (Saccharomyces cervisiae) is commonly used in genomic experimentation. In 1996, the first complete DNA sequence of a eukaryotic genome (yeast) was sequenced. By 1997, the yeast genome has been completely sequenced thanks to a huge international effort involving over 600 scientists in Europe, North America and Japan [12]. With the entire yeast genome sequenced, it is now possible to estimate the number of genes that have significant human homologs. Evidence suggests that of all the protein-encoding genes in yeast, around 30% of them have a statistically significant human homolog [6]. Even with a relatively simple organism such as yeast, 60% of the genes still have no experimentally determined function [6], indicating that there is still much work to be done.

Genetic manipulation in yeast is relatively easy and cheap when compared to manipulation in mammalian systems (when possible), is neither easy, nor cheap. The gestation period for many mammalian subjects is very long, whereas many generations of yeast can be achieved in a relatively short period of time. It has been shown that there are over 70 human genes that complement yeast mutations and this is certain to be an underestimate [42]. With this in mind, information about human genes can be established by studying their yeast homologs and at a considerably more cost-effective way.

Yeast is an extremely useful model organism for eukaryotic biology and there is considerable justification for forging efforts to uncover the functionality of the remaining 60% of yeast genes whose function is still not known. It is commonly believed that many of the remaining yeast genes may represent the most efficient path to understanding diseases such as colonic cancer.

1.2 LIMS

It is possible that the existing system (described in 1.4) could form part of a LIMS (Laboratory Information Management System). The system aims to automate part (or whole) of the work carried out by scientists, which fits nicely into the ideology of a LIMS. The following subsection will provide an introduction to LIMSs and give some detail as to the implementation approaches for LIMSs.

1.2.1 What is a LIMS?

The modern laboratory exists in an environment that produces a large amount of data. With the advent of new technologies, both the quality and quantity of information is increasing exponentially. This increase of data can cause significant problems and methods are needed to manage it. One such method is a LIMS [5].

A LIMS provides a way of automating part of the laboratory system. In a traditional laboratory 75% of the total cost comes from manpower. Removing the need for some human interaction can significantly reduce overheads. The primary function of most laboratories is to provide validated information under some sort of time constraint and then based on that information, allow customers to make decisions [19]. Nowadays, traditional record keeping solutions are simply not up to the task. A LIMS can be of great importance in integrating laboratory operations with the laboratory itself. One of the most important aims of a LIMS is the integration of many different subprocesses, bringing together and consolidating the efforts of potentially many individuals and consequently speeding up the whole process.

LIMSs can save considerable amounts of time and dramatically improve the level of data access for all stakeholders of any given project. This is where a LIMS can become extremely beneficial. The sooner the user is notified of a problem, the sooner that problem can be fixed and the less the solution will cost [19][14][5]. The ideal LIMS should help provide the documentation to ensure that a laboratory and all of its operations exist in compliance.

LIMSs have been used for over 20 years and the technology has evolved considerably during this time. This section aims to provide the reader with a brief introduction to LIMS and help explain the benefits and problems with LIMS.

1.2.2 Benefits of a LIMS

A LIMS provides benefits for many of the users of a laboratory. However, a LIMS does represent an expense that must be considered. This expense will almost certainly have to be justified by a level of higher management. The following is a brief outline of several of the main benefits identified and realised from current users of LIMSs.[i] [38]

- Information can be obtained with the click of a button rather than having to dig through files.
- Years of data can be kept easily without the need for traditional archiving.
- The improvement of business efficiency.
- Improvement of data quality (all the instruments are integrated).
- Automated login, tracking and management.
- Automated customer reports (Turnaround Time, Work Load).
- Automated Integration of Hand-held LIMS devices.
- Automated Quality Control.
- Daily Quality Reports.
- Easily accessible data via the web.

1.2.3 LIMS Selection

It is said that a successful LIMS implementation is closely linked to the selection process. Selection and implementation of the LIMS is a complex process [20]. It is very important to select the right product, as this will have a major impact on the success of the LIMS project. The selection process should be thought of as an official part of the LIMS implementation process.

There are reasons for not taking a formal selection process for a LIMS. The most prevalent is that most of the systems today all have about 80% functionality in common. In considering the 'Pareto Principle', it can be shown that it is the remaining 20% of functionality that is regarded as the most important [20]. For example, the greatest benefit that a LIMS provides may be in the 20% that is not provided. When selecting a LIMS, it is important to select one that is the 'best fit' for your requirements. There are obvious problems with selecting ones with less functionality than needed, but at the same time, there is little worth in selecting a LIMS with superfluous functionality just because it is available.

LIMS projects require large amounts of time, commitment and money. Making the wrong choice of selection could result in a failed project. It can therefore be argued that a process with such bearing on the success of the overall project should have a formal selection process.

There are many steps involved in the selection process. One of the most important (and frequently missed) steps in LIMS selection is a method known as WPE (Work Process Evaluation). WPE is used primarily to define the role of LIMS within the organisation.

[i] Data taken from a survey paper, in which users had been running LabWare LIMS for at least two years.

Requirements must be clearly specified, concise and well understood. Once these requirements have been clearly defined, the selection process can enter the 'Purchase Process'. This process should include viability studies:

- What is the state of financial health of the company supplying the products? Will they be around in 2 years when you need to upgrade?
- What are the future plans for the company? Are they planning on shifting business strategies, rendering your purchase obsolete?

It is important to consider such concerns when selecting a product to ensure that the companies' resources are spent wisely.

As customisation is an expensive way of achieving a 'correct fit', it is important that a LIMS match your criteria as far as possible with as little need for customisation as possible. The final stage of LIMS selection is the Vendor Audit. It is accepted that the customer is responsible for the system once it has been installed. With this in mind, it is necessary to check that the system has first been engineered properly. It is a chance for the customer to verify that the system fully meets the organisation's criteria and to identify any potential pitfalls. It also provides a platform for the customer to suggest possible improvements or modifications and to establish a working environment with the supplier.

1.2.4 Customisation

LIMS can save vast amounts of time and dramatically improve productivity within the workplace [3]. However, inevitably no two laboratories are going to be the same. Work practices, management structure, strategy, expectations, human involvement, are all going to differ. How can a LIMS be produced so as to satisfy this gantry of differing criteria? The answer is simple. It cannot. The solution is to provide the users with a LIMS, which they themselves can customise and alter to their own ends. This removes the need for the LIMS developer to include specific functionality, rather provide the means for LIMS users to provide their own. As a result, the overall potential functionality for the system is greatly increased.

One point to note is that customisation does not simply involve passing on a shell of a working program to a customer. It may involve extensive testing and analysis of the stakeholder's requirements. User feedback can help a developer produce a new, more efficient process, automating as many activities as possible.

1.2.4.1 What can be customised?

The most important need for customisation is the user interface of a system. If a LIMS is to improve productivity, it must provide an easy-to-use, specifically designed front-end for its users. The user interface must be intuitive, flexible and robust. There must be specific screens for various parts of the system and this should reflect the specificity of the system's domain. This is no different to many other areas of system design. The average Microsoft Word™ user will only ever use 15-20% of the functionality of the application. The determination of exactly which 15-20% is to be used is the hard part and this is where the power of customisation can be applied. By allowing the user to determine which parts they are going to need, avoids the need for a proverbial 'finger in the air' guestimate.

Whole screens can be redesigned (possibly from scratch), making them more intuitive and specific for the individual applications. Unnecessary fields, selections, and choices can be removed. Important areas can be highlighted. The user can drill-down or drill-up menus to the desired level of detail. As a consequence, users will not see functions that they do not need and will not have access to data or functions that are irrelevant to their job. The largest drain on productivity manifests in the communication of a desire of a user to a system. Improving the user interface will have the greatest effect on productivity, as it is the tool that provides the most interaction with the user. The more a system can reduce the amount of typing, clicking and thinking, the greater the improvement to productivity.

Customisation can occur in terms of the type and functionality of instrumentation that is to be used. Some instruments may be passive. Others may require user input and require bi-directional communication to provide feedback. Aspects of the project such as these can be provided and integrated into the LIMS from the start.

Laboratory systems are primarily concerned with the collection and analysis of data. A database (RDBMS) seems the most sensible way to manage this plethora of data. However, it is extremely unlikely that any two unrelated laboratories will have the same structure of data. Therefore, a level of customisation is needed to give the customer the freedom to present the data in the most beneficial way for them. This also applies to areas such as data-entry systems, where a generic form is totally unsuitable.

1.2.4.2 How can you customise?

There are two main ways of providing customisation to a LIMS. The first is to include a scripting language with the LIMS. LIL (Laboratory Interface Language) from LabManager is an embedded high-level language with which the user can write their own methods and routines to automate repetitive tasks [3]. With a language, such as LIL, users can combine parts of the system, utilising available additional functions supplied with LIL. This approach is not unique to LabManager. Most LIMS developers provide some sort of scripting language in order to allow the user to customise, develop, and progress their laboratory system.

Another means of customizing a LIMS is to provide a packaged customised LIMS to the customer right from the start. This is usually advantageous for laboratories conducting research on fairly generic subjects. For example, consider the problem of integrating molecular genetics analysis capabilities into a LIMS. gtLIMS is a pre-customised LIMS that specializes in this area [11]. gtLIMS contains the basic building blocks found in a normal LIMS. However, additional features have been added in order to create gtLIMS[ii] and make it more specialised to its target domain.

[ii] A complete description of gtLIMS is not within the scope of this document. Please refer to the referenced paper.

1.2.5 LIMS Specification

One of the great problems with LIMS selection is the huge variety of vendors from which to consider services. Many of these vendors offer services and products that are not compatible with other similar products from the same domain. It is important to invest in products that are valuable and are not going to become redundant in the future.

To date, a set of standards regarding the development of a LIMS has yet to be produced. It is possible that this is one of the essential pieces of the puzzle still to be put into place. Standards allow universal acceptance of a product and also promote a facilitated development curve. The absence of standards can often put the brakes on a potentially successful technology. Often, the discussion on appropriate standards can take so long that the demand for the product has diminished when an agreement is struck. This is true for network technologies more than any.

Localised standards are emerging such as LECIS (Laboratory Equipment Control Interface Specification) [23]. This technology is concerned with providing a specification to provide a robust standard for communicating between equipment from different controllers and platforms. The problem is that developers are inherently more concerned with improving their core capabilities rather than their interfaces and engineering standard. This can often result in poorly designed device interfaces, which can be very inconvenient for implementers. LECIS aims to define the interactions between the devices and the controllers in order to achieve some level of operation. The degree of flexibility pushes LECIS into more of a specification-type category although in 1999, LECIS was balloted through ASTM and has become standard E1989-98.

Such standards are important so that equipment can interact directly with the controllers, a standard specified by the industry rather than by the individual user. Specifications such as LECIS are sorely needed in an industry that contains so much variety and incompatibility. However, more are needed in order to improve the LIMS development process.

1.2.6 The Development Process

The following section describes some of the tools that are often used in the implementation of a LIMS. This section is by no means exhaustive but endevours to cover some of the basic points of development, such as: development languages, XML, ASPs and web services.

1.2.6.1 Development Languages for LIMS

One of the most important aspects to consider when developing a LIMS is the database technology to be used. The database technologies have not significantly changed over the past few years, so it is reasonable to suggest that the ones around today will be around tomorrow. Most LIMS support relational SQL (Structured Query Language) databases such as Oracle, whilst the newer systems are expanding into the object-oriented technology. This has been supported by the object alternative to SQL, namely OQL (Object Query Language, surprisingly).

The hugely popular object-oriented programming languages Java and C++ have reinforced this object-oriented approach. While most vendors develop their LIMS on the Windows platform, those who have employed a multi-platform approach have done so largely influenced by Java's JVM (Java Virtual Machine) [29]. This allows the same source code to be used on many different platforms, by providing a platform-specific runtime environment for the code rather than the code itself being platform-specific.

Selecting the right language to implement the LIMS is a difficult process. Many aspects must be considered; functionality, readability, portability, maintainability, support, tools, etc [29]. Currently, Java is one of the most flexible and widespread programming languages available. As well as providing platform-independent software development, Java enjoys a plethora of support, including distributed computing, networking, and enterprise technology. Standard and non-standard APIs and other support tools are emerging all the time, making Java an extremely strong contender.

1.2.6.2 XML in a LIMS

XML (eXtensible Markup Language) provides a flexible way of creating a common data format to facilitate the sharing and distribution of information on-line. XML provides a means for obtaining and supplying information about things in a standard way. For example, authors can allow browsers to present information inline with the user's requirements, rather than a standard HTML page. Compared to traditional closely coupled interface methods, XML-based systems can be loosely coupled, which are more maintainable and less expensive [8].

XML fits in nicely with the idea of distributed computing with the use of peer-to-peer message passing [27][29]. In such a system, there are no slave-master relationships, rather each individual system creates, responds to and deals with its own packages. This has two main benefits. Firstly, the peer-to-peer architecture helps to keep the network protocol as generic as possible – systems can learn where neighbours are, rather than having an explicit hierarchical structure. Secondly, a peer-to-peer messaging system will survive if one machine fails. Each system will simply re-route the packages via an alternative route.

The appeal of the XML interface is its ease of development, maintainability and potential for reuse. The XML interface can aid a LIMS in situations such as job and sample submission, LIMS data archiving and the integration of two LIMS in different labs. It is this transfer to and from different laboratories in which XML provides the biggest benefits. Transferring the data in a secure and recognisable format constitutes one of the largest challenges for a current lab. There are often isolated areas of a typical laboratory, each of which must be successfully integrated to exploit fully the laboratory's total potential [17]. This problem is further exacerbated due to there being so many different vendors, offering products that lack the standards for exchanging information between them.

With XML, systems can pass tagged data to each other without having to know how the other system is organised. The system can be expanded, trimmed, or reused without any further need to provide additional or differing functionality.

For example, in a LIMS, each XML file could represent one job. Users can customise this file with their own job descriptions as well as including static data structures (lists, etc.).

Consider the development of an ELN (Electronic Laboratory Notebook). The ELN must be able to liaise with the other components of the laboratory, one component being the LIMS. Within the ELN, the user submits experiments to be analysed by the LIMS and then the results are required back at the ELN. In such a situation, XML capabilities can simply be added to both the ELN and the LIMS, providing a successful means of communication between the two.

ASPs and Web Services are closely linked to one another and are as much a part of the development process as the selection process. It may be important to select vendors that have provided means for ASP integration. Conversely, it may become a development issue as whether and where ASPs and web services are used within the LIMS.

1.2.6.3 ASP for a LIMS

Of all the 'buzzwords' in use today, ASP (Application Service Provider) is one of the largest and most promising. It was predicted that ASPs would revolutionise the face of the industry, changing how systems are implemented and managed. However in spite of this, ASP has failed to produce more than a few decent examples of its use [18]. Hugely optimistic forecasts were made about the future ASP market but with the end of the dot-com boom, it now seems that these are unattainable in light of the industries' self-mistrust. The other problem with ASP is that there is a general lack of clarity and understanding in what is actually offered. In spite of this, there are indications that ASP will have a successful future.

Over the past 7 years, science-based organisations have been looking to outsource their IT operations in order to achieve a business focus on their core competencies. This does raise the question of security. In particular, pharmaceutical companies are reluctant to employ third parties to manage their systems as regulatory and security issues are paramount.

ASP provides a way of using software applications without the need to buy expensive licenses, machines, and support. Functionality of the applications is rented out to the organisation, possibly on a pro rata basis. System maintenance, backup, and recovery are all provided for by the supplier as the source program normally resides with them rather than with the customers. All the functionality of the application is held somewhere else and the customers are only shown a front-end. A lot of money can be saved in various areas such as implementation, installation, upgrading, maintenance, security and support. In modern day businesses, up to 80% of the total cost of system will come from these types of costs, greatly outweighing the initial expense of the actual hardware and software.

One of the drawbacks with ASPs is that it requires a totally radical new way of strategising an organisation's attitude to software purchasing. There is no slow, baby-stepping into an ASP scenario. It must be done all at once with no middle ground. This requires some degree of faith, which can be lacking when there is so much at stake. One huge concern within the LIMS domain (as well as any other security-conscious organisation) is that data is held off-site by a third party. This can raise considerable security issues as well as legality concerns in some cases as information must be passed back and forth continuously. However, confidence in the security of the Internet and the communication abilities of the Internet is growing all the time and rightly so.

There are considerable advantages in considering ASPs. In the IT sector today, there exists more competent competition than ever before. The pressure to develop quality products in the shortest time possible is an extremely prevalent occurrence [18]. ASPs can significantly reduce the time taken to achieve an operational status. Installation, configuration and implementation can all be completed at a greatly increased rate, reducing the time needed to prepare the equipment for development. ASPs should be thought of favourably, especially with regards to tightly scheduled projects. There is no need to spend time choosing and purchasing hardware and software systems. With ASPs, the experience is already there. There is constant available support providing instant resolvement. The other benefit of using an ASP is the assurance that it provides. The total risk taken by the customer is reduced when compared to that of purchasing a whole new system. Many suppliers will have an insurance clause, indicating their responsibility for any loss incurred due to a failure in their system.

ASPs can work within the LIMS environment but first, a solid, reliable service must be offered, providing peace of mind to customers. Once this is achieved, ASPs could represent a considerable step forward for LIMS.

1.2.6.4 Web Services

A web service is effectively an application or application logic that can be accessed through the Internet. Companies pay rent in return for the use of services held on a third party's web-server. By using the standard Internet protocols (HTML and XML amongst others), the web application can pass and receive messages from the user. The ASP (Application Service Provider) may have many customers using the same web service and so the web applications are written with strict specifications in order to allow different users access to similar business methods. Companies can add value and functionality to their services as and when the customers require it [17].

Laboratories can have many different data management tools and needs. These may include LIMSs and ELNs, measuring and analysis machinery, as well as the more logistic areas such as human resources, time planning, and accounting. The vast majority of these tools can either benefit by using or beneficially become a web service. Web technologies such as XML and SOAP (Simple Object Access Protocol[iii]) and the current omnipresence of the Internet have made this kind out application outsourcing technically possible, but there are various hurdles to overcome before a complete success can be made through web servicing [9].

One major issue is trust. The leap of allowing an organisation's sensitive data to be held off-site is a very large one. It is very difficult to place the destiny and future of your business in anyone's hands other than your own [9]. An organisation must be confident that the system will not go down, that it will be secure and safe. In some cases, the option of web services will not be a viable one, due to the nature of the data or the level of importance of the operation. At the moment, safety-critical applications are not suitable for outsourcing given the level of importance, but there are many organisations operating in a non-safety-critical environment that would consider their operations as important. So, are web services suitable for them?

[iii] A protocol defining how XML represents data.

As always, it is necessary to weigh the advantages against the potential hazards. It is reasonable to assume that the supplier will have more specific knowledge and therefore be more proficient at handling the same service than the client. It is the shifting of responsibility that unnerves many customers.

1.2.7 A Look to the Future

Although LIMS can be shown to be advantageous to the running of a modern laboratory, there are many problems and issues that must be addressed in order to ensure a successful project. Almost all laboratories could benefit from a LIMS but for many such a system is prohibitively expensive. A LIMS's implementation requires a colossal amount of time, effort and money. The failing of a LIMS project is an all too often occurrence and results in a great waste of resources, probably including a 'forced career move' of several of the responsible parties. Therefore, it is important to get it right and first time round. Both the expense involved and the high possibility of failure can deter the more modest organisations from attempting a LIMS implementation. Even for the more lucrative companies, of which there are many in the pharmaceutical industry, a LIMS implementation cannot be without hindrances. The shear scope of vendor variety can make the important LIMS selection process complex and confusing. The degree of difference between vendors is largely the result of the lack of specifications for building a LIMS and this often leads to serious compatibility problems. For the LIMS technology to progress in a beneficial and well-engineered way, these problems must be addressed.

One step forward could come from the introduction of a standardised way of implementing a LIMS. Standards must be unambiguous, clear and concise. They provide an encapsulation of the most appropriate way of performing a process in order to avoid repetition of previous mistakes. Standards provide a clear way of performing a process. They also allow one person to carry on with another's work, as all the work should be standardised. Specifications and standards can be used to guide a project owner through a successful LIMS implementation. By introducing standards and specifications, vendors will be forced to provide compatible components. In turn, this will help drive the price of LIMSs down for the consumer, increasing competition amongst the LIMS providers. It will become easier to weigh up the differences between different vendors and give the customers the ability to 'pick and choose' rather than being limited to only one company's products. By providing a standardised process that will actually result in a successful LIMS implementation, the possibility of failure will be reduced (not eliminated). This will open the door for the more meagre companies that could not before consider a LIMS. Now that the risk of failure is lower, a LIMS can be seen as a more solid investment.

There is a difficulty arising from attempting to standardise the LIMS development process. Laboratories are largely centred on experimentation. By definition, experiments are original and novel. This makes it very hard to define standards for them. How do you define a standard for something that can be so varied? It is said that it is this variety of laboratory function that has so far hindered the standardising of processes in the laboratory. However, if the experimentation process is treated more like a process, rather than attempting to consider all possible experiments, then a more accessible ideology of a laboratory can be achieved. For example, an experiment can be thought of as having some basic components[iv]: Analysis, Planning, Implementation, Observation, Validation and

[iv] These are only the very basic ideas of an experiment. Obviously, some experiments will have differing components.

Conclusion. When considered in this way, it seems plausible to construct some kind of standardised process, at the very least fitting around the experiment process.

Typically, standards take a very long time to be produced. This length of time is rightly needed so as to get the standards absolutely correct. However, with an industry that is changing so quickly, standards can often appear too late and consequently, are useless. Generally, standards do not work well with industries and technologies that are changing at a very quick pace.

Good software engineers are inherently lazy and try at all costs to refrain from being innovative. Innovation takes time and is pointless when a solution already exists. Patterns are a tool used by developers to avoid having to re-invent certain solutions. A thorough set of patterns, for use with a LIMS, could significantly increase productivity, speed up implementation, and help reduce the amount of innovation required when developing a LIMS.

1.2.8 Conclusion

Implementing a LIMS is an extremely expensive process, one that must be improved considerably if it is to become more widely available. There exist many technologies of which to take advantage, some of which are described here in this paper. However, there are very prominent risks involved in the implementation of a LIMS. A high failure rate can deter many laboratories from attempting such a project. There are various ways to reduce this risk of failure but none of the afore-mentioned processes provide a total, ideal solution. The guidelines for successfully implementing a LIMS are useful but are by no means complete. What may work for one business may be totally unsuitable for another.

The process of experimentation is so varied that any form of automation seems optimistic. However, a LIMS can work, they have been shown to work and they have been shown to be very profitable when employed correctly. It is necessary that, before committing to any particular LIMS, the consumer sits down and reads all the facts. They should be aware of all the possible pitfalls and how to avoid them. A LIMS is a subject that, if tackled correctly, can yield astonishing results.

1.3 The Existing System

1.3.1 Overview

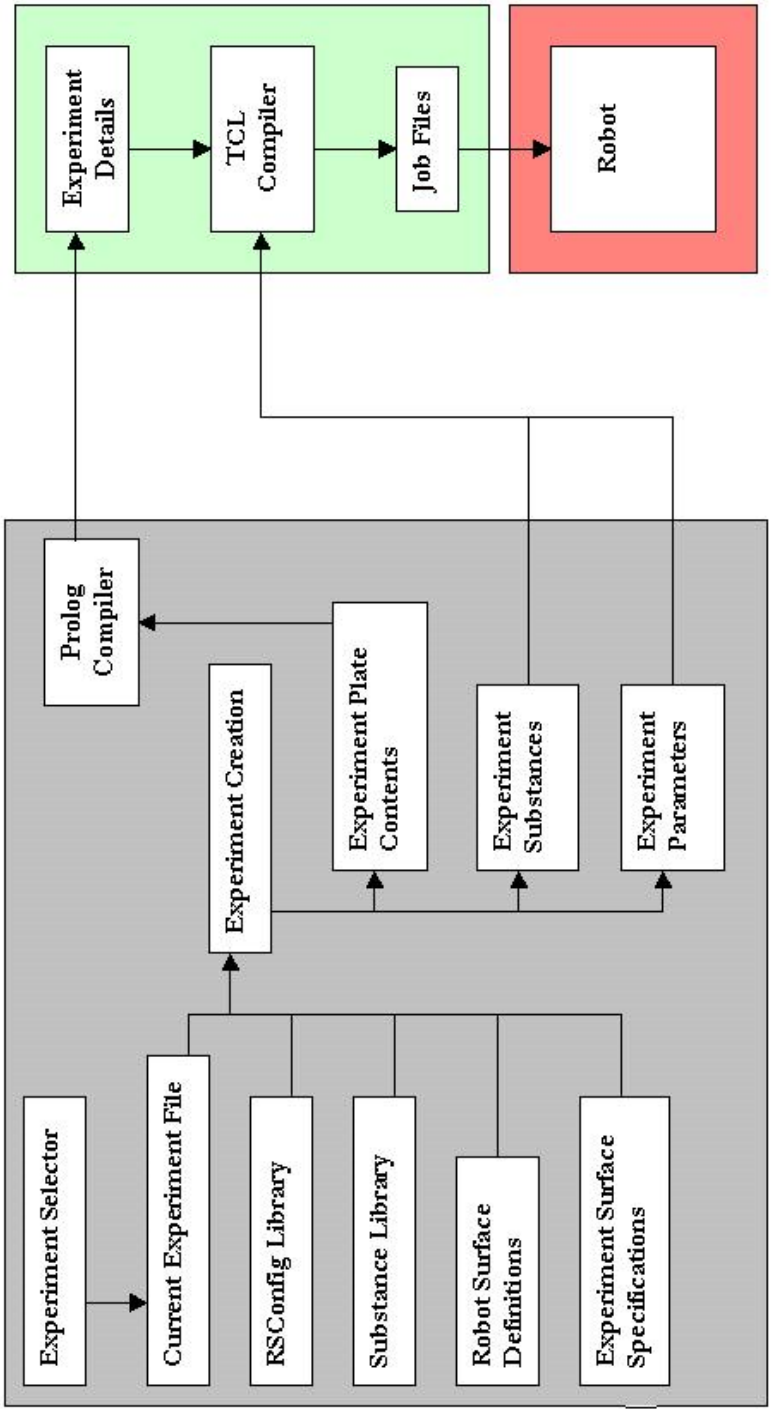


Figure 2 – *An Overview of the Existing System*

1.3.2 Structure

The complete system is comprised of three smaller systems; ASE-Progol, TCL compiler and the Biomek Workstation. The diagram above shows the limits of each of the three subsystems. The following section will discuss the ASE-Progol and Biomek Workstation subsystems and also an analysis of the TCL compiler, upon which this project is based.

1.3.3 Function

The purpose of the system is to develop a framework for automatic experimentation, involving machine learning for the generation of trials and robotics for the execution of the trials to establish accurate hypotheses [13]. This framework is to be used in functional genomics to discover the function of genes [13][2]. Currently, the system is being used to establish metabolic pathways in the amino acid pathway of yeast (Saccharomyces cervisiae).

1.3.4 ASE-Progol

ASE-Progol (Active Selection of Experiments with Progol) is an active learning system, using ILP (Inductive Logic Programming) in order to construct hypotheses. ASE-Progol selects trials to eliminate the hypotheses and then composes the experiments to be executed by the robot. Below is a diagrammatic algorithm of the processes employed during the ASE-Progol system.

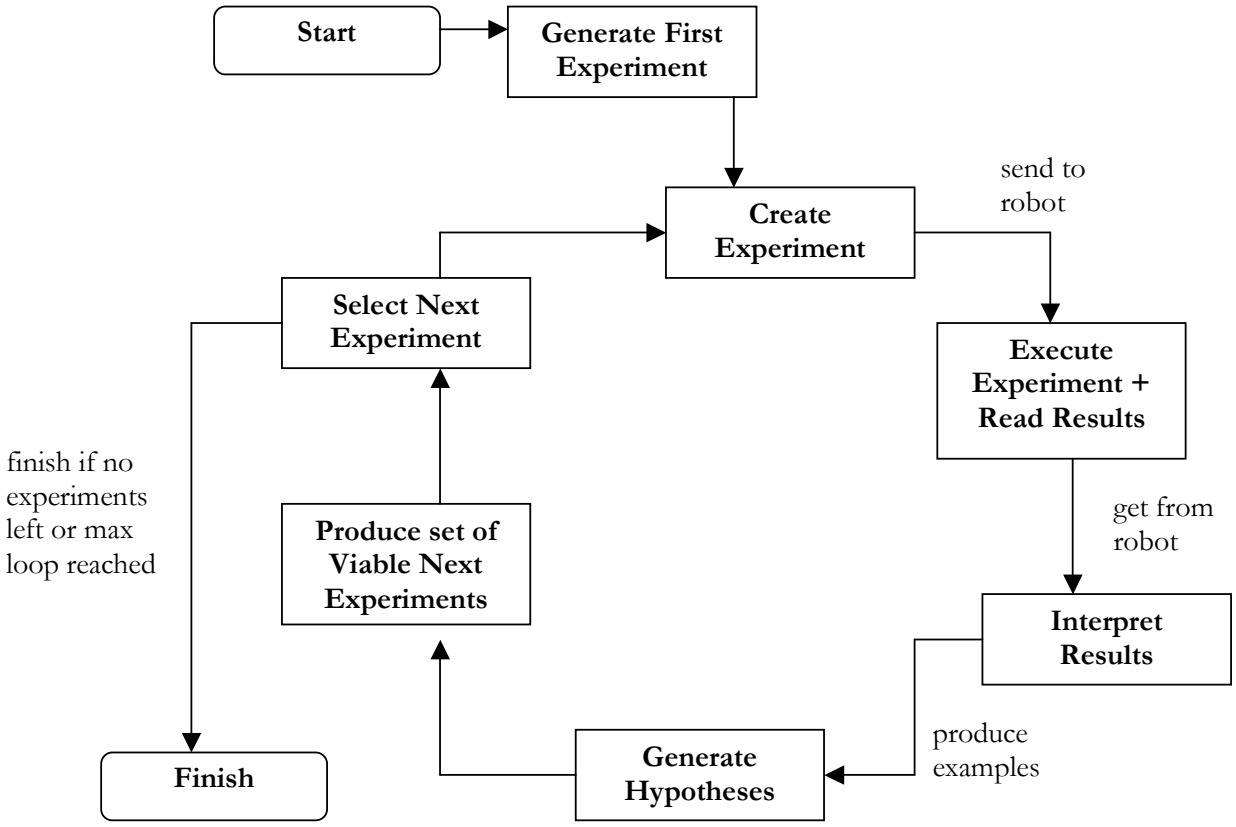


Figure 3 – *The Program Loop for ASE-Progol*

Machine learning systems that produce intelligible results are being increasingly implemented in both scientific and business industries. Such systems are typically *open loop* systems, in which there is no direct link between the machine learning system and the data collection. A *closed loop* machine learning system is one that not only selects the trials, but also is responsible for carrying out the trials in the specified domain.

ASE-Progol is an attempt to partially automate some of the aspects of scientific including; the forming of hypotheses, the creation of trials to discriminate between competing hypotheses, the execution of these trials and the analysis of the trials to establish a validated hypothesis. ASE-Progol was developed with the long-term goal to use the framework in functional genomics to discover the function of genes [35][7].

ASE-Progol has been used to discover how genes participate in the aromatic amino acid pathway of yeast. The cost of chemicals consumed during this process was shown to be five times less using ASE-Progol, when compared to a random strategy. Although ASE-Progol results in an increased cost when compared to a naïve strategy[v], both the naïve and random strategies took significantly longer to reach a final hypothesis [39].

According to [35], there are several distinct phases in ASE-Progol.

- **Hypothesis Generation.** ASE-Progol uses Progol 5.0 to generate hypotheses from observed results and background. The system introduces the examples, one at a time over each iteration and then uses "Theory Completion using Inverse Entailment" [21] for formulation of the hypotheses.

- **Trial Generation.** The first trial that is chosen is the cheapest trial. In the following iterations of the program loop, the trial generator removes any duplicate trials (or trials that have already been performed). The result is a set of no more than 20 trials, which are passed to the classifier.

- **Classifier.** The classifier determines whether the order of each trial is consistent with each hypothesis. ASE-Progol uses the Progol Interpreter inside Progol 5.0 to construct a binary matrix specifying the consistency with the hypotheses. From [35], the binary matrix is one in which entry *ij* is 1 is the outcome of the trial *i* is logically consistent with the hypothesis *j* and 0 otherwise.

- **Trial Selection.** Given the binary matrix, the trial selector computes the expected cost of experimentation for each candidate trial[vi]. The next trial to be performed by the robot will be the trial that minimises the expected cost of the experimentation.

- **Termination.** The loop will terminate if any one of the following conditions is met.

    1. all possible trials have been tried,

    2. the experimental resources have been exhausted,

    3. the number of trials exceeds a limit specified by the user.

[v] A naïve strategy will always choose the cheapest trial from a set of candidate trials.
[vi] expected cost will consist of the chemicals needed for the experiment.

1.3.5 Biomek Workstation

Information documented here in relation to the Biomek Workstation has been obtained from [4].

Bioworks methods are used to control the action of the Biomek workstation. These methods contain details of the tools to be used, the specified actions to be performed and the sequence in which to perform them. Below is a diagram to represent the rough flow of data between the operations. From the diagram, it can be seen how the data 'filters' up the system.

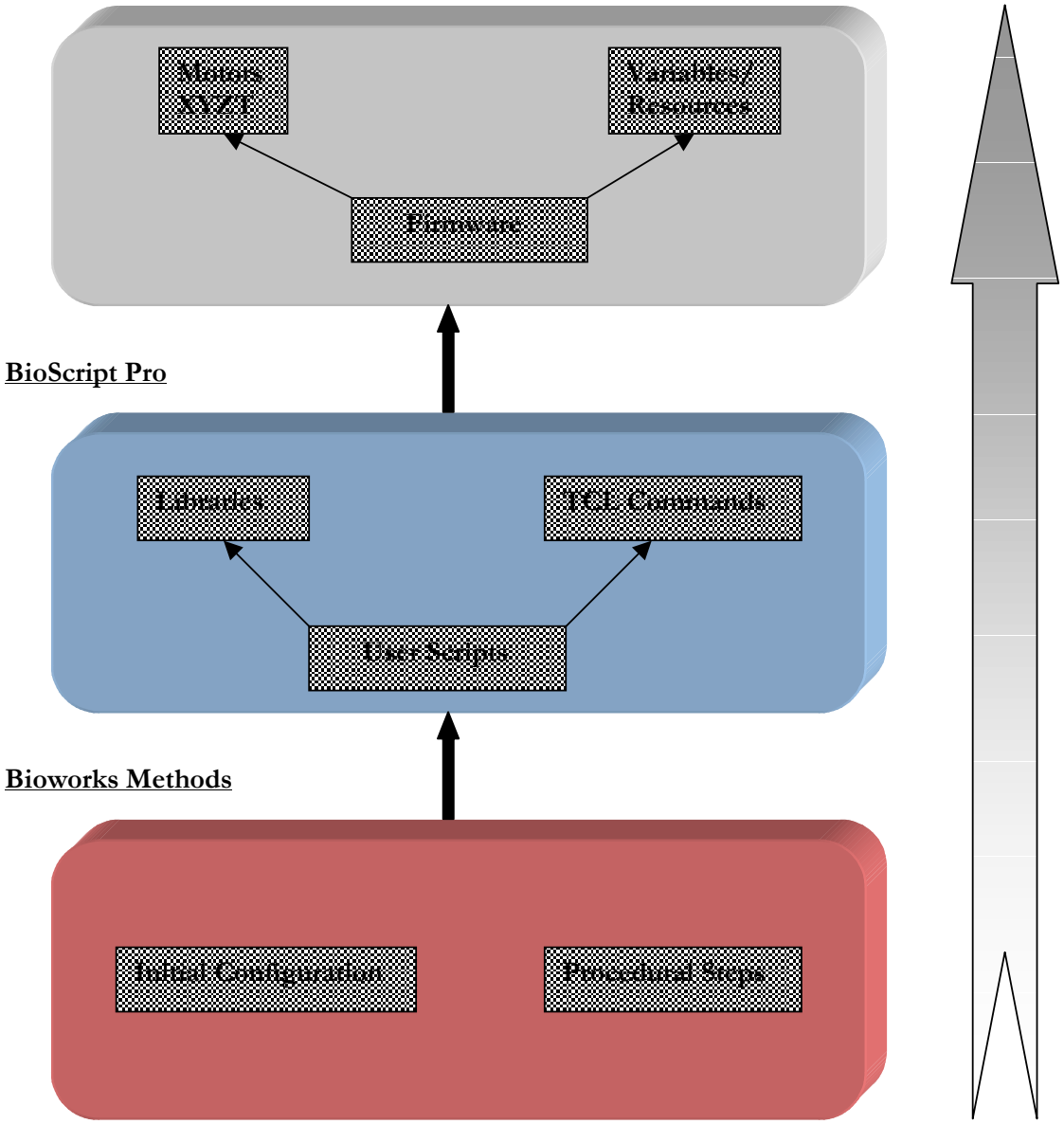


Figure 4 - *An illustration of the filtering of information through the Biomek system*

The initial configuration specifies which tools, lab accessories, and other lab ware is needed for the following experiment. This is an important and necessary step to take. The initial configuration must be set at the beginning of the experiment. Once the experiment has started, it is impossible to add new tools or devices without halting the software completely. It is up to the user to ensure that the setup of the physical machine matches that described in the initial configuration.

Once the initial configuration has been uploaded and checked, then the Biomek starts to perform the functions as set out in the methods. These could include many different types of operations including pipetting liquid from one plate to another or filling plates from another container, or gripping or shaking a substance. The Biomek can only function correctly if the exact dimensions of all the lab equipment are known. However, once this has been achieved, then the user can use higher-level commands to control the Biomek. For example, instead of stipulating exact movement directions in terms of X, Y, Z and T, to access a certain well in a certain plate, the user can simply enter the row and column values of the well that is to be manipulated.

1.3.5.1 Biomek Sensing and Control

1.3.5.1.1 Head Sensing

The system contains sensing on the head in order to maintain control over the manipulation of the lab ware. The system can check the tool type when it picks something up. If the wrong tool is picked up or the intended tool is missing, then an exception is thrown and the Biomek is suspended, waiting for further action from the user.

The Biomek system contains information about the type of tip that is currently being used. Once the system has hold of a tip, either it can put the tip back or it can dispose of it. A tip can be in one of three states, according to the system; clean and empty, dirty and filled, and dirty and empty. If the state of the tip was clean, then the tip can be put back. However, if the state of the tip is dirty, the tip must either be used (if it is full) or it must be deposited. The head can sense whether a tip is missing and it can also detect whether a tip is blocked.

The robot that is being used is a Beckman Coulter 2000 Workstation, primarily a liquid-handling workstation[vii]. The reader is a Wallac Victor 2 plate reader[viii].

" The reader's counting modes cover all the main nonradioactive counting technologies, including fluorometry, TR-fluorometry, luminometry and photometry. It also has shaking and temperature control features. The server is running an IBM PC running Windows NT v3.2. It hosts the robot and reader's software and is connected to the local network and the Internet." [24]

[vii] See www.beckman.com for details
[viii] See www.lifesciences.perkinelmer.com for details

1.3.5.1.2 Work Surface Locations

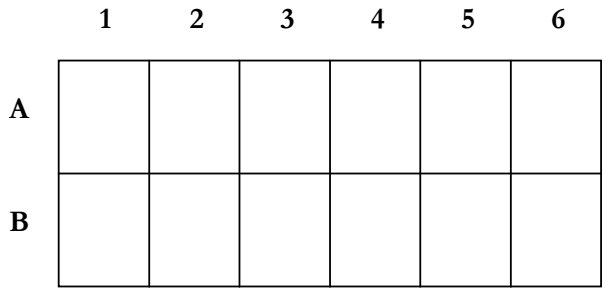


Figure 5 - *An illustration of the workspaces of the Biomek system*

The work surface used by the Biomek workstation is split up into 12 locations as shown in the diagram above. These locations may hold many different things including;

- Tools
- Labware
- Tip racks
- Other devices, such as VICTOR (optical density reader)

It is imperative that the system has the details of these locations. This usually occurs in the form of a configuration file passed to the robot (at the same time as the job files, or immediately before) to document the explicit locations of all the experiment materials. However, there are some restrictions. For example, the location A1 is always reserved for the Optical density reader. These twelve locations are the system's work domain and as such, must be fully accounted. Any piece of lab ware being held at any of the locations may have a label, which the system keeps track of. The system will know where the next clean tip is in a location occupied by a tip rack. Labware and racks can be moved automatically by the system from one location to another, but all labels, marks and unused tip markers will move with them.

1.3.5.1.3 Motion Control

This is monitored by the system with the use of a motion control status flag. When this flag is set to TRUE, then everything is considered to be fine. When this flag becomes set to FALSE, exceptions are thrown and the system begins to fail. There are several reasons why this flag may fail and a few are covered here.

The system monitors its X Y position from the encoding strips along these axes. As well as this, the system also keeps its 'presumed machine position' and 'related motor position'. The system can tell if there have been motion inaccuracies (or stalls) if the measurements from the encoding strips differ to that of the presumed machine position. Obviously, there will always be some difference. The machine's motion will never be 100% accurate, so a minimum is established, a set measurement that gives the calculation scope for error. The control status flag will only fail if the measurements from the encoding strips differ to the presumed machine position by this predefined minimum.

The system can detect a stall in the Z and T directions by taking measurements of the machine when it is in the home position. After a motion, the system will check its positions using the aforementioned methods (as well as some others) and then try and recalibrate itself by amending the differences.

Positioning, speed, acceleration, jerk, grip force can all be controlled and monitored by the system. The system can also detect a collision. Sometimes, there needs to be a collision (for example, when picking something up).

1.3.5.1.4 Method Loop

This involves keeping track of the position within the current method and also keeping track of the method loop exit conditions to see if they are fulfilled. For example, the system needs to know whether it has timed out. This would invoke the system to break out of the loop at the 'next available exit'. The system can break out of the method loop immediately when it tries to pick up 'Next' piece of lab ware, but it discovers it to be of the wrong type or missing.

1.3.5.1.5 Others

Liquid levels can be measured. The wash tool has its own states. The gripper has sensors detecting whether a good grip has been achieved. Much of the internals of the Biomek system are handled within the system itself and not known by the rest of the system. For example, a blocked tip will be handled internally. I.e., a clean tip will be obtained, and operation will continue. This and other experimentation exceptions will be handled within the Biomek system.

1.3.5.2 Workstation Coordinate System

The 3 coordinates used for specifying the movement of the tool head of the Biomek workstation are X, Y, and Z. These can be defined as;

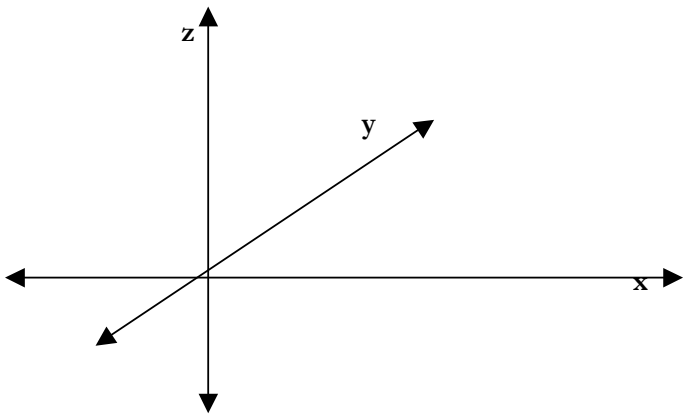


Figure 6 - *An illustration of the Workstation coordinate system*

The diagram above can be thought of as being viewed from the front, facing the workstation. The workstation uses a process called Z-Conversion. The system needs to be aware of where the bottom of the tool head is. This value may change depending on the type of tool attached to the head. Naturally, there are limits for the motors. If the user tries to send the motors to coordinates that it cannot physically go to, the system will stop and abort, returning a error message.

1.3.5.3 The T Coordinate

The T motor, in conjunction with the Z motor, is used to transfer liquids via the pipetting tool. The Z motor controls the up/down movement of the tool head. The T motor controls the amount of liquid that is to be taken up in the pipette. The T motor (along with the Z motor) is principally used in the process of liquid transfers.

1.3.5.4 Liquid Transfers

The Biomek uses the Z and T motors to aspirate and dispense liquid with the pipetting tool. As will be made clear later in the report, liquid transfers form the backbone of the job files that will be passed to the Biomek robot. The Z motor (up/down movement) moves the tool into and out of the liquid, whilst the T motor controls the plunger responsible for aspirating and dispensing the liquid.

There are four main types of variable that affect the process of liquid pipetting delivery. These are:

- **Prewet** – the amount of excess liquid moved into and out of a pipette before pipetting the correct volume, by means of wetting the tip.

- **Blowout** – the amount of excess air that is blown out of a tip, to ensure all the liquid has been dispensed.

- **Deliver** – the amount of excess liquid that will be pipetted in order to transfer the desired volume.

- **Bias** – an amount in order to compensate for the plunger (T motor) inaccuracies.

Please refer to [31] for details of a single liquid transfer.

1.3.6 The TCL Compiler

The TCL compiler is responsible for translating the experiment instructions (supplied from ASE-Progol) to BioScript job files that can be used by the Biomek Workstation to execute the experiments. A version of the TCL compiler already exists and an analysis and can be found in the appendix [31]. Below is a diagram of the requirements of the TCL compiler, taken from [30].

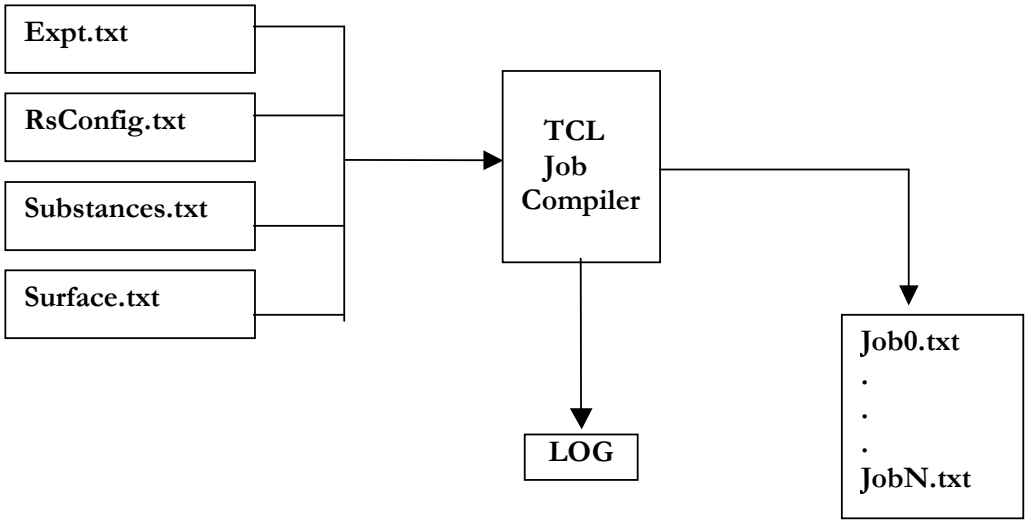


Figure 7 – *Requirements for the TCL Job Compiler*

Please refer to [32] for a specification of the redesigned TCL compiler.

1.3.6.1 A Note on TCL

TCL stands for Tool Command Language and is pronounced 'tickle'. Tcl can be thought of as both a language and a library. Tcl is a simple textual language with simple commands, which can ideally used for issuing commands to interactive programs such as text editors, debuggers, illustrators, and shells [40]. It has simple procedures and is programmable. This means that it can be used to program command procedures that can be more powerful than the standard ones.

Tcl was built with the idea that it would be used as part of a larger system, which in turn would potentially be using a set of other languages. Tcl is ideally used as a scripting language, tying together other parts of the system. Tcl can be used as a library embedded within another application, containing routines and procedures that allow the user to enter built-in Tcl commands to be used from within the application.

"Tcl was designed to make it easy to drop into a lower language when you come across tasks that make more sense at a lower level. In this way, the basic core functionality can remain small and one need only bring along pieces that one particular wants or needs." [40]

1.4 Reengineering and Refactoring

1.4.1 Reengineering

"Reengineering is any activity that;

A) Improves one's understanding of Software.

or

B) Prepares or improves the software itself, usually for increased
   i. Maintainability
   ii. Evolvability
   iii. Reusability" [1]

1.4.1.1 Why should we reengineer?

There are several reasons why it is useful to reengineer.

(i) **Maintenance** – For the majority of software applications, it becomes necessary to add or upgrade software. It is also highly likely that bugs will have developed within the code, no matter how vigorous the design and testing strategies. Upgrading a piece becomes increasingly difficult as the structure of the system decreases[ix]. Reengineering a system can have serious benefits when it comes to maintenance as it strives to do that very thing. I.e., to make tasks such as maintenance easier.

(ii) **Documentation** – both inside and outside the system. This is an important topic and is often neglected. The reengineering of a system, by its nature, can improve the documentation of a system without actually adding any documentation. If the structure of the system is a good one, with clearly setout, well-defined classes and methods, there becomes less need for explicit documentation. Frequent reengineering of a system can also ensure that documentation is kept up to date. Documentation is rarely altered in the case of debugging, or upgrading.

(iii) **Reuse** – When reengineering a system it is often the case that much of the current system can be reused as part of the new system. It follows that, after a project has been reengineered, parts should become more portable and versatile, improving the chance that components of the reengineered system can be reused elsewhere.

(iv) **Integration** – Reengineering can help systems evolve by allowing designers to employ new techniques, which would not have ordinarily have been implemented, within the system. It improves and elongates the thought process of a system, allowing it to become the most well-engineered, evolved, up-to-date system that it can possibly be.

1.4.1.2 How can we reengineer?

There are many reengineering strategies that can be used for any given system. These include Static/Dynamic Extraction (Presentation), Reverse Engineering, Redocumentation, Comprehension, Restructuring, and Refactoring. Two will briefly be described below.

[ix] Please note that this is not always the case, in particular for very small applications. The statement is meant in a general way in order to convey a widely acknowledged opinion.

1.4.1.2.1 Extraction/Presentation

This is the breaking down of the system, analysing the different parts of the system and then the reformation of the system with an improved and significantly simpler system. The diagram below can illustrate this process.

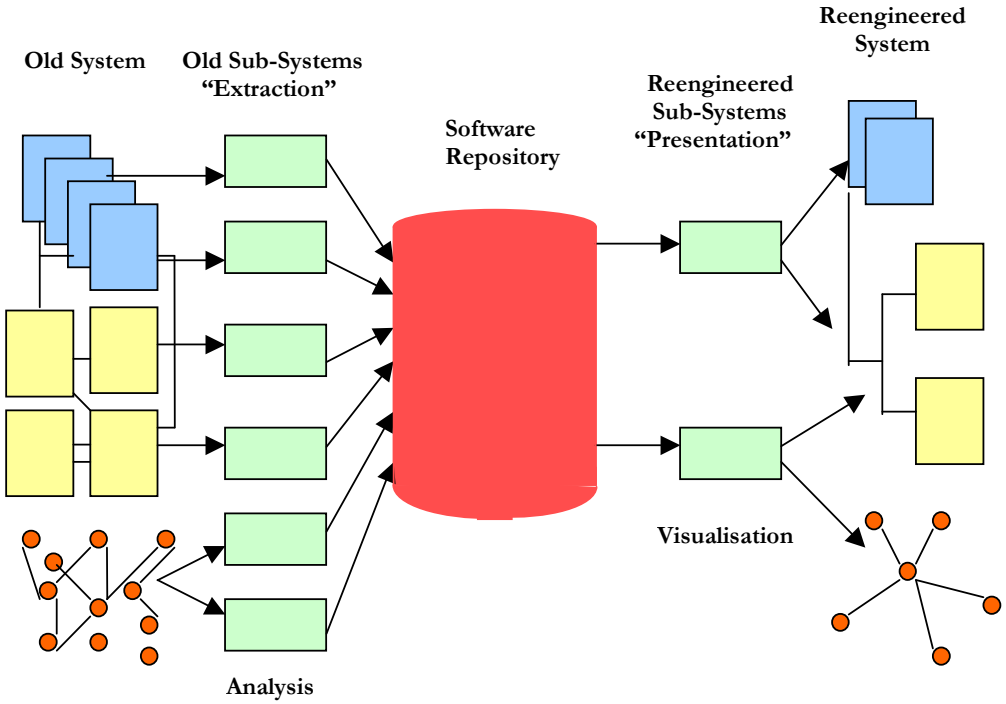


Figure 8 - *An illustration of the Extraction/Presentation process*

This type of approach seems useful when considering the job of reengineering the TCL compiler. One of the main problems with the existing system is the complexity of the system and the convoluted system structure. It would be extremely useful, and indeed fortunate, if the reengineering of the TCL compiler followed the same behaviour as the model system in the diagram above.

1.4.1.2.2 Reverse Engineering

Reverse engineering is used in order to establish how a specific part of a system works. It works by reversing the stream of 0's and 1's of the executable file and using a decompiling tool to transform the stream into the original code. This works with only a varying degree of success. Sometimes, the code produced can be unintelligible and of totally no use. Other times, it can unlock valuable algorithms to the onlooker.

Reverse engineering is not just employed in the Software engineering world. In the automobile industry, the same process is used. A manufacturer may purchase a competitor's car and take it apart, piece by piece, in order to see how it works.

Reverse engineering can be a useful tool when attempting to reengineer a system, the code of which is not available. However, as this is not the case with the TCL compiler, reverse engineering would be of little use.

1.4.2 Refactoring

Refactoring is concerned with the improvement of existing code. It consists of a collection of techniques to improve the structural integrity and performance of existing software. Using code refactoring, badly designed code can be reworked into a well-designed, robust piece of code [43].

However, refactoring can be dangerous. There is always a danger when altering already-working code that the newly 'improved' code may not work. By changing functional code, the developer can introduce bugs into the system. If refactoring is not conducted in a proper fashion, it can significantly reduce the productivity of the system. It is important that refactoring is done properly and systematically, otherwise the code can take on a 'hacked' quality [43]. Like every other area of software engineering, a degree of discipline is needed.

Refactoring does not alter the functionality of a system, rather it attempts to improve the internal structure. If done properly, refactoring can improve many aspects of the code as well as reducing the number of bugs. So, why is refactoring needed?

Systems are essentially designed and then written. Over time, they are altered, updated and improved. The code transforms from being a closely-knit well-designed well-engineered architecture, linked closely to the initial design, to becoming hacked. Changes are made to the system and concern to the overall integrity of the architecture is generally not given. Refactoring helps to reverse this trend by attempting to improve the design of code (which has been hacked) with the use of specific refactoring methods [43]. It is important to apply these methods in a systematic way. If not, they can cause even more confusion.

Chapter

2 Market Analysis

2.1 Problem Definition

The aim of this project is to provide an intermediary system that will "translate" the experiment details from ASE-Progol into machine instructions that can be read by the Biomek Workstation in order to correctly perform the experiments. A typical batch of experiments will result in the production of around 30,000 lines of robot instructions, so it can be seen that an automated system for this process is needed. However, there already exists a working version of the TCL Job Compiler. So, what is the point of building a new one? There are several reasons for this and these shall be explained here.

A brief analysis of the current TCL Job Compiler can be found in [31]. Although this analysis is not complete, it is clear that understanding the current version of the TCL Job Compiler is no easy task. Moreover, the analysis of the current version has proved extremely difficult and, as a result, the analysis remains incomplete. Upon examination of the code for the current TCL Job Compiler, it is obvious that an adequate analysis would take a great deal of time. Regrettably, that time has not been available, but even if it were, it would be debatable as to whether a more in-depth analysis would have been useful. The system is complicated, convoluted, and very poorly documented. There are no documents describing the processes within the TCL Job Compiler, nor many useful comments informing the reader of reasons to pieces of seemingly redundant code.

Furthermore, the original creator of the system has long since left. Ken Whelan has been responsible for maintaining the system, but admits to having only partial knowledge of the system. These consist of the parts he has been asked to maintain and this, in turn, demonstrates the lack of readability within the current system. As mentioned previously, there are no documents, formal or otherwise, supporting the current system.

The new system will be a working reengineered version of the current system. It will exhibit a degree of inheritance, abstraction and polymorphism in order to increase the portability, versatility and maintainability of the system. There will be a set of formal documents that will accompany this report and can be found in Appendix A. It is hoped that these documents will aid future revisions of the code and thus improving the portability, versatility and maintainability of the system.

2.2 Approach

This market analysis will concentrate on the application of the system in the realm of the pharmaceutical industry, in particular the application in LIMSs (Laboratory Information Management System), rather than the market area of compiler creation.

2.3 Market Area

The pharmaceutical industry is one of the most lucrative and fast growing industries today. The following graphs from [16] indicate the impact of the pharmaceutical industry.

The following graph shows the percentage of the nation's GDP (Gross Domestic Product) spent on medicines, during 2000. Note that this figure includes prescription and hospital medicines. Courtesy of [16].

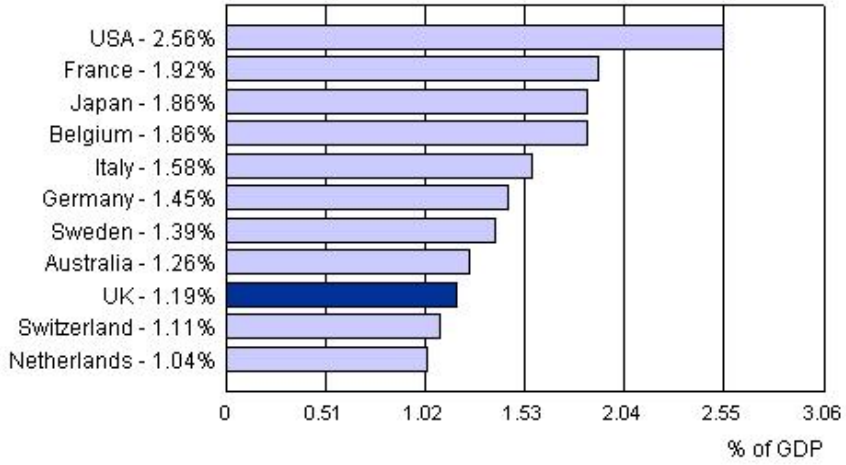


Figure 9 – *Pharmaceuticals as a percentage of GDP*

The following graph illustrates the average expenditure on medicine per person in 2000. Note that these figures include prescription and hospital medicines. Courtesy of [16].

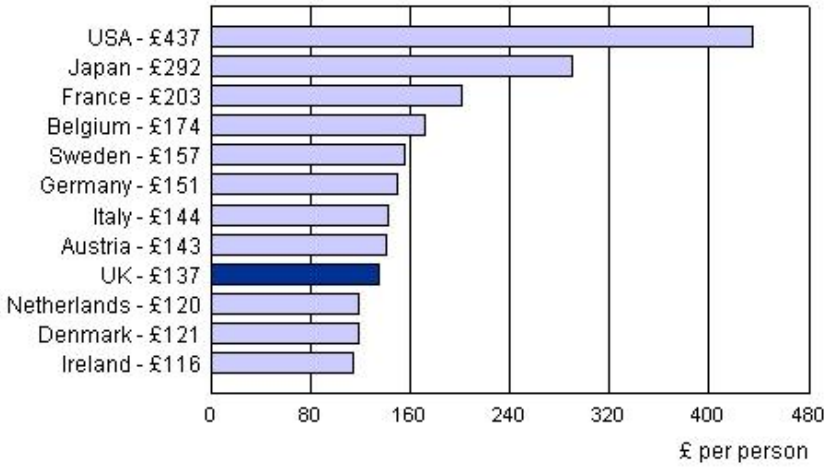


Figure 10 – *Average expenditure on pharmaceuticals per year per capita*

From the above graph, one can easily estimate that in the UK alone, the pharmaceutical revenue will be approximately £8 billion[x]. The obvious economic aspects hardly need to be explained. Pharmaceuticals represent one of the UK's leading manufacturing sectors. In 2001 alone, the pharmaceutical industry brought in a trade surplus of £2.9 billion, with pharmaceutical exports at £9.25 billion – more than £150,000 per employee [36]. Yet, the UK is still some way behind in comparison with our European neighbours. In spite of

[x] Approx. 58 million inhabitants – 58mil x £137 £8 billion.

having two of the largest and most successful pharmaceutical companies in the world, evidence has shown that doctors in the UK are still reluctant to prescribe new medicines. In fact, physicians in neighbouring countries are far more likely to prescribe medicines that have emerged into the market over the past five years [36].

Not only is the pharmaceutical industry large but importantly, it is also profitable. Below is a list of the top ten industries, indexed by their profit as a percentage of revenue (results from 2000).

1.	PHARMACEUTICALS	18.6
2.	COMMERCIAL BANKS	15.8
3.	COMPUTER PERIPHERALS	12.1
4.	TELECOMMUNICATIONS	11.7
5.	BEVERAGES	11.0
6.	SCIENTIFIC, PHOTO, CONTROL EQUIP.	10.6
7.	PUBLISHING, PRINTING	10.5
8.	DIVERSIFIED FINANCIALS	10.1
9.	COMPUTER AND DATA SERVICES	9.2
10.	SECURITIES	8.5

The recent profitability of the pharmaceutical industry has been extremely encouraging. For example, the largest American drug company, Merck, had profits of $6.8 billion in 2000, which was more than the profits of all the Fortune 500 companies in the airline, entertainment, food production, metals and hospitality industries combined.

2.4 Existing (Adaptable) Packages

At present, there are no known 'off-the-shelf' items of software (COTS) that could perform the functions that are proposed here. The main reason for this is that as the system is highly specialised, it would be unprofitable to produce and market commercial packaged software for this purpose. There are software houses as well as bioscience institutes, specialising in automated laboratory systems. If such a company were contracted to perform this task, it would be at great expense.

2.5 Benefits of Developing a Bespoke System

One of the main advantages to using a bespoke system for the TCL Job Compiler is that there are no additional overheads of having to learn and interact with another system. The problem with using commercial system is that there are often redundant parts that are not needed for our system and time would be needed to remove them. Individual add-ons can be incorporated in the system at the request of the client, a process that is not available with packaged software. Users should find this software more usable as it only contains facilities useful for them (the average Word user will only use 10-15% of the available facilities). One extremely influential reason for using bespoke software is cost. Not only is the cost of development and integration much less (in this case only), the department will also be able to maintain and develop its own systems, rather than having upgrade or even re-purchase software at additional expense.

2.6 Risks

As with all projects, there are risks. However, the economic risks associated with this project are very low. As a working version of the system that is to be revised already exists, only the time and labour can be lost. If the project was to be completed and the reengineered system was found to be inferior to the original version, then it would simply not be used. Obviously, this is an undesirable scenario, but if it were to occur, there would be no financial or heuristic loss regarding the operation of the whole project.

There is a danger that an improperly implemented TCL Job Compiler may allow the robot to damage itself using potentially harmful substances. For instance, there is nothing physically stopping the robot from pouring substances on to itself, the analyser, or anyone who may be nearby. These kinds of scenarios have to be identified and dealt with within the project requirements and constraints. However, there is still the possibility that an accident may occur and this can be considered as a possible risk.

There are also the inherent risks of the industry. The technology industry moves at an ever-increasing speed, pulling many other industries, including the pharmaceutical industry, along with it. There is a danger that a system engineered now may soon become out of date. This has partly been accounted for with the selection of Java as the development language. Java is a relatively new language, but has proved itself repeatedly in the object-oriented circuit. It has the benefit of its "write once, run anywhere" strategy, which has helped it achieve object-oriented supremacy among a range of development platforms.

There is also the danger of redundancy of the surrounding systems. The TCL Job Compiler has been designed to run with the ASE-Progol and Biomek systems. If these systems were to become redundant, this would have an effect on the worth of the TCL Job Compiler. The system would have to be redesigned to accommodate the surrounding redundancy.

There is also a danger that if the system is not completed to its fullest extent, then it may be unused, even though it may itself not have become redundant. This might occur due to there being no one to complete and integrate the new TCL Job Compiler into the rest of the system.

2.7 Possibilities for Expansion

The system has several possible areas for expansion. Firstly, the system itself will doubtlessly welcome the opportunity for some stream lining (and possibly some debugging). The whole system process takes approximately 30 hours with the TCL Job Compiler taking only about a minute, so there will not be much gain from much stream lining. However, there are possibilities of cutting down the 30 hours that the Biomek takes to complete the experiments. For example, the Biomek Workstation can handle operating up to eight pipettes at a time. Currently, the system only employs the use of a single pipette at any one time. An upgrade to take advantage of the 8-pipette strategy could significantly reduce the experimentation time. There are other possibilities of a heuristic nature that could be explored and integrated with relative ease to the new system in the future.

2.8 Other Possible Approaches

An alternative approach to reengineering the current TCL Job Compiler could have been to use refactoring techniques, described in 1.4.2. Refactoring is concerned with the improvement of the internal structure of a system without changing the external functionality of that system. When taken in this context, refactoring seems like an ideal solution. However, upon closer inspection of the system that would be refactored, it seems unreasonable to attempt to apply refactoring techniques. It is the opinion of the author that the current TCL Job Compiler is 'beyond refactoring', although refactoring techniques and issues can be taken into account in the reengineering of the current system.

Chapter

3 Requirements and Project Planning

3.1 Approach

This chapter endevours to provide a description of the general approach adopted for this project along with some details of the methods of planning undertaken during the lifetime of the project. It will cover the approach taken towards attaining the requirements for the project (the creation of the Requirements Definition document) as well as some explanations for some of the requirements. There will also be provided, an explanation of some of the techniques used during the project, including some details of time management and development decisions.

3.2 Requirements

A formal document for the Requirements Definition can be found in [30]. Although this section will not cover the requirements to the same depth as [30], some introduction and extra explanation to some parts is necessary. Document [30] aims to provide a description of the services that the system should provide. It also specifies some of the constraints under which the system must operate.

The approach for the conceptualisation of the requirements has been two-fold. The main requirement is that the new system should be a reengineered version of the current version. This led to a process of identification of requirements for the current TCL Job Compiler. Unfortunately, there is no documented form of requirements for the current system. The functional requirements have been ascertained through the identification and analysis of the input and output requirements of the current TCL Job Compiler. These are documented in [30]. Below is a brief description of the functional and non-functional requirements and are referred to in this report only for completeness. To provide any further documentation here would only serve to repeat material available in [30].

3.2.1 Functional Requirements

The functional requirements dictate the services that the system is to provide and are described in [30], section 3. Descriptions of the four input requirements are given; Experiment data, Substance data, Surface data and other configuration data. The only output requirements inherited from the current TCL Job Compiler is the job files (JOB0.txt, JOB1.txt…….JOBN.txt).

3.2.2 Non-Functional Requirements

The non-functional requirements aim to provide the new TCL Job Compiler with a higher standard of operation, previously lacking from the older system. Requirements such as efficiency, versatility and portability are described in [30], section 4.1, 4.2, 4.3.

3.3 Project Management and Planning

3.3.1 Development Model

The reengineering of the TCL Job Compiler does not formally adhere to any particular model, in spite of early attempts to do so. It was intended at the start that the system should be designed and implemented in line with the Unified Process. Upon careful reading of [15], it became clear that the system would not fit into the Unified Process as comfortably as had once been hoped. For example, due to the TCL Compiler being mostly an automated process, it was very difficult to identify more than one actor for the system. With only one actor (i.e., the one who starts the process), it became increasingly difficult to find a reasonable set of use cases. Upon further reading of [15], it became clear that the success of the Unified Process relied heavily on obtaining a relevant set of use cases and as such, would not be entirely suitable for this project.

It was decided that the project's software lifecycle would loosely conform to Boehm's spiral model as a form of evolutionary prototyping [28]. The system will be developed incrementally. Each new feature of the system will be developed and tested informally (as it is implemented). When the feature can be seen to be working correctly, it will be added to the system and work on the next feature will begin. This form of development will result in a system that is in danger of straying from the initial design, due to encountered problems along the way.

Formal Documents will be included at relevant intervals. These will include:

- Requirements Definition [30]
- Analysis of the Current TCL Compiler [31]
- System Design Specification [32]
- Basic Testing Plan [33]
- Subsystem Test Specification [34]

The System Design Specification [32] will represent the design of the completed TCL Job Compiler, although some alterations to the design occurred throughout the implementation process. Formal testing will be conducted after the implementation stage and will follow the guidelines put forth by [33][34]. The extent to which testing will be conducted will be decided after the implementation stage and will depend on the remaining time available.

An alternative development process would have been the Waterfall model. The problem with the Waterfall model is that the development stages are entirely modular and require that they be completed in a specific order. Furthermore, the completion of an earlier stage is required before the commencement of a later one. It was decided that this system should employ a more iterative form of development than the one offered by the Waterfall model.

3.3.2 Planning

"Project planning is concerned with identifying the activities, milestones and deliverables produced by a project. A plan must be drawn up to guide the development towards the project goals." [28]

Effective project planning is essential for a successful project. At the start of the project, a plan should be drawn up and used to drive the project. The plan should make allowances for both forseable and unforseable problems (where possible). The plan should be influenced by the state of the project. That is to say that the project plan is not resistant to change, rather it should be modified throughout the duration of the project.

The following section provides the project planning that was conceived at the beginning of the project. As is normal, the planning had to be modified and updated and the actual time scale of the project is discussed in chapter 7 (Limitations and Evaluation) for comparison purposes.

3.3.3 Effort Estimation

The table below offers an outlined breakdown of the various tasks connected with the project and the estimated amount of time that will be planned for each of the tasks. The time in hours can also be thought of the anticipated effort to complete each of the tasks. Tasks that involve the writing of documents are shown in italics.

Task	**Duration (hours)**
Background	
Problem Analysis/Definition	15
Analysis of Current TCL Compiler	20
Research	50
Approach Analysis	15
Market Analysis	20
Requirements Definition	10
Design	
Outline Design	15
Detailed Design	25
Design Specification	50
Implementation	150
Testing	20
Basic Test Plan	10
Subsystem Test Specification	10
Documentation	
Report	90
Total	**500**

Table 1 – *Effort Estimation*

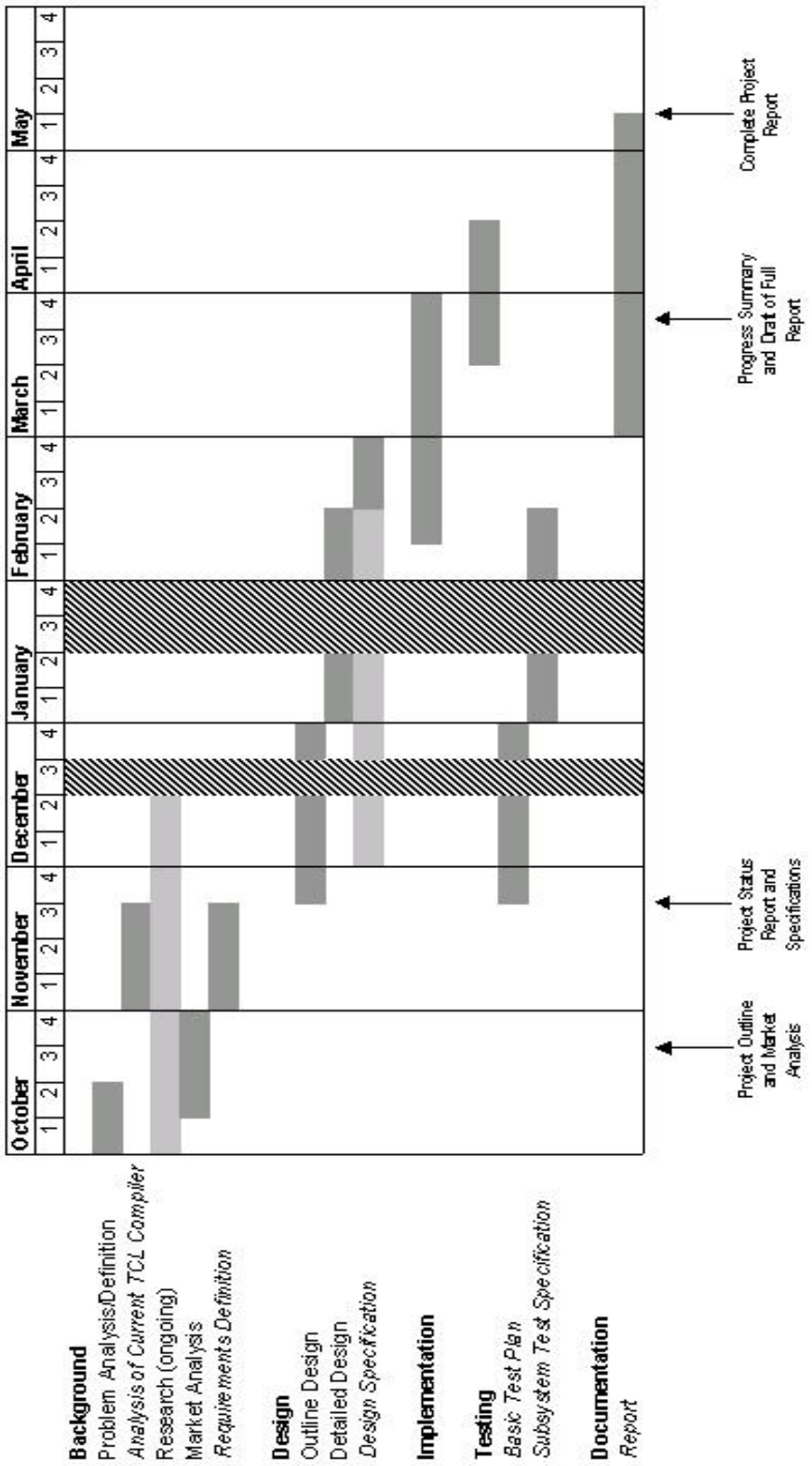


Figure 11 – *Initial project plan*

The above diagram shows the ideal schedule for the tasks during the course of the year. Time has been given to allow for both the Christmas holiday and the exams in late January.

3.3.4 Time Plan

There are several points of discussion regarding the time plan illustrated by the diagram above.

1. The different areas of shading on the diagram represent different intensities of work. This measurement does not attempt to quantify the amount of work to be done; moreover it represents a relative intensity of work in areas where more than one activity is scheduled at the same time.

2. The Requirements Definition and the Analysis of the Current TCL Compiler documents are to be written in parallel. This reflects the approach towards the requirements, described in 3.2. As the requirements are to be extracted from the current TCL Job Compiler, it seems obvious to work on the two documents in parallel.

3. The Design Specification document is to be added to, as long as the outline and detailed design are occurring. It is expected that the design document is going to be large and the plan accommodates for such an expectation.

4. It can be seen from the plan that the testing documents occupy the same time frame as the design activities. This is to reflect the need of the design documents to initiate the testing phase. Please note that the construction of the testing phase is not connected with the implementation phase to ensure that the testing occurs independently of the implementation. However, there is a period of time in which both implementation and testing are occurring. This has been included to accommodate for the presence of regression testing. This is further explained in chapter 6 (Testing).

3.3.5 Risk Analysis and Management

Section 2.6 deals with the associated risks of the system in terms of its marketability. There is also the inherent risk that the time planned for each stage of the development of the system is inadequate. This constitutes the greatest risk for the implementation phase, as there are so many other implications on the development process. It is hoped that possible scenarios regarding the failure to complete important objectives can be identified at an early stage and can therefore be taken into account. It would seem prudent that in such a scenario, features of the system could be withheld in order to complete some more important ones.

Chapter

4 Design

This chapter endevours to cover the design phase of the project. After a description and justification of the development tools and environment, there will be an overview of the approach of the design, followed by a description of the system. Please refer to Chapter 7 (Evaluation and Limitations) for design limitations and other information regarding the success of the design phase.

It is intended that this chapter be read in conjunction with the formal design document [32]. It is not the aim of this chapter to merely repeat material found in [32], but instead to add to it, improving the reader's understanding of the design phase of the project.

4.1 Development

4.1.1 Development Environment

The system is to be developed on a standard home workstation, running Windows 2000©. The reason being that this is the most accessible platform available to the developer at the current time. A Linux environment may have been more suitable for development with some significant software development advantages. However, a Linux build was not available for the primary development machine, due to an un-supported graphics card (GeForce 4).

The robot that is being used is a Beckman Coulter 2000 Workstation, which is primarily a liquid-handling workstation[xi].

4.1.2 Development Language

This is possibly the most important of all the design decisions, as it has the largest bearing on the finished system. There were two distinct development language options and these are described here.

4.1.2.1 TCL

There was an argument that the TCL Job Compiler should be written using TCL (1.4.6.1). This was a reasonable suggest given that the result of the compiler was a form of TCL, BioScript Pro. It was put forward that the compiler should be written in the language that it was compiling to. There are, however, some significant reasons for avoiding an implementation using TCL. Firstly, there is no explicit structure in TCL. It seems to be a primarily function-based language. Methods can be grouped into different TCL files, but other than that, there is no concept of program structure or any of the other facilities that co-exist with some of the modern paradigms today.

TCL was originally constructed for use as a scripting language and, as such, is not suitable for developing large programs. From the start, it lacked arrays and other structure from which to create linked lists. These have since been "added on", but in a way that causes TCL to operate far slower than before. TCL has been around for almost twenty years, but has generally been confined to specific tasks and has failed to present an impact to the same extent as other development languages, such as C or Visual Basic. This is not a

[xi] See www.beckman.com for details

reflection on the quality or success of TCL, rather attempts to highlight the limitations of application for the TCL language. As a result, support tools are scarce, compilers are restricted and there exist few sources of well-documented support. This does not detract from TCL as a language in itself, rather attempts to highlight the fact that TCL was not designed and is not suitable for developing a system such as the TCL Job Compiler.

4.1.2.2 Object-oriented Languages and the Use of Java

The alternative to using TCL was to use an object-oriented programming language, such as C++ or Java. After much deliberation, it was decided that Java was to be used for the development of the TCL Job Compiler for the following reasons. Java is renowned for its "Write once, run anywhere" philosophy, which refers to its method of platform-independence for development. Sun intended Java to be a *platform-neutral* language and so programs are written on a *Virtual Machine* rather than on a specific platform. Developers can write programs for the virtual machine on any platform and can expect the resultant code to run on any other platform. Java supplies the tools and support to provide a quality object-oriented environment. The component architecture of Java brings interoperability to a new level, giving Java a broad industry appeal. It also offers fast development for Web applications, as well as the ability to easily re-use code and provide add-ons to existing structures.

Given the requirements for the TCL Job Compiler (in particular the non-functional requirements) of increasing the portability, versatility and maintainability of the system, Java seems the obvious choice. Indeed, Java has been well noted for performing admirably in these areas [25][26][22][41]. Being an interpreted language, Java will not possess the same performance as the platform-specific languages such as C and C++, but with all the optimisation packages available for Java today, this discrepancy is minimised. Java also contains a well-designed and powerful set of APIs to increase program functionality.

4.2 Approach to Design

The design of the TCL Job Compiler was both an integral and challenging part of this project. This section aims to convey the thought processes that were involved during the design phase of the system. Upon reading this section, the reader should understand how the reengineered system was built and conceptualised from the beginning. This section aims to give the reader an idea of how the design phase began and has therefore been written more or less chronologically.

4.2.1 Study of the Current System

As the aim of this project was to reengineer the current TCL Job Compiler, it seemed prudent to start the design by analysing the current system. This largely prompted the formal document [31]. However, upon beginning the analysis of the current TCL Compiler, it was deemed that it would be both time-consuming and fruitless to provide a full analysis of the current system, hence the incomplete nature of [31]. It became clear that the new system would need to be much more than an updated version of the current system, not least as the current system was too complicated to update. It also became clear that the design would have to be developed using other sources rather than just from the current system.

4.2.2 Analysis of Inputs/Outputs

In order to start building a system to perform the job of the TCL Job Compiler, it seemed likely that most of the system would be centred around the processing and producing of the required inputs and outputs. By analysing the inputs, it was possible to identify the kinds of classes that were to be needed in the implementation. The use of an object-oriented design language helped greatly at this juncture, as the idea of an object could be applied to many of the features in the current setup. For example, from analysing the input files (described in [30]), it seemed obvious that there would likely be an object for a substance and an object for an experiment and so forth. Although this was not always the case, this strategy was found to make considerable headway for the design.

4.2.3 Visualisation and Incremental Build

Once it had been possible to build a set of possible objects for the initial stage of the design process, then it was possible to play around with the objects. This was done using class diagrams, whiteboards and informal UML diagrams, until an adequate structure was uncovered. Most of the structure was conceived from scratch by visualising the functionality of parts of the system. Each part of the system was visualised, interpreted and fitted in one at a time. The system was therefore built incrementally, each subsystem being added to the system once it had been developed and tested. As mentioned before, this type of approach is adopted during Boehm's Spiral development model, except that in this case, there was no rigid structure of iteration. As each subsystem was designed, so it was implemented, approximately in the order of the documentation [32] and 4.2.4.

4.3 Design of the System

As stated before, the aim of this section is not to repeat material found in [32], but to be used together with [32] to provide a more thorough understanding of the design phase, occasionally providing reasoning to specific design issues. Often, there shall be no merit to expanding on the information in [32] and in that case, no information will be added.

Each subsystem shall be covered in the following section, building up a structured model of the finished system.

4.3.1 The Experiment Creation Subsystem

It seemed likely that there would need to be a subsystem that dealt with the creation of the experiments and the substances attached to those experiments. This came largely from the methods described in 4.2.2. There is an experiment file (Expt.txt). This is a list of experiments with the substances that constitute each experiment. There is a corresponding file containing all the substances and their attributes (Substance.txt). It seemed the natural progression to create a subsystem that handled both of these files, extracted the information and created an internal model of the experiments. The result was two separate subsystems, one to deal with the experiments and the other to deal with the substances.

4.3.1.1 The Substance Subsystem

Details of the Substance subsystem can be found in [32] - 3.1.1. The Substance subsystem is very simple. Within SubstanceList, a method 'populateSubstances' handles the file, Substances.txt, reads the substance information and then creates an array of corresponding substances.

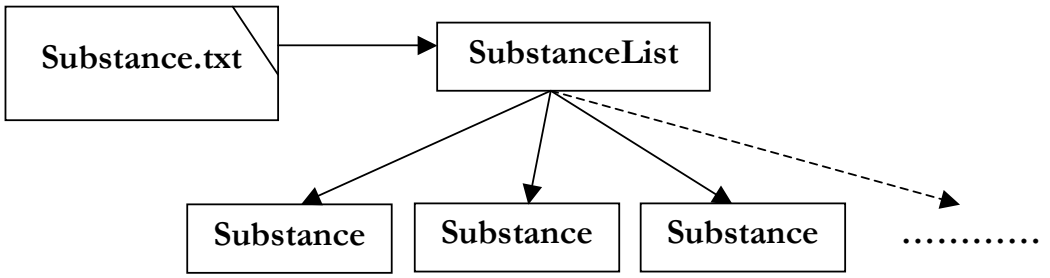


Figure 12 - *An illustration of the Substance Subsystem*

4.3.1.2 The Experiment Subsystem

Details of the Experiment subsystem can be found in [32] - 3.1.2. The Experiment subsystem again is very simple and is designed around the use of the experiment file, Expt.txt. Within ExperimentPlate, the method 'populateExperimentPlate' handles the file (Substance.txt) and creates a set of ExperimentPlates (normally 4), and populates them with the correct experiments taken from Expt.txt.

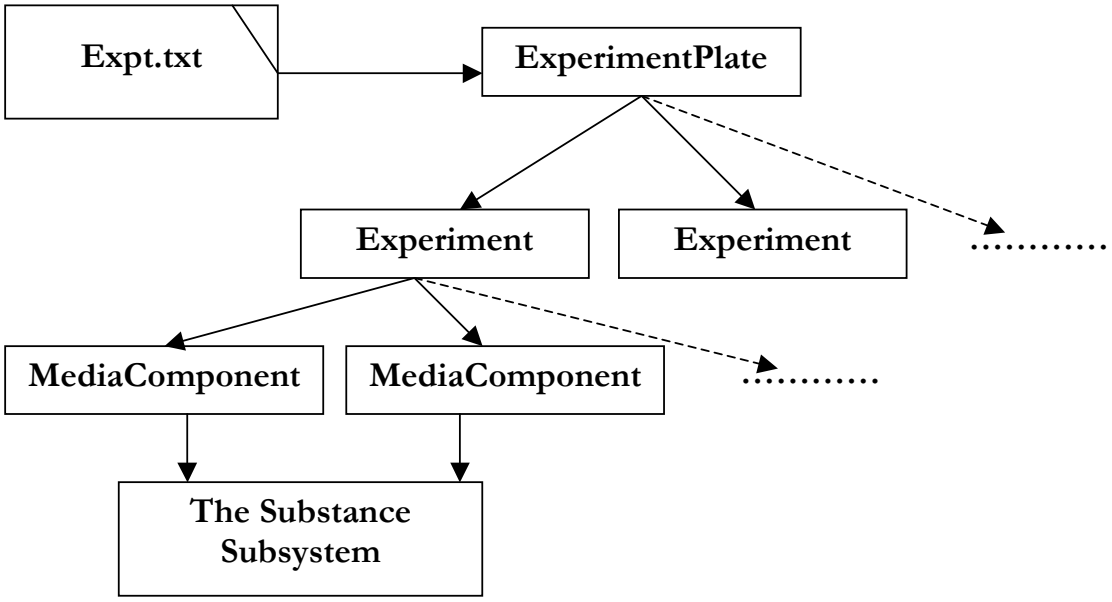


Figure 13 - *An illustration of the Experiment Subsystem*

The most important thing to note about the interaction between the Substance and Experiment subsystems is the difference between a MediaComponent and a Substance. Assuming that a Substance would be linked to an Experiment in some way, it became evident that the amount of the substance in the experiment would have to be stored somewhere, but it did not seem feasible to keep it with the Substance. As a result, a new class was formed, namely MediaComponent. A MediaComponent contains both a Substance and a float representing the amount of Substance. As can be seen in [32] - 3.1.2.1, an Experiment contains MediaComponents, which in turn contain Substances. This helps to separate the Substance and Experiment subsystems and solves the problem that a substance may be used many times in many experiments in differing amounts.

4.3.2 The Log Subsystem

The Log subsystem is a very simple subsystem and is adequately explained in [32] – 3.2. It would not be worthwhile (or even possible) to delve further into the operability of the Log subsystem. For details on the function of the Log subsystem, please refer to [32] – 3.2.1.2.

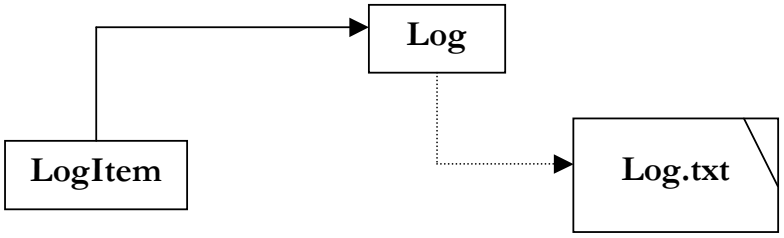


Figure 14 - *An illustration of the Log Subsystem*

4.3.3 The Configurator Subsystem

Details for the Configurator subsystem can be found in [32] – 3.3. Given the approach mentioned in 4.2.2, the next file to look at logically was the JOB0.txt file. This file differed to the other job files in that it consisted wholly of configuration data. Before the Biomek could begin, it would need to be aware of all the tools, labware and other devices operating within its environment. This data was carried by the configuration job file, namely JOB0.txt, and consisted of around 1000 lines of configuration data. In order to handle this configuration data, the Configurator and Surface subsystems were created.

As mentioned before, JOB0.txt consists of around 1000 lines of configuration code. Most of these lines are responsible for setting a specific variable for a specific item (labware, tool, device, etc). The unfortunate result of this is that there must be included, somewhere in the code of the system, a section that sets each of these variables. This will result in approximately 1000 lines of code somewhere in the system.

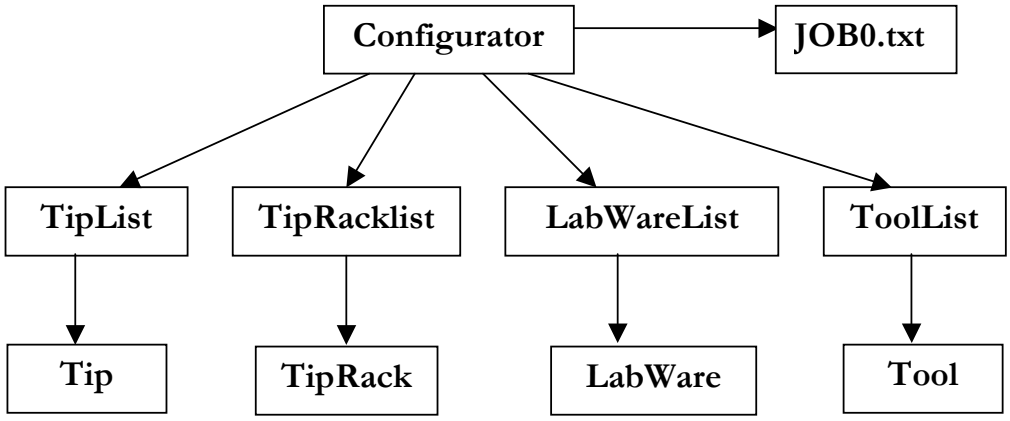


Figure 15 - *An illustration of the Configurator Subsystem*

The module, Configurator, will configure all the devices, tools, and labware for the configuration file and be responsible for initiating the creation of the Biomek surface (see 4.4). One-by-one, the configurator creates and initialises the devices, tools and labware and adds them to the workstation [32] – 2.4.22. After this has completed, the surface is created (using the surface file, Surface.txt), and finally the configuration file, which is comprised of the toString method of the WorkStation, is created.

4.3.4 The Surface Subsystem

It is debatable as to whether the Surface subsystem should constitute a separate subsystem to the Configurator Subsystem, since they are closely tied to each other and both are responsible for the creation of the configuration file. There were two main reasons why it was decided that the Surface subsystem should become its own subsystem.

The first was for the sheer size of the two subsystems. There seemed a natural segregation, however small, between the two and the opportunity to break the system into smaller parts, was taken. The second reason was that, although data from the Configurator would be used later on elsewhere in the system, it would mainly be for creating the configuration file. The Surface subsystem would doubtlessly also be used for creating the configuration file, but would also be used significantly during the creation of the job files.

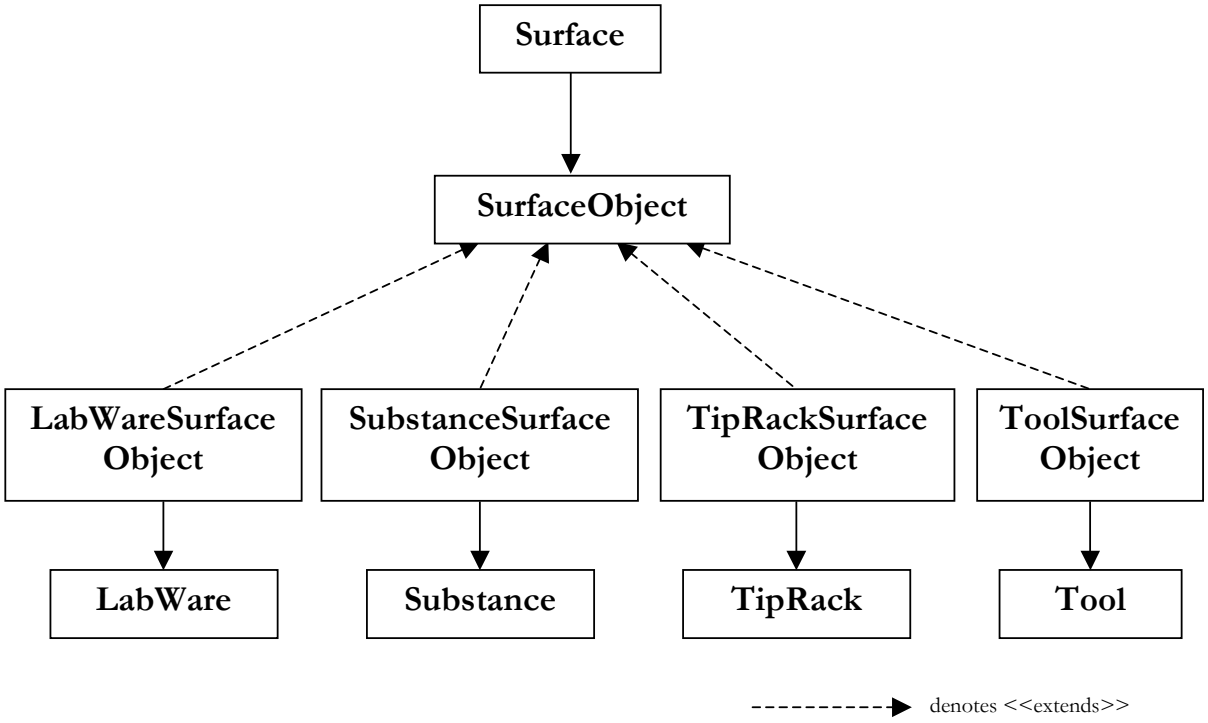


Figure 16 - *An illustration of the Surface Subsystem*

The module, Surface, consists primarily of a 2-dimensional array, capable of holding SurfaceObject items. It was decided that a generic form of surface object would be used, rather than have the surface array hold individual surface objects of a specific type (see [32] – 3.4.1.1 for an illustration of the Surface subsystem). This was due to two factors; the surface objects varied significantly from each other to prevent them being referenced as the same object, however, having significant enough similarities to warrant a hierarchical relationship.

4.3.5 The Builder Subsystem

The Builder subsystem is responsible for the creation of the job files (JOB1.txt… ..JOBN.txt). Within the Builder subsystem is the module, WorkStation. This is an important module as it contains all the information about the environment of the Biomek workstation. Therefore, it contains details on all the tools, devices and labware available within the environment as well as the Surface object that contains details about the location of all the necessary parts of the experiments. As mentioned before, the configuration file is constructed through a call to the toString method within WorkStation.

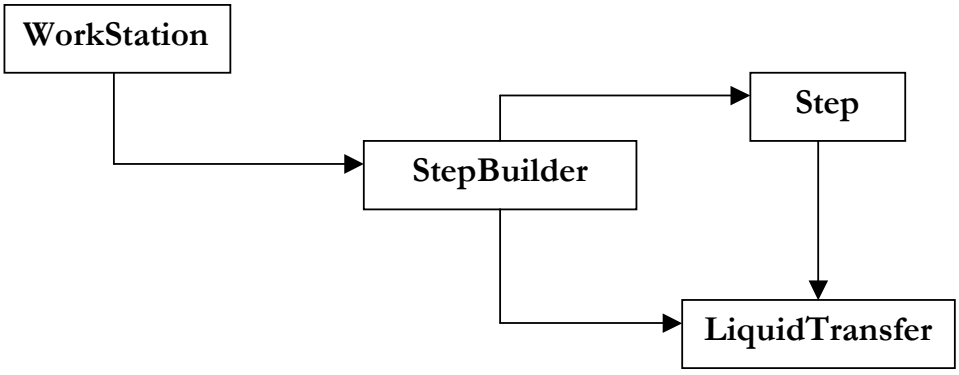


Figure 17 - *An illustration of the Builder Subsystem*

The role of the Builder subsystem is the construction of the job file (JOB1.txt). The Builder subsystem works by taking all the SurfaceLabWare objects within the WorkStation (i.e., all the experiments), and breaking the experiments down into Steps, based on the substance. These experiment steps are grouped into Vectors according to their substance (metabolite, agar or yeast). The steps in each Vector are then extracted and used in a LiquidTransfer operation. The results of these liquid transfers are stored in a string until all the transfers have completed, whereupon the string is exported to a text file (JOB1.txt).

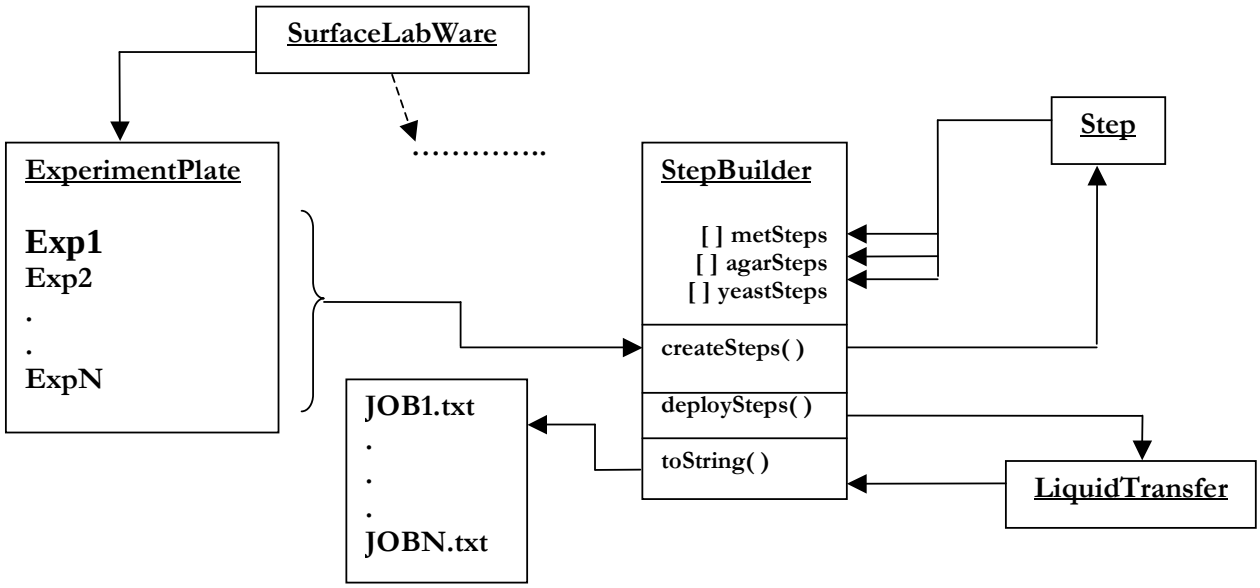


Figure 18 - *An illustration of the functionality of the Builder Subsystem*

A diagram illustrating the functionality of the Builder subsystem can be seen above and is taken from [32] – 3.5.1.2, where a more detailed explanation of the subsystem's function can be found.

4.4 TCL Job Compiler

The following section will describe the process of the complete system. It is intended to represent the culmination of the description of the previous subsystems. The following illustration should provide an explanation to how the subsystems interact with one another.

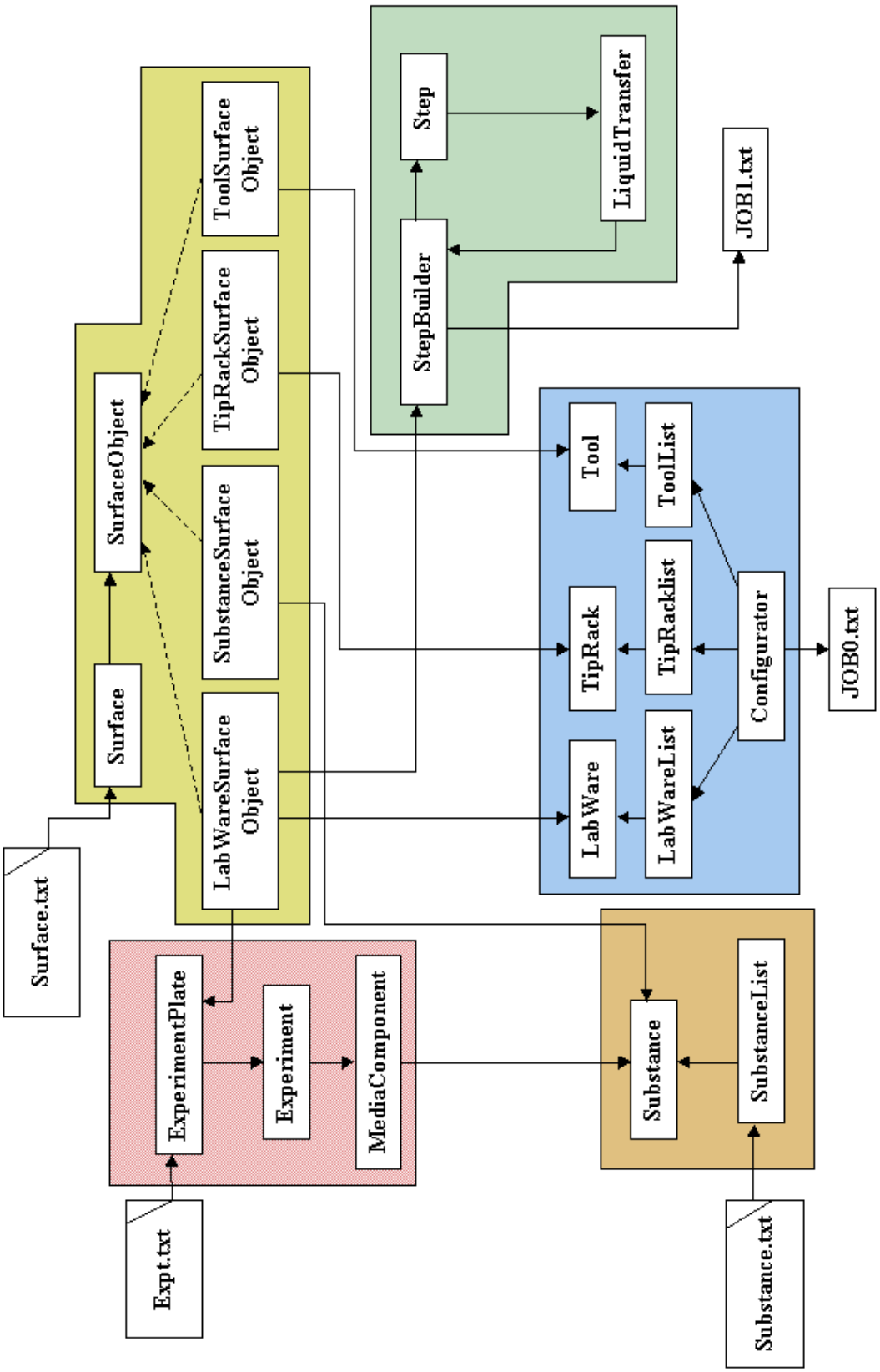


Figure 19 *- An illustration of the complete TCL Job Compiler*

Colours have been used to signify the subsystem boundaries. These are:

- Red – Experiment Subsystem
- Brown – Substance Subsystem
- Yellow – Surface Subsystem
- Blue – Configurator Subsystem
- Green – Builder Subsystem

The following is a step-by-step guide to the intended operation of the TCL Job Compiler. It was necessary to simplify some of the processes at this stage in order retain an understanding of the system.

1. A WorkStation is initialised, using the default constructor.

2. A Configurator object is created, passing to it a reference to the WorkStation.

3. The setup( ) method from Configurator is called, passing to it the substance and surface filenames.

    a. TipList, TipRackList, LabWareList and ToolList are initialised and added to the WorkStation.
    b. SubstanceList is initialised and the array of substances is populated by the method, populateSubstances( ). The SubstanceList is then added to the WorkStation.
    c. The Surface is initialised and the 2-dimensional array is created by parsing the surface file. The surface is then added to the WorkStation.

4. The method, writeJobFile( ) is called passing in the directory in which to place the job file. This method calls the toString( ) method for the WorkStation and writes the result to a file called JOB0.txt in the specified directory.

5. A Vector containing the labware (i.e., the ExperimentPlates) is returned from the WorkStation.

6. For each experiment plate, the method, populateExperimentPlate( ), is called. This populates the ExperimentPlate with Experiments in accordance with the file Expt.txt. The ExperimentPlates are added to the relevant LabWareSurfaceObjects.

7. The substances from each experiment are passed into an object of type Step. This will contain data such as: Substance, amount, source and destination.

8. The Steps are sorted into Vectors by the type of Substance that is being transferred. These are metabolites, agar and yeast.

9. The Vectors are passed into deploySteps( ) one-at-a-time in a specific order (metabolites, agar and then yeast).

10. deploySteps( ) creates a LiquidTransfer object for each Step and executes the create( ) method for the Step. The toString( ) of the LiquidTransfer is captured and appended to a string, maintained within StepBuilder.

11. When all the LiquidTransfers have completed, the string is outputted to a file called JOB1.txt.

Chapter 5

5 Implementation

5.1 Approach

As mentioned in 4.2 (Approach to Design), the implementation phase was carried out simultaneously with the design phase. Once a feature had been designed, it was implemented and then added to the system. This chapter aims to explain the process of implementation and describe some of the methods of implementation used to create the TCL Job Compiler. This chapter will be structured much in the same way as the previous design chapter. Each subsystem will be described in turn, with particular attention paid only to the important or interesting parts. It is not the aim of this report to explain every single part of the system. Indeed, much of the functionality of the new TCL Job Compiler is self-explanatory, thanks to the high level of documentation within the code (i.e., comments). Areas that are not covered explicitly in this chapter may be found in the formal design document [32], or adequately explained within the code itself.

The nature of this chapter is not represented consistently due to the way it was itself implemented. For sections 5.2.4 and 5.2.5, the methods were implemented by first developing algorithms from which to write code. In this case, the algorithms are shown. However, in sections 5.2.1 and 5.2.2, it can be noted that there are no specific algorithms and the methods are described using examples from the actual code alongside textual explanations of the process.

The complete code listing for each of the methods described in this chapter can be found in Appendix B. The complete system code listing has not been included as much of it would be unimportant to the understanding of the system. The complete system code listing is however available[xii].

5.2 Implementation of the Subsystems

5.2.1 The Substance Subsystem

The Substance subsystem is very simple with few significant points of functionality. The main point of interest in the Substance subsystem is the SubstanceList module. The method, populateSubstances( ) is responsible for populating the array within SubstanceList with Substances as directed from the substance file. 5.2.1 aims to describe this method in more detail, revealing how the file is split into its necessary parts. populateSubstances( ) can be found in Appendix B.

There are two forms of string manipulations at work in populateSubstances( ) and they are utilised together to achieve the desired result. Below is one possible version of the substance file that could be used.

```
minimal-agar 0 0 0 0 1 0 1 0
YDR007W:E-2 0 0 0 0 1 0 1 1
C00166:2E-1 0 0 0 0 1 0 1 0
C00108:2E-1 0 0 0 0 1 0 1 0
wild-type:E-2 0 0 0 0 1 0 1 1
YDR354W:E-2 0 0 0 0 1 0 1 1
C00079:2E-1 0 0 0 0 1 0 1 0
YDR035W:E-2 0 0 0 0 1 0 1 1
C00078:2E-1 0 0 0 0 1 0 1 0
YBR166C:E-2 0 0 0 0 1 0 1 1
```

[xii] Please consult author for complete system code listing – Ben Tagger – bnt8@aber.ac.uk

The name appears first, followed by four integer values; the prewet delay, the blow delay, the dispense delay, and the aspiration delay, then four Boolean values; prewet, tip touch, blowout, and knockout.

It is necessary to create Substance objects for each of the substances above and set the specified parameters for each one. Firstly, it is necessary to split the file into parts that contain only one substance. For this purpose, a TextFileReader object is used. A temporary array of Strings is created to store the strings relating to the different substances. The array is then populated with the strings from the file. Please refer to the code below.

```
TextFileReader tReader = new TextFileReader(fileName);

String [ ] tmp = new String [theList.length];
for (int i = 0; i < tmp.length; i++)
{
      String temp = new String(tReader.readString());
      tmp[i] = temp;
}
```

The operation 'tReader.readString( ))' passes in the string up to the end of the line. Therefore, in this case, the array, tmp, is populated with the String up until the end of the line. Hence, the contents of the array at position 0 will be:

```
tmp[0]:

minimal-agar 0 0 0 0 1 0 1 0
```

…and so on for the remaining strings. Once the array of strings (representing each substance) has been created, a StringTokenizer can be used to break each string into its integral parts and these can be used to create a Substance object. First, the tokenizer must be initialised with the correct delimiter. In this case, it is necessary to use a space as a delimiter and so the tokenizer is initialised as such:

```
StringTokenizer st = new StringTokenizer(tmp[i], " ");
```

'tmp[i]' refers to the String from the array that is currently being examined in the for loop. Following this initialisation, a Substance object is created and its name is passed using the StringTokenizer.

```
temp.setName(st.nextToken());
```

The method, nextToken( ), passes the next whole argument (as a string) that has been delimited by whatever is being used as a delimiter (in this case, a space). The remaining attributes for the Substance are captured in the same manner. An example of the parsing of the prewet delay can be seen below.

```
String a = st.nextToken();
int ai = Integer.parseInt(a);
temp.setPrewetDelay(ai);
```

In this same way, each substance from the substance file is passed into the system and added to the array within SubstanceList.

5.2.2 The Experiment Subsystem

Much in the same way as the Substance subsystem, the Experiment subsystem is mostly constructed with very simple modules. The only point of interest and importance is the method, populateExperimentPlate( ), in ExperimentPlate. This method works much in the same way as the method, populateSubstances( ), described above (5.2.1). The listing for populateExperimentPlate( ) can be found in Appendix B at the end of this report. The purpose of populateExperimentPlate( ) is to extract the experiment details from the experiment file, Expt.txt. An example of the experiment file can be seen below.

```
# A1
minimal-agar          120
time          57600

# B1
minimal-agar          120
time          57600
.
.
.
# E9
minimal-agar          120
YDR354W:E-2          20
C00108:2E-1          20
time          57600

# F9
minimal-agar          120
YDR354W:E-2          20
C00108:2E-1          20
time          57600
.
.
.
```
and so on…

It can be seen from the example above that experiments can contain a number of substances with varying amounts. Therefore, the implementation must reflect this and accommodate for this variety. The first challenge is to retrieve the text from the experiment file. The contents of the experiment file were transferred to a single string with the use of a BufferedReader (not explicitly explained here – see Appendix B). Then, the string had to be broken down into single experiments. This was constructed much in the same way as the substances were split in 5.2.1. A StringTokenizer was created that used '#' as a delimiter.

```
StringTokenizer st = new StringTokenizer(temp, "#");
```

An array was created that would contain the experiment strings as shown below.

```
// create an array with the number of tokens as a size
String[] subStr = new String[st.countTokens()];
for (int i = 0; i<subStr.length; i++)
```

```
{
      subStr[i] = st.nextToken();
}
```

subStr is an array that has the string representing a single experiment as each of its elements. It is then necessary to split each string of experiment into its integral parts so that an Experiment object can be built. This became quite a serious problem. The String Tokenizer that Java uses does not recognise the end of the line as a delimiting character. Therefore, when the experiment string was split up, the end of the line ran in with start of the next line. Using space as a delimiter, the results looked like this.

```
# E9
minimal-agar          120
YDR354W:E-2          20
C00108:2E-1          20
time          57600

is delimited to this...
E9minimal-agar
120YDR354W:E-2
20C00108:2E-1
20time
57600
```

This proved extremely difficult to work with, as it required having to split words at designated points and there seemed no simple way of achieving this. After much deliberation and hours spent at trying to split the string satisfactorily, it seemed that the only way to achieve this was to modify the experiment file. This could be done relatively easily as the creation phase of the experiment file, but it was still unfortunate to have an external influence from the TCL Job Compiler. The experiment file was altered to include an extra delimiter at the end of each line. An example of the revised experiment file can be seen below.

```
# A1,
minimal-agar          120,
time          57600,

# B1,
minimal-agar          120,
time          57600,
.
.
.
# E9,
minimal-agar          120,
YDR354W:E-2          20,
C00108:2E-1          20,
time          57600,

# F9,
minimal-agar          120,
YDR354W:E-2          20,
C00108:2E-1          20,
time          57600,
.
.
```

.
and so on…

Notice the delimiters at the end of each line. Now, the string tokenizer can pick up the end of the line using the ',' as the delimiter. Each of the experiment strings is split using the following tokenizer.

```
StringTokenizer tz = new StringTokenizer(subStr[i], ", ");
```

The line above creates the tokenizer for each experiment string using both the ',' and a space as delimiters. Using spaces and the introduced commas as delimiters, the results looked like this.

```
# E9
minimal-agar          120
YDR354W:E-2          20
C00108:2E-1          20
time          57600

is delimited to this...

E9
minimal-agar
120
YDR354W:E-2
20
C00108:2E-1
20
time
57600
```

Having broken the experiment file into the sections as can be seen above, there remained only the matter of identifying each string and creating the resultant Experiment. The first token to be dealt with is the well (in this case, 'E9'). It was necessary to convert the coordinate (E9) to a set of coordinates. This was further complicated by the fact that there were up to 12 possible columns. The solution to this problem was to feed the coordinate string into a character array. Below are two examples for two coordinates, E9 and B12 to show the difference in handling the various sized character arrays.

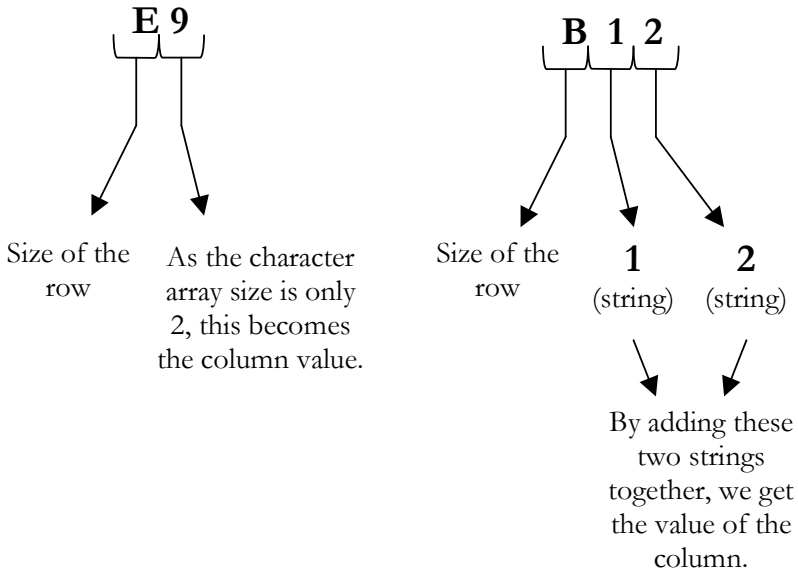


Figure 20 - *An example of the tokenization of the experiment file*

By analysing the first character, the row number could be ascertained. Then, by measuring the size of the character array, it could be established whether the size of the column was either one or two digits. If the size of the character array were two (column being single digit), then the column value would simply be the value of the second element of the character array. However, if the size of the character array were three (showing that the column was a double-digit number), then the value of the column would be 10 plus the value of the third element of the character array.

After setting the rows and columns for the Experiment, it was necessary to add the substances. Given that an Experiment can contain any number of substances, a method was needed to extract that number of substances. Seeing that the first token after the last substance of any experiment was 'time', the following section of code was developed.

```
String next = tz.nextToken();
do
{
      // create instance of new Media component
      MediaComponent media = new MediaComponent();

      // uses the first token
      next.trim();
      media.setSubstance(substanceList.findByName(next));

      // set amount of media component
      next = tz.nextToken();
      next.trim();
      media.setAmount(Integer.parseInt(next));

      exp.addMediaToList(media);
      //move onto next token
      next = tz.nextToken();
} while (!next.equals("time"));
```

For each Substance that exists, this algorithm will add it to the Experiment (as a MediaComponent). The loop will break when the next token encountered is 'time'. This signifies the end of the substances. The next token will therefore be the time value and can be entered as such.

5.2.3 The Log Subsystem

As mentioned in the design section for the Log subsystem, there is nothing complicated or of interest within the Log subsystem. Please refer to the formal document, [32], in which the design and structure of the Log subsystem is adequately explained.

5.2.4 The Configurator and Surface Subsystems

The following part of the system to be explained is the section of code within the Configurator subsystem that creates the surface in the Surface subsystem. It is the final

part of the method, setup( ), in the Configurator class within the Configurator subsystem. A selected part of setup( ) can be found in Appendix B. As in the previous sections, it involves the use of a file. In this case, it is the surface file (Surface.txt). An example of the surface file can be seen below.

```
A1 VICTOR -51.8 33.9 78.1 15
A21 P20L
A22 P200L
A24 Gripper
A3 P250
A4 96-well flat
A5 96-well flat
A6 quarter vertical
B2 P20
B3 96-well flat
B4 96-well flat
B5 quarter vertical
B6 half single
A6A1 YBR166C:E-2 20000.0
A6A2 C00078:2E-1 20000.0
A6A3 YDR035W:E-2 20000.0
A6A4 C00079:2E-1 20000.0
A6A5 YDR354W:E-2 20000.0
A6A6 wild-type:E-2 20000.0
A6A7 C00108:2E-1 20000.0
A6A8 C00108:2E-1 20000.0
B5A1 C00108:2E-1 20000.0
B5A2 C00166:2E-1 20000.0
B5A3 YDR007W:E-2 20000.0
B6A1 minimal-agar 120000.0
```

This file is used to document the locations of all the devices, tools and labware that exist within the Biomek system. It is imperative that the Biomek system be aware of these items and their locations. Therefore, it was necessary to create an *in silico* surface representation and populate it using the surface file (above). The surface was modelled using a 2-dimensional array in the class, Surface.

Before analysing the algorithm, it is necessary to understand the way that the surface is represented in the system.

The Biomek surface can be illustrated by the following diagram.

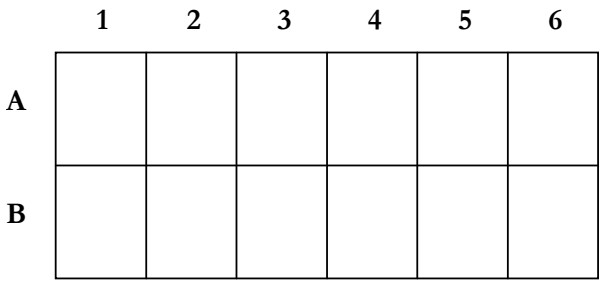


Figure 21 - *An illustration of the Biomek workspace*

Each of these 12 locations can hold a device, tool or piece of lab ware or any number of these, or conversely it can be empty. The system model must be able to adequately represent this. For example, consider the following line from the surface file.

```
A6A1 YBR166C:E-2 20000.0
```

The 'A6A1' means that within the workspace, A6, there is an area, A1, in which there is 20 ml of the substance 'YBR166C:E-2' (A type of mutated Yeast – referred to as a knockout). In order to model this situation correctly, it was decided that the 2-dimensional array would consist of one array of 12 locations (one for each workspace) and the other array representing the possible segments of the locations.

Consider the following entry from the surface file,

```
A6A1 YBR166C:E-2 20000.0
```

This would cause the system to place the substance 'YBR166C:E-2' into the array at [6][1].

Also the entry,

```
B4 96-well flat
```

This would cause the system to place the LabWare object referenced as '96-well flat' into the array at [10][0]. This is in location 10 (B4 = 10) and is referenced as segment 0, indicating that the workspace is not segmented.

The following algorithm is used to set up the surface.

1. Create the StringTokenizer for string containing the line from the surface file, using spaces as a delimiter.

2. Call nextToken( ) and pass the contents to a temporary string (coord). This string should contain the coordinates for the surface object (e.g., A1, A22, or B6A4).

3. If the coord string is 2 characters long (e.g., A1)

    a. If the 1st character is 'B', then the location = 6 + 2nd character.
    b. Otherwise, the location = 2nd character.

4. If the coord string is 3 characters long (e.g., A22)

    a. Same as 3.a.
    b. Same as 3.b.
    c. Segment = 3rd character.

5. If the coord string is 4 characters long (e.g., B6A4)

    a. Same as 3.a.
    b. Same as 3.b.
    c. Segment = 4th character.

6. Call nextToken( ) and pass the contents to a temporary string (type).

7. If the string 'type' is 'VICTOR', then the tool, victor, is added to the surface.

8. Check if 'type' is the name of a tool. If it is, then add the tool to the surface.

9. Check if 'type' is the name of a tip rack. If it is, then add the tip rack to the surface.

10. Check if 'type' is the name of a piece of labware. If it is, then add the labware to the surface.

11. Check if 'type' is the name of a substance. If it is, then add the substance to the surface.

12. Add the surface to the WorkStation.

5.2.5 The Builder Subsystem

There are several parts of the Builder subsystem that need to be explained. The Builder subsystem represents the greatest area of functionality in the system and, as such, contains the locus of control for the rest of the system.

The first two areas of interest are from the StepBuilder class within the Builder subsystem. The first method, createSteps( ), is responsible for extracting the substances from the Experiments and aligning them into Vectors of Steps. The second method, deploySteps( ), is responsible for taking these Vectors of Steps and executing the appropriate LiquidTransfers in the correct order.

The third area of interest is the create( ) method with the LiquidTransfer class. This method creates the bulk of the job file, which itself is constructed from many liquid transfers.

5.2.5.1 createSteps( )

The following algorithm forms the basis of the method, createSteps( ). The full listing of createSteps( ) can be found in Appendix B.

1. Retrieve a Vector containing all the Surface LabWare.
2. Create a Vector and populate it with the ExperimentPlates for the LabWareSurfaceObject's ExperimentPlate.
3. For each ExperimentPlate…
    a. Retrieve the array of Experiments and copy it to a temporary array.
    b. For each Experiment
        i. Retrieve the array of MediaComponents
        ii. For each MediaComponent…
            - Create a Step for the MediaComponent.
            - If the first letter is 'm', then place Step in Agar Vector.
            - If the first letter is 'C', then place Step in Metabolite Vector.
            - If the first letter is 'Y' or 'w', then place Step in the Yeast Vector.
4. Repeat 3 until all ExperimentPlates have been analysed.

5.2.5.2 deploySteps( )

deploySteps( ) is one of the simpler methods that is to be explicitly described in this chapter, but is included due to its importance. The method, deploySteps( ), takes all the steps and creates liquid transfers for them in the correct order, meanwhile building up jobString (JOB1.txt). The full listing of deploySteps( ) can be found in Appendix B. The following algorithm documents the process of deploySteps( ).

1. Initialise the integer 'id' to 0.
2. For each Step in the Metabolite Step Vector…
    a. Create a LiquidTransfer object, passing in 'id'.
    b. Call the method, create( ), passing in the Step details.
    c. Append the jobString with the liquid transfer (LiquidTransfer.toString( )).
    d. Increment 'id'.
3. For each Step in the Agar Step Vector…
    a. Create a LiquidTransfer object, passing in 'id'.
    b. Call the method, create( ), passing in the Step details.
    c. Append the jobString with the liquid transfer (LiquidTransfer.toString( )).
    d. Increment 'id'.
4. For each Step in the Yeast Step Vector…
    a. Create a LiquidTransfer object, passing in 'id'.
    b. Call the method, create( ), passing in the Step details.
    c. Append the jobString with the liquid transfer (LiquidTransfer.toString( )).
    d. Increment 'id'.

Upon completion of this method, the string, jobString, should contain all the data for creating the job file (JOB1.txt). The method, writeJobFile( ), takes jobString and writes it to a specified text file.

5.2.5.3 Liquid Transfer

The LiquidTransfer class contains, arguably, the most important sections of implementation, as it is responsible for the construction of the job file (JOB1.txt) that controls the Biomek system. The basis for the LiquidTransfer method, create( ), came from two sources. The first was a description of a sample liquid transfer from the Biomek User guide [4]. The second was a thorough analysis of the existing job file from the current TCL Job Compiler. The second source proved distinctly more useful in the construction of the new transfer process. This section will present the analysis of the existing job file and the resultant algorithm for the new TCL Job Compiler.

The following analysis is from a sample job file from the operation of the current TCL Job Compiler. It consists of a single liquid transfer and is described in [31].

```
Log "---- Step: Md"

Liquid_Transfer_Create
    Log "Executing LiquidTransfer: Md"

    Pipetting_Tool_Init
        Tool attach P20

        Tip_Attach
            Log "Attaching tip 4W (row 1, column 1)"
            Tip attach dispose P20 P20 P20
            Log "Attaching to P20 tip in row 1, column 1"

    Move Abs [Coord B5 A2] 46.869999

    Well_CheckVolume
        Log "Checking volumes: 19980.0, 23600.000000"

    Pipetting_Tool_Aspirate
        Log "Aspirating C00166:2E-1 from B5A2 (volume 20.0)"

        PipettingTool_Prewet

            PipettingTool_SuckAir
                Move Abs T 15.6502

            Move Abs Z 9.26959999

            PipettingTool_SetDirty
                putres system pod dirty_tip 1

            Move Rel T 17.8872
            Delay 0
            Move Abs Z 44.869999
            Move Abs T 15.6502
            Delay 0

            PipettingTool_SquirtAir
                Move Abs T 9.000000

            Delay 0
```

```
    PipettingTool_SuckAir
        Move Abs T 15.6502

    Move Abs Z 9.26959999

    PipettingTool_SetDirty
        putres system pod dirty_tip 1

    Move Rel T 14.5822
    Delay 0
    Move Rel T -1.322
    Move Abs Z 46.869999

Move Abs [Coord A4 A4] 22.17

Well_CheckVolume
    Log "Checking volumes: 20.0, 362.760010"

PipettingTool_Dispense
    PutVal tools P20 max_velocity 2
    Move Abs Z 18.5725
    Move Abs T 15.6502
    PutVal tools P20 max_velocity 25.000000
    Delay 0
    Move Abs Z 20.170000

    PipettingTool_SquirtAir
        Move Abs T 9.000000
     Delay 0

Well_AddSubstance
    Log "Performing Well_AddSubstance; Well: A4A4; C00166:2E-1"

Log "Transfer from B5A2 to A4A4 complete, volume 20.0"
```

The arrows attempt to show the scope of each of the TCL functions used in the construction of the liquid transfer. By analysing each of the functions in depth, it was possible to establish exactly how the liquid transfer is composed. From there, an algorithm for the construction of the transfer could be developed and this led to the development of the code for the liquid transfer. The algorithm for a single liquid transfer can be seen below. The full listing of create( ) (LiquidTransfer) can be found in Appendix B.

Variables Needed:

Source
Destination
Substance
Amount

Start:

1. Find the appropriate tool based on the amount to be transferred.
2. Attach the tool.
3. Attach the tip for that tool.
4. Move the tool to the Source.
5. Check that there is enough substance at the source.
6. If a prewet is needed:
    a. Suck some air in.
    b. Move tip into the substance.
    c. Suck up some substance.
    d. Move tool up a bit.
    e. Push out the substance.
    f. Push out the air.
7. If a blowout is needed:
    a. Suck some air in.
8. Suck up the appropriate amount of substance.
9. Move the tool up.
10. Move the tool the to the destination.
11. Check that there is enough room in the destination.
12. Move the tool down to an appropriate height.
13. Push the substance out.
14. Move the tool back up again.
15. Push all the air out of the tool.

Chapter 6

6 Testing

6.1 Overview of the Chapter

The aim of this chapter is to provide a description of the testing phase for the project. There are two formal documents, [33][34], which describe the processes of testing that are expected to occur during the system development and it is expected that these be used in conjunction with this chapter for a complete idea of the testing phase. During the chapter, the details of the testing phase will be documented, followed by a description of the test cases that were employed. Finally, some examples of the test cases will be presented.

6.2 What Will Be Tested?

As described in [32], there will be several layers of testing. The unit testing involves the testing of each individual component, ensuring that not only are they syntactically correct, but that they are semantically correct and they also meet their design specification. This level of testing will not be documented in this chapter.

The second layer of testing is the Module testing. This involves the testing of a group of related components in isolation. The need for this level is determined by the size of the system as the module testing can sometimes overlap with the subsystem testing. However, module testing can often take place during the implementation phase. Therefore, this chapter does not cover it.

The third layer of testing is the subsystem testing. During this level, a set of test cases is established, documenting the expected test results. This will constitute the start of the documented testing for the system. The fourth layer deals with the testing of the overall system and the fifth layer is concerned with the performance of the system within its target environment (i.e., How well does it work?).

6.3 What Will Be Tested For?

Errors, bugs and faults are present in every system and the TCL Job Compiler is not exception. The purpose of this testing phase is to uncover as many of them as possible and, at the same time, possibly suggest feasible solutions. A selection of problems, likely to be encountered in the TCL Job Compiler is as follows.

6.3.1 Typographical Errors

A missing semicolon or uneven brackets can cause significant program failures with repercussions throughout the entire system. Errors, such as these are invariably present in all code and can be detected adequately using static testing. These errors are generally found and corrected during compilation in either the implementation or testing phases.

6.3.2 General Coding/Syntactic Errors

These errors include things such as having ‘for’ loops in the wrong place or that can never terminate, or an ‘if’ statement that can never be reached. Calling methods with the wrong parameters will also cause problems. Again, these problems can normally be detected at compilation, however some will not. I.e., a ‘for’ loop that will never terminate.

6.3.3 Communication/Interfacing Errors

Communications between classes, modules, subsystems, systems, computers and users can often cause problems due to errors in the code. These can often be difficult to find and even more difficult to provide adequate solutions for. For the TCL compiler system, parsing the configuration files may cause problems, when attempting to tokenise the strings within the text files.

6.3.4 Design Flaws

Flaws that lie within the project may be the result of an inadequate or erroneous design. In some cases as this, the design phase must be revisited to improve or modify the design as is befitting.

6.4 How Will It Be Tested?

There are five types of testing to be used for the testing of the TCL Job Compiler are described below here.

6.4.1 Static Testing

This would usually primarily involve code walkthroughs. However, due to the nature of the project and the fact that there is only one developer (designer, tester, etc.), walkthroughs may not be worthwhile or even possible. Even though there is only one person involved in the project, some aspects of walkthroughs can still be observed. For example, the code can still be examined as a series of paths and the most likely paths of the system can be ascertained during this process. Static testing is usually the first method of testing to be employed during a project and can be useful in spotting syntactical and typographical errors.

6.4.2 Black box testing

Black box testing can be used as a follow up process to the static testing to further explore possible paths through the system. Black box testing can be useful when the tester knows the function of the component with respect to the operations on data inputs, but is unsure as to how the component functions on the inside. Black box testing is most suitable for testing top-level systems.

6.4.3 White box (Structural) testing

White box testing (glass box testing) is carried out when the inner workings of a component are known and the test cases can be constructed with respect to this knowledge. The tester uses knowledge about the structure of the component to derive test data and test cases. White box testing allows the tester to use many of the possible paths through a component, rather than simply the paths that are most likely to be used. It is not possible to use every possible path, but a subset of test cases can be established given the unit's function and likely problems.

6.4.4 Interface testing

Interface testing is concerned with the communications between modules, subsystems and subsystems. It is primarily concerned with the errors encountered during these periods of communication and aims to monitor and improve the way that these components co-operate.

6.4.5 Regression testing

The fixing and removing of bugs and errors can introduce new errors into the code and the design. Therefore, the testing process must be iterative and repeated until the system is adequately error-free.

6.5 Test Cases

For this section, one test case from each subsystem shall be documented as an example of the types of test cases performed. The complete list of test cases has not been included in the report for the sake of relevance. Please refer to Chapter 8 for a description of the limitations with regards to the testing phase. As in the Implementation chapter, a complete listing of the code used in this section can be found in Appendix B.

6.5.1 Substance Test Case

The objective of this test case is to test the internal structure of the Substance module, by providing a setDetails( ) method. The values are entered as expected results and the output is observed. Notice the translation of the integer to Boolean values for the prewet, tip touch, blowout and knockout variables.

SubstanceTest.java - Initial test of the Substance sub-system. Involving:

- Substance.java
- MediaComponent.java
- SubstanceTest.java

SubstanceTest.java allows the user to input some data manually (using a setDetails() method). The test harness will then present the data to the user in a formatted manner.

Below are some scripts relating to this process:

```
c:\project>java SubstanceTest

Entering Details...
Please enter the amount and all details
Enter amount 120

Enter Name: Minimal Agar

Enter Prewet Delay: 50

Enter Blow Delay: 100

Enter Dispense Delay: 120
```

```
Enter Aspirate Delay: 60

Prewet Needed?: 0

Tiptouch Needed?: 1

Blow Out Needed?: 1

Knockout Needed?: 0

Displaying Details...

Amount: 120
 Name:  Minimal Agar
 Prewet Delay:  50
 Blow Delay:    100
 Dispense Delay:        120
 Aspirate Delay:        60
 Prewet Neeeded:        false
 Tiptouch Needed:       true
 Blowout Needed:        true
 Knockout Needed:       false
```

6.5.2 Experiment Test Case

The objective of the following test case was to test the handling of the experiment file (Expt.txt). The test harness that was used functioned much in the same way as the final populateExperimentPlate( ) method in ExperimentPlate and was used as a prototype for that method. The expected output was to provide on-screen details of the experiments and their constituate parts that could be validated against the original experiment file.

Below is a selection of output referring to a single Experiment.

```
Well: 6 11
Time: 57600

Substance 1

Amount: 120
Name:                   minimal-agar
Prewet Delay:           0
Blow Delay:             0
Dispense Delay:         0
Aspirate Delay:         0
Prewet Neeeded:         true
Tiptouch Needed:        false
Blowout Needed:         true
Knockout Needed:        false

Substance 2

Amount: 20
Name:                   wild-type:E-2
Prewet Delay:           0
Blow Delay:             0
Dispense Delay:         0
Aspirate Delay:         0
Prewet Neeeded:         true
Tiptouch Needed:        false
Blowout Needed:         true
Knockout Needed:        true

Substance 3

Amount: 20
Name:                   C00166:2E-1
Prewet Delay:           0
Blow Delay:             0
Dispense Delay:         0
Aspirate Delay:         0
Prewet Neeeded:         true
Tiptouch Needed:        false
Blowout Needed:         true
Knockout Needed:        false
```

This can be compared against the following exert from the experiment file.

```
# F11,
minimal-agar          120,
wild-type:E-2          20,
C00166:2E-1          20,
time          57600,
```

It can be seen that the two extracts are displaying the same experiment.

6.5.3 Configurator Test Case

The following is a basic test case that aims to test the internal structure of the TipList module. The values of the Tips are entered in a test harness and the toString( ) for the TipList was then outputted to the screen. This test harness also aimed to test the compatibility of the toString( ) method for use in the configuration file (Notice the use of 'CreateRes' Statements.)

Below is a selection of output referring to a single TipList.

```
CreateRes tips P20
PutVal tips P20 maxvel 1.0
PutVal tips P20 minvol 0.25
PutVal tips P20 slowstep 0.0
PutVal tips P20 smargin 0.63
PutVal tips P20 length 38.1
PutVal tips P20 shldiam 6.86
PutVal tips P20 shoulder 9.78
PutVal tips P20 sensing false
PutVal tips P20 maxvol 23.0
```

6.5.4 Surface Test Case

The following test case tests the correct operation of the Surface subsystem and part of the Configuration subsystem. The surface is propagated using the surface file (Surface.txt), which is passed in using a test harness.

Below is a selection of output referring to the Surface.

```
A1 VICTOR -51.8 33.9 78.1 1
A21 P20L
A22 P200L
A24 Gripper
A3 P250
A4 96-well flat
A5 96-well flat
A6 quarter vertical
B2 P20
B3 96-well flat
B4 96-well flat
B5 quarter vertical
B6 half single
A6A1 YBR166C:E-2 20000.0
A6A2 C00078:2E-1 20000.0
A6A3 YDR035W:E-2 20000.0
A6A4 C00079:2E-1 20000.0
A6A5 YDR354W:E-2 20000.0
```

```
A6A6 wild-type:E-2 20000.0
A6A7 C00108:2E-1 20000.0
A6A8 C00108:2E-1 20000.0
B5A1 C00108:2E-1 20000.0
B5A2 C00166:2E-1 20000.0
B5A3 YDR007W:E-2 20000.0
B6A1 minimal-agar 120000.0

Log "-----------------------------------------------------
Log "Setting up surface..."
Log "-----------------------------------------------------
Log "Creating VICTOR."
Log "Line: A2 P20L"
Log "Creating surface object P20L"
PutVal loc A2 thang_type 2
PutVal loc A2 nLayer 0
Log "Line: A2 P200L"
Log "Creating surface object P200L"
PutVal loc A2 thang_type 2
PutVal loc A2 nLayer 0
Log "Line: A2 Gripper"
Log "Creating surface object Gripper"
PutVal loc A2 thang_type 2
PutVal loc A2 nLayer 0
Log "Line: A3 P250"
Log "Creating surface object P250"
PutVal loc A3 lid_status 0
PutVal loc A3 sl_dst_wedge 0
PutVal loc A3 last_tip_col 0
PutVal loc A3 sl_dst_stack 0
PutVal loc A3 sl_src_wedge 0
PutVal loc A3 sl_dst_shelf 0
PutVal loc A3 sl_src_stack 0
PutVal loc A3 group_tips_used 0
```

The test harness calls the toString( ) method for the Surface module. Notice how the toString( ) method has been configured to be compatible for use in the configuration file.

6.5.5 Builder Test Case

The following test case for the Builder subsystem was expanded to provide the point of execution for the entire system. In this case, the communication of all the subsystems is tested with the expected output as the job file (JOB1.txt) being identical (to all intent and purposes) to the job file constructed with the use of the current TCL Job Compiler. Below is a selection of output from the job file from the new system.

```
Log "Executing LiquidTransfer: 0"
Tool attach P20L
Log "Attaching tip P20 with some grid reference (not sure yet)"
Tip attach dispose P20 P20 P20L
Log "Attaching to P20 with some grid reference (not sure yet)"
Move Abs [Coord B5 A2] 46.872
Log "Checking volumes: 19960.0 23600.0"
Log "Aspirating C00166:2E-1 from B5A2 (volume 20.0)"
Move Abs T 15.6502
Move Abs Z 9.2696
putres system pod dirty_tip 1
Move Rel T 17.8872
Delay 0
Move Abs Z 44.87
Move Abs T 15.6502
Delay 0
Move Abs T 9.0
Delay 0
Move Abs T 15.6502
Move Abs Z 9.2696
putres system pod dirty_tip 1
Move Rel T 14.5822
Delay 0
Move Rel T -1.322
Move Abs Z 46.872
Move Abs [Coord A4 A4] 22.167
Log "Checking volumes: 20.0 362.76"
PutVal tools P20L max_velocity 2
Move Abs Z 18.5725
Move Abs T 15.6502
PutVal tools P20L max_velocity 25.0
Delay 0
Move Abs Z 20.17
Move Abs T 9.0
Delay 0
Log "Transfer from B5 A2 to A4 A4, C00166:2E-1, Volume: 20.0
COMPLETE."
```

This must be compared to the same liquid transfer from the job file created by the current TCL Job Compiler (see below).

```
Log "---- Step: Md"
Log "Executing LiquidTransfer: Md"
Tool attach P20
Log "Attaching tip 4W (row 1, column 1)"
Tip attach dispose P20 P20 P20
Log "Attaching to P20 tip in row 1, column 1"
Move Abs [Coord B5 A2] 46.869999
Log "Checking volumes: 19980.0, 23600.000000"
Log "Aspirating C00166:2E-1 from B5A2 (volume 20.0)"
Move Abs T 15.6502
Move Abs Z 9.26959999
putres system pod dirty_tip 1
Move Rel T 17.8872
Delay 0
Move Abs Z 44.869999
Move Abs T 15.6502
Delay 0
Move Abs T 9.000000
Delay 0
Move Abs T 15.6502
Move Abs Z 9.26959999
putres system pod dirty_tip 1
Move Rel T 14.5822
Delay 0
Move Rel T -1.322
Move Abs Z 46.869999
Move Abs [Coord A4 A4] 22.17
Log "Checking volumes: 20.0, 362.760010"
PutVal tools P20 max_velocity 2
Move Abs Z 18.5725
Move Abs T 15.6502
PutVal tools P20 max_velocity 25.000000
Delay 0
Move Abs Z 20.170000
Move Abs T 9.000000
Delay 0
Log "Performing Well_AddSubstance; Well: A4A4; Substance: C00166:2E-1"
Log "Transfer from B5A2 to A4A4 complete, transfer list 9V, volume
20.0"
```

It can be seen that, although there are differences between the two versions, these differences are merely cosmetic (i.e., the Log entries). The numerical values for each movement of the Biomek are identical, which indicates that the job file from the new TCL Job Compiler could be successfully used in place of the current system. Please refer to chapter 8 for limitations of the system.

Chapter 7

7 Limitations and Evaluation

7.1 Introduction

The purpose of this chapter is to provide an evaluation for the SEM49060 project. It is necessary to once again examine the requirements, this time in parallel with the recognised achievements of the project. Throughout the course of the project, various limitations have been identified but not yet explored within this document. It is the aim of this chapter to identify and describe the limitations encountered during the project, together with a description of possible alternative approaches. This chapter will also attempt to provide some of the abandoned approaches taken during the project. Having described the limitations for the project, a more informed approach to the evaluation can be taken.

During this chapter, the limitations of each phase of the project will be discussed. It is hoped that the discussion of the limitations of the project will also highlight some of the author's regrets regarding each phase of the project. This will be followed by a discussion and evaluation of the project as a whole. Finally, there will be a section describing a look to the future for the TCL Job Compiler.

7.2 Requirements Limitations

7.2.1 Vagueness of Requirements

One of the initial problems with the requirements phase of the project was the process of requirement identification. Through initial meetings with supervisors, it was unclear as to the exact specifications of the proposed system. After a time, the project requirement emerged as "simply" to reengineer the current TCL Job Compiler. This was found to be particularly disadvantageous for the requirements phase for the following reason. Whereas in a more conventional project, the requirements would be constructed through customer-liaising and technical knowledge, the requirements for the proposed system needed to be developed from the requirements of the current system. The consequence was that there needed to be an analysis of the current system before any requirements for the proposed system could be developed.

Given that the current system needed to be analysed and, due to the nature of the current system (7.2.2), the development of the requirements was delayed. This resulted in a knock-on effect throughout the phases and was regrettably unavoidable. Having analysed the current TCL Job Compiler and encountered the problems documented below, it was found that nothing more than a requirements definition could be adequately composed [30]. It would have been preferable to have had both a requirements definition and specification, but this was not possible given the state of the current system.

7.2.2 The Current TCL Job Compiler

Given the initial requirement of reengineering the current TCL Job Compiler, it seemed worthwhile to conduct an analysis of current system in order to extract the requirements for the new system. Although the analysis of the current system undoubtedly contributed in part towards the success of the project in other areas, it was found to be of limited use for the construction of requirements for the following reasons.

Firstly, it was found that the analysis of the current system could not be completed. This was due to the considerable complexity and readability of the current system. The current TCL Job Compiler consists of approximately 4,000 lines of TCL (Tool Command

Language); a language unfamiliar to the author at the time. There were also many more lines of configuration data held in various files, which had to be considered. Comments were extremely scarce. The only areas that contained readable comments were the sections that had been maintained or revised[xiii]. There were no supporting documents, formal or otherwise, regarding the operation of the TCL Job Compiler. All the points of functionality had to be extracted and analysed manually in order to construct an analysis of the system. The formal document offered in this project [31] is by no means complete. However, even this incomplete stage of analysis took many hours to achieve, due to the lack of structure present in the current system.

It was deemed that an exhaustive analysis of the current system would take too long and be of limited use. Therefore, the analysis was halted and an alternative method for the development of requirements was sought. These included building on the analysis of the current system and using input/output analysis, visualisation and conceptualisation, details of which can be found in 4.2 (Approach to Design).

7.2.3 Current System Knowledge

One of the problems with analysing the current TCL Job Compiler was that there was insufficient knowledge of the system. Not only were there were no supporting documents available for the system, there were very few comments describing the functionality of the system. To make matters worse, the author of the system had long since left the department. The result was that there was no one with a complete knowledge of the system. This made the job of analysing the current system extremely difficult. Ken Whelan was responsible for maintaining the system and, as such, held knowledge on selected parts. However for other parts, there was no point of contact for support. Consequently, the majority of the system was analysed with no support of any kind.

This further impacted the speed of which the analysis could be conducted and the commencements of the remaining project phases. It was this lack of support that contributed to the incompleteness of [31].

7.2.4 System Accountability

One of the problems encountered during the construction of the requirements, was the lack of accountability in the current TCL Job Compiler. It was difficult to know whether the requirements were correct or complete when there were substantial parts of the current system that were still not understood. It was very hard to understand the function of certain parts of the system. This was partly due to the lack of comments, but also due to other readability factors such as:

- Unhelpful naming,
- Redundant code mixed with working code,
- The use of multiple, complicated, similarly-named variables,
- Complicated data flow,
- Apparent duplication of functionality.

As a result of this, there is the danger that a potentially important design issue was overlooked, simply because it could not be 'deciphered'.

[xiii] Maintainence carried out by Ken Whelan.

7.3 Project Planning Limitations

7.3.1 System Knowledge

As mentioned in 7.2, the inability to sufficiently analyse the current TCL Job Compiler resulted in a difficulty of producing a set of adequate requirements. This had serious implications for the planning of the project, as it was unclear exactly how much effort would be needed in each phase, in particular the implementation phase. In fact, it only became clear how much implementation would be needed when it was 'too late' to alter the project requirements. The promise of delivering the system became considerably more daunting when it was uncovered that it should generate around 30,000 lines of robot instructions. With hindsight, a more achievable requirement would have been attempted after a more vigorous inspection of the proposal.

7.3.2 External Time Planning

During the planning of the project, the inclusion and budgeting of other commitments was largely ignored. Although there was time allowed for Christmas and for the January exams, other activities, most notably the major survey paper (due in for late January), were not accounted for in the original time plan. It was generally considered that any other activity could be 'fit' around the project commitments. In most cases this proved a successful strategy. In the case of the survey paper, it did not.

The survey paper, due in after the Christmas break, enveloped much of the time before and after Christmas. This time had originally been scheduled for the development of the design and testing documents. However, the time was not available for these and so the design, testing and, by implication, the implementation phases were pushed back several weeks. This was regrettable but unavoidable. The project phases were immediately re-scheduled and the result was a higher workload and a reduced testing period (refer to 7.5).

7.3.3 Development Model

As described in 3.3.1 (Project Planning, Development Model), the selection of an appropriate development model was a problematic task. It was originally hoped that the reengineering of the TCL Job Compiler could be conducted using the Unified Process as a development model [15]. However, little evidence could be found that the Unified Process could be successfully employed for a reengineering project. Moreover the Unified Process works primarily from use cases. For this project, there seemed to be an insufficient amount of use cases due to the low number of actors (refer to [15] for details of use cases and actors).

It would be difficult to class the project into a specific development model. As detailed in 3.3.1, it was intended that the project loosely conform to Boehm's Spiral development model. Looking back, it seems that the project took a different approach to the one originally intended.

The development of the project certainly contains elements of Boehm's Spiral model, but there are also significant similarities to the processes of XP (Extreme Programming) and Agile methods in the way that each feature was designed and added without specific requirements. One piece of evidence to support this was the way the formal design document [32] was written retrospectively as each feature was developed.

7.4 Design Limitations

7.4.1 Size Estimation

One of the biggest limitations of the design phase was the estimation of the size of the system. It was anticipated that the formal design document [32] would be reasonably large but it was not expected to be as sizeable as it turned out to be. The number of modules required for the implementation of the TCL Job Compiler was far greater than expected. It is likely that there are considerable opportunities for optimising the finished system. However, time constraints have shortened the amount of effort that could be spent on this process. The erroneous estimations for the design and implementation phases resulted in the phases running over the scheduled time, further infringing on the time set aside for testing.

7.4.2 The Surface Subsystem

One of biggest surprises of the project was the need for the Surface subsystem. During the conceptualisation of the bulk of the design, the need for the Surface subsystem was overlooked. This was partly due to the reasons described in 7.2.2, 3 and 4. When it came to creating the configuration file (JOB0.txt), it became necessary to place the various experiment items (devices, tools, labware, etc.) into the Biomek workstation. The locations of these items were held in a file called Surface.txt (See 5.2.4 for an example of Surface.txt).

The manipulation of the surface file could not be found anywhere in the current TCL Job Compiler and therefore, it was assumed that the file would not cause too much difficulty and would not be complicated to implement. However, when it came to the implementation of the configuration file, it was found that a separate subsystem would be needed in order to place the items in the correct locations *in silico*. It was necessary to do this for two reasons. Firstly, the surface locations were needed in order to successfully create the configuration file. Secondly, the system needed to know the locations of the experiment materials during the creation of the liquid transfers.

Even after establishing the need for the Surface subsystem, it proved very difficult to find evidence of surface manipulation in the current system.

7.4.3 Locus of Control

One of the limitations of the design was that a control module was not specifically designed to control the TCL Job Compiler. In retrospect, it would have been beneficial to create a purpose-built module to control the operation of the system. Currently, the control of operation lies with the test program, BigDemo.java (see Appendix B), which has been adequately implemented to run the system. However, as the name suggests, it is merely a test harness.

7.5 Implementation Limitations

7.5.1 External Alterations

Given the initial requirement that the new TCL Job Compiler should offer exactly the same functionality as the current version, it was particularly unfortunate to have to alter any external entities. However, this was necessary in the case of the experiment file. Section 5.2.2 documents the need to change the experiment file (Expt.txt) in order to include a delimiter at the end of each line. It is unfortunate that the system has imposed a change externally, but there appeared to be no other way of delimiting the experiment file correctly.

7.5.2 Log Implementation

The Log subsystem is arguably the simplest of all the subsystems within the TCL Job Compiler. Implementing the Log subsystem was not considered as important as some other areas of the system, given its simplicity. For this reason, the Log subsystem has yet to be included in the system. It is important to note that the Log subsystem is fully functioning and ready to be integrated into the system.

The Log subsystem was omitted for two reasons. Firstly, the construction of a log should be a thoughtful and important process. The log is designed to give the user feedback on program activity, possibly notifying of any failures. It is important that the log should contain clear, concise and readable information. It is the feeling of the author that there was not sufficient time to provide a high quality log. It was therefore considered better practice to omit the log and let someone else give it the proper time it deserves. Secondly, for the purposes of this project, the job file (JOB1.txt) gave sufficient feedback as to the state of the project. For example, during a program crash, it would be clear exactly where the program failed by analysing the job file, seeing the last item recorded.

7.5.3 Front End

It is regrettable that an adequate front-end to the TCL Job Compiler could not be implemented. It was secretly hoped that this would be possible but due to the time constraints, it was not. A GUI (Graphical User Interface) would not have brought anything other than aesthetic benefits to the system, as there is very little user interaction. This will doubtlessly be a development for the future, where the TCL Job Compiler will hopefully be expanded to include other functions and options(see 7.9).

7.6 Testing Limitations

7.6.1 Time Constraints

As mentioned previously in the preceding descriptions of phase limitations, the various time constraints played a part in the shaping of the development of the TCL Job Compiler, as would be expected. Unfortunately, it was the testing phase that suffered the most through changes and revision of the schedule. There were two main reasons for this.

Firstly, the completion of the system was of paramount importance to the project and therefore, it took priority. The level of importance on completing the finished system is a result of the level of credit given for 'Achievement' in the marking of the SEM490 project.

It was decided that the design and implementation phases were the most important to complete with whatever time left available for testing. This approach was further reinforced by the relatively low credit available for the testing phase.

Upon completion of the design and implementation phases, it transpired that there was ample time to conduct an adequate testing process, given the credit available for the phase.

7.6.2 Test Cases

It is important to note that although there is less credit awarded, the testing phase remains a very important part of the development of the TCL Job Compiler. Given the importance of the experiments and the costs of the equipment, it is important to have a fully tested implementation of the TCL Job Compiler. However, due to time constraints and for the reasons mentioned above, the TCL Job Compiler was not tested to the fullest extent possible. Moreover, one of the limitations of the testing phase is the thoroughness of the test cases. There simply wasn't enough time to complete the necessary number of test cases needed to constitute a release stage for the system.

7.7 System Limitations

7.7.1 Pipette Changes

All of the experiments within the Biomek system are created using minimal agar as growth media. Minimal agar is a jelly at room temperature. This creates the problem of transferring the minimal agar into each well. To achieve this, the agar is heated until it turns to a liquid and can be therefore transferred as such. The problem is that, having heated the agar to liquid form, it will slowly cool to jelly and block the tip of the pipette. This will impede any further liquid transfers. The current TCL Job Compiler solves this problem by explicitly changing the tip every number of transfers. This prevents the tips from blocking.

Currently, the new TCL Job Compiler has no function for preventing a tip blockage for the following reason. After examining the Biomek documentation [4], it states that the workstation detects whether a tip is blocked and automatically replaces the blocked tip with a clean one. It will then resume normal operation. Presently, it is unconfirmed as to whether this is the case. However, if not, then the original tip-blocking method can be applied to the new system easily enough.

7.7.2 Completeness of the System

Apart from the limitations stated above (pipette changes and testing), the TCL Job Compiler is ready to be integrated into the existing system. The configuration file (JOB0.txt) is grammatically sound and the author is confident that it can be used without further modification (maintenance when necessary). The job file (JOB1.txt) is more difficult to validate. The size of the job file (30,000 lines) makes it very difficult to say whether it is correct or not. It is certain that the system should undergo further testing before the job file is used with the Biomek system.

7.8 System Evaluation

7.8.1 Metrics

It was originally intended that metrics would be used to evaluate the final TCL Job Compiler. Metrics can be used to measure various attributes of a software system in order to achieve an idea of overall quality. It is particularly applicable for a project that is concerned with the reengineering of an older system. Presumably, a requirement for a reengineering project is the improvement of quality. Therefore, it would be prudent to try and prove that an improvement of quality has taken place.

It was not possible to employ metrics during this project for two reasons. Firstly, time constraints resulted in there being insufficient time to conduct an adequate suite of metrics. The idea of metrics[xiv] was presented to the year through a semester one module. Unfortunately, the time plan for the project had already been established by the time metrics had been introduced to the year. Although the time plan could have been altered to include metrics, it was decided that there were other more important areas to focus on, particularly in light of the increased time constraint (refer to 7.3.2).

Secondly, the comparison of metrics relies on having one value to compare to each other. Although the author acknowledges that it would have been possible to conduct metrics on the current TCL Job Compiler, it would have been extremely difficult to do so, particularly for some of the more complex metrics, such as LCOM (Lack of Cohesion Of Methods). Therefore, it would not be possible to compare the systems using metrics in any case.

7.8.2 System Discussion

The previous sections have focused on the limitations on the reengineering of the TCL Job Compiler. It is therefore likely that the previous sections may have painted the project in an unsuccessful light. However, this is not the case.

As a whole the project was a success, given the ambitious requirement of reengineering the current TCL Job Compiler. Please refer to 7.7.2 above for details of the project completeness. The system resulted in being larger than had originally been thought. It was feared that it would not be possible to produce a working version of the TCL Job Compiler. At one point of the project, it seemed only probable to have the functionality to produce the configuration file for the final product.

However, with much hard work, it was possible to implement the fully working system, quite an accomplishment when considering the lack of support and technical knowledge for the original system. It is the opinion of the author that many of the shortcomings of the project (those already identified) are of little significance when considering the overall achievement of the system (i.e., developing a fully working system).

[xiv] Metrics taught as part of SEM3510 (Semester 1) by eds.

7.9 The Future for the TCL Job Compiler

It is hoped that the reader understands the difficulty associated with the reengineering of the TCL Job Compiler. In reading this chapter, as well as the rest of the report, it should be clear as to the downfalls of the original system. The fact that no one fully understands the current system is a testament to its user-unfriendly complexity. The state of the current system destined it to lifecycle of maintenance by code hacking.

In comparison the new system endevours to provide a more maintainable environment. It does this by offering improved:

- Readability
- Portability
- Versatility
- and as a consequence, Maintainability.

These improved attributes allow the TCL Job Compiler to be more easily maintained, upgraded and appended. As mentioned in 2.7, the Biomek system can operate up to 8 pipettes at a time. This introduces the possibility of increasing productivity by up to 8 times. Although this has not been implemented in the existing new system, it should not represent too much of a problem if someone should decide to include this in the future.

There are other possibilities for improving the productivity of the TCL Job Compiler. Other heuristic approaches can be researched and explored and then implemented with greater ease thanks to the new TCL Job Compiler. These kinds of alterations would be unheard of when using the current TCL Job Compiler.

Chapter 8

References

[1] *ARNOLD 93* – dictionary definition (translated from French)

[2] Attwood T.K, Parry-Smith D.J – Introduction to Bioinformatics, *Cell and Molecular Biology in Action Series, Pearson Education Limited,* 1999.

[3] Bartow, G. Custom LIMS Help Maximize Laboratory Productivity. *Scientific Computing and Instrumentation Online* (Oct 1998) Available www.scamag.com.

[4] BioScript Pro Programmer's Guide*, Document 609848-AA.*

[5] Boeijen, F.P.M. The Total Qualified Laboratory. *Scientific Computing and Instrumentation Online* (Nov 1999) Available www.scamag.com.

[6] Botstein D., Chervitz S.A. & Cherry J.M. Genetics: Yeast as a model organism. *Science*, 277, (1997).

[7] Bryant C.H., Muggleton S.H., Oliver S.G., Kell D.B., Reiser P., King R.D. – Combining Inductive Logic Programming, Active Learning and Robotics to Discover the Function of Genes. *Linkoping University Electronic Press, Linkoping, Sweden.* December 2001.

[8] Coles, S. An XML Interface to LIMS. *Scientific Computing and Instrumentation Online* (Nov 1999) Available www.scamag.com.

[9] Dugdale, D. Web Services Come of Age. *DevX Online Articles* (Dec 2000). Available www.devx.com.

[10] I.J. Farkas, H. Jeong, T. Vicsek, A.-L. Barabási3 & Z.N. Oltvai1 - *The topology of the transcription regulatory network in the yeast, S. cerevisiae* - http://xxx.lanl.gov/ftp/cond-mat/papers/0205/0205181.pdf

[11] Ganjei, J.K & Bergen, A.W. LIMS Customization for Biomedical Research. *Scientific Computing and Instrumentation Online* (May 2001) Available www.scamag.com.

[12] A. Goffeau et al – Life with 6000 Genes. *From Science Magazine: Enhanced Perspectives.*

[13] Heiter P., Boguski M. - Functional Genomics: It's all how you read it., *Viewpoints, Science, Vol 278*, pp601, 24th October 1997.

[14] Hunkapiller, T & Hood, L. LIMS and the Human Genome Project. *Bio/Technology Vol 9 p1344-1345* (Dec 1991).

[15] Hunt J. – The Unified Process for Practitioners, Object Oriented Design, UML and Java. *Springer – Practitioner Series,* 2000.

[16] IMS World Review 2001, OECD **http://www.oecd.org/std/nahome.htm**, downloaded August 2001, Office for National Statistics.

[17] Jones, J.H. Extending Laboratory Data Management with Web Services. *Scientific Computing and Instrumentation Online* (Nov 2001) Available www.scamag.com.

[18] Joyce, J.R. Searching for the Service in ASP. *Scientific Computing and Instrumentation Online* (Nov 2001) Available www.scamag.com.

[19] Mcdowall, R.D. A Matrix for the Development of a Strategic Laboratory Information Management System. *Analytical Chemistry Vol 69 No.20 p896A – 901A* (Oct 1993).

[20] Miller, S. A Practical Approach to LIMS Selection. *Scientific Computing and Instrumentation Online* (May 2001) Available www.scamag.com.

[21] Muggleton S.H., Bryant C.H. – Theory Completion using Inverse Entailment, *In Proc. of the 10th International Workshop on Inductive Logic Programming (ILP-00),* Berlin, 2000. Springer-Verlag.

[22] O'Reilly – Wireless Java Opens the Door for New Handheld Applications: O'Reilly's "Learning Wireless Java" Brings Developers Up to Speed*, O'Reilly Press Room*, January 17th 2002, available at: http://press.oreilly.com/wirelessjava.html

[23] Redman, J. The Laboratory Equipment Control Interface Specification. *Scientific Computing and Instrumentation Online* (Nov 1999) Available www.scamag.com.

[24] Reiser P.G.K., King R.D., Kell D.B., Muggleton S.H., Bryant C.H., Oliver S.G. – Developing a Logical Model of Yeast Metabolism. *Linkoping University Electronic Press, Linkoping, Sweden.* August 1998.

[25] Rofrano J.J. – Java Portability by Design, *Dr. Dobb's Journal*, June 1999 – available at: http://www.ddj.com/documents/s=902/ddj9906c/9906c.htm

[26] ScreamingMedia – Chordiant Software Inc - Sails Through Java Portability Tests, *Market News Publishing*, March 28th 2002.

[27] Smith, K. Data Transfer Using XML. *Scientific Computing and Instrumentation Online* (Nov 2000) Available www.scamag.com.

[28] Sommerville I. – Software Engineering 5th Edition. *Addison-Wesley*. 1995

[29] Staab, T.A. Next Generation LIMS. *Scientific Computing and Instrumentation Online* (Nov 1999) Available www.scamag.com.

[30] Tagger B. – Requirements Definition, *Available in Appendix A*, 25/11/02

[31] Tagger B. – Analysis of the Current TCL Job Compiler, *Available in Appendix A*, 1/12/02

[32] Tagger B. – System Design Specification, *Available in Appendix A*, 27/02/03

[33] Tagger B. – Basic Test Plan, *Available in Appendix A*, 14/01/03

[34] Tagger B. – Subsystem Test Specification, *Available in Appendix A*, 28/02/03

[35] Tamaddoni A., Muggleton S.H. – Closed Loop Machine Learning: Complexity of ASE-Progol. Jan 2002

[36] *The Association of the British Pharmaceutical Industry,* www.abpi.org.uk/statistics/section

[37] Thein S.L. – Thalassaemia Prototype of a Single Gene Disorder with Multiple Phenotypes. *Department of Haematological Medicine* - www.ish2002.org/main1/pdf/822.pdf

[38] Webber, J. A survey of LIMS Satisfaction. *Scientific Computing and Instrumentation Online* (Nov 2000) Available www.scamag.com.

[39] Website - http://www.scms.rgu.ac.uk/staff/chb/closedloop.html

[40] Website - http://tcl.sourceforge.net/faqs/tcl/part1.html

[41] Wired - Microsoft Says Java Is Best on Windows, June 17th 1997, *available at:* http://www.wired.com/news/technology/0,1282,4491,00.html

[42] XREFdb – *Data held in from the National Center for Biotechnology. Information at:* http://www.ncbi.nlm.nih.gov/Bassett/cerevisiaie/index.html.

[43] Fowler, M. – Refactoring: Improving the Design of Existing Code. *Booch, Jacobson and Rumbaugh Object Technology Series,* 2000.

Appendix

Part A: Requirements Definition
Part B: Analysis of the Current TCL Job Compiler
Part C: System Design Specification
Part D: Basic Test Plan
Part E: Subsystem Test Specification

REQUIREMENTS DEFINITION

Author: Ben Tagger (bnt8)
Date: 25/11/2002
Version: 2.0
Status: Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of the Document

This document aims to provide a requirements definition of the TCL compiler system. This software requirements definition will provide an abstract description of the services that the system should provide, taking into account the constraints under which the system must operate.

1.2 Scope

This document should only describe the external behaviour of the TCL compiler and should not be concerned with design characteristics or any of the internal complexities. Consequently, the requirements should not be written with the use of an implementation model.

1.3 Objectives

This document should be written in such a way that it is understandable for a customer with no specialised knowledge of the system/industry. It is therefore preferable for the requirements definition to be expressed in natural language and intuitive diagrams. From this document, the reader should be able to establish the primary functions of the TCL compiler and be able to identify some of the constraints of the proposed system. One of the objectives of this document is to achieve a level of clarity for the reader to completely understand the requirements of this TCL compiler. It is also important to achieve a distinct difference between functional and non-functional requirements.

2. BASIC REQUIREMENTS

The most basic and all-encompassing requirement for this project is that the new system completely mimics the old system. That it, there should be no functional difference, externally. This section will describe some of the different types of requirement and then section 3 will describe the requirements in more detail.

2.1 Functional Requirements

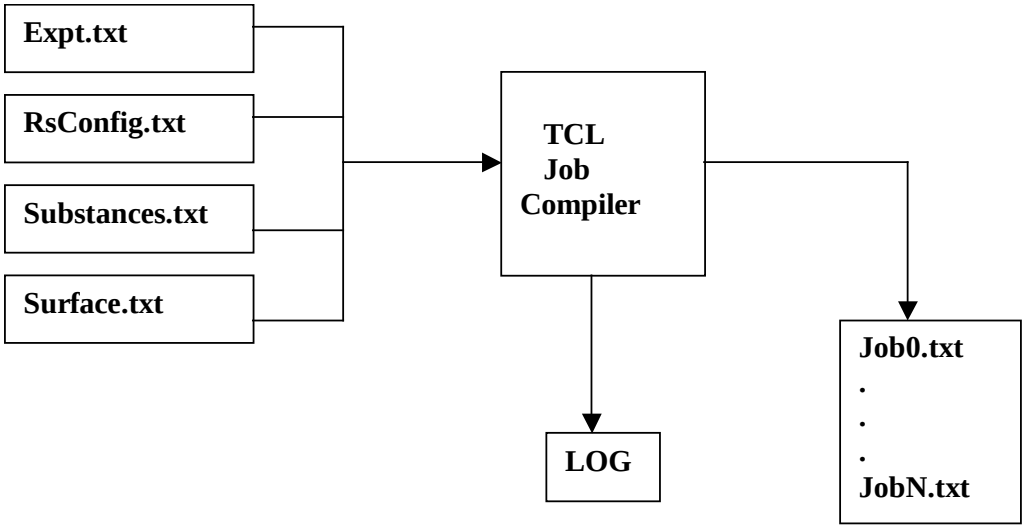
The functional requirements describe the services that the system will provide. It should describe how the system should react under certain data inputs and in certain situations. It will also describe the output that is expected from the system.

2.2 Non-functional Requirements

The non-functional requirements state the constraints that are placed upon the system. These can include timing constraints, constraints on the development process and implications of possible safety-related hazards. It will also include some of the standards that are applicable to the project.

3. FUNCTIONAL REQUIREMENTS

This section will describe some of the functional requirements. For a description of what is meant by functional requirements, please refer to 2.1. Below is an overview of the inputs and outputs that are relevant to the TCL compiler.



3.1 Input Requirements

Below is a description of the inputs that will be required to be handled by the TCL compiler system. These files will be made available by the ASE-Progol system and will placed in specific directories to be accessed by the TCL compiler. The four input files that can be seen above are all that is required for the operation of the TCL compiler. These four files must be implemented and integrated into the new TCL compiler system. There must be some process of "reading in" the information contained within these configuration files and then this data must be handled in some appropriate way.

3.1.1 Experiments

The experiments constructed during the ASE-Progol phase of operation are placed in the file, Expt.txt. This file contains the details of all the experiments that are to be prepared by the Biomek station. It contains information of every experiment that is to be conducted in every single well. An example of an experiment, as documented within Expt.txt, can be seen below:

```
# E11
minimal-agar          120
wild-type:E-2          20
C00166:2E-1           20
time          57600
```

The above data set would constitute a single experiment. ‘E11’ constitutes the well in which the experiment is to be performed. Then, there is a list of the substances and amounts that comprises the experiment. Lastly, the time dictates the length of time taken for the experiments.

3.1.2 Tools

The configurations of all the tools that are to be used within the Biomek station are contained within the RsConfig.txt file. The contents of this file will need to be passed on to the Biomek workstation by some means.

3.1.3 Substances

A description of all the possible substances that can be used in the experiments are supplied within the Substances.txt file. The substances named within this file need to match up with the substances documented for the experiments in Expt.txt. If they do not, then the attributes of some substances in the Expt.txt, will be missing. An example of the Substances file is shown below:

```
minimal-agar 0 0 0 0 1 0 1 0
YDR007W:E-2 0 0 0 0 1 0 1 1
C00166:2E-1 0 0 0 0 1 0 1 0
C00108:2E-1 0 0 0 0 1 0 1 0
```

3.1.4 Surfaces

The configuration of the work surface of the Biomek workstation is contained within the Surfaces.txt file. This file is for use within the TCL compiler in order to process the experimentation steps correctly. For example, the following line:

```
B3 96-well flat
```

indicates that in the B3 section of the work surface, there is a flat-based plate with the standard 96 wells (8x12).

3.2 Output Requirements

Below is a description of the files that are to be outputted from the TCL compiler system.

3.2.1 Job files

The job files are created within the TCL compiler and passed to the Biomek processing system. The creation of these job files constitutes the main function of the TCL compiler. These job files consist of lines of instructions written in a scripting TCL language, called BioScript PRO. The way these files are created and the order in which the commands are written will have a bearing effect on the correctness of operation as well as the speed and efficiency of operation of the Biomek workstation. The job files are created as a set of files, the first being called JOB0.txt and each subsequent files being JOB1.txt, JOB2.txt and so on. JOB0.txt is concerned with setting up the Biomek workstation, containing all the parameters and configuration material needed. Subsequent job files then contain the commands for the operation of the robot scientist. An example of a command contained within the job files is shown below:

```
Log "Aspirating C00166:2E-1 from B5A2 (volume 20.0)"
Move Abs T 15.6502
Move Abs Z 9.26959999
putres system pod dirty_tip 1
Move Rel T 14.5822
Delay 0
Move Rel T -1.322
Move Abs Z 46.869999
Move Abs [Coord A4 B4] 22.17
```

This selection of files describes an aspiration process. The details of this process will not be explained explicitly here (largely as they are not themselves fully understood), but it is clear that coordinates and measurements are placed into the system to control various parts of the robot. The job files are constructed using these basic commands (Move, Delay, Put, etc).

3.2.2 Log

A log will be kept as part of the output process. To a degree, the job files can be used as a log to the extent that they record the output of the TCL compiler. The log file will contain purpose-built log information, but will be most useful when used in conjunction with the job files.

3.3 Error handling

The system should provide some error handling. There is no sign of any kind of error handling in the present TCL compiler. The system should, at the very least, be capable of detecting that an error has occurred, inform the user, and prevent the system from performing any further actions that could constitute a danger. Although, the TCL compiler should not be described as a safety-critical (or even a safety-related) system, it is important to note that there are some safety implications to be taken into consideration. The control of any kind of machine should be considered from a safety-oriented point of view and the TCL compiler is no exception. For example, the robot scientist has the capability to pour a liquid (possibly hazardous) onto parts of itself. This has an obvious economic implication and is clearly undesirable. Adequate error handling can have a direct effect on these kinds of scenarios.

4. NON-FUNCTIONAL REQUIREMENTS

The following section describes the non-functional requirements identified to date for the TCL compiler system. For a description of what is covered by the term non-functional requirements, please see 2.2.

4.1 Efficiency

The system will need to plan the sets of experiments in such a way that allows the Biomek machine to work in a sufficiently efficient manner, which does not endanger either itself, or any potential users. The Biomek robot works with hazardous materials and as such, must be provided with adequate protection.

4.2 Versatility

Versatility involves providing a system that can operate on more than one level. The solution it provides should be able to be applied to more than one domain (the domain it was primarily designed for). The more versatile a system can be made, the more useful it can be for future use.

4.3 Portability

Portability is concerned with providing a system that can be used in a different domain, without the requirement of too much reworking. The system should be able to be moved between systems easily and with as little low-level work as possible.

4.4 Job files

There are some constraints on the way the job files are written. Firstly, the job files must be written using BioScript PRO (TCL). Although there are no current time constraints, it is beneficial to have the job files completed in the fastest time possible. The current system completes in about five minutes, so it shouldn't be too difficult to undercut that time. However, it is the experimentation process that takes the greatest amount of time. This can take times up to two days, so any time that can be saved here is extremely desirable. The job files should be engineered so that the experiments can be completed in the fastest time possible.

ANALYSIS OF THE CURRENT TCL COMPILER

Author: Ben Tagger (bnt8)
Date: 01/12/2002
Version: 3.0
Status: Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of the Document

This document aims to provide an analysis of the current TCL Job Compiler. This document endevours to explain and describe some of the processes encountered in the current system, providing some flow control diagrams where appropriate. It was hoped that this analysis would provide some useful information for the construction of a new system and potentially form some basis for the design of the new system.

1.2 Scope

This document will only attend to specific points of operation of the current TCL Job Compiler. It will only be concerned with certain areas of useful functionality and is not intended to be complete.

1.3 Objectives

This document aims to provide assistance in the conception, design and implementation of the new TCL Job Compiler. The document will probably only be useful for those people who have worked, or are familiar with, the current TCL Job Compiler. Upon reading this document, the reader should be aware of some of the functionality and workings of the current TCL Job Compiler and also should be aware, first hand, to some of its shortcomings.

1.4 Constraints of the Document

Upon reading this document, it should be obvious that it is not a complete analysis and should not be considered as such. The original intention was that this document would be complete and exhaustive, but upon starting the analysis, it was concluded that a complete analysis would be difficult, time-consuming and be of limited value.

2. ANALYSIS OF THE CURRENT TCL COMPILER

2.1 Overview

Creating an idea of the whole system in this case has proved to be extremely difficult. There are over 4000 lines of TCL code as well as many more lines of configuration data, which have to be thoroughly worked through in order to ascertain a complete picture of the system. There appears to be little definite structure within the code and there are very few useful comments. This makes the design of the system very hard to

represent, both through initial understanding of the system and when trying to demonstrate it on paper. However, the following section provides an attempt to present the design of the existing system and hopefully provide the reader with an insight into why the current system needs to be professionally reengineered.

It is important to remember that the following section does not aim to provide a complete analysis of the workings of the TCL Job Compiler, but simply a brief analysis on what is considered to be the most important areas.

2.2 Starting the Compiler

The compiler is started using a shell file (or batch file when using MSDOS). This basically just specifies some working directories for the compiler to use and also starts the compiler system with the line;

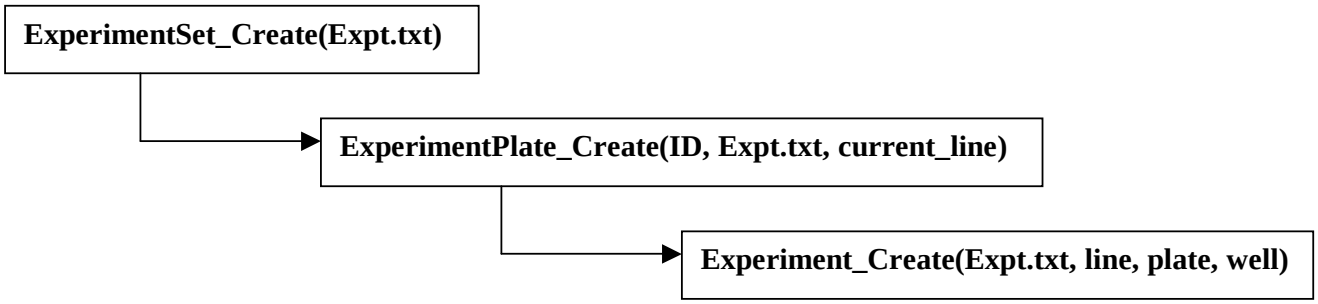
```
$TCLSH $RS_HOME/Compiler/compiler.tcl $RS_HOME $EXPT
```

This line calls the TCL analyzing tool and passes **Compiler.txt** to it with some parameters. Compiler.txt is the first TCL file that is called. Its purpose is largely to provide configuration details for the system, setup the system, and to set the system running. It also cleans up the arrays and registers used during the system process, deleting all of the elements of the various variables. It then calls a serious of definition files, which provide descriptions and measurements of various parts of the system. For example, it provides the dimensions of the various pipetting tools and details of the volumes of liquid that they can manipulate. Finally, the **System_Run()** procedure is called.

The **System_Run()** procedure has two main functions. The first is the creation and setting up of the experiments. The second is concerned with performing the experiments. These will be discussed concurrently. This is also the order in which the functions occur in the system.

2.3 The 'Create' Process

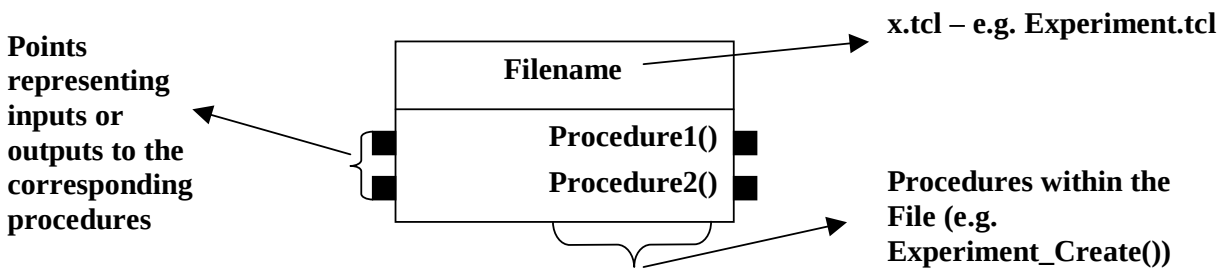
The experiments are arranged in an informal hierarchy. An ExperimentSet contains a number of ExperimentPlates and in turn, a ExperimentPlate contains a number of Experiments. It should be made clear that there are no formal connections between these three items, only a methodological idea. The following is an extremely simplified graphical representation of how the experiments are initialised and set up, and how the entities are seen to be given an impression of hierarchy.

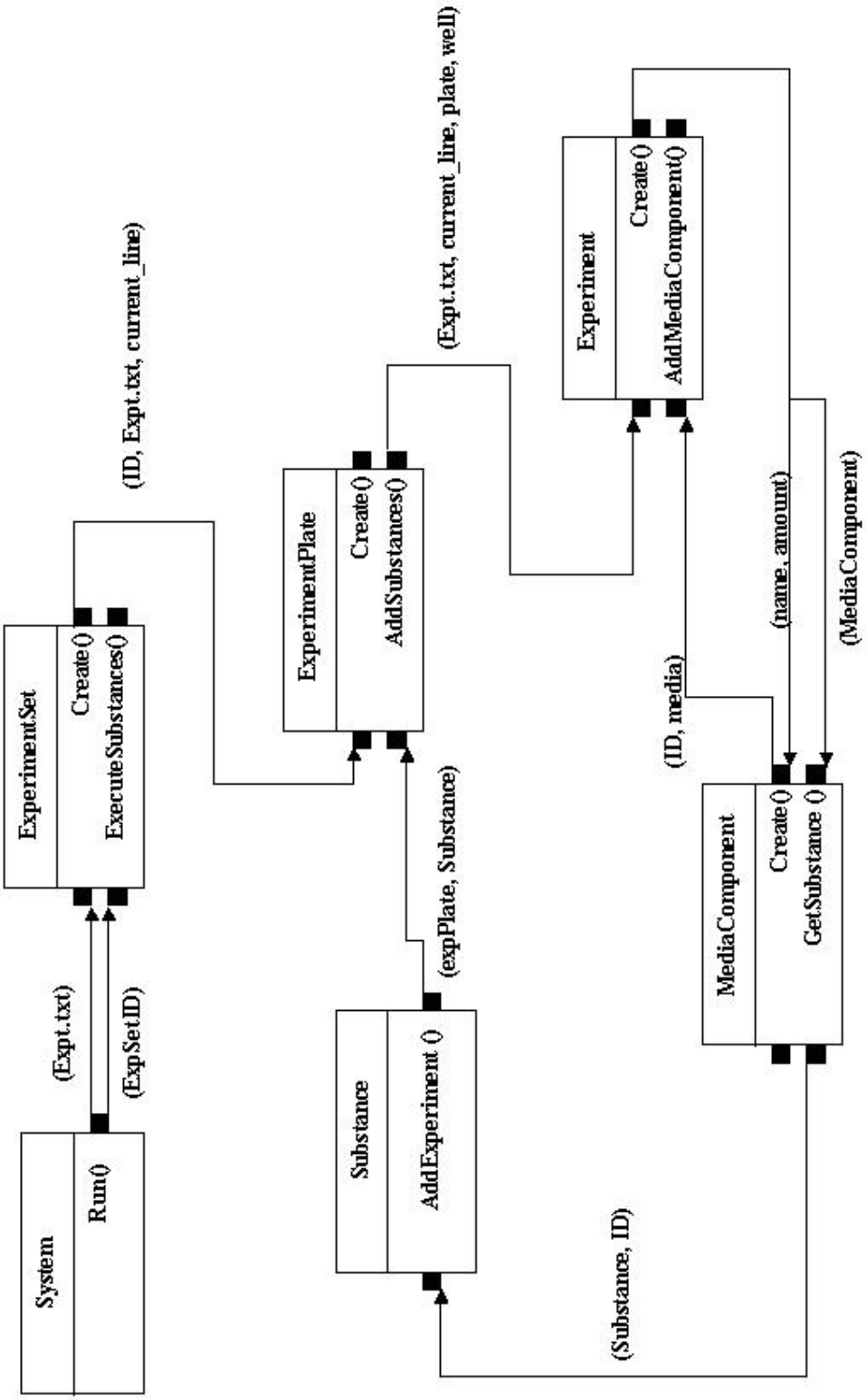


The following diagram is an illustration of the data flow for the current system regarding the setting up of the experiments. It would have been preferable to use UML to demonstrate this. However this was not possible as the current system does not employ use of the entities that UML is generally used for (classes, class attributes, objects, etc.). It is hoped that the diagram is comprehendible nonetheless.

The boxes represent the various groups of procedures found in the current system. At first glance, these may look like classes and the procedures below them, relating to the class. However this is slightly misleading. The current system groups common procedures according to their function and placement. For example, the procedures that control the properties and functionality of the Experiments are all listed in a file called Experiment.tcl. This gives the impression of a kind of structure without explicitly defining it. So, for example, the create procedure within the Experiment.tcl file is not called create() as it is suggested in the diagram, rather it is called Experiment_Create().

The terminology can therefore be thought of as the following;




System
Run()
(Expt.txt)
(ExpSetID)
ExperimentSet
Create()
ExecuteSubstances()
(ID, Expt.txt, current_line)
ExperimentPlate
Create()
AddSubstances()
(Expt.txt, current_line, plate, well)
Substance
AddExperiment ()
(expPlate, Substance)
Experiment
Create()
AddMediaComponent()
(ID, media)
(name, amount)
(MediaComponent)
MediaComponent
Create()
GetSubstance ()
(Substance, ID)

2.4 Setting up of the Experiments

The following is a step-by-step account of how the experiments are set up in the current system. The experiment creation is initiated from the System_Run() procedure.

1. The System_Run() procedure calls ExperimentSet_Create() procedure and passes to it a link to the Experiment data outputted by the ASE-Progol system (Expt.txt). Expt.txt is a text file that lists each and every experiment to be completed. It is assumed that each experiment contained in Expt.txt will occupy on well in one plate.

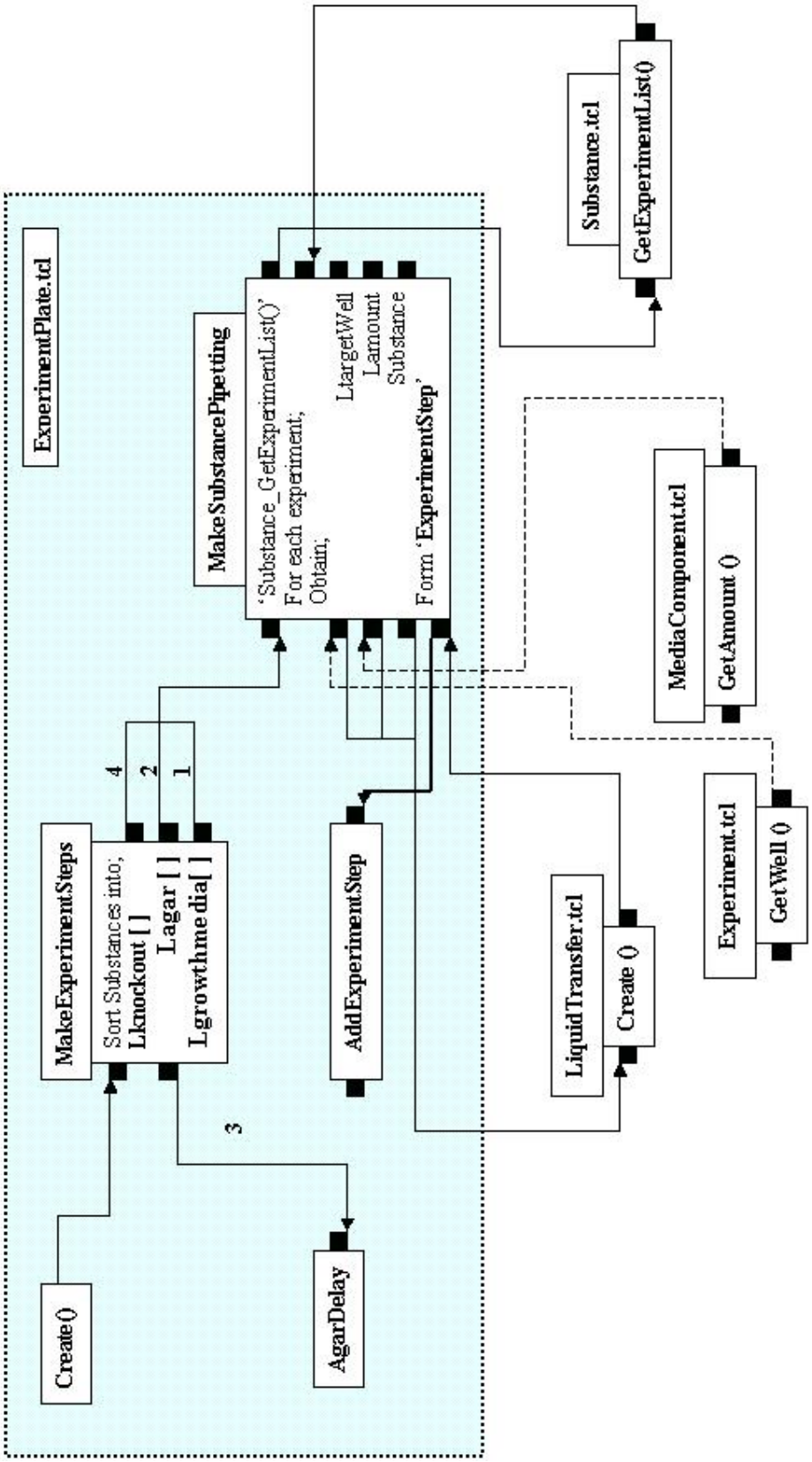
2. The ExperimentSet_Create() procedure creates a ExperimentSet variable which is a 2-dimensional array consisting of an ID and a number of ExperimentPlates. The procedure then creates a number of ExperimentPlates using the ExperimentPlate_Create() procedure, passing to it an ID, Expt.txt, and a pointer to the first line of Expt.txt. The procedure will continue to create ExperimentPlates until there are no more left to create. ExperimentSet_Create() must create an ExperimentPlate before adding it to the ExperimentSet.

3. The ExperimentPlate_Create() procedure creates an ExperimentPlate, with various attributes. After a labware tools check, it retrieves a pointer to the first available well of the plate (this is referred to as A1). The procedure then calls the Experiment_Create() procedure passing to it, Expt.txt, the current line of Expt.txt, the plate and the well. The idea is to create the experiment in that line of Expt.txt in the designated well (indicated by the variable inputs- plate, well). The procedure will continue to call Experiment_Create() until there are no more Experiments to create.

4. The Experiment_Create() procedure creates and sets up an Experiment using the details from the given line in Expt.txt. The procedure creates a MediaComponent using the name and amount of the substance specified within Expt.txt. The system keeps a record of all the substances used (possibly within Substance.tcl).

5. Finally, a pointer to the experiment is added to the substance and a record of the substance used is passed to the ExperimentPlate.

In this way, a whole set of experiments are built up. Again, it is necessary to mention that this description is by no means a complete analysis of the process involved, but does strive to capture the most important areas of design.


ExperimentPlate.tcl
MakeExperimentSteps
Sort Substances into;
Lknockout [ ]
Lagar [ ]
Lgrowthmedia[ ]
MakeSubstancePipetting
'Substance_GetExperimentList()'
For each experiment;
Obtain;
LtargetWell
Lamount
Substance
Form 'ExperimentStep'
AddExperimentStep
Create()
AgarDelay
Substance.tcl
GetExperimentList()
MediaComponent.tcl
GetAmount ()
LiquidTransfer.tcl
Create ()
Experiment.tcl
GetWell ()
1
2
3
4

2.5 Creating the Experiment Steps

Looking at the diagram on the previous page is no easy task. It seems very convoluted and the reason for that is that it simply easy convoluted. It differs from the previous diagram in that there are no labels documenting the variables passed between procedures. It was deemed that this would make the diagram even more cluttered and serious reduces its readability. The variables will be mentioned during the following textual analysis of this process.

The creation of the experiment steps is initiated from the ExperimentPlate_Create () procedure with the line;

```
ExperimentPlate_MakeExperimentSteps $curId
```

An ID to an ExperimentPlate (that contains a substance list) is passed to the procedure, MakeExperimentSteps.

1. The substance list of the ExperimentPlate is sorted into three different categories. These are lknockout (Yeast), lagar, and lgrowthmedia.

2. Each item from the lgrowthmedia category is passed to the MakeSubstancePipetting procedure along with its ExperimentPlate.

3. Within MakeSubstancePipetting(), Substance_GetExperimentList is called. This returns a list of all the experiments that the given substance is used in. So, MakeSubstancePipetting() now has a record of all the experiments that involve this specific substance.

4. For each experiment that uses the substance, Experiment_GetWell and MediaComponent_GetAmount() are called to return the target well that the substance is to be deposited in and the required amount that is to be deposited.

5. The target well, amount and substance variables are passed to the LiquidTransfer_Create() procedure. This procedure creates an ExperimentStep, which is passed back to MakeSubstancePipetting().

6. ExperimentPlate_AddExperimentStep is called and the newly created ExperimentStep is passed to it, thus adding the experiment step to the experiment plate.

7. Steps 2 – 6 are repeated, but using the lagar category.

8. ExperimentPlate_AgarDelay is called as the agar needs time to cool down before other operations can take place.

9. Steps 2 – 6 are repeated, but using the Lknockout category.

In this way, a series of experiment steps are sequentially built up in the right order until a complete set exists.

Another part of the system worth mentioning here is the ExperimentSet_ ExecuteSubstances() procedure. This seems to be a fairly linear procedure that runs through the various experiment steps of each of the mediums. It is here that the experiment steps are actually executed for all the plates in the entire experiment set.

2.6 A Liquid Transfer

Arguably the most important part of the TCL Job Compiler is the process of transferring a liquid. The robot, a Beckman Coulter 2000, is a workstation primarily for the transferring of liquids. The experiments are built and exectued using sequences of liquid transfers. The job file consists of many hundreds of liquid transfers and, therfore, it seems prudent to explore the creating of a liquid transfer within the current TCL Job Compiler.

The following analysis is from a sample job file from the operation of the current TCL Job Compiler. The arrows attempt to display the scope of each of the TCL functions.

```
Log "---- Step: Md"

Liquid_Transfer_Create
    Log "Executing LiquidTransfer: Md"

    Pipetting_Tool_Init
      Tool attach P20

      Tip_Attach
         Log "Attaching tip 4W (row 1, column 1)"
         Tip attach dispose P20 P20 P20
         Log "Attaching to P20 tip in row 1, column 1"

    Move Abs [Coord B5 A2] 46.869999

    Well_CheckVolume
        Log "Checking volumes: 19980.0, 23600.000000"

    Pipetting_Tool_Aspirate
       Log "Aspirating C00166:2E-1 from B5A2 (volume 20.0)"

       PipettingTool_Prewet

          PipettingTool_SuckAir
              Move Abs T 15.6502

          Move Abs Z 9.26959999

          PipettingTool_SetDirty
              putres system pod dirty_tip 1

          Move Rel T 17.8872
          Delay 0
          Move Abs Z 44.869999
          Move Abs T 15.6502
          Delay 0

          PipettingTool_SquirtAir
              Move Abs T 9.000000

          Delay 0
```

```
    PipettingTool_SuckAir
        Move Abs T 15.6502

    Move Abs Z 9.26959999

    PipettingTool_SetDirty
        putres system pod dirty_tip 1

    Move Rel T 14.5822
    Delay 0
    Move Rel T -1.322
    Move Abs Z 46.869999

Move Abs [Coord A4 A4] 22.17

Well_CheckVolume
    Log "Checking volumes: 20.0, 362.760010"

PipettingTool_Dispense
    PutVal tools P20 max_velocity 2
    Move Abs Z 18.5725
    Move Abs T 15.6502
    PutVal tools P20 max_velocity 25.000000
    Delay 0
    Move Abs Z 20.170000

    PipettingTool_SquirtAir
        Move Abs T 9.000000
     Delay 0

Well_AddSubstance
    Log "Performing Well_AddSubstance; Well: A4A4; C00166:2E-1"

Log "Transfer from B5A2 to A4A4 complete, volume 20.0"
```

3. FURTHER WORK

Please refer to the chapters 4 and 5 of the report (3) for details of how the analysis of the current TCL Job Compiler contributed to the project.

SYSTEM DESIGN SPECIFICATION

Author: Ben Tagger (bnt8)
Date: 27/02/2003
Version: 4.0
Status: Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to provide the reader with an outline of the system. It should provide a detailed plan from which a system can be implemented. It should address the system in a logical, progressive fashion detailing the sub-systems one by one until a complete picture of the whole system is achieved. This document should also be of use for subsequent alterations and maintenance of the system when required.

1.2 Scope

This document specifies the design outlines so that the analysis, implementation and testing phases can be conducted completely and correctly. As this document is not required as part of the SEM490 project, there may be absent or erroneous items contained within the document. Apologies are made in advance for this, but it should be noted that the workload for a SEM490 project is such that a complete, exhaustive set of documents is quite impossible (as they are not directly required as deliverables).

1.3 Objectives

To provide an outline of the system that is to be implemented and to provide a historical reference/manual for the current system.

2. OUTLINE OF STRUCTURE

2.1 Introduction

The following section will describe the system design of the TCL compiler.

2.2 Outline of System Function

The following are requirements for the TCL compiler system:

1. Establish a connection with the various configuration files: Substances.txt, Expt.txt, RSConfig.txt, etc.
2. Analyse the data contained within these files, parse and tokenize the files to a logical representation of the experiment configuration within the system.
3. Create a series of JOB files (JOB0.txt, JOB1.txt…..JOBN.txt) and populate them with the Bioscript pro (TCL) used by Biomek Workstation to conduct the experiments.

2.3 Module Descriptions

The following section provides a brief description of all the modules that will be implemented in the final TCL Compiler system.

2.3.1 Substance Class

A class representing a single substance as described by the configuration file, Substances.txt. There are ten substances described in this file, each with nine attributes each and the class Substance aims to represent a single one.

2.3.2 MediaComponent Class

An experiment that contains a certain substance will have a corresponding amount of the substance. The class, MediaComponent represents a component within a single experiment – so, a substance and an amount.

2.3.3 SubstanceList Class

A class that represents the list of the ten substances described in the configuration file, Substances.txt.

2.3.4 Experiment Class

A class representing a single experiment contained within a single well of an experiment plate. The class Experiment will contain information regarding the experiment, including the media components used, the location (well) and the length of the experiment.

2.3.5 ExperimentPlate Class

A class representing a plate of experiments. The plate can be of any size, but the default is 8 X 12, in accordance with the plate standard (insert standard). This class provides methods for adding experiments and also for populating an ExperimentPlate with Experiments.

2.3.6 LogItem Class

A class that represents a single Log entry. LogItem will contain a single string relating to the log entry.

2.3.7 Log Class

This class handles the LogItem objects. It provides a single log item and provides the user with the ability to write the log to a file.

2.3.8 Tip Class

A class that represents a single type of tip used in the experiments.

2.3.9 TipRack Class

A class that represents the tip racks used in the experiments.

2.3.10 LabWare Class

A class that represents the types of labware used in the experiments. (i.e., the plates and wells)

2.3.11 Tool Class

A class that represents the different types of tool used in the experiments. (i.e., the grippers and the pipettes).

2.3.12 TipList Class

A class that holds a list of the types of tips.

2.3.13 TipRackList Class

A class that holds a list of the types of tip racks.

2.3.14 LabWareList Class

A class that holds a list of the types of lab ware.

2.3.15 ToolList Class

A class that holds a list of the types of tools.

2.3.16 Surface Class

A class that represents the surface area of the Biomek robot work station. This surface area is split into 12 sub-areas, each of which may be partitioned again. The details of the surface are supplied in the file, Surface.txt. The Surface will hold an array of SurfaceObjects (See 2.3.17).

2.3.17 SurfaceObject Class

A class that represents a type of surface object that can be placed in the surface. SurfaceObject is to be the super class of several other types of more specific surface object.

2.3.18 TipRackSurfaceObject Class

A class that represents a tip rack as a surface object. It extends SurfaceObject.

2.3.19 LabWareSurfaceObject Class

A class that represents a piece of lab ware a surface object. It extends SurfaceOBject.

2.3.20 ToolSurfaceObject Class

A class that represents a tool as a surface object. It extends SurfaceObject.

2.3.21 SubstanceSurfaceObject Class

A class that represents a substance as a surface object. It extends SurfaceObject.

2.3.22 WorkStation Class

A class that represents the entire experiment environment.This class will contain details of tools, tips, tipracks, labware, substances and surface. It will also contain some parameters attributed to the current state of the robot (Biomek environment).

2.3.23 Configurator Class

This class is primarily responsible for the creation of the configuration file (JOB0.txt). It will be responsible for handling the substance and surface configuration files, setting up the substances and the surface correctly and writing JOB0.txt to a specified location.

2.3.24 LiquidTransfer Class

This class will be responsible for creating the liquid transfers that form the basis of the job file (JOB1.txt).

2.3.25 StepBuilder Class

This class isolates all the steps (liquid transfers) needed to create the experiment platform, separates the steps into various vectors based on their substance and then executes a liquid transfer for each step in a pre-defined order.

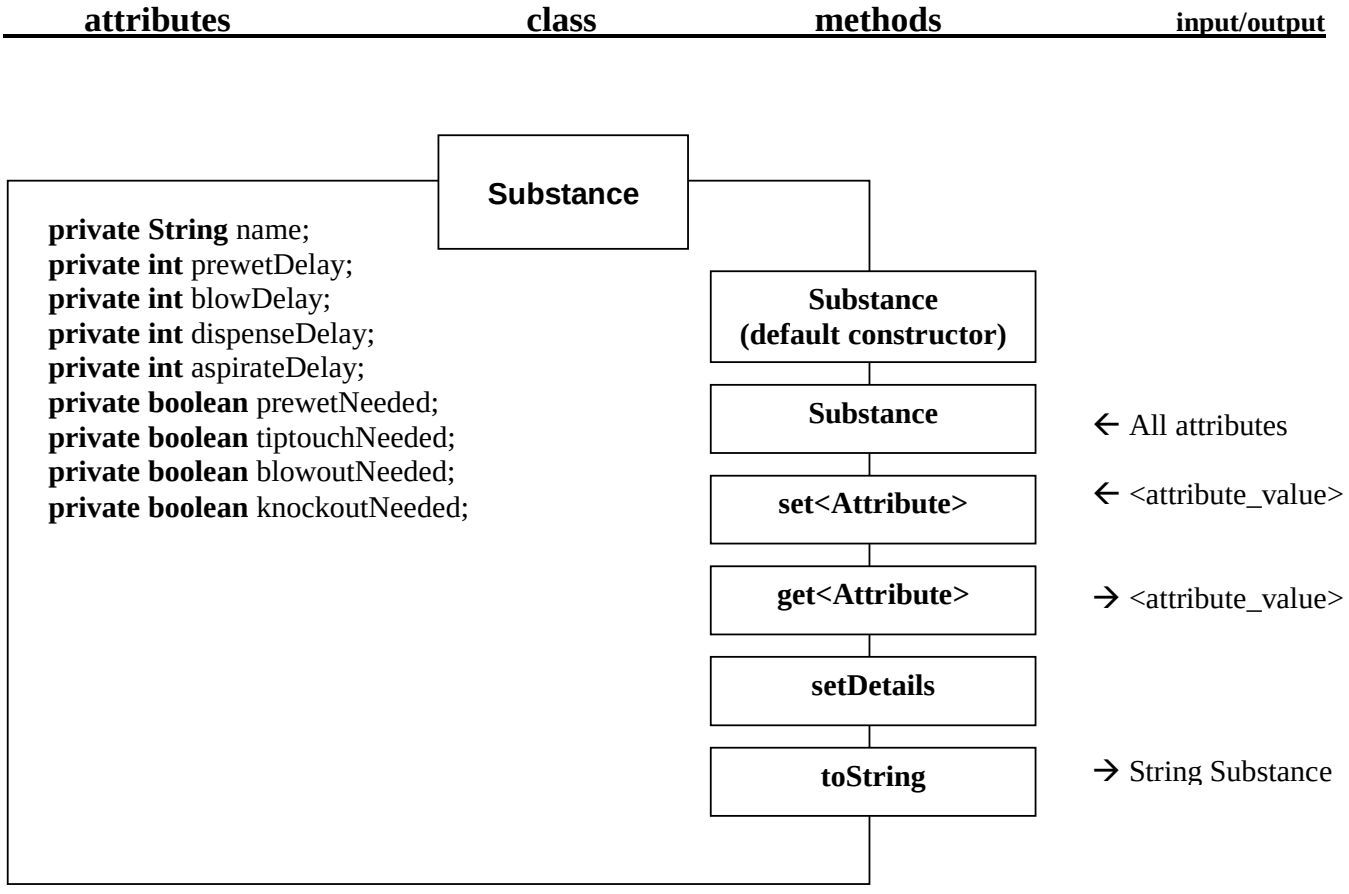
2.3.26 Step Class

This class represents a single step (liquid transfer) in the experiment process.

2.4 Detailed Description of each Module

2.4.1 Substance Class

Module Name : Substance
Type of Module : Data Structure
Module Description : A class representing a single substance as described by the configuration file, Substances.txt. There are ten substances described in this file, each with nine attributes each and the class Substance aims to represent a single one.



Constructors

- Default Constructor – takes no parameters
- Alternative Constructor – takes all attributes to create a Substance.

Attributes

- String name – the name of the substance (e.g. minimal agar)..
- int prewetDelay – time in hundreth second for liquid to rise into tip when prewetting.
- int blowDelay – time in hundreth second for draining, when blowing.
- int dispenseDelay – delay after dispensing.

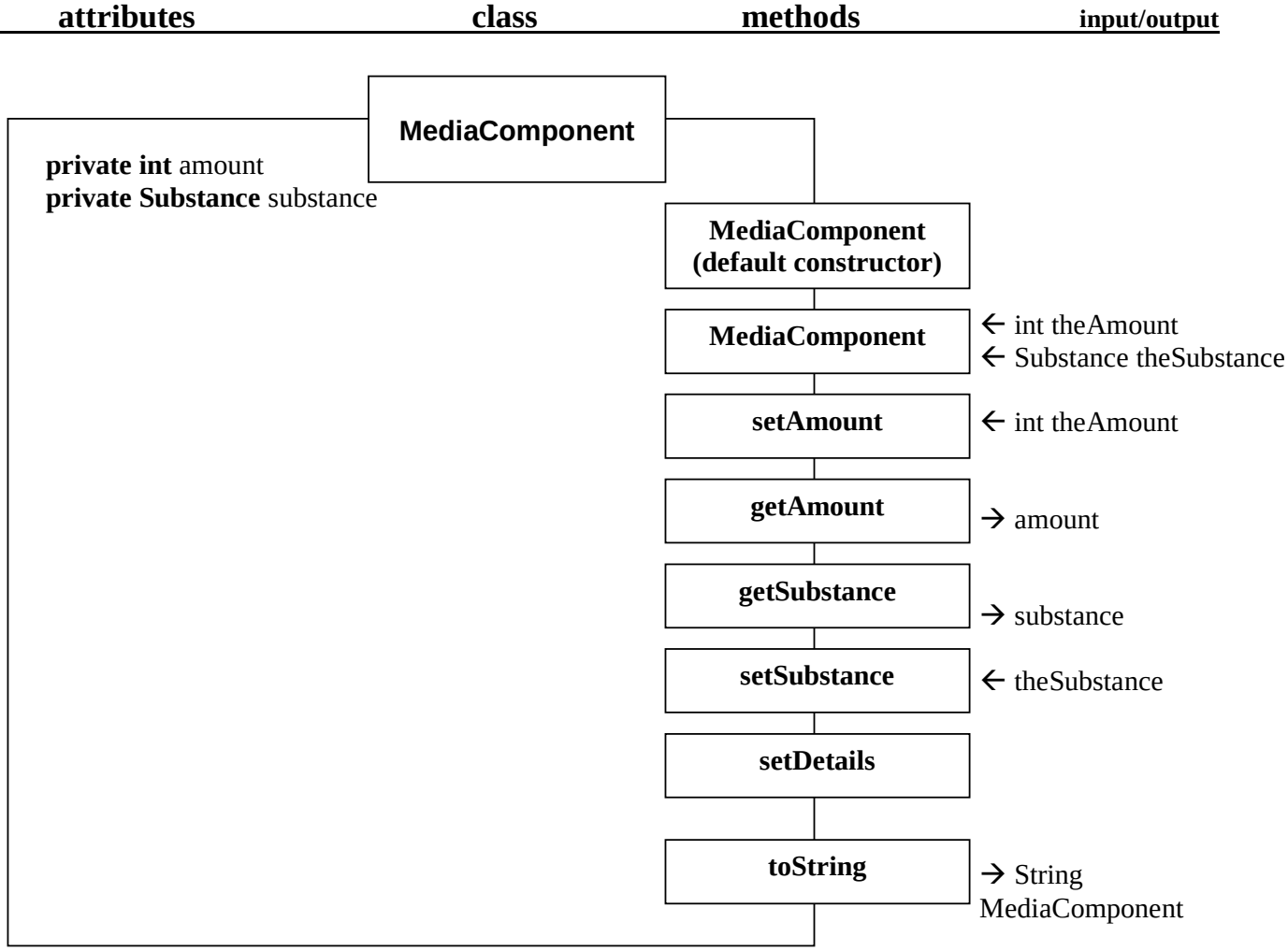
- int aspirateDelay – delay after aspirating.
- boolean prewetNeeded – Is a prewet needed ?
- boolean tiptouchNeeded – Is a tip touch needed ?
- boolean blowoutNeeded – Is a blow out needed ?
- boolean knockout – not actually sure right now…

Methods

- gets and sets for all class attributes
- setDetails for use in debugging and testing
- toString – returns a formatted string of a single substance

2.4.2 MediaComponent Class

Module Name : MediaComponent
Type of Module : Data Structure
Module Description : An experiment that contains a certain substance will have a corresponding amount of the substance. The class, MediaComponent represents a component within a single experiment – so, a substance and an amount.



Constructors

- Default Constructor – takes no parameters
- Alternative Constructor – takes two parameters – theAmount and theSubstance

Attributes

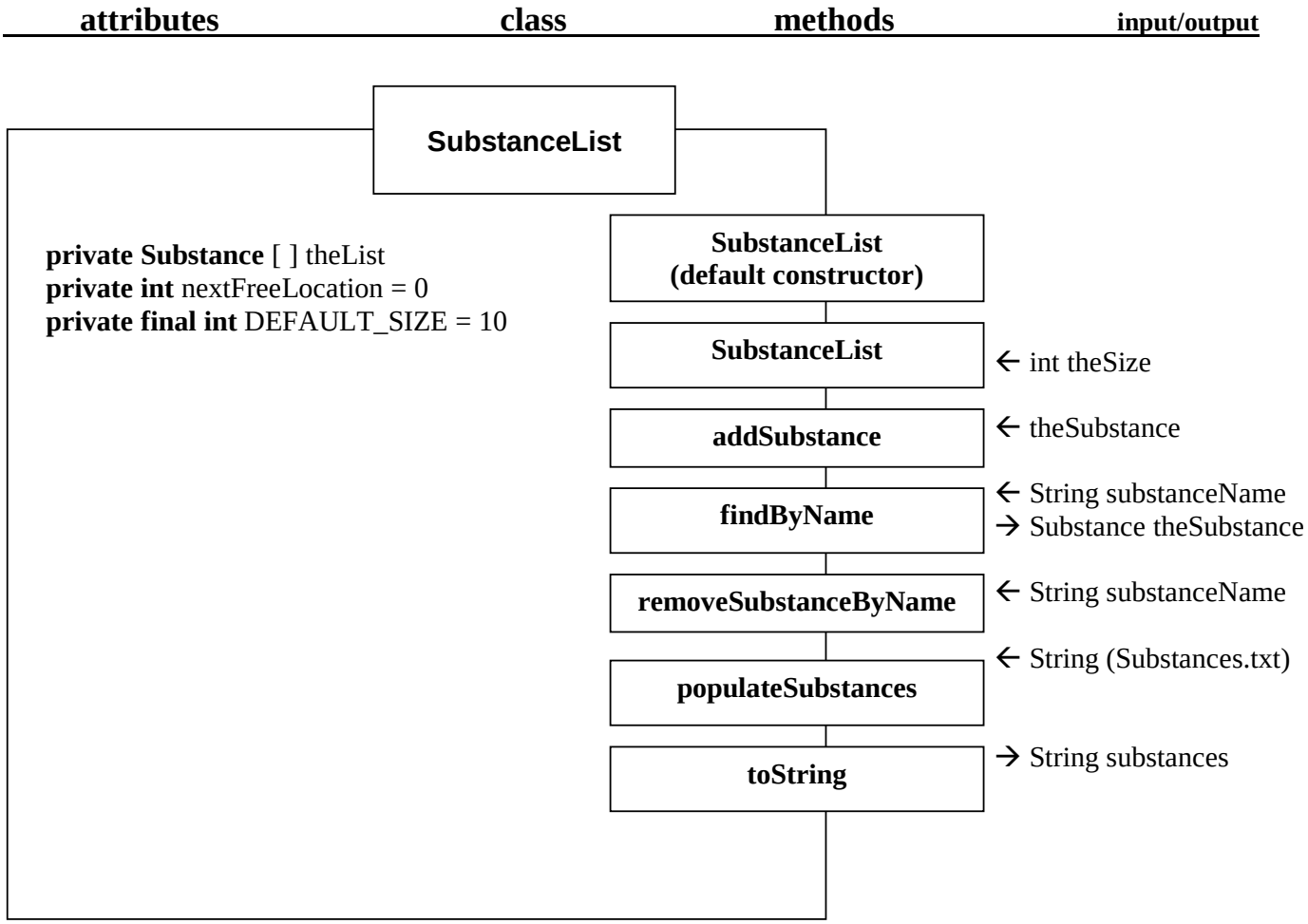
- int amount– the amount of the substance (e.g. minimal agar 120)
- Substance substance – the substance attributed to the Media Component

Methods

- gets and sets for all class attributes
- setDetails for use in debugging and testing
- toString – returns a formatted string of a Media Component – consisting of the Substance toString and the amount.

2.4.3 SubstanceList Class

Module Name : MediaComponent
Type of Module : Data Structure
Module Description : A class that represents the list of the ten substances described in the configuration file, Subsances.txt.



Constructors

- Default Constructor – takes no parameters – creates theList using DEFAULT_SIZE
- Alternative Constructor – takes one parameter, theSize, and creates theList with size specified in theSize.

Attributes

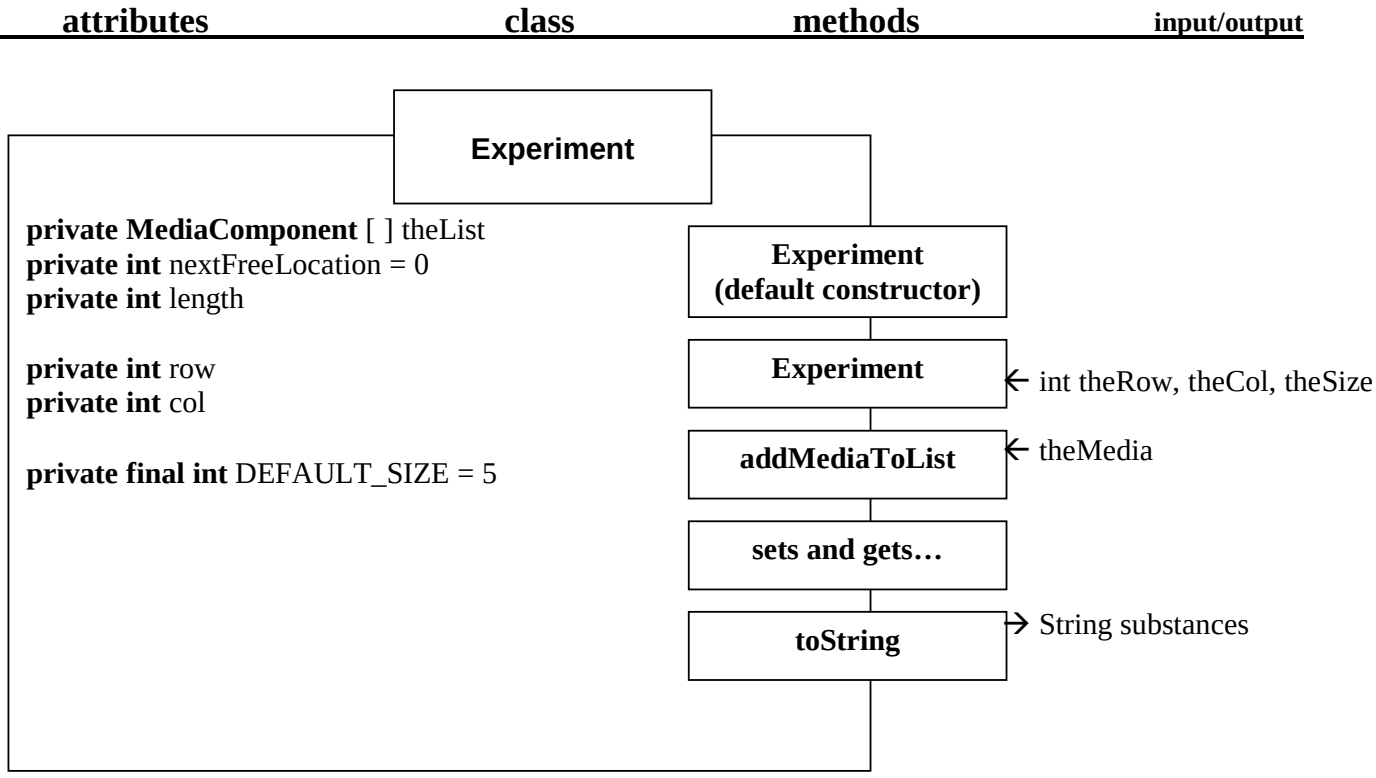
- Substance [ ]theList – array of Substances.
- int nextFreeLocation – integer used as a pointer to keep track of the array and where the next substance to be added would go.
- final int DEFAULT_SIZE – integer used to specify the size of the array of theList. The default is ten, as there are ten substances in Substances.txt.

Methods

- addSubstance – Takes in a substance as an argument and appends it to theList, updating nextFreeLocation appropriately.
- findByName – a method that takes in a String representing the name of a substance (e.g. minimal-agar) and returns the Substance object relating to that name.
- removeSubstanceByName – a method that takes in a String representing the name of the substance to be removed. The Substance object relating to the string is then removed from theList (not sure if this method is needed).
- populateSubstances – a method that takes in a file name (most likely to be Substances.txt) and, with the use of a string tokenizer, populates theList with the substances found in the file given as an argument.
- toString – returns a formatted string of the set of substances (most likely from a file, such as Substances.txt).

2.4.4 Experiment Class

Module Name : Experiment
Type of Module : Data Structure
Module Description : A class representing a single experiment contained within a single well of an experiment plate. The class Experiment will contain information regarding the experiment, including the media components used, the location (well) and the length of the experiment.



Constructors

- Default Constructor – takes no parameters – creates theList using DEFAULT_SIZE
- Alternative Constructor – takes three parameters – theRow and theCol specify the well that the experiment is to take place in. theSize specifies the size of the array for the media components.

Attributes

- MediaComponent [ ] theList – representing all the media components that are being used for the experiment
- int nextFreeLocation – integer used as a pointer to keep track of the array and where the next media component to be added would go.
- in length – integer used to represent the length of time for the experiment.
- int row – the row of the experiment
- int col – the column value of the experiment

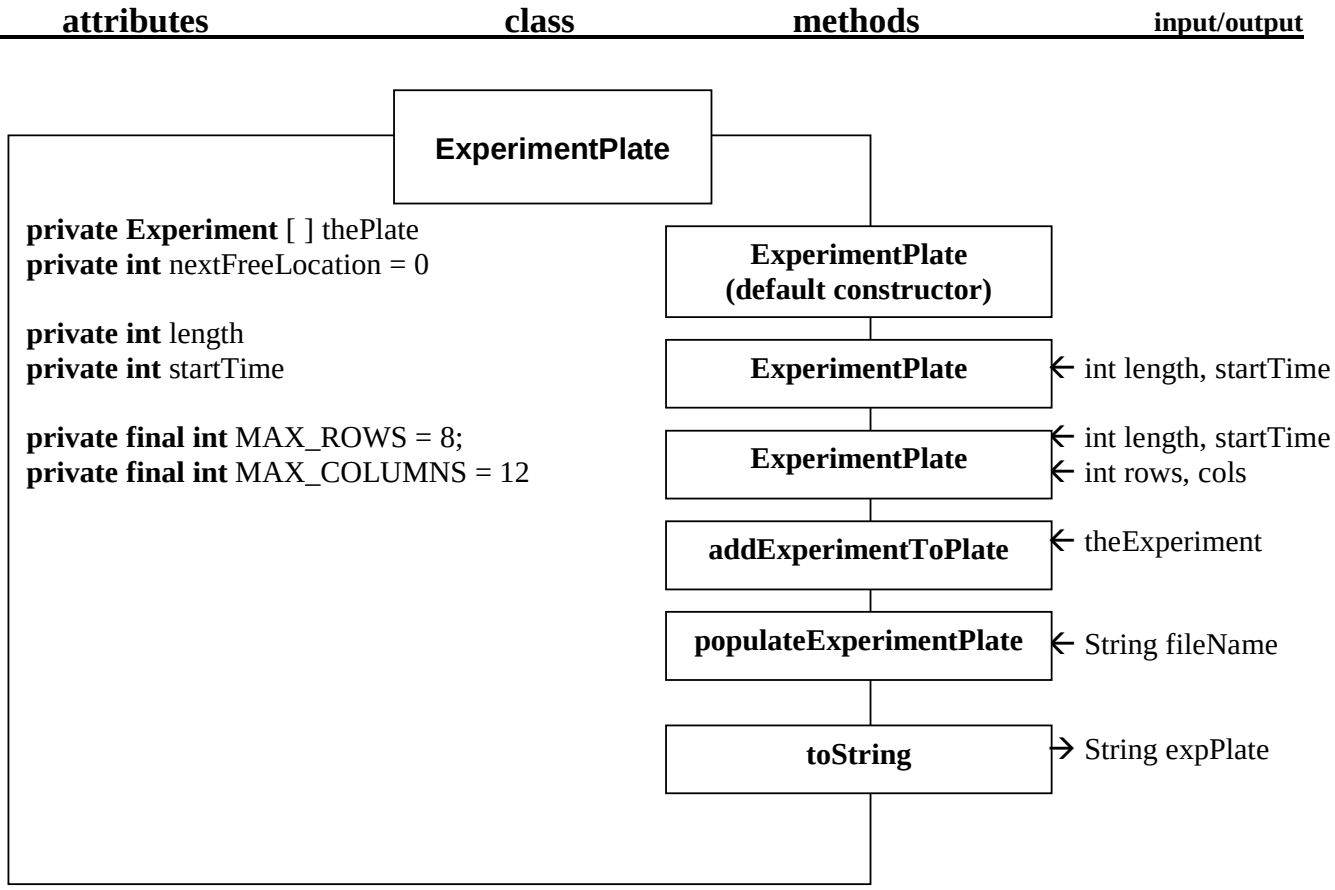
- final int DEFAULT_SIZE – integer used to specify the size of the array of theList. The default is five, as there are at most five media components used in any given experiment.

Methods

- gets and sets for some class attributes (length, row, col)
- addMediaToList – takes a MediaComponent and adds it to theList, updating nextFreeLocation appropriately.
- toString – returns a formatted string of an Experiment– consisting of the experiment length, then a list of the Media Components used in the experiment.

2.4.5 ExperimentPlate Class

Module Name : Experiment
Type of Module : Data Structure
Module Description : A class representing a plate of experiments. The plate can be of any size, but the default is 8 X 12, in accordance with the plate standard (insert standard). This class provides methods for adding experiments and also for populating an ExperimentPlate with Experiments.



Constructors

- Default Constructor – takes no parameters – creates thePlate using MAX_ROWS and MAX_COLUMNS
- Alternative Constructor – takes two parameters – length represents the total length of the whole experiment plate (should be the greatest length of any of the experiments in the plate?) startTime represents the start time of the experiment plate – this could be a counter – not sure yet. Creates thePlate using MAX_ROWS and MAX_COLUMNS
- Alternative Constructor – takes four parameters – length, startTime as described for the constructor above. thePlate is created using the value of 'rows x cols' as length.

Attributes

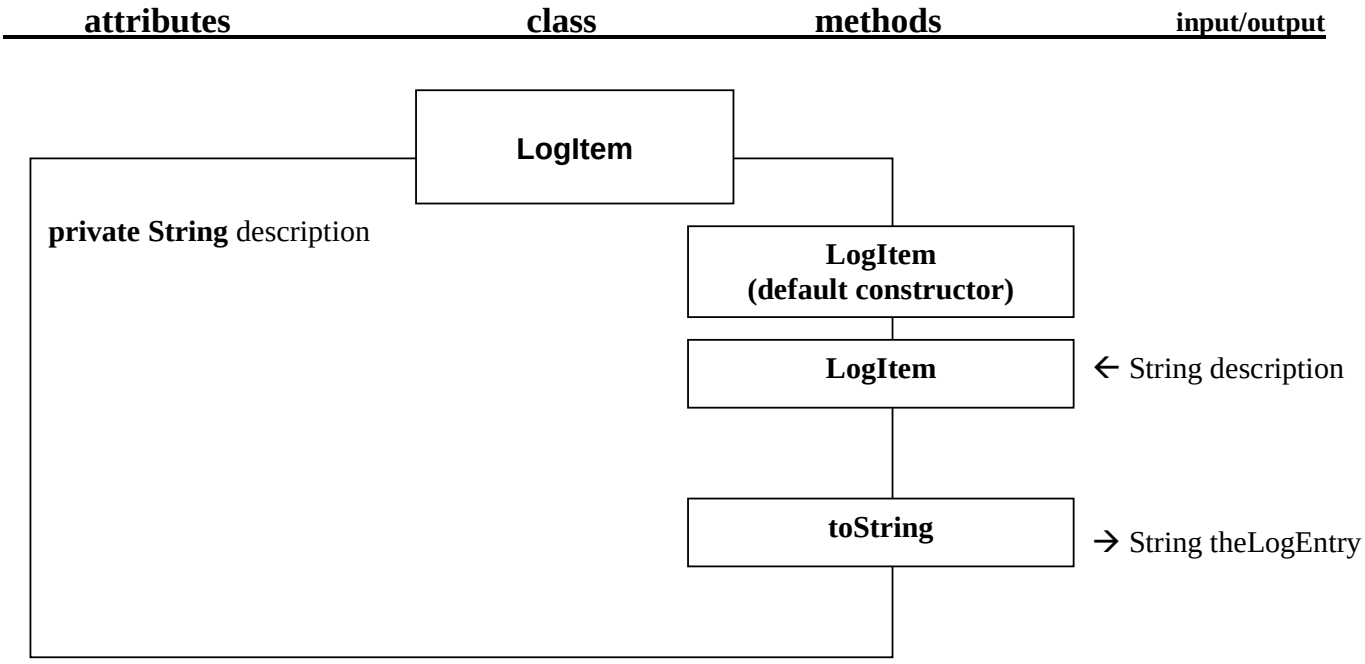
- Experiment [ ] thePlate – representing all the experiments that are being conducted in this experiment plate.
- int nextFreeLocation – integer used as a pointer to keep track of the array and where the next experiment to be added would go.
- in length – integer used to represent the length of time for this experiment plate.
- startTime – of the experiment plate – maybe a counter.
- final int MAX_ROWS, MAX_COLUMNS – integers used to specify the size of the array of thePlate. The default values are eight and twelve respectively (indicating a 8 X 12 plate)

Methods

- addExperimentToPlate – takes an Experiment and adds it to thePlate, updating nextFreeLocation appropriately.
- populateExperimentPlate - a method that takes in a file name (most likely to be Expt.txt) and, with the use of a string tokenizer, populates thePlate with the experiments found in the file given as an argument.
- toString – returns a formatted string of an Experiment plate– consisting of the details of each experiment in the plate, their media components and attributes.

2.4.6 LogItem Class

Module Name : LogItem
Type of Module : Data Structure
Module Description : A class that represents a single Log entry. LogItem will contain a single string relating to the log entry.



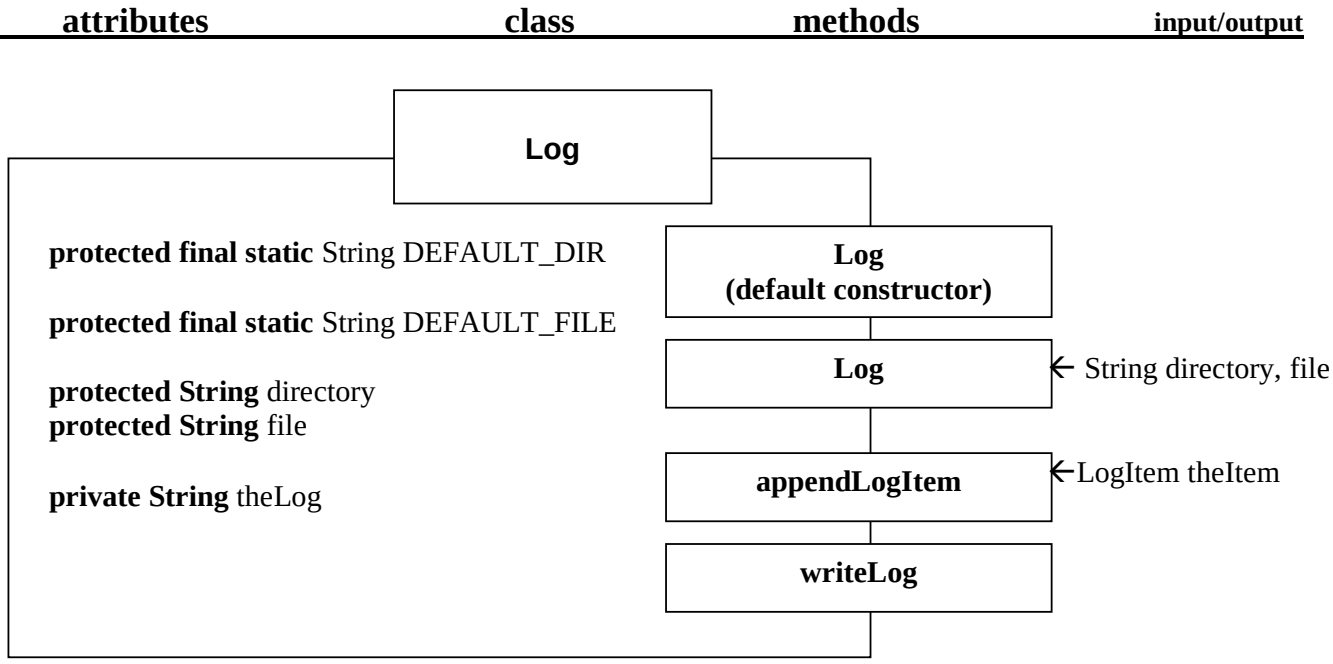
Constructors
- Default Constructor – takes no parameters.
- Alternative Constructor – takes one parameter – description which is the log entry

Attributes
- description – represents the log entry

Methods
- toString method returns a string representing the log entry

2.4.7 Log Class

Module Name : Log
Type of Module : Data Structure
Module Description : This class handles the LogItem objects. It provides a single log item and provides the user with the ability to write the log to a file.



Constructors

- Default Constructor – takes no parameters. The Default constructor initialises the attributes, file and directory, using the default attributes, DEFAULT_DIR and DEFAULT_FILE.
- Alternative Constructor – takes two parameters, which are used to intialise the variables, file and directory.

Attributes

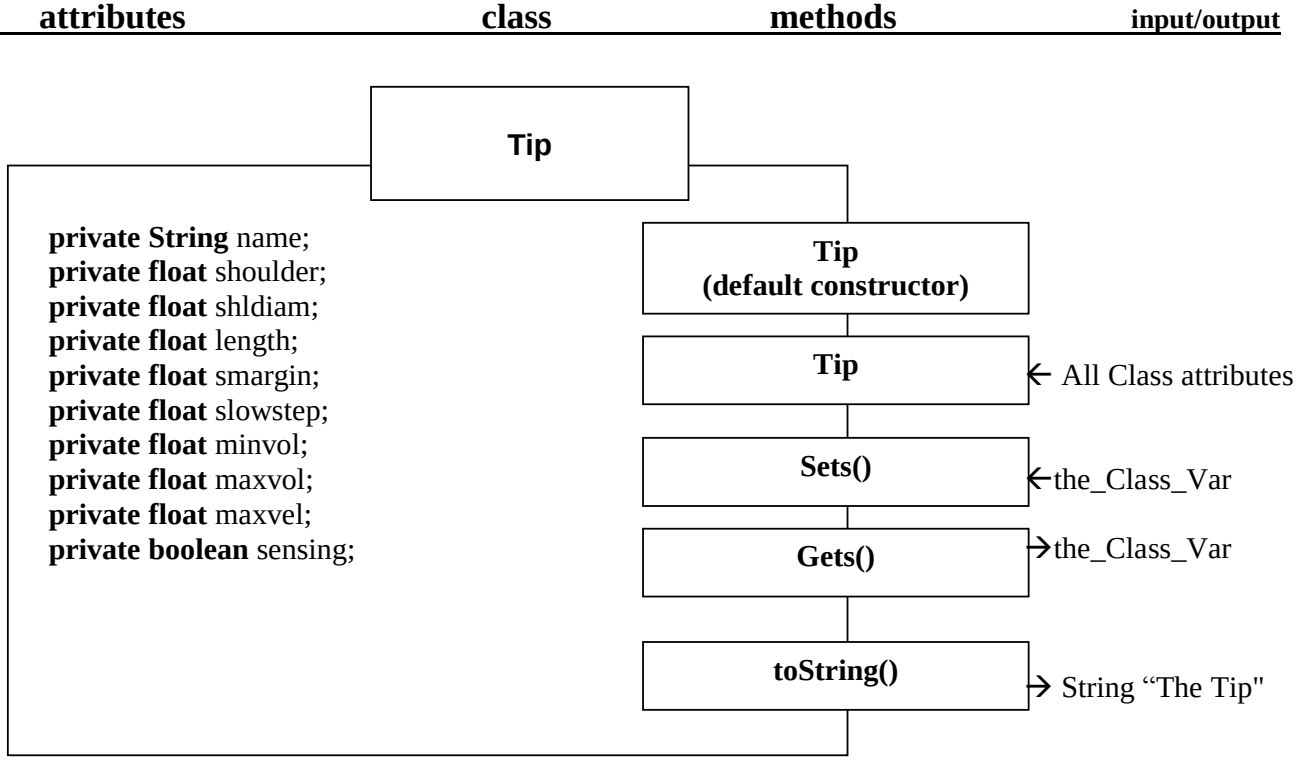
- DEFAULT_DIR will be set to something such as "c:/test" for debugging purposes.
- DEFAULT_FILE will be set to something such as "log" for debugging purposes.
- directory – a string representing the directory of the log file.
- file – a string representing the name of the log file.
- theLog – a String representing the whole of the log file.

Methods

- appendLogItem – take in a LogItem object (consisting of a descrption, see 2.4.6) and append it to the string, theLog.
- writeLog – takes the string, theLog, and writes it to a text file (stipulated by ‘file’ and ‘directory’).

2.4.8 Tip Class

Module Name : Tip
Type of Module : Data Structure
Module Description : A class that represents a single type of tip used in the experiments.



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes ten parameters corresponding to the class attributes and are used to intialise them.

Attributes

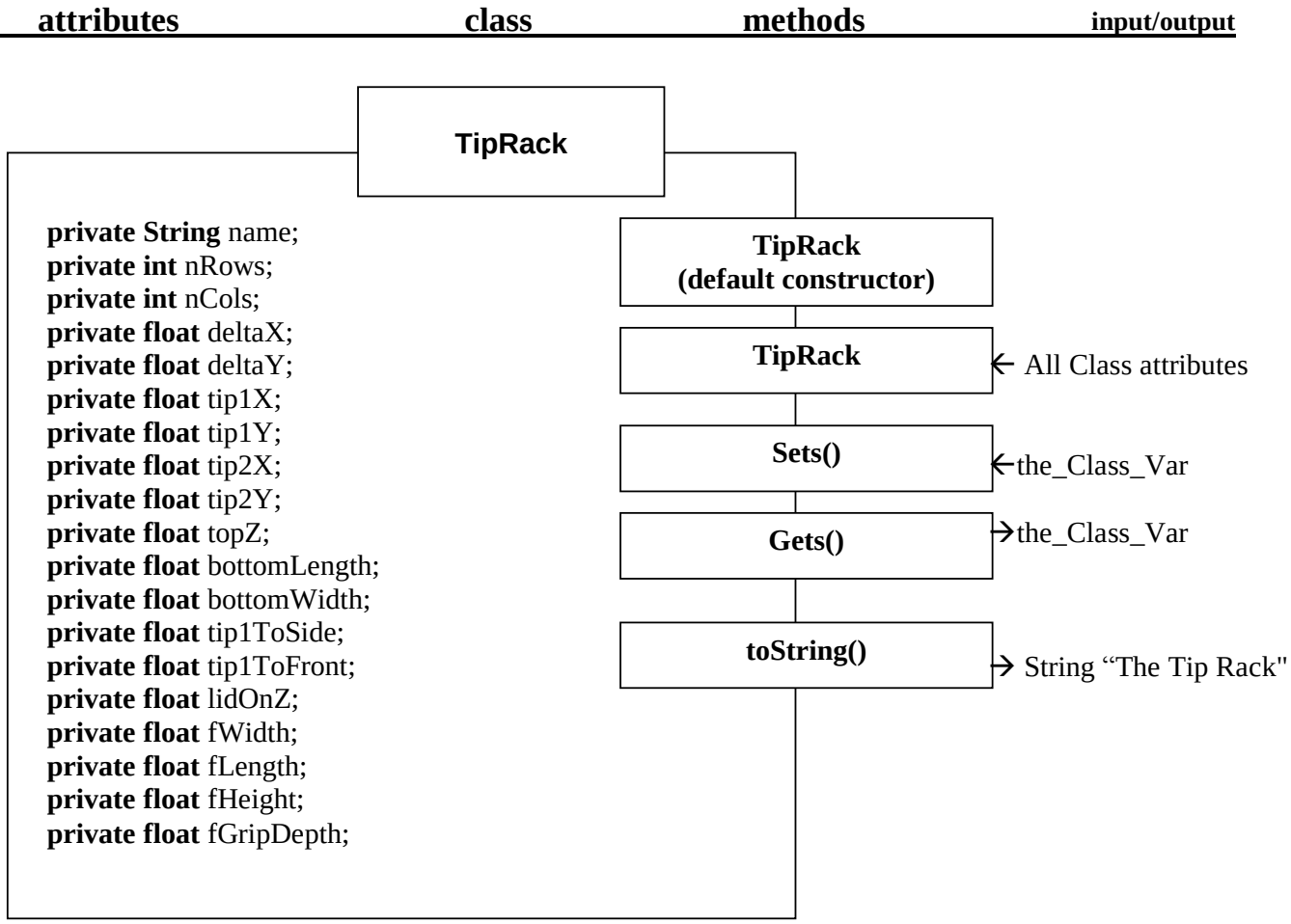
- String name – represents the name of the Tip. I.e., P20, P250, etc.
- Please refer to the Biomek BioScript Pro Programmer's Guide for further information for the other attributes.

Methods

- Sets and Gets for the class attributes.
- toString() – a string representing the tip, compatable for direct use within the configuration job file (JOB0.txt)

2.4.9 TipRack Class

Module Name : TipRack
Type of Module : Data Structure
Module Description : A class that represents a single tip rack used in the experiments. Tip racks typically hold a set of tips.



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes 19 parameters corresponding to the class attributes and are used to intialise them.

Attributes

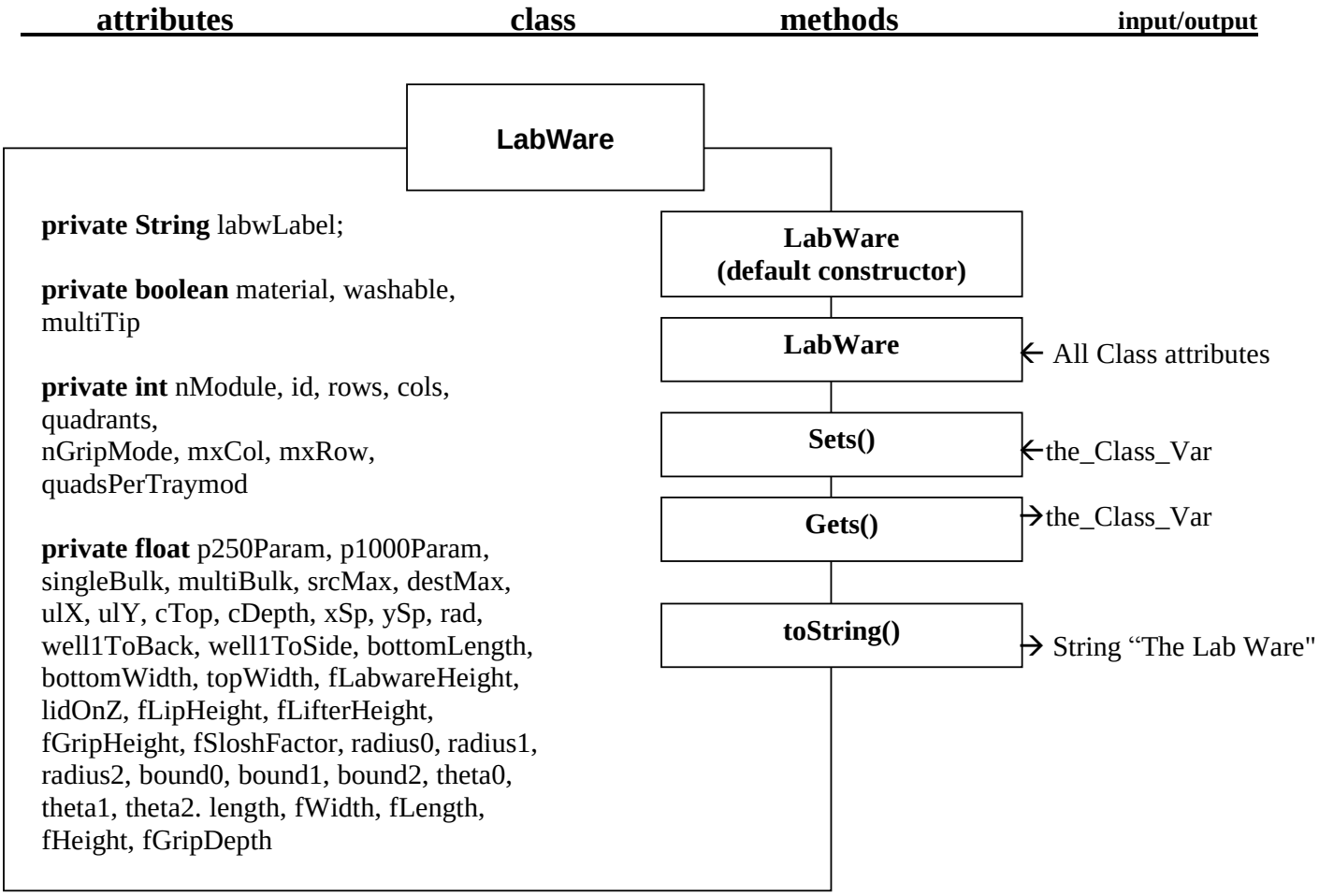
- String name – represents the name of the tip rack. I.e., P20, P250, etc.
- Please refer to the Biomek BioScript Pro Programmer's Guide for further information for the other attributes.

Methods

- Sets and Gets for the class attributes.
- toString() – a string representing the tip rack, compatable for direct use within the configuration job file (JOB0.txt)

2.4.10 LabWare Class

Module Name : LabWare
Type of Module : Data Structure
Module Description : A class that represents piece of lab ware used in the experiments (i.e., the plates and wells).



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes 50 parameters corresponding to the class attributes and are used to intialise them.

Attributes

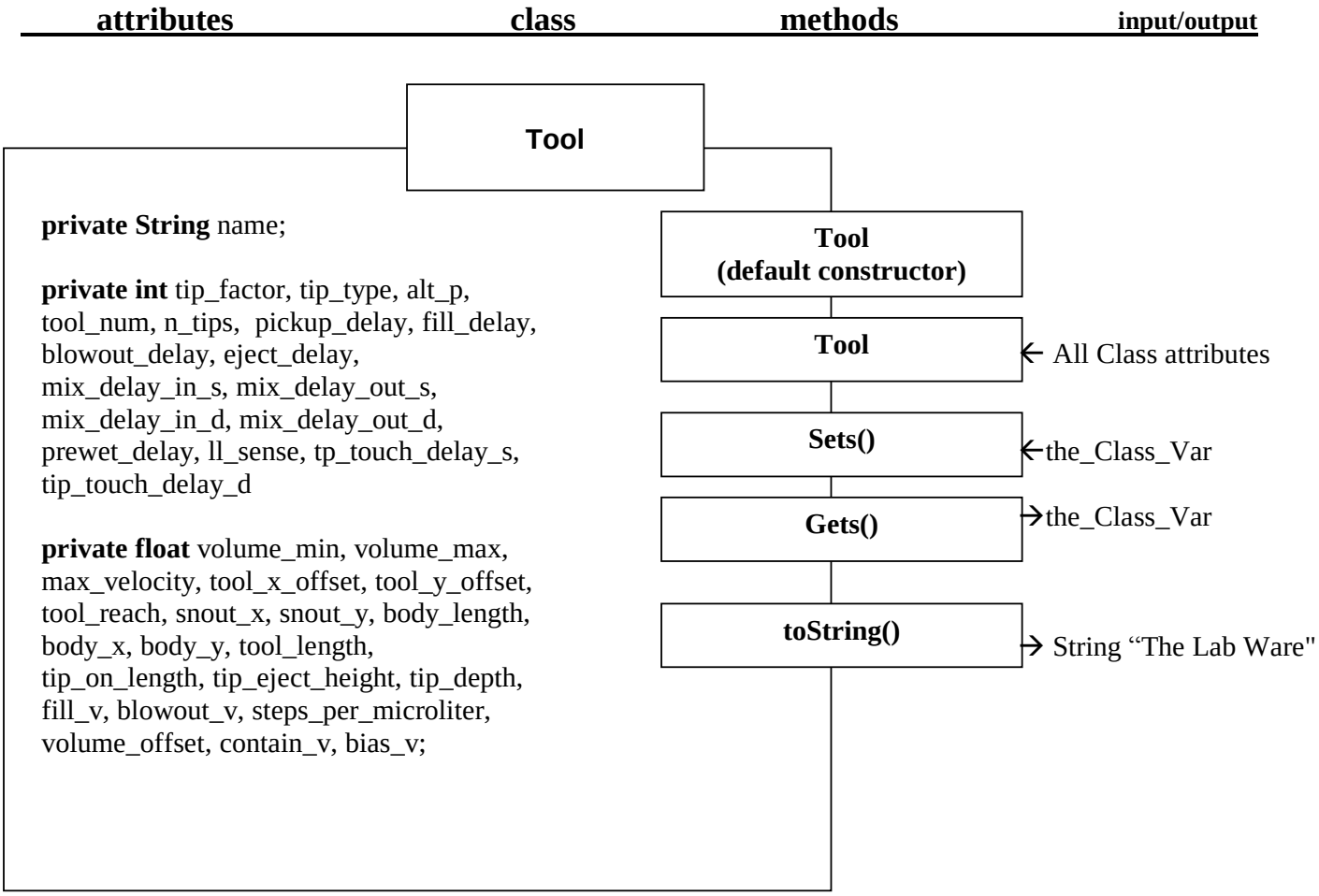
- String name – represents the name of the Lab ware. I.e., 96-Well Deep, quarter horizontal, etc.
- Please refer to the Biomek BioScript Pro Programmer's Guide for further information for the other attributes.

Methods

- Sets and Gets for the class attributes.
- toString() – a string representing the Lab ware, compatable for direct use within the configuration job file (JOB0.txt)

2.4.11 Tool Class

Module Name : Tool
Type of Module : Data Structure
Module Description : A class that represents the different types of tool used in the experiments. (i.e., the grippers and the pipettes).



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes 37 parameters corresponding to the class attributes and are used to intialise them.

Attributes

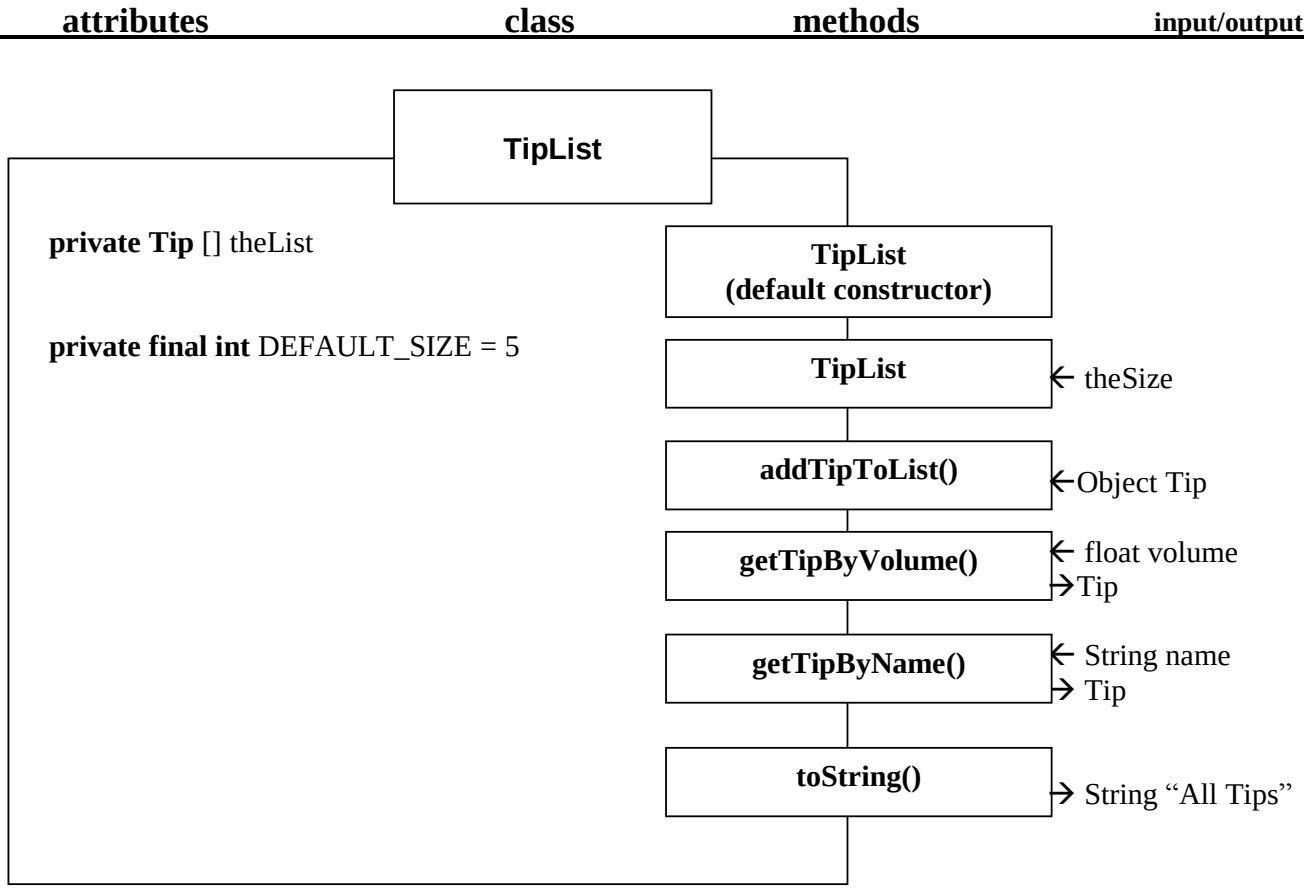
- String name – represents the name of the Tool. I.e., Gripper, P250L, etc.
- Please refer to the Biomek BioScript Pro Programmer's Guide for further information for the other attributes.

Methods

- Sets and Gets for the class attributes.
- toString() – a string representing the Tool, compatable for direct use within the configuration job file (JOB0.txt)

2.4.12 TipList Class

Module Name : TipList
Type of Module : Data Structure
Module Description : A class that holds all the tips available for the experiment.



Constructors

- Default Constructor – takes no parameters. Initiates the array using the DEFAULT_SIZE.
- Alternative Constructor – takes one parameter, the Size, which allows the user to define the size of theList.

Attributes

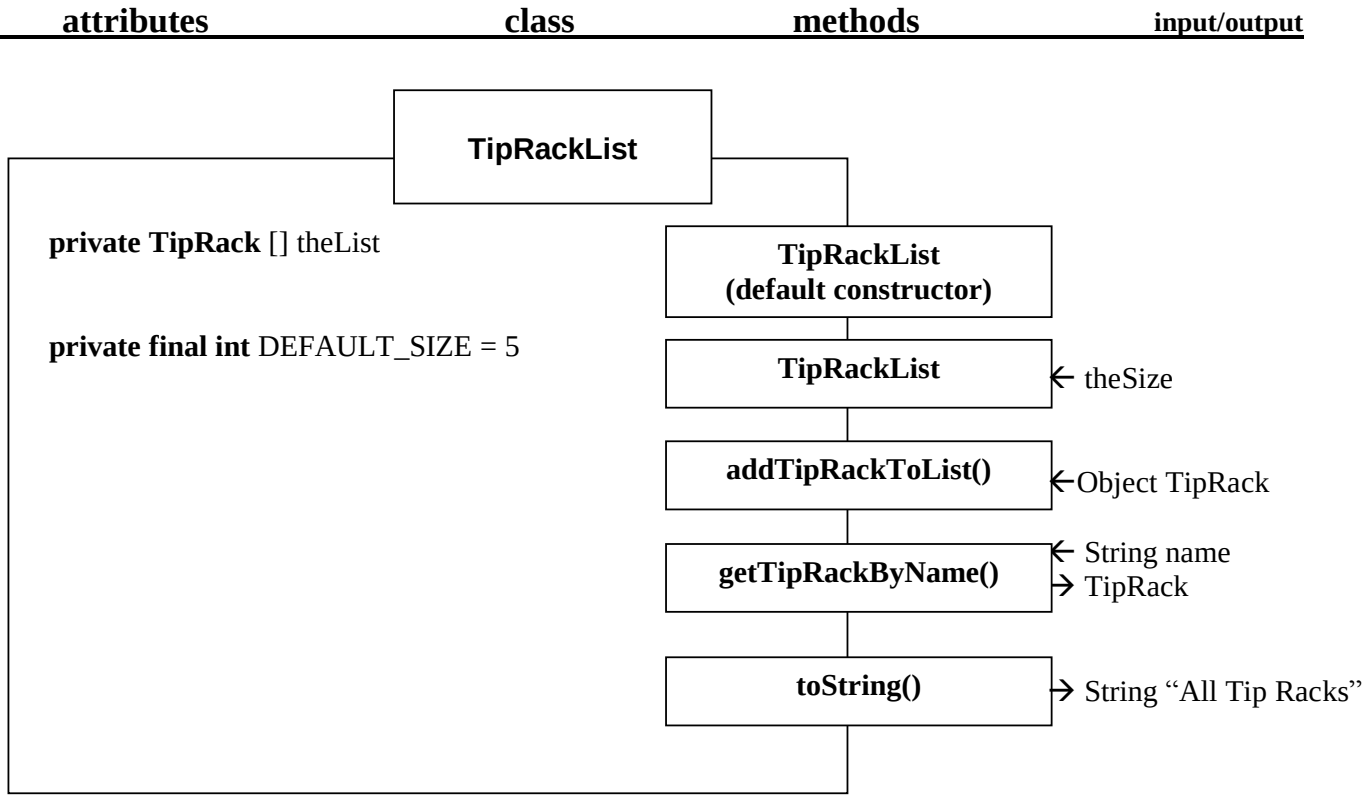
- theList – an array that holds all of the available Tips for the experiment process.
- DEFAULT_SIZE – the size that theList is initiated with when the default constructor is invoked.

Methods

- AddTipToList – takes a tip and adds it to theList. Tips are added in no particular order, other than the order in which they are added by the user.
- getTipByVolume – passes in a volume as a parameter and returns the tip that is appropriate for use for that volume.
- getTipByName – passes in a string representing the name of the tip needed and returns the tip with that name.
- toString() – returns the toString of all the tips contained within theList. Compatable for use in the configuration file (JOB0.txt).

2.4.13 TipRackList Class

Module Name : TipRackList
Type of Module : Data Structure
Module Description : A class that holds all the tip racks available for the experiment.



Constructors

- Default Constructor – takes no parameters. Initiates the array using the DEFAULT_SIZE.
- Alternative Constructor – takes one parameter, the Size, which allows the user to define the size of theList.

Attributes

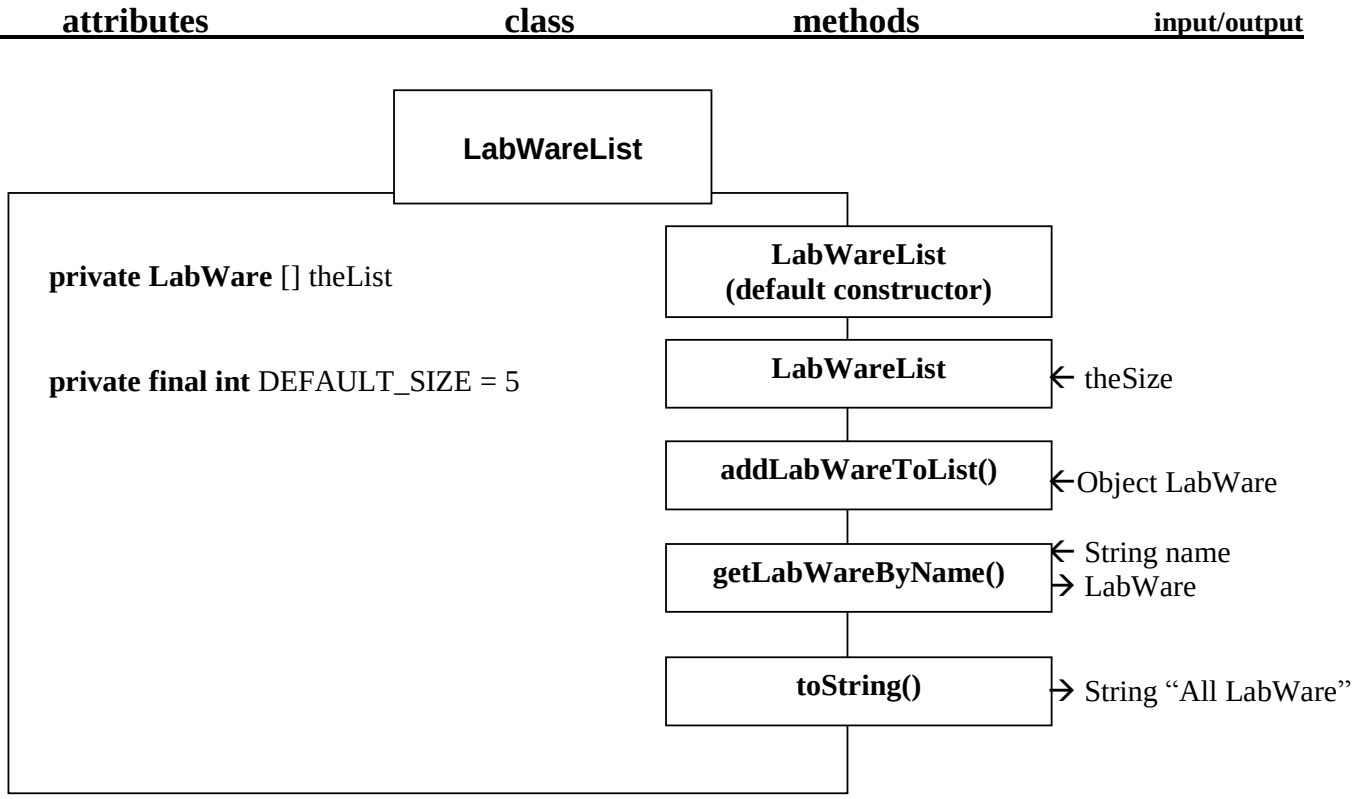
- theList – an array that holds all of the available Tip racks for the experiment process.
- DEFAULT_SIZE – the size that theList is initiated with when the default constructor is invoked.

Methods

- AddTipRackToList – takes a tip rack and adds it to theList. Tip racks are added in no particular order, other than the order in which they are added by the user.
- getTipRackByName – passes in a string representing the name of the tip rack needed and returns the tip rack with that name.
- toString() – returns the toString of all the tip racks contained within theList. Compatable for use in the configuration file (JOB0.txt).

2.4.14 LabWareList Class

Module Name : LabWareList
Type of Module : Data Structure
Module Description : A class that holds all of the lab ware available for the experiment.



Constructors

- Default Constructor – takes no parameters. Initiates the array using the DEFAULT_SIZE.
- Alternative Constructor – takes one parameter, the Size, which allows the user to define the size of theList.

Attributes

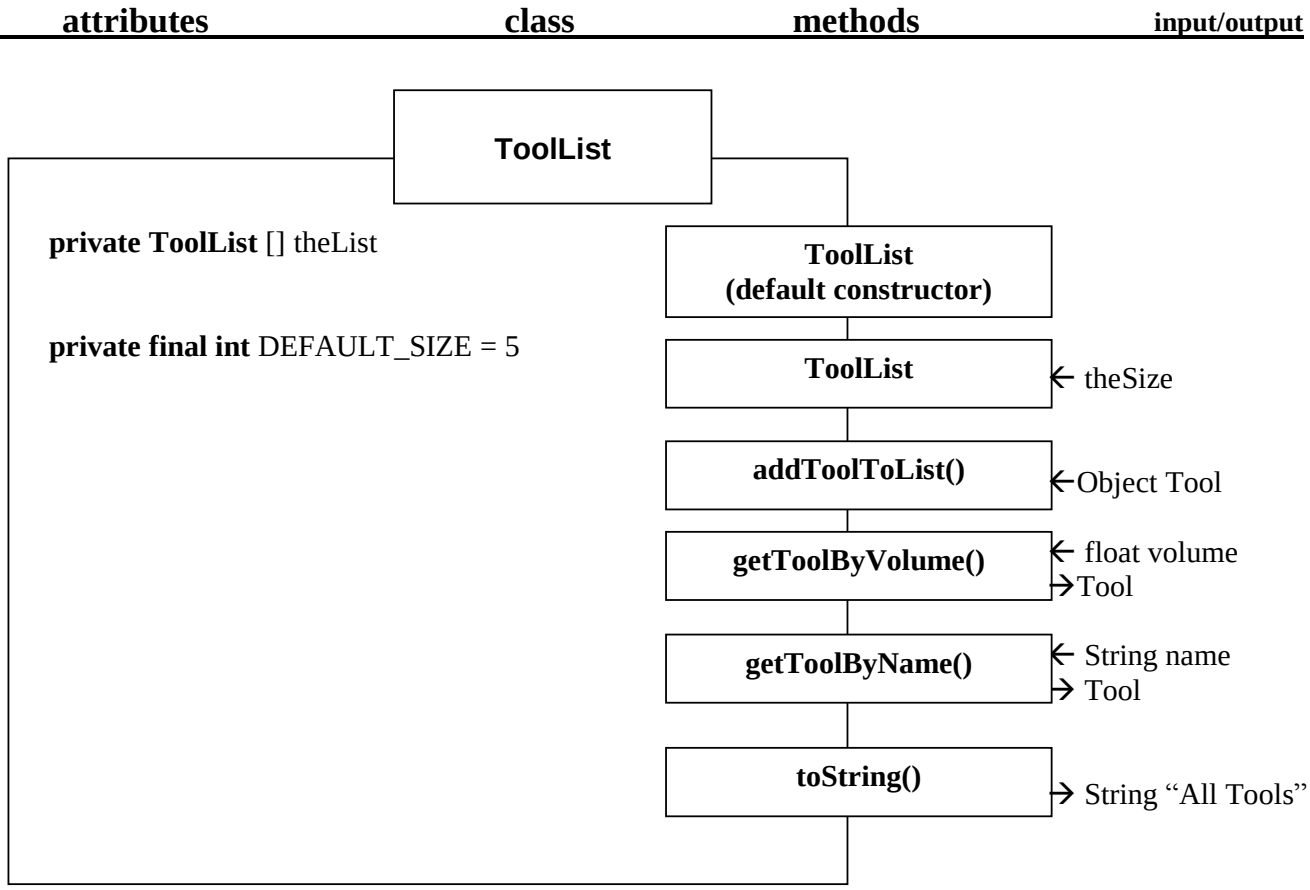
- theList – an array that holds all of the available Lab ware for the experiment process.
- DEFAULT_SIZE – the size that theList is initiated with when the default constructor is invoked.

Methods

- AddLabWareToList – takes a lab ware object and adds it to theList. Lab ware objects are added in no particular order, other than the order in which they are added by the user.
- getLabWareByName – passes in a string representing the name of the lab ware needed and returns the lab ware object with that name.
- toString() – returns the toString of all the lab ware contained within theList. Compatable for use in the configuration file (JOB0.txt).

2.4.15 ToolList Class

Module Name : ToolList
Type of Module : Data Structure
Module Description : A class that holds all the tools available for the experiment.



Constructors

- Default Constructor – takes no parameters. Initiates the array using the DEFAULT_SIZE.
- Alternative Constructor – takes one parameter, the Size, which allows the user to define the size of theList.

Attributes

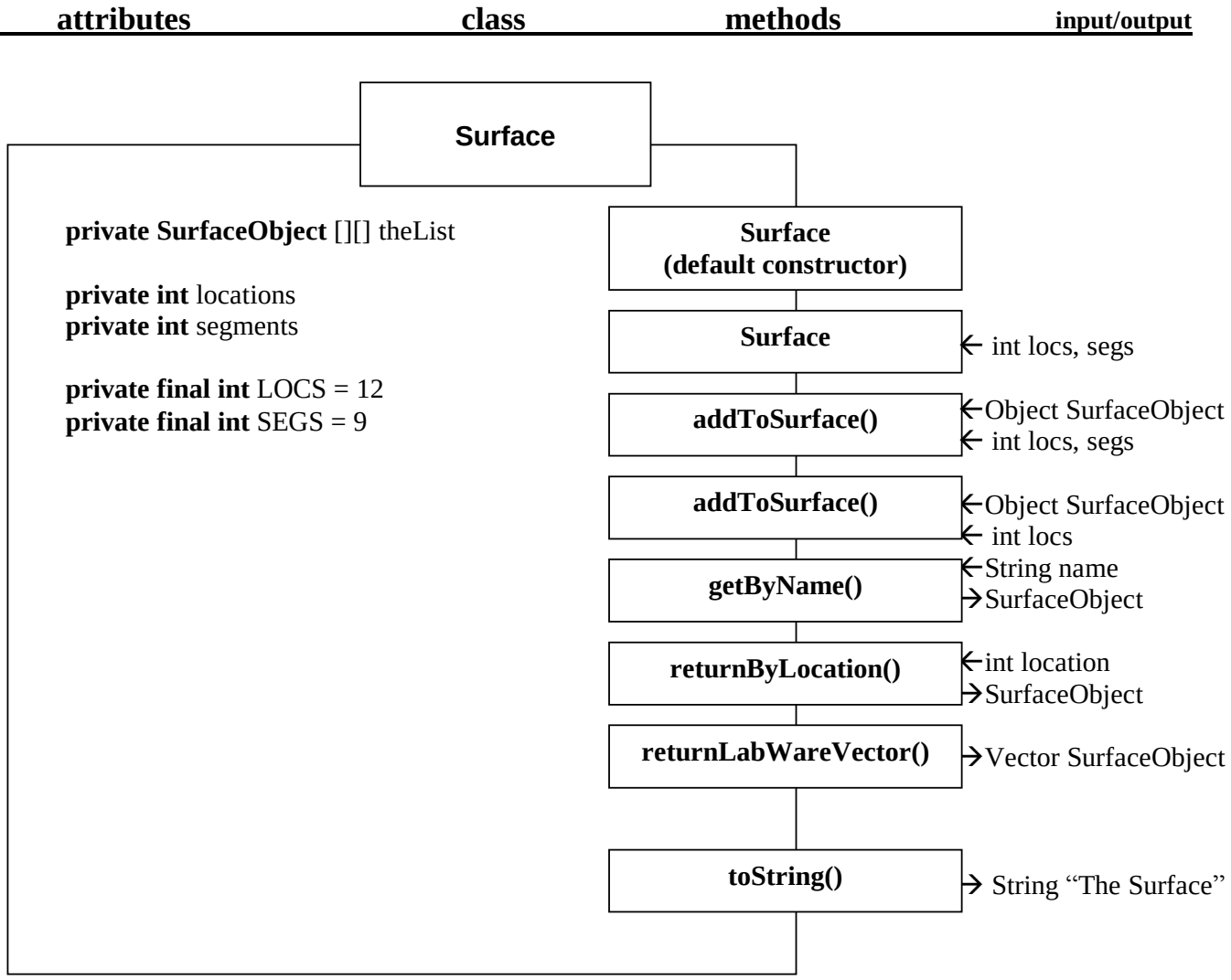
- theList – an array that holds all of the available Tools for the experiment process.
- DEFAULT_SIZE – the size that theList is initiated with when the default constructor is invoked.

Methods

- AddToolsToList – takes a Tool and adds it to theList. Tools are added in no particular order, other than the order in which they are added by the user.
- getToolByVolume – passes in a volume as a parameter and returns the tool that is appropriate for use for that volume.
- getToolByName – passes in a string representing the name of the tool needed and returns the tool with that name.
- toString() – returns the toString of all the tools contained within theList. Compatable for use in the configuration file (JOB0.txt).

2.4.16 Surface Class

Module Name : Surface
Type of Module : Data Structure
Module Description : A class that represents the surface area of the Biomek robot work station. This surface area is split into 12 sub-areas, each of which may be partitioned again. The details of the surface are supplied in the file, Surface.txt. The Surface will hold an array of SurfaceObjects (See 2.3.17).



Constructors

- Default Constructor – takes no parameters. Initiates the array using the LOCS and SEGS.
- Alternative Constructor – takes two parameters, locs and segs, which allows the user to define the size of theList.

Attributes

- theList – a 2D array that holds all of the objects in the Biomek work station surface. Objects are held of type SurfaceObject.
- int locations – the surface is split into 12 parts as below:

	1	2	3	4	5	6
A						
B						

The rows are signed alphabetically and the columns are numbered. Therefore the surface illustrated above would range from A1 to B6. For simplification purposes, the coordinates of the surface locations have been transferred to a single integer value. So, the integer location would be used to signify the coordinates as follows:

1	2	3	4	5	6
7	8	9	10	11	12

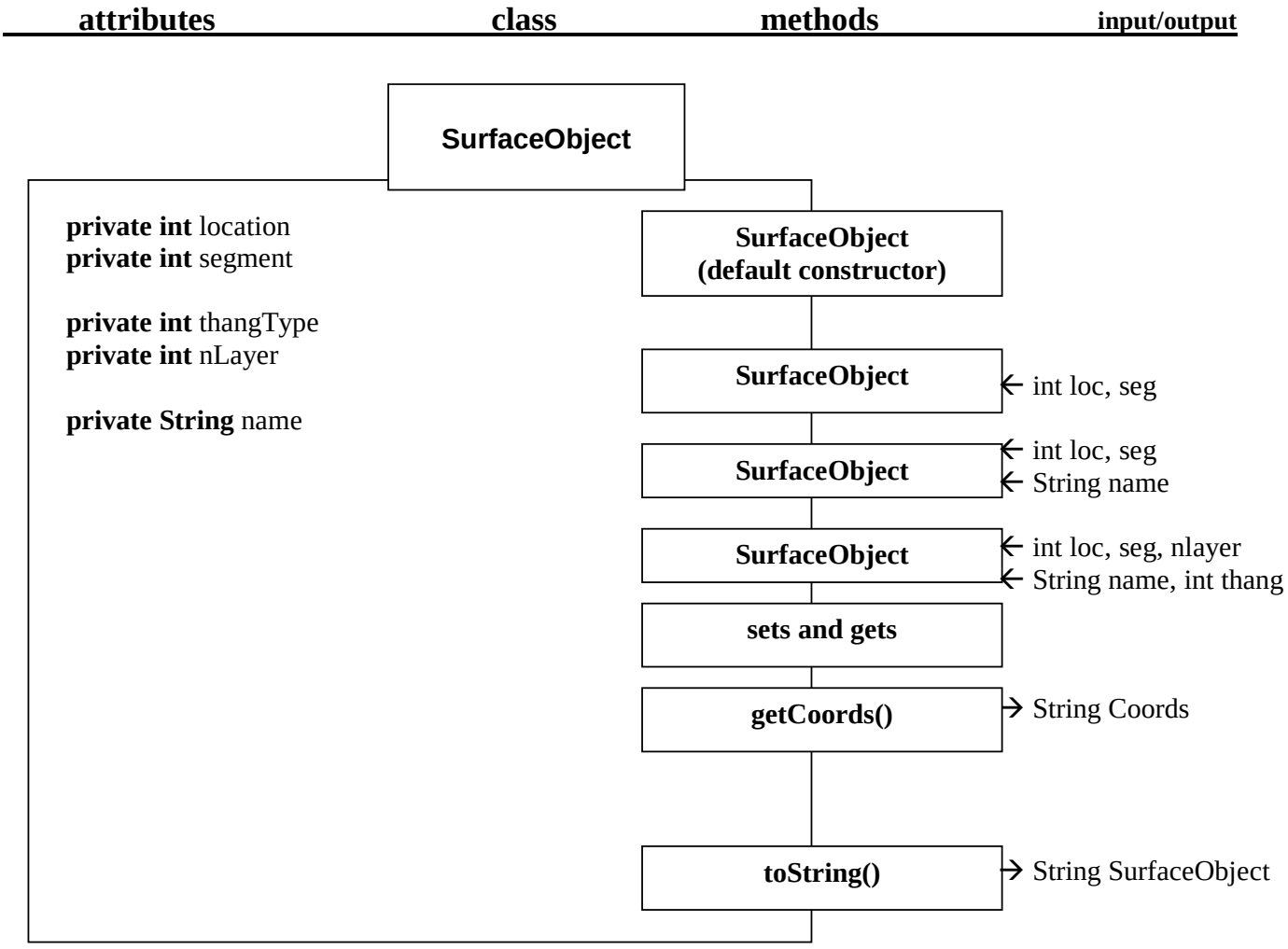
- int segments – each location can be partitioned again into a number of segments. This integer represents the number of segments that may be attached to a certain location.

Methods

- AddToSurface – adds a SurfaceObject to theList – passed with an explicit location and segment to place the object.
- AddToSurface – adds a SurfaceObject to theList – passed with only a location and is added to theList at the specified location and at the next available segment.
- getByName – passes in a string as a parameter and returns the surface object with that name.
- returnByLocation – passes an integer corresponding to the location of the desired surface object. This method is useful when trying to get the labware of a certain location.
- returnLabWareVector – returns a Vector, consisting of all the labware held on the surface. Useful when populating experiment plates.
- toString() – returns the toString of all the materials contained in the surface. Compatable for use in the configuration file (JOB0.txt).

2.4.17 SurfaceObject Class

Module Name : SurfaceObject
Type of Module : Data Structure
Module Description : A class that represents a type of surface object that can be placed in the surface. SurfaceObject is to be the super class of several other types of more specific surface object.



Constructors

- Default Constructor – takes no parameters.
- 2nd Constructor – takes two parameters, loc and seg, which indicate where on the surface the surface object is residing.
- 3rd Constructor – takes three parameters, loc and seg (described above) and name, which is the name taken from a subclass to indicate the nature of the Surface object.
- 4th Constructor – takes five parameters, to initialise all of the class attributes.

Attributes

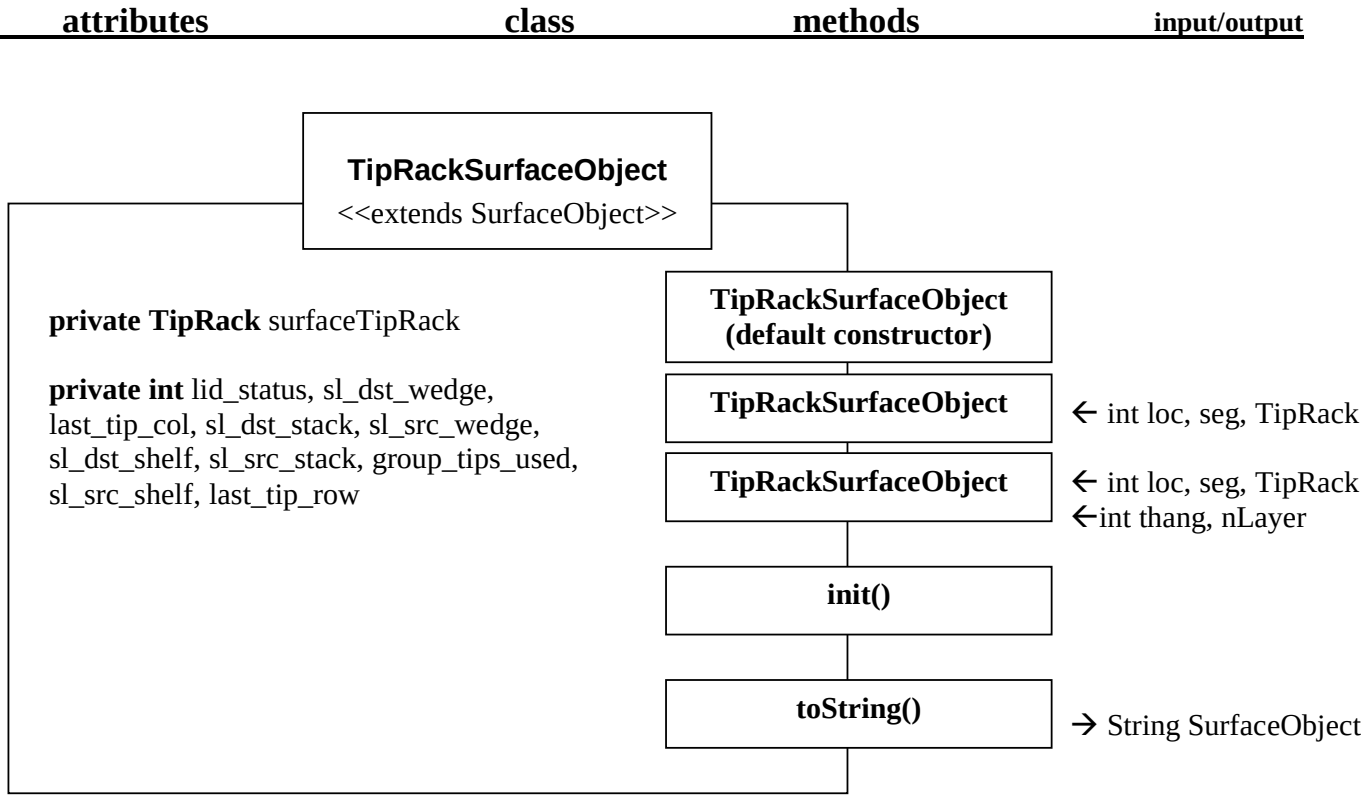
- location – indicates the location of the surface object within the surface. For details of the nature of location, please refer to 2.4.16.
- segment – indicates the segment of the surface object within the location in the surface. For details of the nature of location and segment, please refer to 2.4.16.
- thangType and nLayer are two common attributes for surface objects. For details, please refer to the BioScript Pro programmers's guide.

Methods

- getCoords – returns a formatted string representing the coordinates of the surface object. This method is primarily used in the construction of JOB1.txt and is called from within LiquidTransfer.
- toString() – returns the toString of the surface object. Compatable for use in the configuration file (JOB0.txt).

2.4.18 TipRackSurfaceObject Class

Module Name : TipRackSurfaceObject
Type of Module : Data Structure
Module Description : A class that represents a tip rack as a surface object. It extends SurfaceObject.



Constructors

- Default Constructor – calls the super default constructor and init().
- 2nd Constructor – takes three parameters, loc and seg, described in 2.4.16, and the TipRack object. This is the object that is to occupy the surface location. Calls the super constructor passing in loc and seg, and the name of the tip rack from TipRack.getName().
- 3rd Constructor – same as the second constructor, but also takes thangType and nLayer (described in 2.4.16).

Attributes

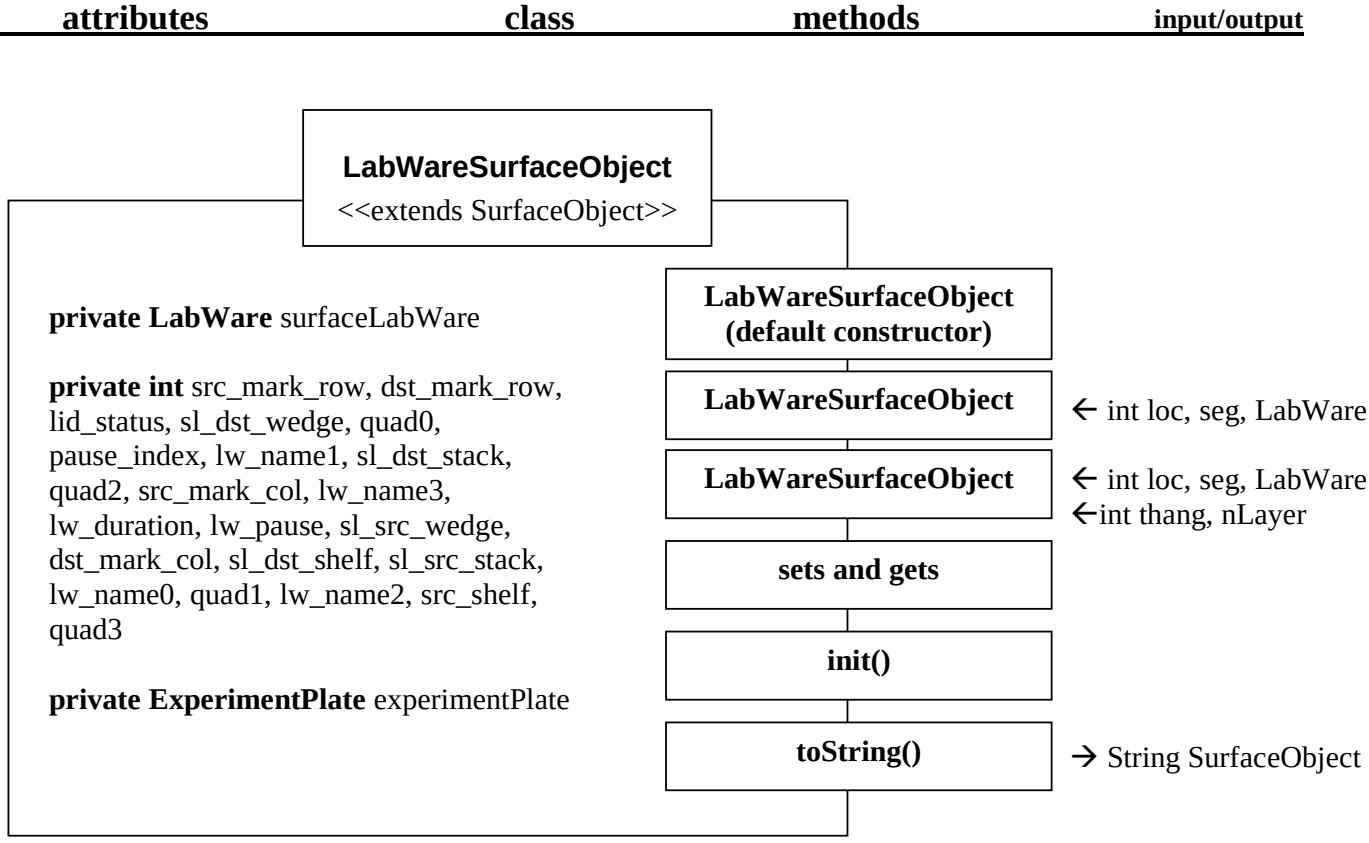
- TipRack – this is the tip rack that is to occupy the surface location represented by the SurfaceObject.
- Other surface variables – included for completeness.

Methods

- init() – initialises all the additional surface attributes.
- toString() – returns the toString of the surface object. Compatable for use in the configuration file (JOB0.txt).

2.4.19 LabWareSurfaceObject Class

Module Name : LabWareSurfaceObject
Type of Module : Data Structure
Module Description : A class that represents a piece of lab ware as a surface object. It extends SurfaceObject.



Constructors

- Default Constructor – calls the super default constructor and init().
- 2nd Constructor – takes three parameters, loc and seg, described in 2.4.16, and the LabWare object. This is the object that is to occupy the surface location. Calls the super constructor passing in loc and seg, and the name of the tip rack from LabWare.getName().
- 3rd Constructor – same as the second constructor, but also takes thangType and nLayer (described in 2.4.16).

Attributes

- surfaceLabWare – this is the lab ware that is to occupy the surface location represented by the SurfaceObject.
- experimentPlate – this is the experiment plate that is to reside within the labware.

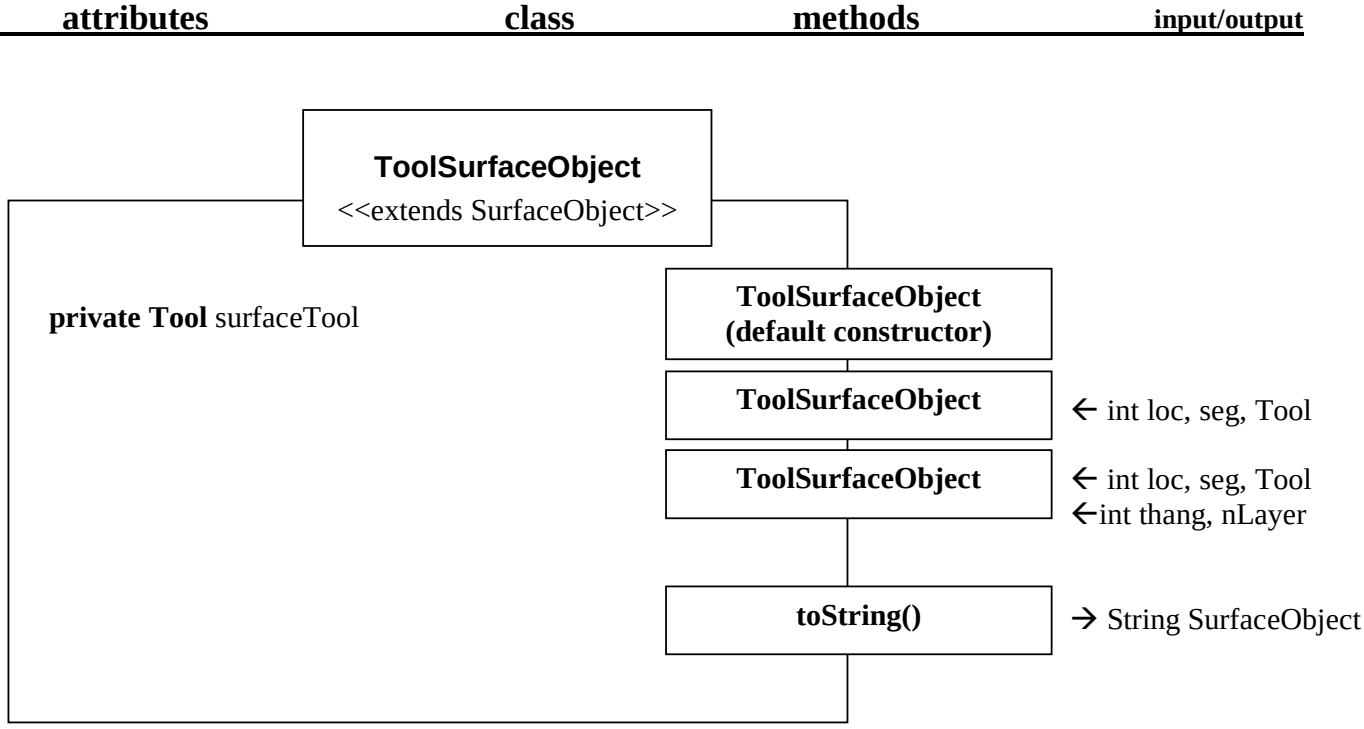
- Other surface variables – included for completeness.

Methods

- init() – initialises all the additional surface attributes.
- toString() – returns the toString of the surface object. Compatable for use in the configuration file (JOB0.txt).

2.4.20 ToolSurfaceObject Class

Module Name : ToolSurfaceObject
Type of Module : Data Structure
Module Description : A class that represents a tool as a surface object. It extends SurfaceObject.



Constructors

- Default Constructor – calls the super default constructor.
- 2nd Constructor – takes three parameters, loc and seg, described in 2.4.16, and the Tool object. This is the object that is to occupy the surface location. Calls the super constructor passing in loc and seg, and the name of the tool from Tool.getName().
- 3rd Constructor – same as the second constructor, but also takes thangType and nLayer (described in 2.4.16).

Attributes

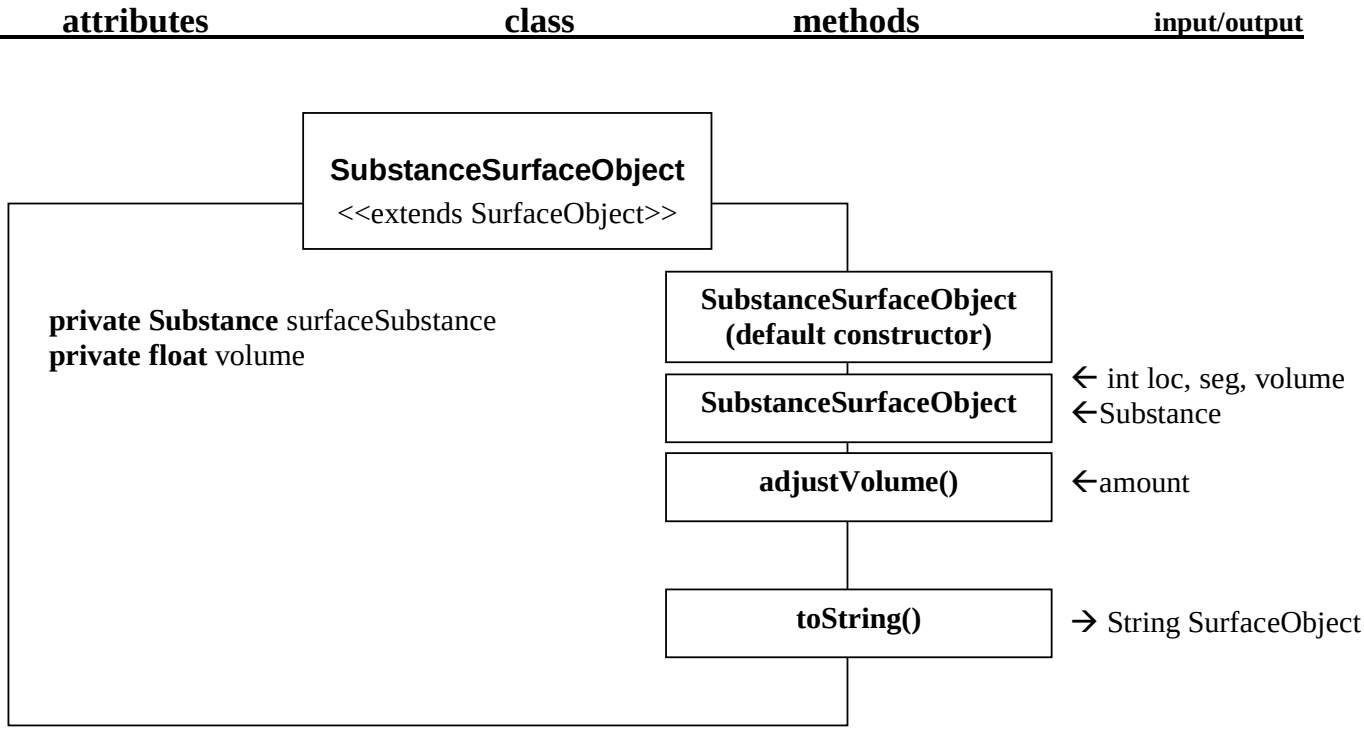
- surface Tool – this is the tool that is to occupy the surface location represented by the SurfaceObject.

Methods

- toString() – returns the toString of the surface object. Compatable for use in the configuration file (JOB0.txt).

2.4.21 SubstanceSurfaceObject Class

Module Name : SubstanceSurfaceObject
Type of Module : Data Structure
Module Description : A class that represents a substance as a surface object. It extends SurfaceObject.



Constructors

- Default Constructor – calls the super default constructor.
- 2nd Constructor – takes four parameters, loc and seg, described in 2.4.16, and the Substance. This is the object that is to occupy the surface location. Calls the super constructor passing in loc and seg, and the name of the substance from surfaceSubstance.getName().

Attributes

- surface Tool – this is the tool that is to occupy the surface location represented by the SurfaceObject.
- volume – this indicates the volume of substance that is available. This class is primarily intended to be used to represent the substances when they form reservoirs for the creation of the experiments.

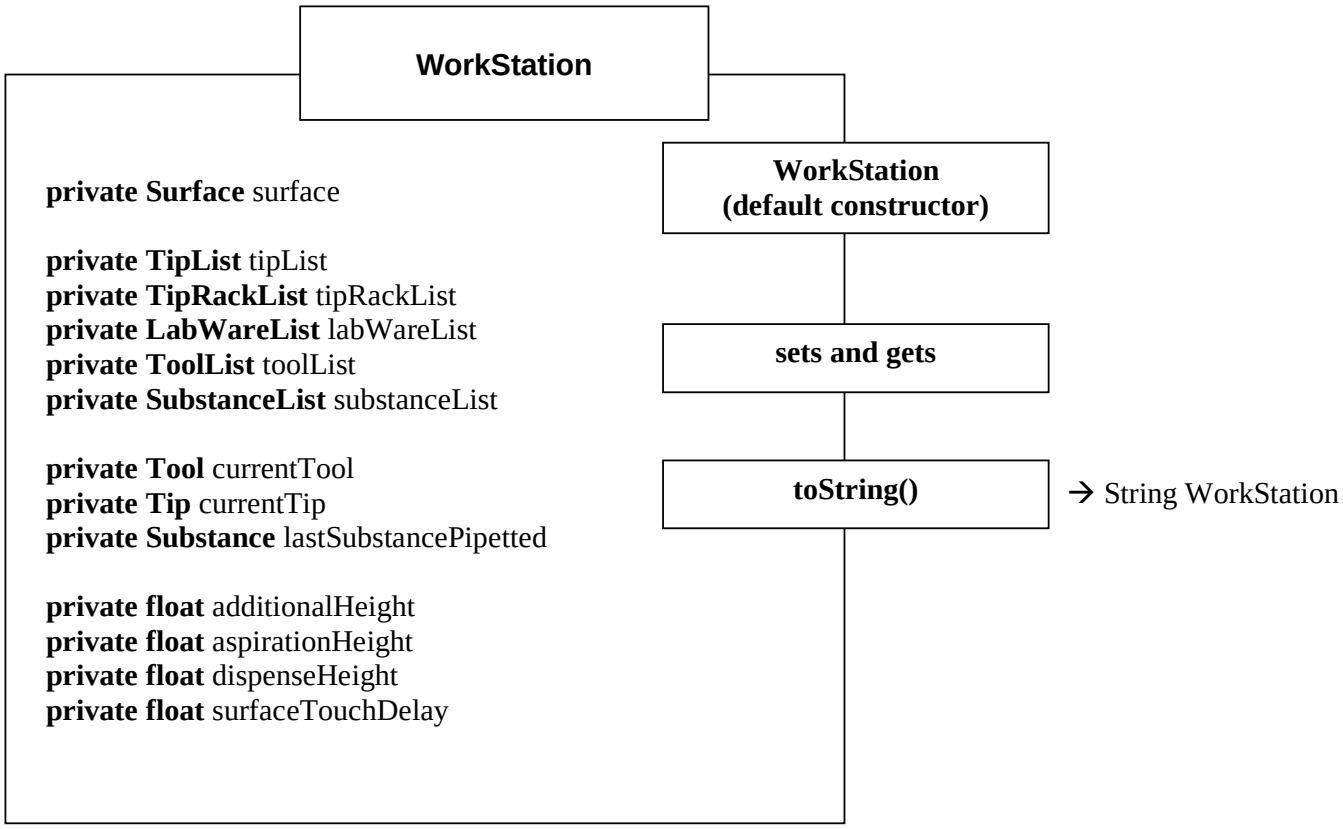
Methods

- adjustVolume() takes an amount and adjusts the volume by that amount. This method is used when keeping track of the levels of substances during the experiment creation process.
- toString() – returns the toString of the surface object. Compatable for use in the configuration file (JOB0.txt).

2.4.22 WorkStation Class

Module Name : WorkStation
Type of Module : Data Structure
Module Description : A class that represents the entire experiment environment.This class will contain details of tools, tips, tipracks, labware, substances and surface. It will also contain some parameters attributed to the current state of the robot (Biomek environment).

attributes **class** **methods** **input/output**



Constructors
- Default Constructor – takes no parameters.

Attributes
- surface – represents the surface of the BioMek work station.
- tipList – the list of tips.
- tipRackList – the list of tip racks.
- labWareList – the list of lab ware.

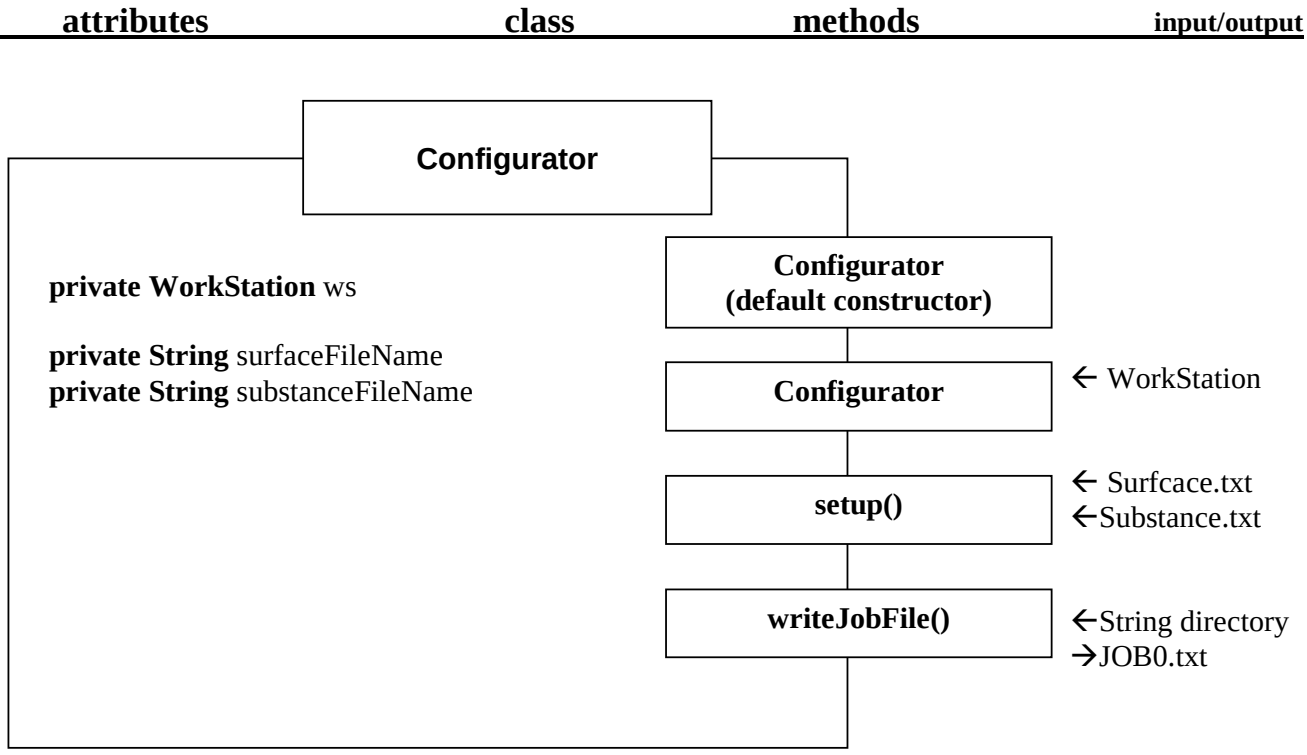
- toolList – the list of tools.
- substanceList – the list of substances.
- currentTool – the tool currently held by the arm.
- currentTip – the tip currently being used by the system.
- lasSubstancePipetted – keeps a record of the last substance to be pipetted, so that the system can tell if a tip is contaminated.
- float variables – specified by the system in RSConfig.txt.

Methods

- toString() – method that initiates the call to all attributed classes and produces a string that forms the basis of the configuration file (JOB0.txt).

2.4.23 Configurator Class

Module Name : Configurator
Type of Module : Data Builder
Module Description : This class is primarily responsible for the creation of the configuration file (JOB0.txt). It will be responsible for handling the substance and surface configuration files, setting up the substances and the surface correctly and writing JOB0.txt to a specified location.



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes a workstation as a parameter

Attributes

- ws – the workstation that is being currently used.
- surfaceFileName – the name of the surface filename (i.e., Surface.txt)
- substanceFileName – the name of the substance filename (i.e., Substance.txt)

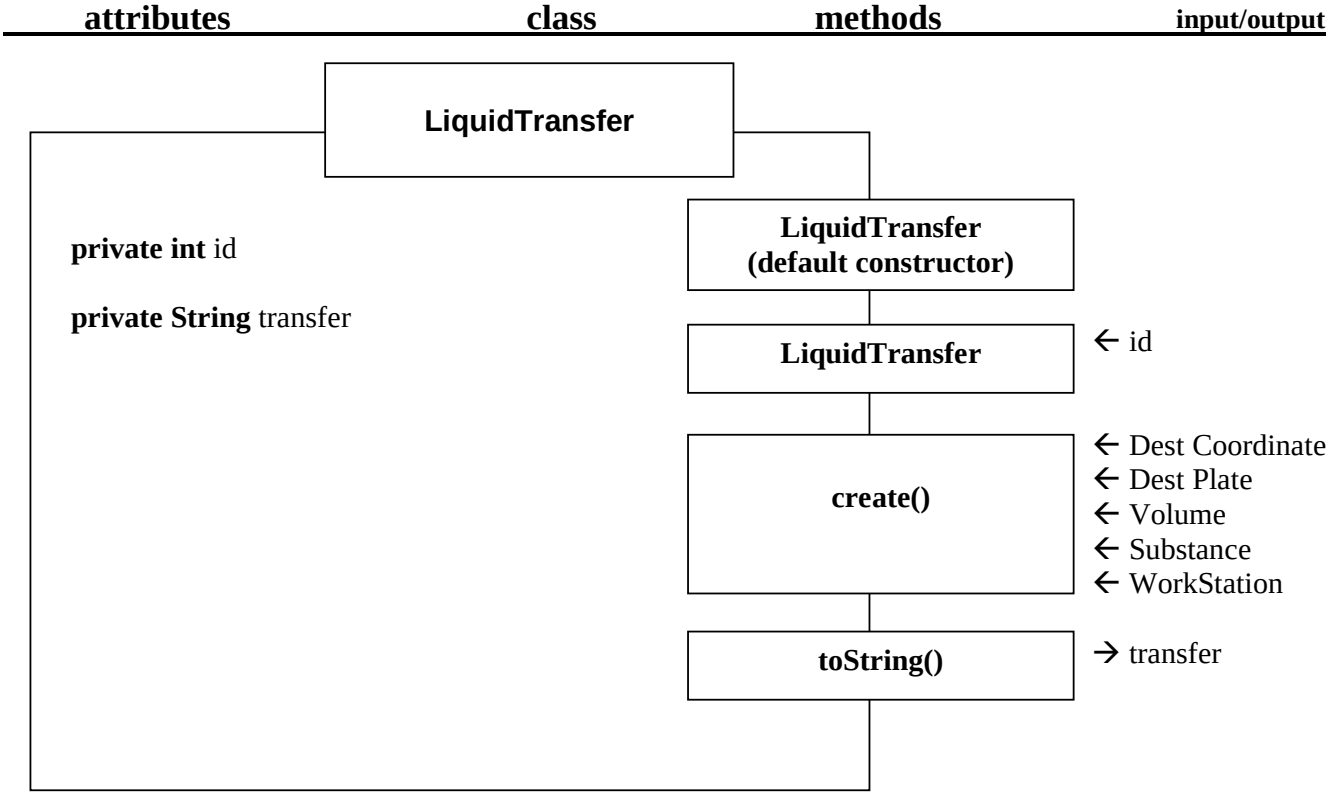
Methods

- setup( ) – This method is responsible for the intiation of the entire workstation. Currently, the details of all the tips, tip racks, devices and labware are to be located within the setup( ) method. This method will also setup the surface of the WorkStation. Details of how this is achieved along with further details of the workings of setup( ) can be found in the project report.

- writeJobFile( ) – This method is responsible for the output of the configuration file (JOB0.txt). It does this simply by calling the toString( ) method from the WorkStation object (ws) and writes the resultant string into the file, JOB0.txt (within the directory passed as a string to the method).

2.4.24 LiquidTransfer Class

Module Name : LiquidTransfer
Type of Module : Data Builder
Module Description : This class will be responsible for creating the liquid transfers that form the basis of the job file (JOB1.txt).



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes an ID as a parameter. This id will be used in the JOB file to identify a specific liquid transfer.

Attributes

- id - will be used in the JOB file to identify a specific liquid transfer.
- transfer – the string that will form the Liquid transfer and form part of the JOB file.

Methods

- create( ) – This method is responsible for the creation of the liquid transfer. Parameters that specify the transfer are passed into the method, as well as the workstation that is to host the transfer. The algorithm for the create method can be seen below. This algorithm was created with the use of two major sources. There was a liquid transfer algorithm in the oringinal system, which can be seen in [31]. There is also a rough guide for a liquid transfer contained in [4].

Variables Needed:

1. Source
2. Destination
3. Substance
4. Amount

Start:

1. Find the appropriate tool based on the amount to be transferred.
2. Attach the tool.
3. Attach the tip for that tool.
4. Move the tool to the Source.
5. Check that there is enough substance at the source.
6. If a prewet is needed:
    a. Suck some air in.
    b. Move tip into the substance.
    c. Suck up some substance.
    d. Move tool up a bit.
    e. Push out the substance.
    f. Push out the air.
7. If a blowout is needed:
    a. Suck some air in. (?)
8. Suck up the appropriate amount of substance.
9. Move the tool up.
10. Move the tool the to the destination.
11. Check that there is enough room in the destination.
12. Move the tool down to an appropriate height.
13. Push the substance out.

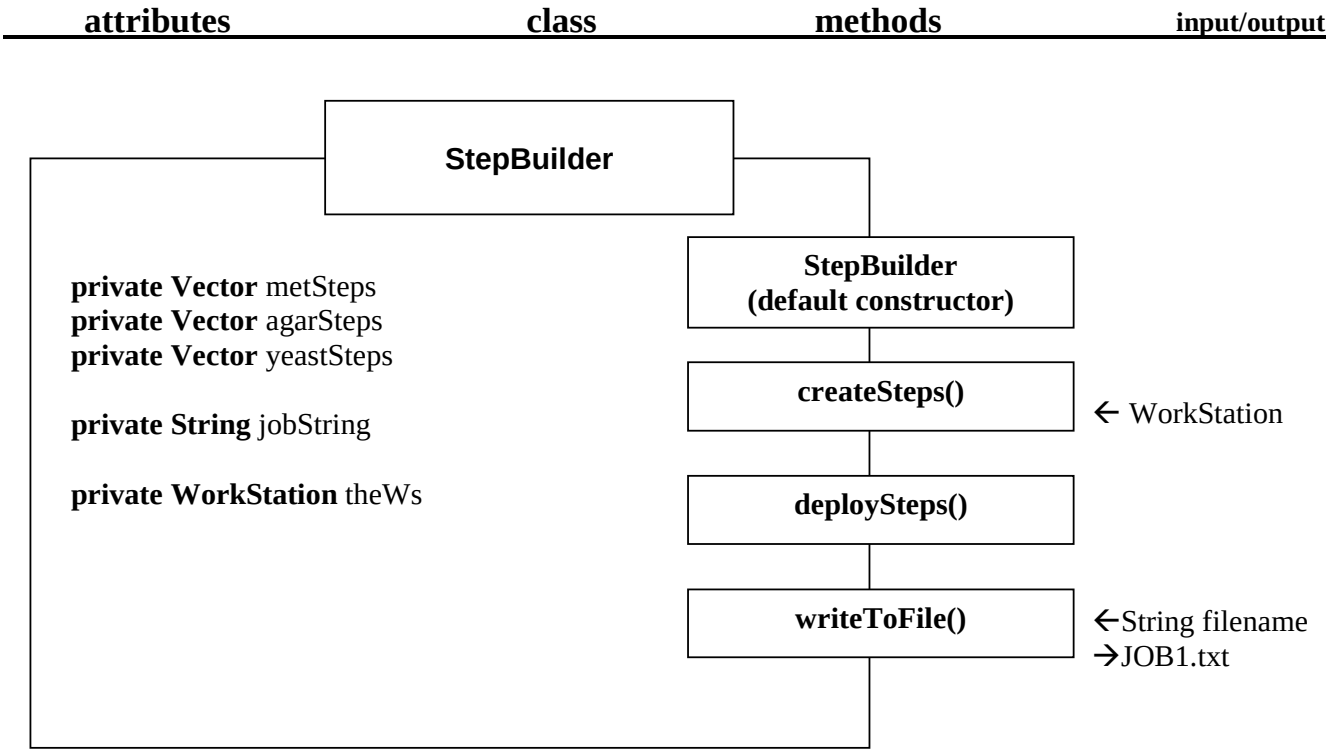
```
14.   Move the tool back up again.

15.   Push all the air out of the tool.
```

- toString( ) – during the create( ) method, the string ‘transfer’ is created and appended. When the liquid transfer algorithm (seen above) has completed, then the toString( ) of the LiquidTransfer object is passed back, returning the transfer string.

2.4.25 StepBuilder Class

Module Name : StepBuilder
Type of Module : Data Builder
Module Description : This class isolates all the steps (liquid transfers) needed to create the experiment platform, separates the steps into various vectors based on their substance and then executes a liquid transfer for each step in a pre-defined order.



Constructors

- Default Constructor – takes no parameters. Initialises the vectors metSteps, agarSteps, and yeastSteps. Creates vectors with an intitial size of 30 elements with a growth factor of one element.

Attributes

- metSteps, agarSteps, and yeastSteps – the steps (liquid transfers) that constitute the experiment.
- jobString – the string that will represent the whole of the job file (JOB1.txt)
- theWs – the work station that is to host the experiments.

Methods

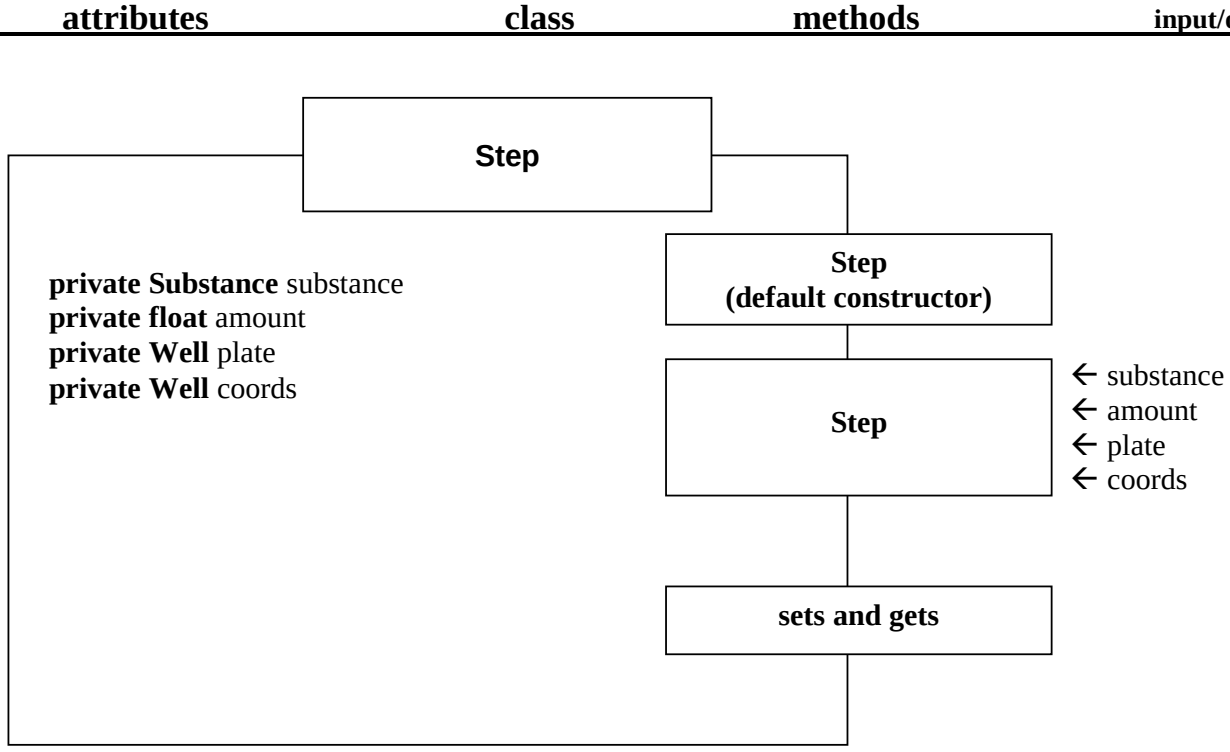
- createSteps( ) – this method iterates over every single susbtance for every well contained within the experiment. A step is created for each of these substances and placed in the appropriate vector depending on whether the substance is agar, yeast or another metabolite.

- deploySteps( ) – this method takes the step vectors one-by-one (in a designated order) and creates the liquid transfers for each one at a time, passing and appending the resulting liquid transfer string to jobString.

- writeToFile( ) – write the contents of jobString to the designated file (JOB1.txt).

2.4.26 Step Class

Module Name : Step
Type of Module : Data Structure
Module Description : This class represents a single step (liquid transfer) in the experiment process.



Constructors

- Default Constructor – takes no parameters.
- Alternative Constructor – takes and intialises four parameters – substance, amount, plate, coords.

Attributes

- subtance – the substance that is to be transferred.
- amount – the volume of substance that is to be transferred.
- plate – the destination plate of the transfer.
- coords – the destination coordinates of the transfer.

3. ANALYSIS OF SUBSYSTEMS

The following section will describe the various subsystems that comprise the TCL compiler system. The subsystems will be described in turn until a picture of the whole system is built up, whereupon an analysis of the whole system will be provided. It is always difficult to know exactly at what level to break up a system into subsystems. A subsystem (or package) is usually characterised by the set of services it provides. A subsystem can of course contain other subsystems.

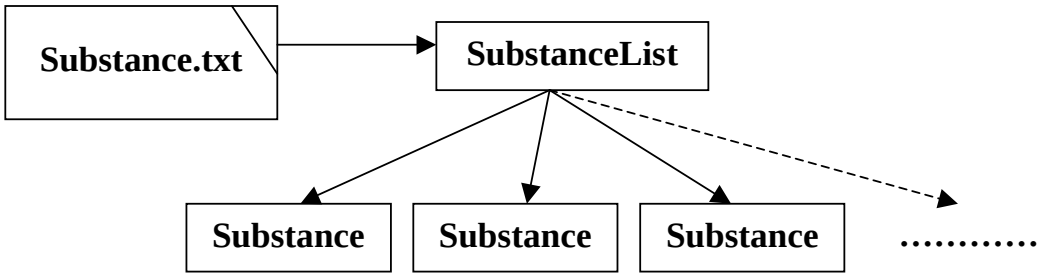
3.1 The Experiment Creation Subsystem

The experiment subsystem is primarily concerned with the creation of a set of experiment plates, within those plates a set of experiments and within those plates, a set of substances. The experiment subsystem can be divided into two subsystems: the Subtance subsystem and the Experiment subsystem. These are described below.

3.1.1 The Substance Subsystem

3.1.1.1 Overview

The substance subsystem is concerned with the creation and instantiation of the collection of substances that are to be used in the experiments. A diagrammatic representation of the substance subsystem is shown below.



As can be seen in the above diagram, the substance subsystem consists of the classes, SubstanceList, Substance and the text file Substance.txt.

3.1.1.2 Functionality

The function of the substance subsystem is to read a list of substances from the text file, Substances.txt, and create the appropriate amount of substances, held in an array within SubstanceList, and populate the Substance objects with the data from Substances.txt. The method populateSubstances() in SubstanceList will tokenize the file Substances.txt and create an array of Substances in accordance with that file.

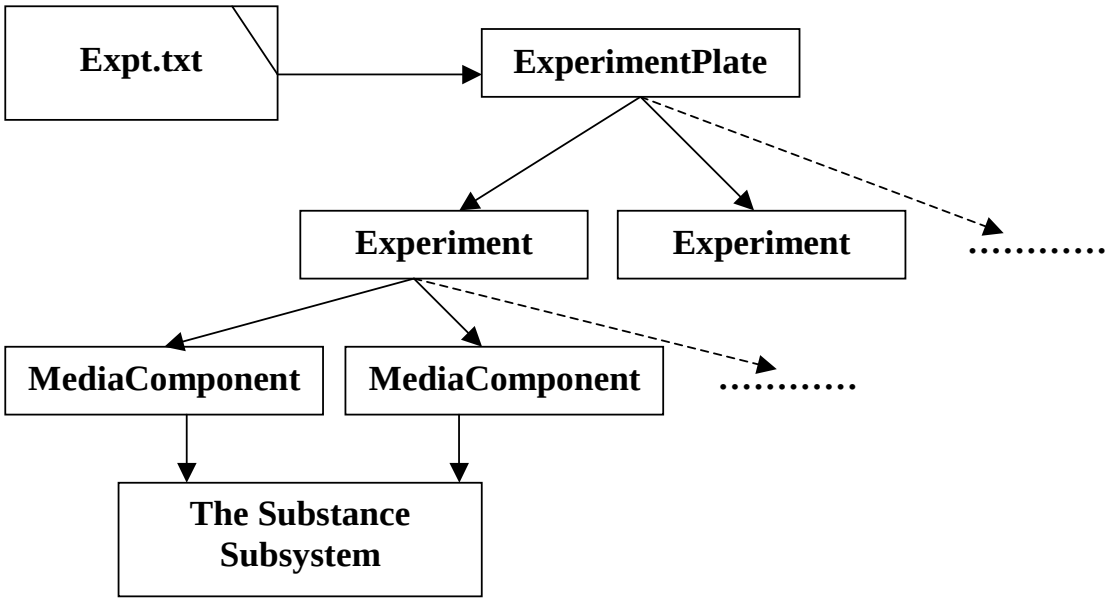
3.1.1.3 Dependencies

The substance subsystem is connected with the experiment subsystem alone and provides access to substances that are used in the experiment subsystem. The operation of the substance subsystem requires the availability of the Substances.txt file.

3.1.2 The Experiment Subsystem

3.1.2.1 Overview

The experiment subsystem is concerned with the creation of the sets of experiments and, as such, is primarily designed around the implementation of the experiment data file, namely Expt.txt. A diagrammatic representation of the experiment subsystem is shown below.



ExperimentPlate contains an array of Experiments. Experiment contains an array of MediaComponents. A MediaComponent consists of a substance and an amount.

3.1.2.2 Functionality

The populateExperimentPlate() method in the class ExperimentPlate, will tokenize the file Expt.txt and create an array of Experiments in accordance with that file.

3.1.2.3 Dependencies

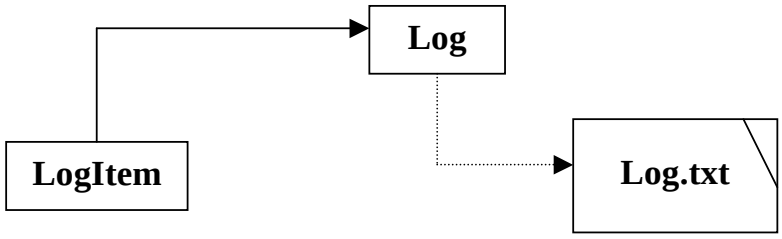
To date, there is only a dependency with the Substance subsystem (co-dependency). There will undoubtedly be other dependencies.

3.2 The Log Subsystem

The log subsystem is concerned with the creation and maintenance of a log with regard to the TCL compiler in the creation of the experiment process. When a significant event occurs, the log subsystem will be used to make a note of the event, along with any other information that may be considered important, such as possible errors and faults detected at that point. The log is an important part of the system as it allows the user to follow the trace of the program whilst it is running. Were the system to fail, there would be a record of exactly (or approximately) where it failed.

3.2.1.1 Overview

The log subsystem is constructed with two modules. These are Log and LogItem. The diagram below illustrates the relationship between the two modules.



3.2.1.2 Functionality

When something is to be added to the log, an object of type LogItem is created. LogItem simply contains a formatted log string with the description of the log entry embedded. The LogItem is added to the Log. At certain time intervals (namely at program completion, or a halt), the Log writes its contents (all the LogItems) to a file, here named ‘Log.txt’.

3.2.1.3 Dependencies

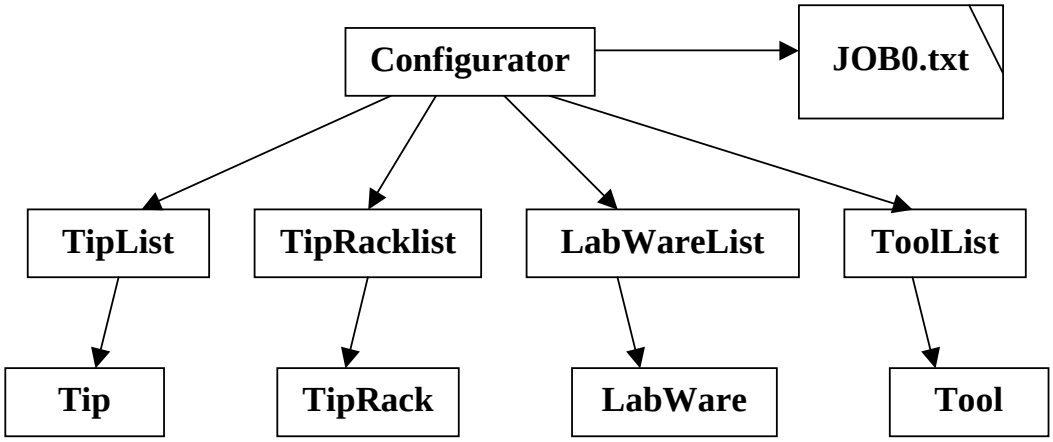
The Log subsystem has no external dependencies.

3.3 The Configurator Subsystem

The main function of the Configurator subsystem is the preparation and construction of the configuration file, namely JOB0.txt. There are other functions. The Configurator subsystem is responsible for setting up all the variables for the devices employed throughout the experimentation process. These classes and objects are therefore used elsewhere in the system as will be detailed accordingly.

3.3.1.1 Overview

The following diagram illustrates how the Configurator subsystem is constructed. The Configurator susbystem deals with all the available devices and tools that can be used in the experimentation process. It does not contain information as to what devices and tools are actually being used in the current experiment implementation.



3.3.1.2 Functionality

The Configurator subsystem is responsible for the construction of the configuration job file (JOB0.txt). To do this, the data for the different devices, labware and tools for the experiment must be available. In this implementation, this data is supplied in Configurator. The Configurator creates and intitialises each piece of labware, device and tool one at a time, with the variables expicitly defined within the code. The Configurator controls the variables for the WorkStation upon which it was called for.

3.3.1.3 Dependencies

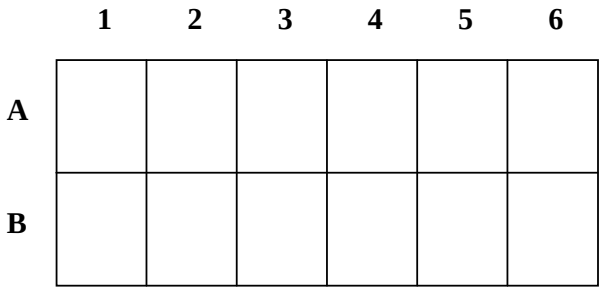
To be announced…

3.4 The Surface Subsystem

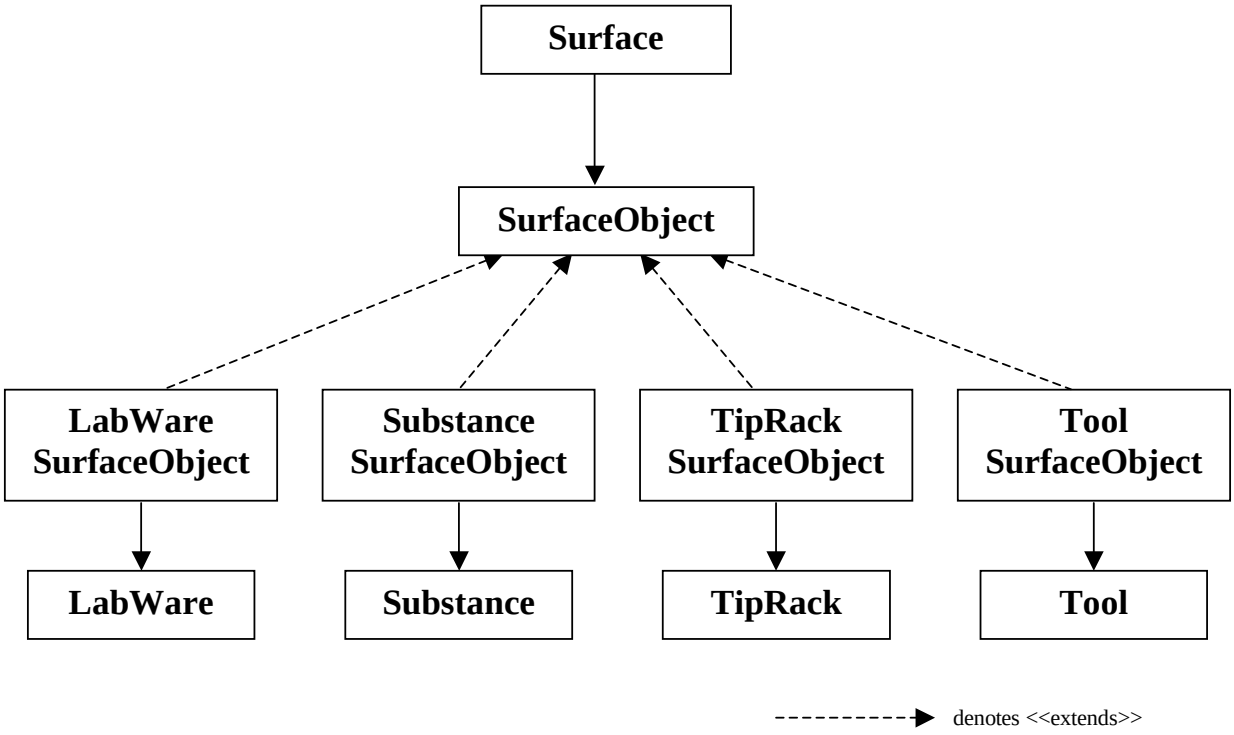
The surface subsystem deals with the initialising and maintaining of the surface of the biomek workstation. It will contain the information on exactly what devices and tools are being used in the current experiment implementation.

3.4.1.1 Overview

The biomek workstation has a set of 12 “workspaces”. These workspaces can be again split into a number of separate areas. There are some restrictions as to what can be placed where. For example, the optical density reader (VICTOR) must be situated in the workspace, designated as A1. The diagram below illustrates the entire workspace of the Biomek environment.



The system must know the exact contents of each of these workspaces and this is the purpose of the Surface subsystem. The diagram below illustrates the structure of the Surface subsystem.



3.4.1.2 Functionality

Surface contains a 2D array of SurfaceObjects. This 2D array should represent the surface of the Biomek environement (as displayed on the previous page). Four classes, LabWareSurfaceObject, SubstanceSurfaceObject, TipRackSurfaceObject, and ToolSurfaceObject, all extend SurfaceObject.

It is necessary to initialise the surface for two main reasons. Firstly, the configuration file (JOB0.txt) contains the details of the surface configuration so that the Biomek workstation knows the locations of all its equipment. Secondly, when constructing the liquid transfers for the job file (JOB1.txt), the system must know the surface configuration in order to construct the details of the liquid transfers. I.e., where to get the substance from, where to move the robot handler to, etc.

3.4.1.3 Dependencies

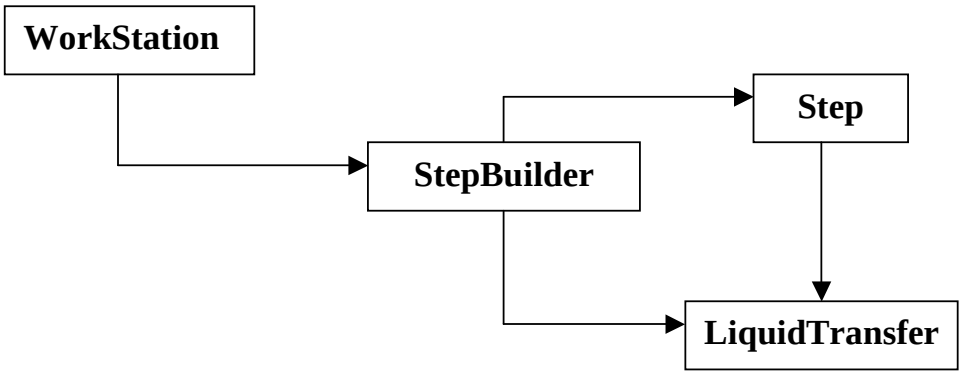
To be announced.

3.5 The Builder Subsystem

The Builder subsystem is responsible for the construction of the job file (JOB1.txt). Within the Builder subsystem is the class WorkStation. This module contains the representation of the entire Biomek environment.

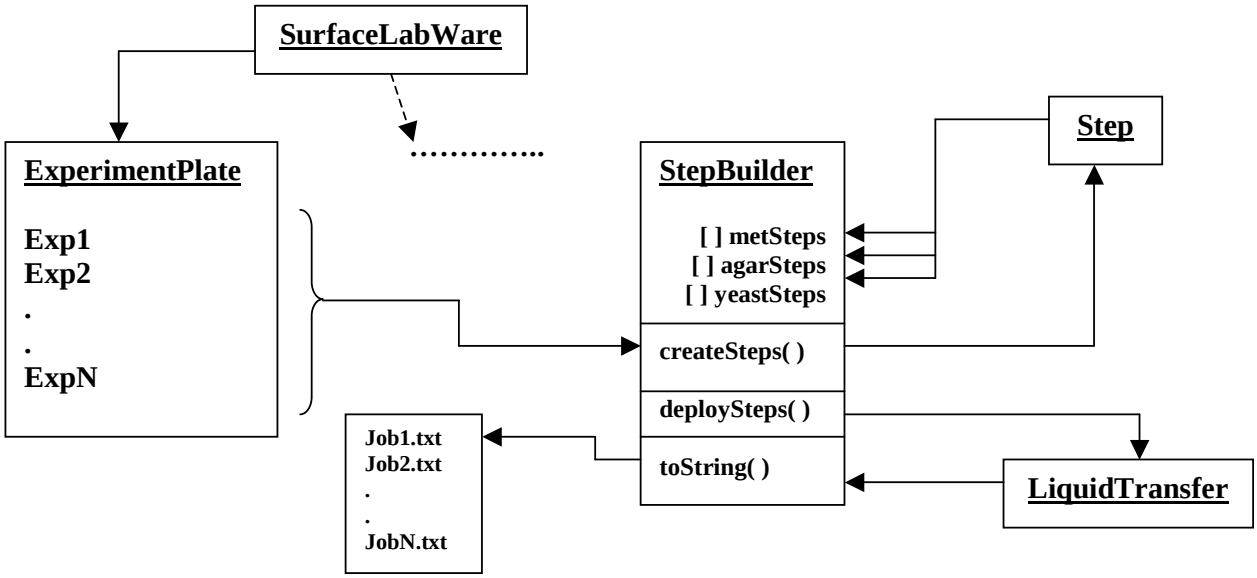
3.5.1.1 Overview

The Builder subsystem functions by passing in all the experiment plates (labware), identifying all of the different substances used in each well of each plate and separating all of these into steps. The result is an amount of steps that can be segregated according to their type (whether they are metabolite, yeast or agar steps).



3.5.1.2 Functionality

The functionality of the Builder subsystem is illustrated in the diagram below.



createSteps( ) identifies each piece of labware located on the surface of the current work station. It parses through each experiment of each experiment plate of each piece of labware, identifying each liquid transfer and creating a Step object for that transfer and storing it in the appropriate Vector (according to whether it is a metabolite, agar or yeast step).

deploySteps( ) passes each step (in a designated order) to the create( ) method in LiquidTransfer. Here, the liquid transfer is created as a set of instructions, and passed back and appended to a job string contained within StepBuilder. After deploySteps( ) has completed, the job String is passed to an appropriate job file. In this way, the job files are constructed.

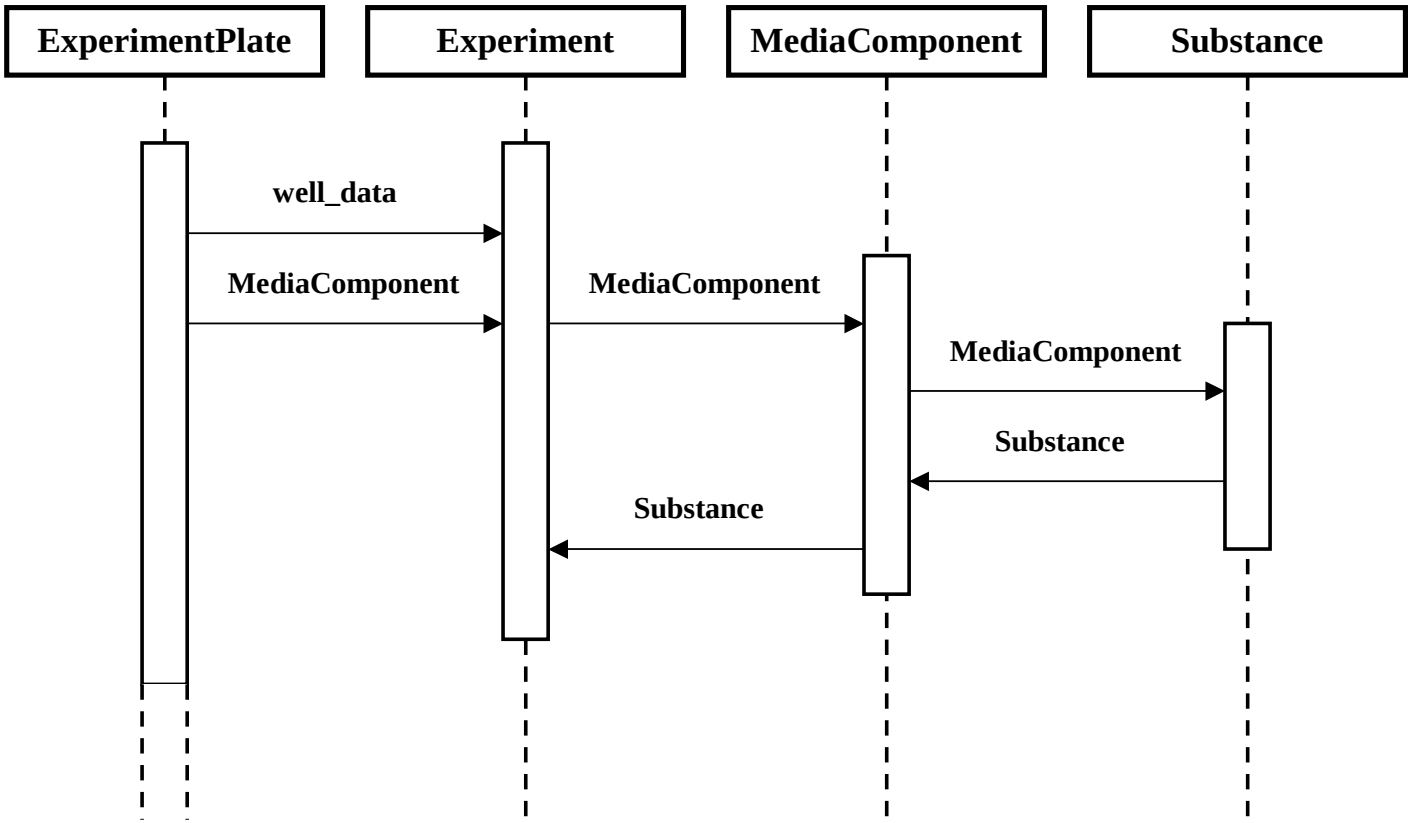
3.5.1.3 Dependencies

To be announced

4. SUBSYSTEM SEQUENCE DIAGRAMS

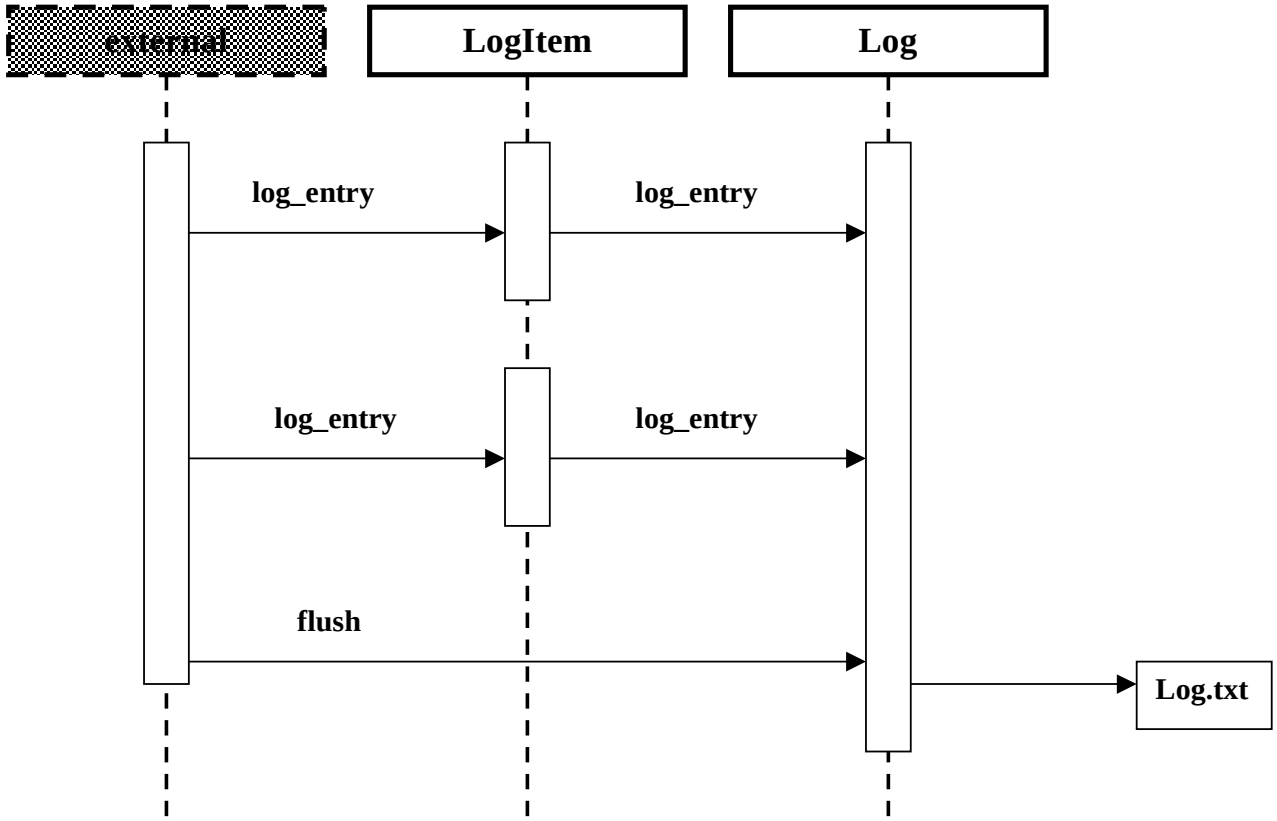
The following section contains sequence diagrams for each of the five subsystems.

4.1 The Experiment Creation Subsystem

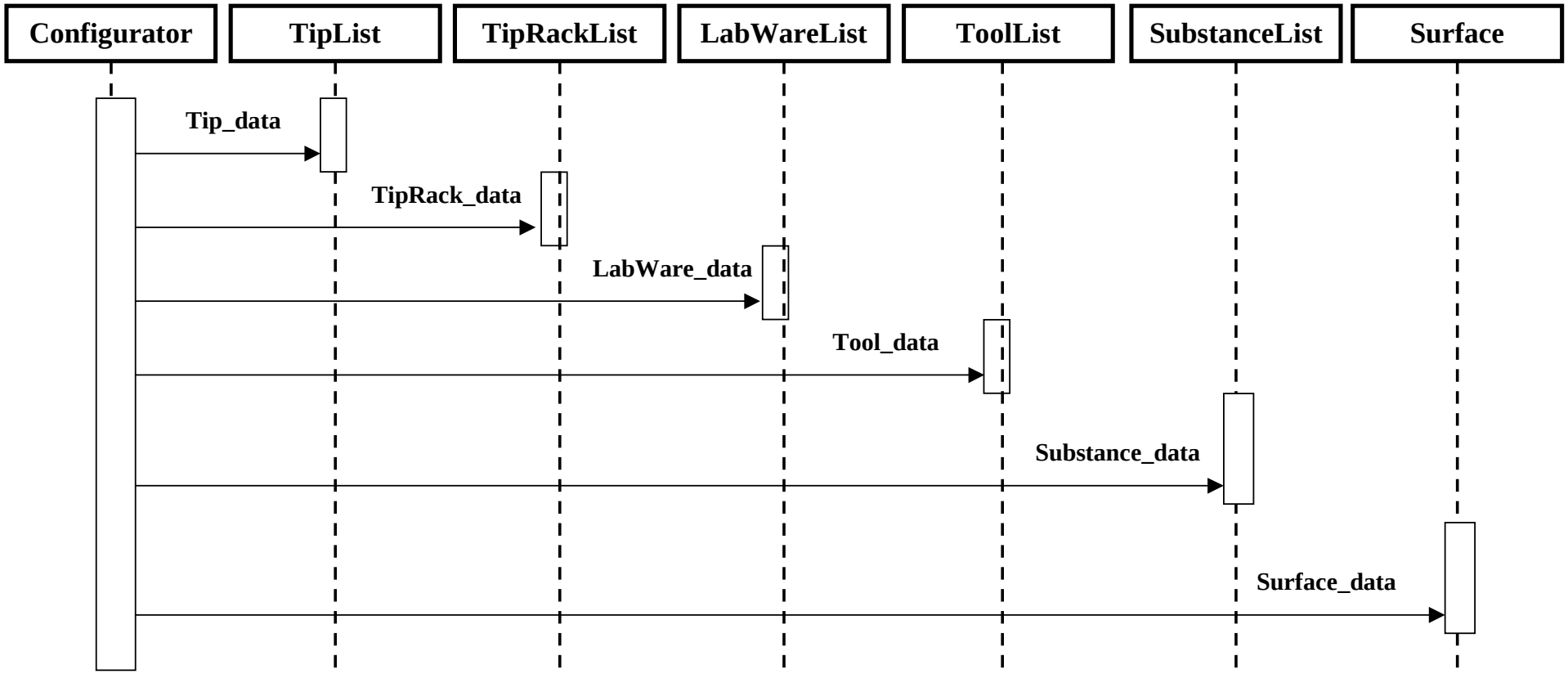


4.2 The Log Subsystem

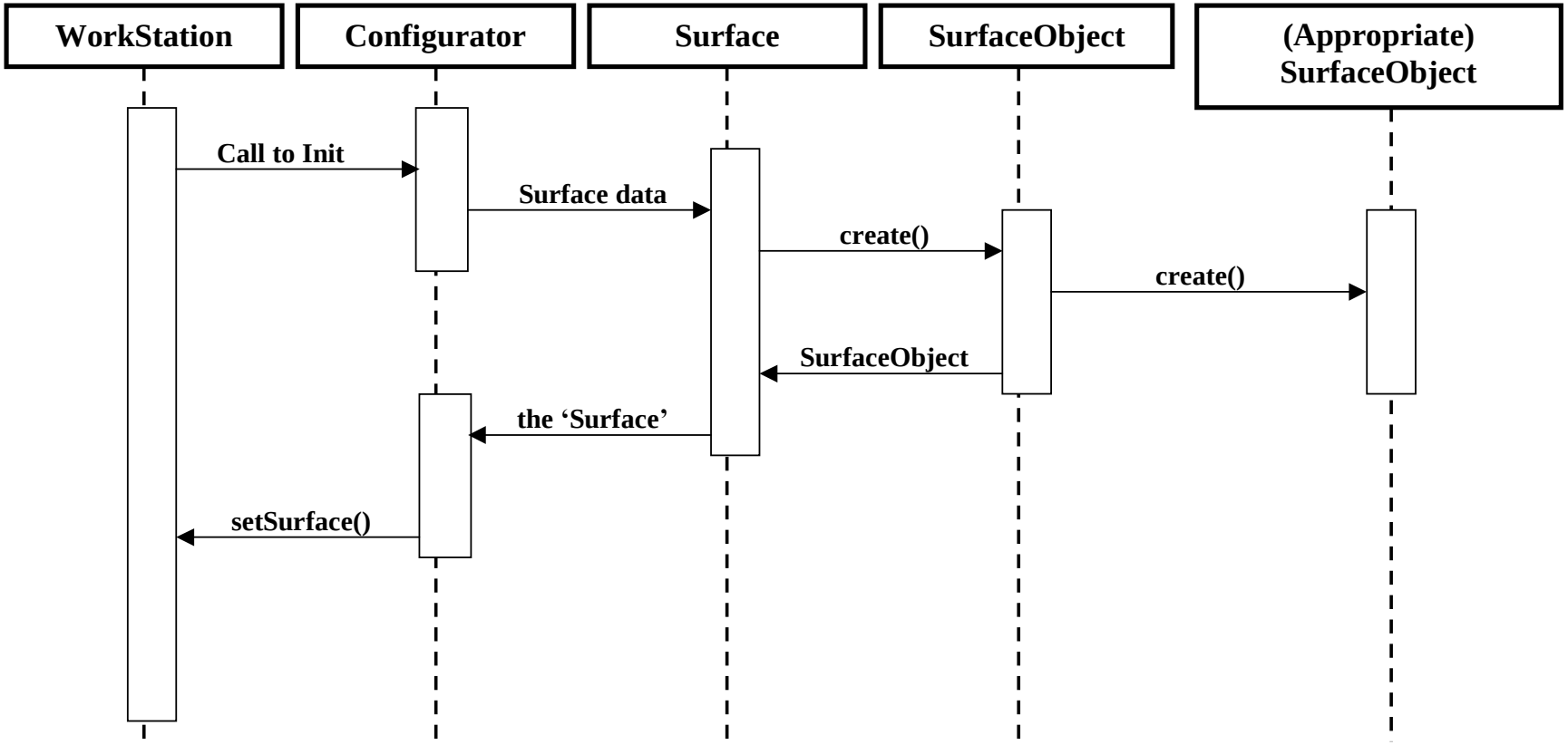
Showing two complete log entries.



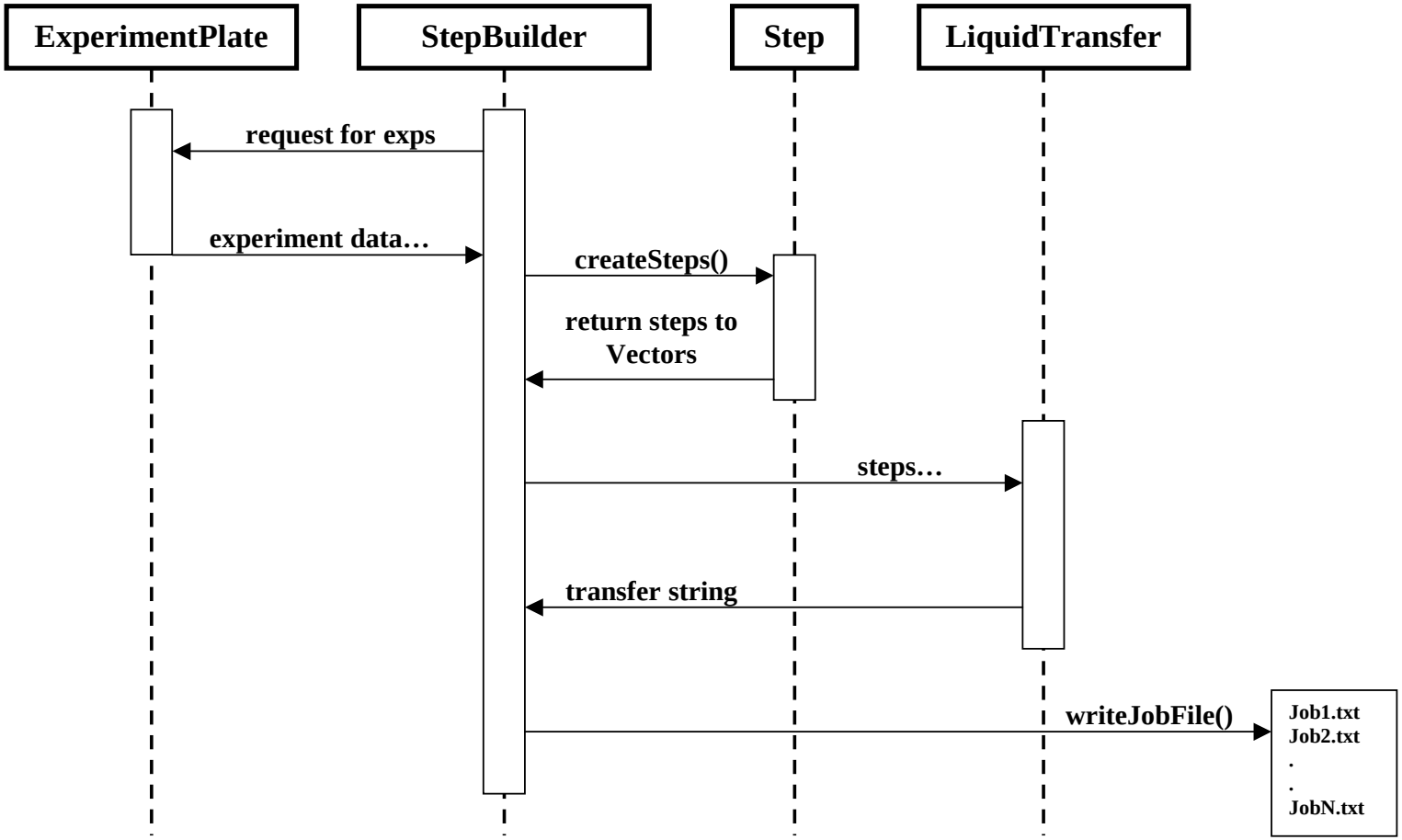
4.3 The Configurator Subsystem



4.4 The Surface Subsystem



4.5 The Builder Subsystem

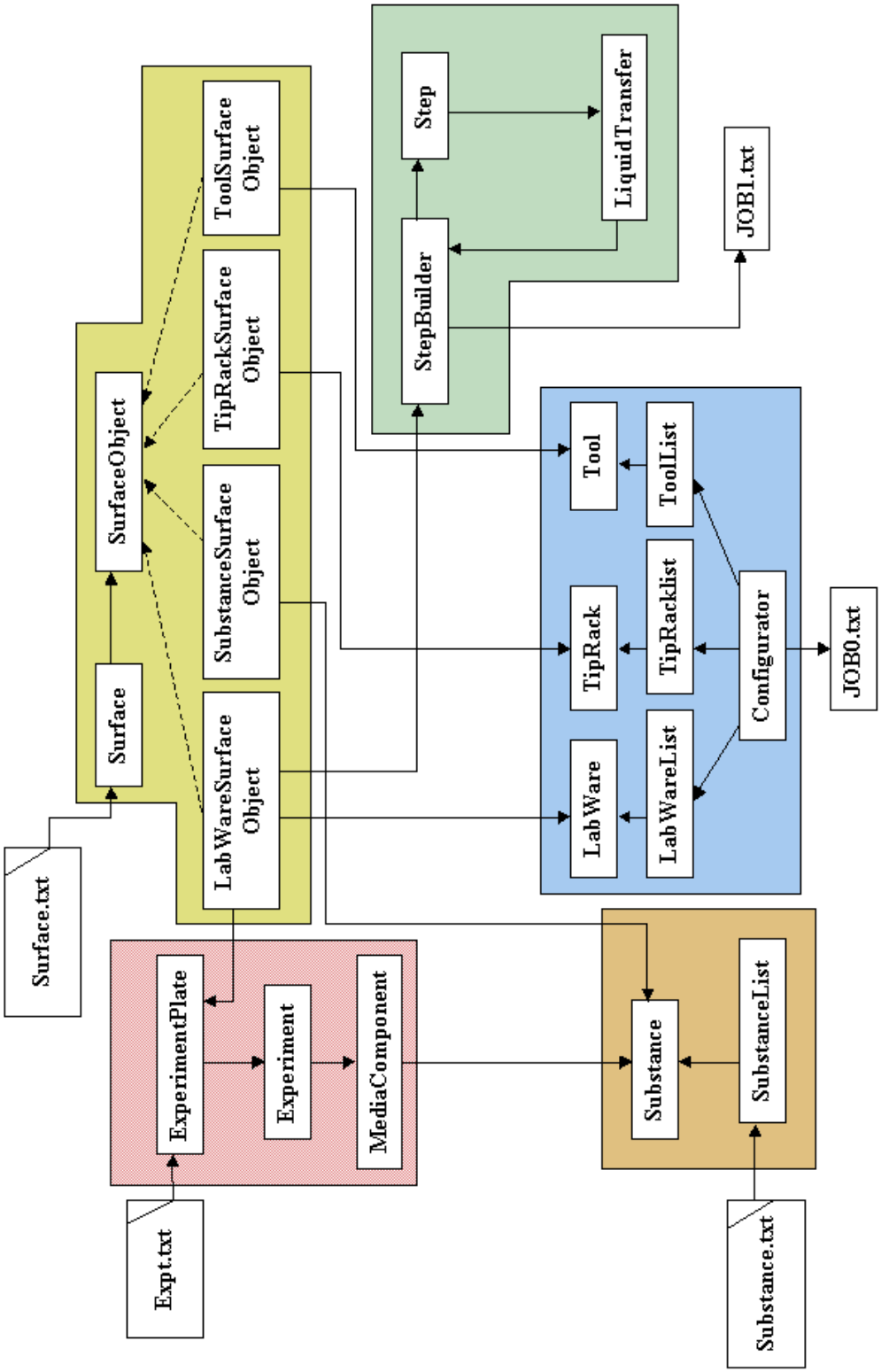


5. TCL JOB COMPILER

5.1 Overview

The following is a diagram of the entire TCL Job Compiler. It shows the subsystems connected to each other and attempts to display the flow of data within the system. As the system becomes more complex, it becomes more difficult to adequately represent the system graphically. The following diagram has been simplified in terms of the level of communication that is expected to occur between the subsystems. However, given the constraints of the graphical representation, there is an ample level of detail.

5.2 Illustration



5.3 Description of TCL Job Compiler

Colours have been used to signify the subsystem boundaries. These are:

- Red – Experiment Subsystem
- Brown – Substance Subsystem
- Yellow – Surface Subsystem
- Blue – Configurator Subsystem
- Green – Builder Subsystem

The following is a step-by-step guide to the intended operation of the TCL Job Compiler. It was necessary to simplify some of the processes at this stage in order retain an understanding of the system.

1. A WorkStation is initialised, using the default constructor.

2. A Configurator object is created, passing to it a reference to the WorkStation.

3. The setup( ) method from Configurator is called, passing to it the substance and surface filenames.

    a. TipList, TipRackList, LabWareList and ToolList are initialised and added to the WorkStation.
    b. SubstanceList is initialised and the array of substances is populated by the method, populateSubstances( ). The SubstanceList is then added to the WorkStation.
    c. The Surface is initialised and the 2-dimensional array is created by parsing the surface file. The surface is then added to the WorkStation.

4. The method, writeJobFile( ) is called passing in the directory in which to place the job file. This method calls the toString( ) method for the WorkStation and writes the result to a file called JOB0.txt in the specified directory.

5. A Vector containing the labware (i.e., the ExperimentPlates) is returned from the WorkStation.

6. For each experiment plate, the mehod, populateExperimentPlate( ), is called. This populates the ExperimentPlate with Experiments in accordance with the file Expt.txt. The ExperimentPlates are added to the relevant LabWareSurfaceObjects.

7. The substances from each experiment are passed into an object of type Step. This will contain data such as: Substance, amount, source and destination.

8. The Steps are sorted into Vectors by the type of Substance that is being transferred. These are metabolites, agar and yeast.

9. The Vectors are passed into deploySteps( ) one-at-a-time in a specific order (metabolites, agar and then yeast).

10. deploySteps( ) creates a LiquidTransfer object for each Step and executes the create( ) method for the Step. The toString( ) of the LiquidTransfer is captured and appended to a string, maintained within StepBuilder.

11. When all the LiquidTransfers have completed, the string is outputted to a file called JOB1.txt.

BASIC TESTING PLAN

Author: Ben Tagger (bnt8)
Date: 14/01/2003
Version: 2.0
Status: Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to outline the basic testing strategy to be followed throughout the course of this project. This document will not provide an exhaustive outline, rather a tactical plan of the general approach to be adopted.

1.2 Scope

This document is intended to provide guidelines for the planning and executing of the testing of the TCL Compiler system and subsystems.

1.3 Objectives

This document aims to describe the processes to be employed throughout the testing of the TCL compiler system. The document should provide a basic guideline for the construction of individual test cases. The test cases may differ in accordance with the application. Various strategies are to be employed, including: black box (input/output) testing, path testing and white box (structural) testing. The objectives of testing will include the location and removal or masking of as many errors as possible, to ensure correct operation and compliance with the system requirements. One of the objectives of the testing process is to determine the causes of as many errors as possible and to possibly suggest some solutions.

2. LEVELS OF TESTING

Testing will usually occur on various levels in order to ensure thorough testing of the whole system at all levels of operation. These levels of testing are not mutually exclusive and all relate to one another. Therefore, it is necessary that testing occur in an explicit manner, however it is important that testing occur in a general sequence. This section endeavours to describe the testing processes that are to be followed during the course of this project.

2.1 Unit Testing – 1st Level

This level of testing involves testing the individual components, ensuring that they meet their design specification. This level of testing can be carried out during the implementation phase and will not require the use of formal testing plans or specifications. Some standard testing strategies should be adhered to in this process.

2.2 Module Testing – 2nd Level

The level of module testing will involve the testing of a group of related components in isolation. Again, this level of testing can be carried out during the implementation phase and does not require the use of formal testing plans or specifications. Some standard testing strategies should nevertheless be adhered to during this process.

2.3 Subsystem Testing – 3rd Level

This level of testing will involve producing a test specification for the subsystem, establishing a series of test cases with their expected test results. This plan is then to be followed precisely. Separate interfaces may be used in order to test the various subsystems, as there may not be adequate built-in interfaces.

2.4 Testing against Requirements – 4th Level

This level will test the overall level of functionality of the whole system, as well as the non-functional aspects of the system, such as the speed and safety of the system. The safety can be weighed against a safety specification (not yet supplied) and the speed can be ascertained through assessment by benchmarking.

2.5 Acceptance Testing – 5th Level

The final stage of testing will involve integrating and testing the new system within the overall project environment. In this case, the environment will be the Progol experimentation environment. The TCL compiler will take in configuration files produced by the ASE-Progol system and operate correctly to produce the job files to be used by the Biomek workstation. Acceptance testing will involve the correct operation of these processes in an adequately safe, fast and informative manner.

3. TEST CASES

Each test case will consist of the input data, output data, and behaviour of any given test run. During formal testing, a test case must be established before it is executed. The test case will be selected by the user/developer, who will select appropriate data based on the type of testing that is to occur.

4. ERROR, BUG AND PROBLEM TYPES

Errors, faults and bugs can arise in programs for many different reasons and from many different sources. The purpose of the testing phase is to uncover as many of these as possible and to suggest possible solutions. This section will describe some of the different types of problems typically encountered in a project.

4.1 Typographical Errors

A missing semicolon or uneven brackets can cause significant program failures with repercussions throughout the entire system. Errors, such as these are invariably present in all code and can be detected adequately using static testing. These errors are generally found and corrected during compilation in either the implementation or testing phases.

4.2 General Coding/Syntactic Errors

These errors include things such as having ‘for’ loops in the wrong place or that can never terminate, or an ‘if’ statement that can never be reached. Calling methods with the wrong parameters will also cause problems. Again, these problems can normally be detected at compilation, however some will not. i.e., a ‘for’ loop that will never terminate.

4.3 Communication/Interfacing Errors

Communications between classes, modules, subsystems, systems, computers and users can often cause problems due to errors in the code. These can often be difficult to find and even more difficult to provide adequate solutions for. For the TCL compiler system, parsing the configuration files may cause problems, when attempting to tokenise the strings within the text files.

4.4 Design Flaws

Flaws that lie within the project may be the result of an inadequate or erroneous design. In some cases as this, the design phase must be revisited to improve or modify the design as is befitting.

5. TYPES OF TESTING

There are five types of testing described below here. All are applicable for use during the testing phase for this project. However to what extent each is to be used has yet to be established.

5.1 Static Testing

This would usually primarily involve code walkthroughs. However, due to the nature of the project and the fact that there is only one developer (designer, tester, etc.), walkthroughs may not be worthwhile or even possible. Even though there is only one person involved in the project, some aspects of walkthroughs can still be observed. For example, the code can still be examined as a series of paths and the most likely paths of the system can be ascertained during this process.

5.1.1 Suitability of method

Static testing is usually the first method of testing to be employed during a project and can be useful in spotting syntactical and typographical errors.

5.2 Black box testing

Black box testing can be used as a follow up process to the static testing to further explore possible paths through the system. Black box testing can be useful when the tester knows the function of the component with respect to the operations on data inputs, but is unsure as to how the component functions on the inside.

5.2.1 Suitability of method

Black box testing is most suitable for testing top-level systems.

5.3 White box (Structural) testing

White box testing (glass box testing) is carried out when the inner workings of a component are known and the test cases can be constructed with respect to this knowledge. The tester uses knowledge about the structure of the component to derive test data and test cases.

5.3.1 Suitability of method

White box testing allows the tester to use many of the possible paths through a component, rather than simply the paths that are most likely to be used. It is not possible to use every possible path, but a subset of test cases can be established given the unit's function and likely problems.

5.4 Interface testing

Interface testing is concerned with the communications between modules, subsystems and subsystems. It is primarily concerned with the errors encountered during these periods of communication and aims to monitor and improve the way that these components co-operate.

5.5 Regression testing

The fixing and removing of bugs and errors can introduce new errors into the code and the design. Therefore, the testing process must be iterative and repeated until the system is adequately error-free.

6. DOCUMENTATION OF RESULTS OF TESTING

At this time, it is unclear exactly how the results of the testing phase are to be documented. It is not clear to what depth the testing results are to be described within the 'story' report at this time. At this point in time, it seems likely that only the most prudent areas of testing completed will be documented within the final report.

SUBSYSTEM TEST SPECIFICATION

Author: Ben Tagger (bnt8)
Date: 28/02/2003
Version: 3.0
Status: Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to identify the items that are to be tested and the tasks to be performed. This document is concerned with the testing of the subsystems of the TCL Job Compiler. A basic testing plan for the overall system can be found in [33]. Each subsystem must be clearly understood so that this testing strategy can be carried out in a correct fashion. A detailed analysis of the design of the subsystems can be found in [32].

1.2 Scope

This document aims to set out clearly and precisely the items that are to be tested and the approach to the subsystem testing. It should provide a more specific test plan than [33]. The idea is that this document is derived from [32] in response to the design outline of the subsystems and overall system. This document also makes the test specification for the system possible. Although there is no formal time constraint for the completion of subsystem testing, it is obvious that this part must be completed in due time to allow sufficient time for system testing.

1.3 Objectives

The objective of this document is to provide a plan of testing to ensure that all TCL Job Compiler subsystems fulfil the design and requirement specifications in order to minimise errors. Criteria that the TCL Job Compiler must meet are listed in section 3 of this document

2. TEST ITEMS

A test item can be thought of as a subsystem of the TCL Job Compiler. This may be composed of one or more classes. The following section endevours to identify and describe the test items that are to be dealt with in this document.

2.1 Subsystems

The following subsystems represent a section of the TCL Job Compiler, which, where possible, can be tested independently from the rest of the system. Listed below are the subsystems.

2.1.1 The Substance Subsystem

The substance subsystem is concerned with the creation and instantiation of the collection of substances that are to be used in the experiments. The substance subsystem contains the following classes:

- Substance class – [32], section 2.4.1
- SubstanceList class – [32], section 2.4.3
- The substance subsystem also employs use of the substance file (Substance.txt).

2.1.2 The Experiment Subsystem

The experiment subsystem is concerned with the creation of the sets of experiments and, as such, is primarily designed around the implementation of the experiment data file, Expt.txt. The experiment subsystem contains the following classes:

- Experiment class – [32], section 2.4.4
- ExperimentPlate class – [32], section 2.4.5
- MediaComponent class – [32], section 2.4.2
- Expt.txt – the file that contains the details of the experiments.

2.1.3 The Log Subsystem

The log subsystem is concerned with the creation and maintenance of a log with regard to the operation of the TCL Job Compiler in the creation of the experiment process. When a significant event occurs, the log subsystem will be used to make note of the event, along with any other points of information that may be important. The Log subsystem contains the following classes:

- LogItem class - [32], section 2.4.6
- Log class – [32], section 2.4.7
- Log.txt – the text file that the log will be kept.

2.1.4 The Configurator Subsystem

The configurator subsystem is concerned with the preparation and construction of the configuration file, namely JOB0.txt. The configurator subsystem is also responsible for the setting up of all the devices and tools that can be used in the experiment process. The configurator subsystem contains the following classes.

- Configurator class – [32], section 2.4.23
- Tip class – [32], section 2.4.8
- TipList class – [32], section 2.4.12
- TipRack class – [32], section 2.4.9
- TipRackList class – [32], section 2.4.13

- LabWare class – [32], section 2.4.10
- LabWareList class – [32], section 2.4.14
- Tool class – [32], section 2.4.11
- ToolList class – [32], section 2.4.15
- JOB0.txt - the text file that contains the configuration data for the Biomek workstation.

2.1.5 The Surface Subsystem

The surface subsystem deals with the initialising and maintaining of the surface of the Biomek workstation. It will handle information on exactly what devices and tools are being used in the current experiment implementation. The surface subsystem contains the following classes.

- Surface class – [32], section 2.4.16
- SurfaceObject class – [32], section 2.4.17
- LabWareSurfaceObject class – [32], section 2.4.19
- SubstanceSurfaceObject class – [32], section 2.4.21
- TipRackSurfaceObject class – [32], section 2.4.18
- ToolSurfaceObject class – [32], section 2.4.20

2.1.6 The Builder Subsystem

The Builder subsystem is responsible for the construction of the job file (JOB1.txt). Within the builder susbystem is the class WorkStation. This module contains the representation of the entire Biomek environment. The builder subsystem contains the following classes.

- WorkStation class – [32], section 2.4.22
- StepBuilder class – [32], section 2.4.25
- Step class – [32], section 2.4.26
- LiquidTransfer –[32], section 2.4.24

3. FEATURES TO BE TESTED

The subsystem testing process must ensure that the software is functional, reasonably error-free, and complies with the design specification. The following section will identify the features of each subsystem that are to be tested.

3.1 Features of the Substance Subsystem

Design specification [32] references for detailed module descriptions.

- Substance class – section 2.4.1
- SubstanceList class – section 2.4.3

3.1.1 Compliance with design specification

3.1.1.1 Substance

The Substance must:

- Contain the appropriate variables.
- Provide an adequate toString( ) method for a formatted representation of the substance.

3.1.1.2 SubstanceList

The SubstanceList must:

- Hold an array of Substance.
- Provide methods for maintaining the array.
- Provide a method for reading from a file (Substance.txt) and populating the array.

3.1.1.3 Substance and SubstanceList

The Substance and SubstanceList classes must communicate with each other.

3.1.1.4 Input/Output

The subsystem should be able to deal with the file input of the substances text file. This should include abnormal file detection, missing file detection and other IO errors.

3.1.2 Handling of expected parameters

Substance has two constructors; a default constructor and an alternative constructor that takes eight parameters (1 string, 4 integers, 4 booleans). SubstanceList also has two constructors; a default constructor and an alternative constructor that takes one parameter (an integer) to establish the size of the array. These constructors must all function with the passing of normal paramaters in standard order.

- Correct storage of variables.
- The SubstanceList should hold an array of Substances.

3.1.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.
- Test that the subsystem can deal with large and zero integers.
- Test the correct handling of a large or zero array of Substance.

3.1.4 Handling of invalid parameters

The Substance subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.
- Correct handling of non-boolean variables passed in where booleans are expected.
- Correct handling of non-integer variables passed in where integers are expected.
- Correct handling of parameters passed in an unexpected order.
- Correct handling of non-Substance items being added to the array (such as Strings).

3.2 Features of the Experiment Subsystem

The experiment subsystem contains the following classes:

- Experiment class – [32], section 2.4.4
- ExperimentPlate class – [32], section 2.4.5
- MediaComponent class – [32], section 2.4.2

3.2.1 General Compliance with the design specification

3.2.1.1 Experiment

The Experiment must:

- Hold an array of MediaComponent.
- Provide a method for adding a MediaComponent to the array.
- Provide an adequate toString( ) method for a formatted representation of the Experiment.

3.2.1.2 ExperimentPlate

The ExperimentPlate must:

- Hold an array of Experiment.
- Provide methods for maintaining the array.
- Provide a method for reading from a file (Expt.txt) and populating the array.

3.2.1.3 MediaComponent

The MediaComponent must:

- contain an instance of a Substance as a class attribute.

3.2.1.4 Experiment and ExperimentPlate

The Experiment and ExperimentPlate classes must communicate with each other.

3.2.1.5 Input/Output

The subsystem should be able to deal with the file input of the experiments text file. This should include abnormal file detection, missing file detection and other IO errors.

3.2.2 Handling of expected parameters

Experiment has two constructors; a default constructor and an alternative constructor that takes three parameters (3 integers). ExperimentPlate has three constructors; a default constructor and two alternative constructors; one that takes two parameters (2 integers), the other takes four parameters (4 integers). These constructors must all function with the passing of normal paramaters in standard order.

- Correct storage of variables.
- The Experiment should hold an array of MediaComponent.
- The ExperimentPlate should hold an array of Experiment.

3.2.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.
- Test that the subsystem can deal with large and zero integers.
- Test the correct handling of a large or zero array of Substance.

3.2.4 Handling of invalid parameters

The Experiment subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.
- Correct handling of non-boolean variables passed in where booleans are expected.
- Correct handling of non-integer variables passed in where integers are expected.
- Correct handling of parameters passed in an unexpected order.
- Correct handling of non-Experiment or non-MediaComponent items being added to the arrays (such as Strings).

3.3 Features of the Log Subsystem

The Log subsystem contains the following classes:

- LogItem class - [32], section 2.4.6
- Log class – [32], section 2.4.7

3.3.1 General Compliance with the design specification

3.3.1.1 LogItem

The LogItem must:

- Contain a string that will represent the log entry.

3.3.1.2 Log

The Log must:

- Write to the log file.
- Maintain a string that will represent the entire log and be able to append items (of text) to it.

3.3.1.3 LogItem and Log

The LogItem and Log classes must communicate with each other.

3.3.1.4 Input/Output

The output file should be standardised, coherent, delimited in some way.

3.3.2 Handling of expected parameters

LogItem has two constructors; a default constructor and an alternative constructor that takes one parameters (String). Log has two constructors; a default constructor and an alternative constructor that takes two parameters (2 strings). These constructors must all function with the passing of normal paramaters in standard order.

- Correct storage of variables.

3.3.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.

3.3.4 Handling of invalid parameters

The subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.

3.4 Features of the Configurator Subsystem

The configurator subsystem contains the following classes.

- Configurator class – [32], section 2.4.23
- Tip class – [32], section 2.4.8
- TipList class – [32], section 2.4.12
- TipRack class – [32], section 2.4.9
- TipRackList class – [32], section 2.4.13
- LabWare class – [32], section 2.4.10
- LabWareList class – [32], section 2.4.14
- Tool class – [32], section 2.4.11
- ToolList class – [32], section 2.4.15

3.4.1 General compliance with design specification

3.4.1.1 Configurator

The Configurator must:

- Provide a method that prepares the WorkStation for the implementation of the configuration file.
- Be able to read from a file.
- Be able to write to a file.

3.4.1.2 Tip

The Tip must:

- Contain the appropriate variables.
- Provide a formatted string of a tip for the configuration file through the object's toString( ) method.

3.4.1.3 TipList

The TipList must:

- Hold an array of Tip.
- Provide a method to add a Tip to the array.
- Provide a method that returns a Tip based on its volume.
- Provide a method that returns a Tip by its name.

3.4.1.4 TipRack

The TipRack must:

- Contain the appropriate variables.

- Provide a formatted string of a TipRack for the configuration file through the object's toString( ) method.

3.4.1.5 TipRackList

The TipRackList must:

- Hold an array of TipRack.
- Provide a method to add a TipRack to the array.
- Provide a method that returns a TipRack by its name.

3.4.1.6 LabWare

The LabWare must:

- Contain the appropriate variables.
- Provide a formatted string of a LabWare for the configuration file through the object's toString( ) method.

3.4.1.7 LabWareList

The LabWareList must:

- Hold an array of LabWare.
- Provide a method to add a LabWare to the array.
- Provide a method that returns a LabWare by its name.

3.4.1.8 Tool

The Tool must:

- Contain the appropriate variables.
- Provide a formatted string of a Tool for the configuration file through the object's toString( ) method.

3.4.1.9 ToolList

The ToolList must:

- Hold an array of Tool.
- Provide a method to add a Tool to the array.
- Provide a method that returns a Tool by its name.

3.4.1.10 Communication

The classes must be able to communicate correctly between each other.

3.4.1.11 Input/Output

The subsystem should be able to deal with the file input of the experiments text file. This should include abnormal file detection, missing file detection and other IO errors. The output file should be standardised, coherent, delimited in the correct way.

3.4.2 Handling of expected parameters

The constructors of each of the modules described above must all function correctly with the passing of normal parameters in the correct order.

- Correct storage of variables.
- The arrays should be tested with respect to holding their intended objects.

3.4.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.
- Test that the subsystem can deal with large and zero integers.
- Test the correct handling of a large or zero array of the appropriate type.

3.4.4 Handling of invalid parameters

The subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.
- Correct handling of non-boolean variables passed in where booleans are expected.
- Correct handling of non-integer variables passed in where integers are expected.
- Correct handling of parameters passed in an unexpected order.
- Correct handling of non-intended items being added to the arrays (such as Strings).

3.5 Features of the Surface Subsystem

The Surface subsystem contains the following classes.

- Surface class – [32], section 2.4.16
- SurfaceObject class – [32], section 2.4.17
- LabWareSurfaceObject class – [32], section 2.4.19
- SubstanceSurfaceObject class – [32], section 2.4.21
- TipRackSurfaceObject class – [32], section 2.4.18
- ToolSurfaceObject class – [32], section 2.4.20

3.5.1 General compliance with design specification

3.5.1.1 Surface

The Surface must:

- Hold a 2-dimensional array representing the surface of the Biomek Workstation.
- Provide a method of adding SurfaceObject to the array in the correct fashion (i.e., in the correct place).

- Provide a method that returns the SurfaceObject by its name.
- Provide a method that returns the SurfaceObject by its location in the array.
- Provide a method that returns a vector consisting of all the LabWare currently residing in the surface.
- Provide a toString( ) method that is used to provide the string for the configuration file (JOB0.txt).

3.5.1.2 SurfaceObject

The SurfaceObject must:

- Contain the appropriate variables.
- Provide a method that returns the surface coordinates of the SurfaceObject.
- Provide a formatted string of a SurfaceObject for the configuration file through the object's toString( ) method.

3.5.1.3 LabWareSurfaceObject

The LabWareSurfaceObject must:

- Extend SurfaceObject.
- Contain the appropriate variables.
- Contain an ExperimentPlate.

3.5.1.4 SubstanceSurfaceObject

The SubstanceSurfaceObject must:

- Extend SurfaceObject.
- Contain the appropriate variables.
- Contain a Substance.

3.5.1.5 TipRackSurfaceObject

The TipRackSurfaceObject must:

- Extend SurfaceObject.
- Contain the appropriate variables.
- Contain a TipRack.

3.5.1.6 ToolSurfaceObject

The ToolSurfaceObject must:

- Extend SurfaceObject.
- Contain the appropriate variables.
- Contain a Tool.

3.5.1.7 Communication

The classes must be able to communicate correctly between each other.

3.5.1.8 Input/Output

The subsystem should be able to deal with the file input of the experiments text file. This should include abnormal file detection, missing file detection and other IO errors. The output file should be standardised, coherent, delimited in the correct way.

3.5.2 Handling of expected parameters

The constructors of each of the modules described above must all function correctly with the passing of normal parameters in the correct order.

- Correct storage of variables.
- The arrays should be tested with respect to holding their intended objects.

3.5.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.
- Test that the subsystem can deal with large and zero integers.
- Test the correct handling of a large or zero array of the appropriate type.

3.5.4 Handling of invalid parameters

The subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.
- Correct handling of non-boolean variables passed in where booleans are expected.
- Correct handling of non-integer variables passed in where integers are expected.
- Correct handling of parameters passed in an unexpected order.
- Correct handling of non-intended items being added to the arrays (such as Strings).

3.6 Features of the Builder Subsystem

The Builder subsystem contains the following classes.

- WorkStation class – [32], section 2.4.22
- StepBuilder class – [32], section 2.4.25
- Step class – [32], section 2.4.26
- LiquidTransfer –[32], section 2.4.24

3.6.1 General compliance with design specification

3.6.1.1 WorkStation

The WorkStation must:

- Contain all the appropriate variables for modelling the Biomek environment, according to [32].

3.6.1.2 StepBuilder

The StepBuilder must:

- Contain the appropriate variables.
- Provide a method that creates the steps (liquid transfers) from the Substances in the ExperimentPlates.
- Provide the functionality to use Vectors.
- Provide a method that executes the appropriate LiquidTransfers and appends them to the jobString.
- Be able to write to a file.

3.6.1.3 Step

The Step must:

- Contain the appropriate variables.

3.6.1.4 LiquidTransfer

The LiquidTransfer must:

- Contain the appropriate variables.
- Contain an id.
- Provide a method that creates a Liquid transfer operation, given the necessary data to assimilate a string and to pass that string back.

3.6.1.5 Communication

The classes must be able to communicate correctly between each other.

3.6.1.6 Input/Output

The subsystem should be able to deal with the file input of the experiments text file. This should include abnormal file detection, missing file detection and other IO errors. The output file should be standardised, coherent, delimited in the correct way.

3.6.2 Handling of expected parameters

The constructors of each of the modules described above must all function correctly with the passing of normal parameters in the correct order.

- Correct storage of variables.
- The Vectors should be tested with respect to holding their intended objects.

3.6.3 Handling of extreme parameters

The subsystem must be able to handle the input of values in excess or less than likely values.

- Test that the subsystem can deal with long strings.
- Test that the subsystem can deal with large and zero integers.

- Test the correct handling of a large or zero array of the appropriate type.

3.6.4 Handling of invalid parameters

The subsystem must be able to handle invalid parameters.

- Correct handling of non-string variables passed in where strings are expected.
- Correct handling of non-boolean variables passed in where booleans are expected.
- Correct handling of non-integer variables passed in where integers are expected.
- Correct handling of parameters passed in an unexpected order.
- Correct handling of non-intended items being added to the arrays (such as Strings).

4. APPROACH

The following section describes the testing approaches that will be employed whilst testing the TCL Job Compiler. It is important to note that not all the approaches will be suitable for every testing occurrence, and each must be applied where most appropriate.

4.1 Static Testing

This would usually primarily involve code walkthroughs. However, due to the nature of the project and the fact that there is only one developer (designer, tester, etc.), walkthroughs may not be worthwhile or even possible. Even though there is only one person involved in the project, some aspects of walkthroughs can still be observed. For example, the code can still be examined as a series of paths and the most likely paths of the system can be ascertained during this process.

4.1.1 Suitability of method

Static testing is usually the first method of testing to be employed during a project and can be useful in spotting syntactical and typographical errors.

4.2 Black box testing

Black box testing can be used as a follow up process to the static testing to further explore possible paths through the system. Black box testing can be useful when the tester knows the function of the component with respect to the operations on data inputs, but is unsure as to how the component functions on the inside.

4.2.1 Suitability of method

Black box testing is most suitable for testing top-level systems.

4.3 White box (Structural) testing

White box testing (glass box testing) is carried out when the inner workings of a component are known and the test cases can be constructed with respect to this knowledge. The tester uses knowledge about the structure of the component to derive test data and test cases.

4.3.1 Suitability of method

White box testing allows the tester to use many of the possible paths through a component, rather than simply the paths that are most likely to be used. It is not possible to use every possible path, but a subset of test cases can be established given the unit's function and likely problems.

4.4 Interface testing

Interface testing is concerned with the communications between modules, subsystems and subsystems. It is primarily concerned with the errors encountered during these periods of communication and aims to monitor and improve the way that these components co-operate.

4.5 Regression testing

The fixing and removing of bugs and errors can introduce new errors into the code and the design. Therefore, the testing process must be iterative and repeated until the system is adequately error-free.

5. PASS/FAIL CRITERIA

This section is used to identify explicitly what is meant by 'passing' or 'failing' a certain item of testing. For this project, an explicit list of pass/fail criteria is not called for and, indeed, time certainly not permit one. Please refer to the report (Story) for more details of the testing process. The previous sections adequately demonstrate the direction of the test process that is to be carries out. Furthermore, the exact criteria for a pass or fail should seem clear, given the nature of the layout of the previous sections.

Appendix

B

populateSubstances()
populateExperimentPlate()
setup()
createSteps()
deploySteps()
create()
SubstanceTest.java
ExperimentTokenizer.java
TipListDemo.java
SurfaceDemo.java
BigDemo.java

populateSubstances( )

```
/*
 * A method that loads a list of substances from a given file
 * @param String fileName - the name of the file with the
 * substances (i.e. Substances.txt)
 */

public void populateSubstances(String fileName)
        throws TextIOException, Exception
{
      TextFileReader tReader = new TextFileReader(fileName);

      // Load the substances into a temporary array as sets of
      // Strings from Substances.txt
      String [] tmp = new String [theList.length];
      for (int i = 0; i < tmp.length; i++)
      {
            String temp = new String(tReader.readString());
            tmp[i] = temp;
      }

      // Load the description of the Substances into individual
      // Substance objects.
      for (int i = 0; i < tmp.length; i++)
      {
            // Using string tokenizer with space as delimiter
            StringTokenizer st = new StringTokenizer(tmp[i], " ");

            Substance temp = new Substance();

            temp.setName(st.nextToken());

            // Must deal with the String - needs INT!
            String a = st.nextToken();
            int ai = Integer.parseInt(a);
            temp.setPrewetDelay(ai);

            String b = st.nextToken();
            int bi = Integer.parseInt(b);
            temp.setBlowDelay(bi);

            String c = st.nextToken();
            int ci = Integer.parseInt(c);
            temp.setDispenseDelay(ci);

            String d = st.nextToken();
            int di = Integer.parseInt(d);
            temp.setAspirateDelay(di);

            // Dealing with the String - need Boolean
            String e = st.nextToken();
            int eb = Integer.parseInt(e);
            if (eb == 1)
            {
                  temp.setPrewetNeeded(true);
            }
```

```
                else
                {
                      temp.setPrewetNeeded(false);
                }

                String f = st.nextToken();
                int fb = Integer.parseInt(f);
                if (fb == 1)
                {
                      temp.setTiptouchNeeded(true);
                }
                else
                {
                      temp.setTiptouchNeeded(false);
                }

                String g = st.nextToken();
                int gb = Integer.parseInt(g);
                if (gb == 1)
                {
                      temp.setBlowoutNeeded(true);
                }
                else
                {
                      temp.setBlowoutNeeded(false);
                }

                String h = st.nextToken();
                int hb = Integer.parseInt(h);
                if (hb == 1)
                {
                      temp.setKnockoutNeeded(true);
                }
                else
                {
                      temp.setKnockoutNeeded(false);
                }

                addSubstance(temp);
          }
}
```

populateExperimentPlate( )

```
/*
* A method to populate a plate with experiments using a file
* e.g. Expt.txt
*/

public void populateExperimentPlate(String fileName, SubstanceList substanceList)
      throws TextIOException, Exception
{

      File myFile = new File(fileName);

      BufferedInputStream a = new BufferedInputStream(new FileInputStream(myFile));
      InputStreamReader b = new InputStreamReader(a);
      BufferedReader d = new BufferedReader(b);

      String temp = "";
      for (int i = 0; i<512; i++)
      {
            temp += d.readLine();
            temp.trim();
      }

      // First, break the file into single experiments. - array
      // Using string tokenizer with # as delimiter
      StringTokenizer st = new StringTokenizer(temp, "#");

      // create an array with the number of tokens as a size
      String[] subStr = new String[st.countTokens()];
      for (int i = 0; i<subStr.length; i++)
      {
            subStr[i] = st.nextToken();
      }

      // Setup Substances
      //SubstanceList substanceList = new SubstanceList();
      //substanceList.populateSubstances("C:/test/substances.txt");

      // Create an Experiment Plate
      ExperimentPlate plate = new ExperimentPlate();
      // Loop that builds the experiments
      for (int i = 0; i< subStr.length;i++)
      {
            Experiment exp = new Experiment();
            // Create new String tokenizer to tokenize the experiment
            StringTokenizer tz = new StringTokenizer(subStr[i], ", ");


            //**********************WELL*****************************//

            // First token is the well - split into characters
            char [] wellArray = tz.nextToken().toCharArray();
            // sort out row

            if (wellArray[0] == 'A')
            {
```

```
      exp.setRow(1);
}
if (wellArray[0] == 'B')
{
      exp.setRow(2);
}
if (wellArray[0] == 'C')
{
      exp.setRow(3);
}
if (wellArray[0] == 'D')
{
      exp.setRow(4);
}
if (wellArray[0] == 'E')
{
      exp.setRow(5);
}
if (wellArray[0] == 'F')
{
      exp.setRow(6);
}
if (wellArray[0] == 'G')
{
      exp.setRow(7);
}
if (wellArray[0] == 'H')
{
      exp.setRow(8);
}


// sort out column
if (wellArray.length > 2)
{
      // column is greater than 9
      String tempInt = new String
          (Character.toString(wellArray[1]) +
             Character.toString(wellArray[2]));
      int ti = Integer.parseInt(tempInt);
      exp.setCol(ti);
}
else
{
      // column is between 1 and 9
      String tempInt = new String

      (Character.toString(wellArray[1]));
      int ti = Integer.parseInt(tempInt);
      exp.setCol(ti);

}
exp.initWell();

  /******************Substances***************************/
// "time" represents the end of the Substances
// gets the first token
```

```
            String next = tz.nextToken();
            do
            {
                  // create instance of new Media component
                  MediaComponent media = new MediaComponent();
                  // uses the first token
                  next.trim();
                  media.setSubstance(substanceList.findByName(next));
                  // set amount of media component
                  // move onto next token
                  next = tz.nextToken();
                  next.trim();
                  media.setAmount(Integer.parseInt(next));

                  exp.addMediaToList(media);
                  //move onto next token
                  next = tz.nextToken();
            } while (!next.equals("time"));


              /**********************Time*****************************/
            // the above nextToken() purges the String - 'time'
            exp.setLength(Integer.parseInt(tz.nextToken()));
            this.addExperimentToPlate(exp);
      }
      System.out.println("Populated Experiment Plate");
}
```

A Selected Section of setup( )

```
// Going to start adding things to the surface

// so we need to create a surface.
Surface theSurface = new Surface();

TextFileReader tReader = new TextFileReader(surfFileName);

// Load the substances into a temporary array as sets of
// Strings from Substances.txt
String [] tmp = new String [25];
for (int ni = 0; ni < tmp.length; ni++)
{
      String temp = new String(tReader.readString());
      tmp[ni] = temp;
      System.out.println(tmp[ni]);
}

for (int mi = 0; mi < tmp.length; mi++)
{
      // Using string tokenizer with space as delimiter
      StringTokenizer st = new StringTokenizer(tmp[mi], " ");

      // First, get the coordinates - the first token
      String coord = st.nextToken();
      System.out.println(coord);
      int locNum = 0;    // will be a number between 1 and 12.
      int segNum = 0;    // will be between 1 and 8
      // If the token is only 2 characters, then A1 etc...
      if (coord.length() <= 2)
      {
            if (coord.charAt(0) == 'B')
            {
                  locNum = 6 + Integer.parseInt(coord.valueOf(coord.charAt(1)));
            }
            else
            {
                  locNum = Integer.parseInt(coord.valueOf(coord.charAt(1)));
            }
      }

      if (coord.length() == 3) // means there is a segement as well
      {
            if (coord.charAt(0) == 'B')
            {
                  locNum = 6 + Integer.parseInt(coord.valueOf(coord.charAt(1)));
            }
            else
            {
                  locNum = Integer.parseInt(coord.valueOf(coord.charAt(1)));
            }
            segNum = Integer.parseInt(coord.valueOf(coord.charAt(2)));
      }

      if (coord.length() >= 4) // means there is a segement as well
      {
```

```
      if (coord.charAt(0) == 'B')
      {
            locNum = 6 + Integer.parseInt(coord.valueOf(coord.charAt(1)));
      }
      else
      {
            locNum = Integer.parseInt(coord.valueOf(coord.charAt(1)));
      }
      segNum = Integer.parseInt(coord.valueOf(coord.charAt(3)));
}

// ok, so we've got the coordinates - location on surface.

String type = st.nextToken();

// Need to check the line for each different possible surface object.


// If it's VICTOR - will always be first?
if (type.equals("VICTOR"))
{
      // we want a new SurfaceObject
      SurfaceObject victor = new
      SurfaceObject(locNum, segNum, "VICTOR");

// Add the victor to the surface
theSurface.addToSurface(victor, locNum, segNum);
}

// Now, do the tools
if (type.equals("Gripper"))
{
      ToolSurfaceObject gripper = new ToolSurfaceObject
       (locNum, segNum, 2, 0, ws.getToolList().getToolByName(type));
      theSurface.addToSurface(gripper, locNum, segNum);
}

if(type.equals("P20L"))
{
      ToolSurfaceObject p20L = new ToolSurfaceObject
       (locNum, segNum, 2, 0, ws.getToolList().getToolByName(type));
      theSurface.addToSurface(p20L, locNum, segNum);
}
if(type.equals("P200L"))
{
      ToolSurfaceObject p200L = new ToolSurfaceObject
       (locNum, segNum, 2, 0, ws.getToolList().getToolByName(type));
      theSurface.addToSurface(p200L, locNum, segNum);
}
if(type.equals("P1000L"))
{
      ToolSurfaceObject p1000L = new ToolSurfaceObject
       (locNum, segNum, 2, 0,      ws.getToolList().getToolByName(type));
      theSurface.addToSurface(p1000L, locNum, segNum);
}

// Now the tip racks
```

```
	if(type.equals("P20"))
	{
		TipRackSurfaceObject p20 = new TipRackSurfaceObject
		 (locNum, segNum, 4, 0,  ws.getTipRackList().getTipRackByName(type));
		theSurface.addToSurface(p20, locNum, segNum);
	}
	if(type.equals("P250"))
	{
		TipRackSurfaceObject p250 = new TipRackSurfaceObject
		 (locNum, segNum, 4, 0, `ws.getTipRackList().getTipRackByName(type));
		theSurface.addToSurface(p250, locNum, segNum);
	}
	if(type.equals("P1000"))
	{
		TipRackSurfaceObject p1000 = new TipRackSurfaceObject
		 (locNum, segNum, 4, 0, ws.getTipRackList().getTipRackByName(type));
		theSurface.addToSurface(p1000, locNum, segNum);
	}

	// Now the labware...

	if(type.equals("96-well") | type.equals("half") |
	    type.equals("quarter") | type.equals("reservoir"))
	{
		String ext = st.nextToken();
		LabWareSurfaceObject well = new LabWareSurfaceObject
		  (locNum, segNum, 3, 0, ws.getLabWareList().getLabWareByName
		  ("{" + type + " " + ext+ "}"));
		theSurface.addToSurface(well, locNum, segNum);
	}

	// Now, the subtances
	// Is it a substance?
	if (theSubstanceList.findByName(type)!= null)
	{
		// First, get the amount
		float am = Float.parseFloat(st.nextToken());
		SubstanceSurfaceObject subSurObj = new
			SubstanceSurfaceObject(locNum, segNum,
				theSubstanceList.findByName(type),am);
		theSurface.addToSurface(subSurObj, locNum, segNum);
	}
}
ws.setSurface(theSurface);
```

createSteps( )

```
/*
* A method that takes the Surface labware one-by-one of the surface
* and iterates throught the Experiment plate, creating the
* liquid transfer steps.
*/

public void createSteps(WorkStation ws)
{
   theWs = ws;

   // Need to get the Experiment plates...
   Vector expPlates = new Vector(1,1);

   // Need all the relevant labware objects from surface
   // extract experiment plates from labware surface objects
   Vector labWareSurface = ws.getSurface().returnLabwareVector();
   for (int i = 0; i < labWareSurface.size(); i++)
   {
      LabWareSurfaceObject temp = (LabWareSurfaceObject)labWareSurface.get(i);
      expPlates.add(temp.getExperimentPlate());
   }

   // go through plates and create the liquid transfer steps
   for (int i = 0; i < expPlates.size(); i++)
   {
      if(((LabWareSurfaceObject)labWareSurface.get(i)).getName().equals("{96-well
      flat}"))
      {
         // get a link to the experiment array
         ExperimentPlate temp = (ExperimentPlate)expPlates.get(i);

         // set up a Well for the plate
         Well thePlateWell = new Well();
         int thePlateLoc = ((LabWareSurfaceObject)labWareSurface.get(i)).getLocation();
         if (thePlateLoc > 6)
         {
            thePlateWell.setRow(2);
            thePlateWell.setCol(thePlateLoc-6);
         }
         else
         {
            thePlateWell.setRow(1);
            thePlateWell.setCol(thePlateLoc);
         }
         System.out.println(thePlateWell);

         try
         {
            Experiment [] tmp = temp.getPlate();

            // sort into various vectors
            for (int n = 0; n < tmp.length; n++)
            {
               // get the Medias - they will form the Liquid transfers
```

```
            MediaComponent [] mediaList = tmp[n].getList();

            for (int p = 0; p < mediaList.length; p++)
            {
              if (mediaList[p] != null)
              {

                Step theStep = new Step(mediaList[p].getSubstance(),
                mediaList[p].getAmount(), thePlateWell, [n].getWell());
                // step is a agar step
                if (theStep.getSubstance().getName().charAt(0) == 'm')
                {
                  agarSteps.add(theStep);
                }
                // step is a metabolite step
                if (theStep.getSubstance().getName().charAt(0) == 'C')
                {
                  metSteps.add(theStep);
                }
                // step is a yeast step
                if (theStep.getSubstance().getName().charAt(0) == 'Y' |
                theStep.getSubstance().getName().charAt(0) == 'w')
                {
                  yeastSteps.add(theStep);
                }
              }
            }
          }
        }
        catch (Exception e)
        {
              System.out.println("Error/Fault/Exception in Step Creation" + e);
        }
      }
    }
    System.out.println("Created Steps.");
}
```

deploySteps( )

```
 /*
 * A method that takes all the steps and creates liquid transfers
* for them in the correct order and builds up jobString (JOB1.txt)
*/
public void deploySteps()
{
  LiquidTransfer temp;
  Step theStep;
  int id = 0;

  // First, the metabolite steps.
  System.out.println("\nStarting Metabolite Steps.\n");
  for (int i= 0; i< metSteps.size(); i++)
  {
    // create the Liquid Transfer for the step
    temp = new LiquidTransfer(id);
    theStep = (Step)metSteps.get(i);

    temp.create(theStep.getCoords(),
      theStep.getPlate(),
        theStep.getAmount(),
          theStep.getSubstance(),
            theWs);
    jobString += temp;
    id++;
  }
  System.out.println("\nFinished Metabolite Steps.");

  // First, the metabolite steps.
  System.out.println("\nStarting Agar Steps.\n");
  for (int i= 0; i< agarSteps.size(); i++)
  {
    // create the Liquid Transfer for the step
    temp = new LiquidTransfer(id);
    theStep = (Step)agarSteps.get(i);

    temp.create(theStep.getCoords(),
      theStep.getPlate(),
        theStep.getAmount(),
          theStep.getSubstance(),
                theWs);
    jobString += temp;
    id++;
  }
  System.out.println("\nFinished Agar Steps.");

  // First, the metabolite steps.
  System.out.println("\nStarting Yeast Steps.\n");
  for (int i= 0; i< yeastSteps.size(); i++)
  {
    // create the Liquid Transfer for the step
    temp = new LiquidTransfer(id);
    theStep = (Step)yeastSteps.get(i);
```

```
      temp.create(theStep.getCoords(),
        theStep.getPlate(),
          theStep.getAmount(),
            theStep.getSubstance(),
              theWs);
      jobString += temp;
      id++;
    }
    System.out.println("\nFinished Yeast Steps.");
}
```

create( ) (LiquidTransfer)

```
/*
* The create() method for when the LiquidTransfer object exists
* but is not yet instantiated.
* @param theSource - the well that contains the source liquid
* @param theDest - the well that is to receive the liquid
* @param theVolume - the amount of liquid that is to be transferred
* @param theSubst - the substance that is to be transferred
* @param theWs - the Work Station (environment)
*/
public void create(Well theDestCoord,
                   Well theDestPlate,
                   float theVolume,
                   Substance theSubst,
                   WorkStation theWs)
{
  /*********************************************************
  * IMPORTANT NOTE:                                        *
  * The "transfer += ......" parts                         *
  * are adding the string together that will form JOB1.txt *
  *********************************************************/

  // start the transfer string
  transfer += "\nLog \"Executing LiquidTransfer: " + id + "\"";

  /************************* 1.*****************************/
  // Find the appropriate tool based on the amount to be transferred.
  Tool theTool;
  theTool = theWs.getToolList().getToolByVolume(theVolume);

  /************************* 2.*****************************/
  // Attach the tool.
  // We may already have the correct tool... Check
  if (theWs.getCurrentTool() == null ||
      !theTool.getName().equals(theWs.getCurrentTool().getName()))
  {
    // we don't have the correct one so we must change it.
    transfer += "\nTool attach " + theTool.getName();
    theWs.setCurrentTool(theTool);
  }

  /************************* 3.*****************************/
  // Attach the tip for that tool.
  // First, get the tip...
  // If the last tip is dirty with the substance for this liquid
  // transfer, then we can use it again?
  if (theWs.getLastSubstancePipetted() == null ||
      !theWs.getLastSubstancePipetted().getName().equals(theSubst.getName()))
  {
    // then we must attach a new tip
    Tip theTip = new Tip();
    theTip = theWs.getTipList().getTipByVolume(theVolume);
    theWs.setCurrentTip(theTip);

    transfer += "\nLog \"Attaching tip " + theWs.getCurrentTip().getName() +
```

```
    " with some grid reference (not sure yet)\"";
  transfer += "\nTip attach dispose " + theWs.getCurrentTip().getName() + " " +
    theWs.getCurrentTip().getName() + " " + theWs.getCurrentTool().getName();

  transfer += "\nLog \"Attaching to " + theWs.getCurrentTip().getName() +
     " with some grid reference (not sure yet)\"";

}

theWs.setLastSubstancePipetted(theSubst);

/************************* 4.*****************************/
// Move the tool to the Source. This command always occurs.
// Get the source surface object - should be a substance.

SubstanceSurfaceObject src =
       (SubstanceSurfaceObject)theWs.getSurface().getByName(theSubst.getName());

LabWareSurfaceObject temp1 =
     (LabWareSurfaceObject)theWs.getSurface().returnByLocation(src.getLocation());

LabWare theSrcLabWare = temp1.getSurfaceLabWare();

transfer+= "\nMove Abs [Coord " + src.getCoords() + " A" +
     src.getSegment() + "] " + (theWs.getAdditionalHeight() +
          theSrcLabWare.getFLabwareHeight());

/************************* 5.*****************************/
// Check that there is enough substance at the source.

src.adjustVolume( - theVolume);

if (src.getVolume() < 0)
{
  System.out.println("Ran out of substance at the source. Should quit!");
}
else
{
  transfer+= "\nLog \"Checking volumes: " + (src.getVolume() - theVolume) +
       " " + theSrcLabWare.getSrcMax() + "\"";
}

// Start of the Aspiration process....
transfer+= "\nLog \"Aspirating " + src.getSurfaceSubstance().getName() +
     " from " + src.getCoords() + "A" + src.getSegment() +
          " (volume " + theVolume + ")\"";

/************************* 6.*****************************/
float aspirationZ = (theWs.getAspirationHeight() * theSrcLabWare.getCDepth()) +
     theSrcLabWare.getCTop() - theSrcLabWare.getCDepth();
// If a prewet is needed:
if (src.getSurfaceSubstance().getPrewetNeeded() == true)
{
  // a.  Suck some air in.
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getHeightWithAir();

  // b.  Move tip into the substance.
```

```
  transfer+= "\nMove Abs Z " + aspirationZ;

  // Mark the tip as dirty. Not sure what this does?..Do it anyway
  // I don't think this is necessary...
  transfer+= "\nputres system pod dirty_tip 1";

  // c.  Suck up some substance.
  float prewetT;
  prewetT = (theWs.getCurrentTool().getVolumeOffset() +
    (theWs.getCurrentTool().getSteps() * (theVolume +
         5 + theWs.getCurrentTool().getBiasV())));
  transfer+= "\nMove Rel T " + prewetT;
  transfer+= "\nDelay " + src.getSurfaceSubstance().getPrewetDelay();

  // d.  Move tool up a bit.
  transfer+= "\nMove Abs Z " + theSrcLabWare.getCTop();

  // e.  Push out the substance.
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getHeightWithAir();
  transfer+= "\nDelay " + src.getSurfaceSubstance().getBlowDelay();

  // f.  Push out the air.
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getFillV();
  transfer+= "\nDelay " + src.getSurfaceSubstance().getDispenseDelay();

}

// 7.  If a blowout is needed:
if (src.getSurfaceSubstance().getBlowoutNeeded() == true)
{
  // a.  Suck some air in. (?)
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getHeightWithAir();
}

transfer+= "\nMove Abs Z " + aspirationZ;
transfer+= "\nputres system pod dirty_tip 1";

// 8.  Suck up the appropriate amount of substance.
float amountT = (theWs.getCurrentTool().getVolumeOffset() +
  (theWs.getCurrentTool().getSteps() * (theVolume +
    theWs.getCurrentTool().getBiasV())));
transfer+= "\nMove Rel T " + amountT;
transfer+= "\nDelay " + src.getSurfaceSubstance().getAspirateDelay();

transfer+= "\nMove Rel T " + (-(theWs.getCurrentTool().getSteps() *
  theWs.getCurrentTool().getBiasV()));

// 9.  Move the tool up.
transfer+= "\nMove Abs Z " + (theWs.getAdditionalHeight() +
  theSrcLabWare.getFLabwareHeight());


// 10. Move the tool the to the destination.
LabWareSurfaceObject theDestLabWareSurfaceObject =
(LabWareSurfaceObject)theWs.getSurface().returnByLocation(theDestPlate.getLocation()
);
```

```
LabWare theDestLabWare = theDestLabWareSurfaceObject.getSurfaceLabWare();

transfer+= "\nMove Abs [Coord " + theDestPlate.toString() + " " +
theDestCoord.toString() + "] " + (theWs.getAdditionalHeight() +
  theDestLabWare.getFLabwareHeight());

// End of the Aspiration Process

// 11. Check that there is enough room at the destination.
Experiment destExp =
theDestLabWareSurfaceObject.getExperimentPlate().returnByLoc(theDestCoord.getLocatio
n());

if (destExp.getVolume() > (theDestLabWare.getDestMax() - theVolume))
{
  System.out.println("Not enough room at the Destination. Should quit!");
}
else
{
  transfer+= "\nLog \"Checking volumes: " + theVolume +
    " " + theDestLabWare.getDestMax() + "\"";
  destExp.updateVolume(theVolume);
}

// Start of the dispensation process
// 12. Move the tool down to an appropriate height.
transfer+= "\nPutVal tools " + theWs.getCurrentTool().getName() +
   " max_velocity 2";
float dispenseZ = (theWs.getDispenseHeight() * theDestLabWare.getCDepth()) +
  theDestLabWare.getCTop() - theDestLabWare.getCDepth();

transfer+= "\nMove Abs Z " + dispenseZ;

// 13. Push the substance out.
if (src.getSurfaceSubstance().getBlowoutNeeded() == true)
{
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getHeightWithAir();
}
else
{
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getFillV();
}
transfer+= "\nPutVal tools " + theWs.getCurrentTool().getName() +
   " max_velocity " + theWs.getCurrentTool().getMaxVelocity();

transfer+= "\nDelay " + src.getSurfaceSubstance().getDispenseDelay();

// 14. Move the tool back up again.
transfer+= "\nMove Abs Z " + theDestLabWare.getCTop();

// 15. Push all the air out of the tool.
if (src.getSurfaceSubstance().getBlowoutNeeded() == true)
{
  transfer+= "\nMove Abs T " + theWs.getCurrentTool().getFillV();
  transfer+= "\nDelay " + src.getSurfaceSubstance().getBlowDelay();
}
if (src.getSurfaceSubstance().getTiptouchNeeded() == true)
```

```
  {
    float tiptouchZ = theDestLabWare.getCTop() - theDestLabWare.getCDepth() +
        (((destExp.getVolume() + theVolume) / theDestLabWare.getDestMax()) *
             theDestLabWare.getCDepth()) + 1;
    transfer+= "\nMove Abs Z " + tiptouchZ;
    transfer+= "\nDelay " + theWs.getSurfaceTouchDelay();
    transfer+= "\nMove Abs Z " + (theDestLabWare.getCTop() +
      theWs.getAdditionalHeight());
  }

  // Print out details of the transfer
  transfer += "\nLog \"Transfer from " + src.getCoords() + " A" +
       src.getSegment() + " to " + theDestPlate.toString() + " " +
            theDestCoord.toString() + ", " + theSubst.getName() +
                  ", Volume: " + theVolume + " COMPLETE.\"";

  System.out.print(".");
}
```

SubstanceTest.java

```
/*****************************************************************
 * SubstanceTest.java - a test class to test Substance.java       *
 * Just testing basic input and output                 *
 * @author - Ben Tagger                                   *
 * @version - 05/02/03                                   *
 *****************************************************************/

import aber.util.TextIOException;

import java.io.*;

public class SubstanceTest
{

    public void test() throws TextIOException
    {
        System.out.println("\nEntering Details...");
        MediaComponent theMedia = new MediaComponent();
        theMedia.setDetails();

        System.out.println("\nDisplaying Details...");
        System.out.println(theMedia);

    }

    public static void main (String [] args) throws TextIOException
    {
        SubstanceTest theDemo = new SubstanceTest();
        theDemo.test();
    }
}
```

ExperimentTokenizer.java

Please refer to populateExperimentPlate( ) for details of the ExperimentTokenizer test harness (these two entries are virtually identical).

TipListDemo.java

```
public class TipListDemo
{
    public void test() throws Exception
    {
        // create tip list
        TipList theList = new TipList(3);

        // create and add tips
        Tip a = new Tip("P20",
                        (float)9.780000,
                        (float)6.860000,
                        (float)38.099998,
                        (float)0.630000,
                        (float)0.000000,
                        (float)0.250000,
                        (float)23.000000,
                        (float)1.000000,
                        false);
        theList.addTipToList(a);

        Tip b = new Tip("P250",
                        (float)9.780000,
                        (float)6.860000,
                        (float)38.099998,
                        (float)0.630000,
                        (float)0.000000,
                        (float)0.250000,
                        (float)23.000000,
                        (float)1.000000,
                        false);
        theList.addTipToList(b);

        Tip c = new Tip("P1000",
                        (float)9.780000,
                        (float)6.860000,
                        (float)38.099998,
                        (float)0.630000,
                        (float)0.000000,
                        (float)0.250000,
                        (float)23.000000,
                        (float)1.000000,
                        false);
        theList.addTipToList(c);

        System.out.println(theList);
    }


    public static void main (String [] args) throws Exception
    {
        TipListDemo theDemo = new TipListDemo();
        theDemo.test();
    }
}
```

SurfaceDemo.java

```
public class SurfaceDemo
{
    public void test() throws Exception
    {
        // Set up Workstation
        WorkStation newStation = new WorkStation();

        // Set up some intitial variables
        newStation.setAspirationHeight((float)0.01);
        newStation.setAdditionalHeight((float)2.0);
        newStation.setDispenseHeight((float)0.85);
        newStation.setSurfaceTouchDelay((float)0.5);

        // Set up the Configurator for the WorkStation
        Configurator newConfig = new Configurator(newStation);
        newConfig.setup("C:\\test\\Substances.txt", "C:\\test\\Surface.txt");

        System.out.println(newStation.getSurface());
    }



    public static void main (String [] args) throws Exception
    {
        SurfaceDemo theDemo = new SurfaceDemo();
        theDemo.test();
    }
}
```

BigDemo.java

```
/*
 * This is a big demo to test everything
 * @author Ben Tagger
 * @version started 12/03/03
 */

import java.util.*;
import java.io.*;

public class BigDemo
{
    public static void main (String [] args) throws Exception
    {
        // Set up Workstation
        WorkStation newStation = new WorkStation();

        // Set up some intitial variables
        newStation.setAspirationHeight((float)0.01);
        newStation.setAdditionalHeight((float)2.0);
        newStation.setDispenseHeight((float)0.85);
        newStation.setSurfaceTouchDelay((float)0.5);

        // Set up the Configurator for the WorkStation
        Configurator newConfig = new Configurator(newStation);
        newConfig.setup("C:\\test\\Substances.txt", "C:\\test\\Surface.txt");
        // create Job0.txt
        newConfig.writeJobFile("C:\\test\\");

        // Now, for the real problem - JOB1.txt
        // Need to populate the surface with Experiments
        // So, we just want the labware...

        Vector labWareSurface = newStation.getSurface().returnLabwareVector();
        for (int i = 0; i < labWareSurface.size(); i++)
        {
            // populated the experiments for each relevant piece of labware.
            LabWareSurfaceObject surfaceObject =
            (LabWareSurfaceObject)labWareSurface.get(i);
            ExperimentPlate thePlate = new ExperimentPlate();

            if (surfaceObject.getName().equals("{96-well flat}"))
            {
                 try
                {
                      thePlate.populateExperimentPlate
                      ("C:\\test\\Expt.txt", newStation.getSubstanceList());
                }
                catch (Exception e)
                {
                     System.out.println("Failed to Populate Experiment Plates: " + e);
                }
            }
            surfaceObject.setExperimentPlate(thePlate);
        }
```

```
        // create the step builder
        StepBuilder newBuilder = new StepBuilder();
        newBuilder.createSteps(newStation);
        newBuilder.deploySteps();
        newBuilder.writeToFile("C:\\test\\JOB1.txt");
    }
}
```