Jason Barrie Morley

Calendar and Contacts Synchronization Server

Computer Science Tripos Part II

Clare College, 2004

May, 2004

Proforma

Name:	*Jason Barrie Morley*
College:	*Clare College*
Project Title:	*Calendar and Contacts Synchronization Server*
Examination:	*Computer Science Tripos Part II*
Word Count:	*11,800 (Approximately)*
Project Originator:	*Jason Barrie Morley*
Supervisor	*Dr. Graham Titmus*

Original Aims of the Project

To implement an extensible calendar and contacts client-server synchronization system capable of merging data from two different sources. This system would provide a method whereby support for additional platforms, devices and formats might be easily added. Additionally, it was hoped that an extensive server implementation would allow for the handling of performance and storage limitations with client devices.

Work Completed

While the original aims proved to be more ambitious than expected, they were fulfilled by authoring a custom module for a Java SyncML[1] implementation, Sync4j[2]. As this avoided the need to implement a communications protocol and corresponding client side applications, it allowed for greater focus on the server side implementation and various approaches to the problem of merging data and of displaying the data on the server.

Special Difficulties

The limited documentation and, at times, buggy implementation of Sync4j architecture presented a number of problems which affected the reliability of the system. Additionally, this slowed development, with much time spent becoming familiar with the framework and getting help from the mailing list.

---

[1] www.openmobilealliance.org
[2] http://www.sync4j.org

Declaration of Originality

I, Jason Barrie Morley of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1 Introduction ... 1
1.1 Synchronization ... 1
1.1.1 Topologies ... 1
1.1.2 Mapping ... 2
1.1.3 Change Detection ... 2
1.1.4 Conflict Handling ... 2
1.2 Previous Work ... 3
1.2.1 vCards and vCalendars ... 3
1.2.2 SyncML ... 3
1.2.3 OpenGroupware ... 3
1.2.4 MultiSync ... 4
Chapter 2 Preparation ... 5
2.1 Requirements Analysis ... 5
2.2 Work to be Undertaken ... 6
2.2.1 Transport and Protocol ... 6
2.2.2 Sync4j v1.2 ... 6
2.3 Design Model ... 9
2.4 Testing ... 9
2.5 Tools ... 10
2.5.1 Language ... 10
2.5.2 External Libraries ... 10
2.5.3 Backup ... 10
Chapter 3 Implementation ... 11
3.1 Sync4j Module Authoring ... 11
3.2 Package Structure ... 13
3.2.1 Versit Utilities and Objects - uk.co.in7.Versit ... 14
3.2.2 Helper Functions - uk.co.in7.utils ... 21
3.2.3 Synchronization Items - uk.co.in7.insync ... 21
3.2.4 Persistent Storage - uk.co.in7.insync.database ... 25
3.2.5 Sync4j Interface - uk.co.in7.insync.source ... 29
3.2.6 User Interface - uk.co.in7.insync.ui ... 29
3.2.7 Mapping and Conflict Handling - uk.co.in7.insync.engine ... 33
3.3 Sync4j 2.0 ... 33
Chapter 4 Evaluation ... 35
4.1 Testing ... 35
4.1.1 Unit Testing ... 35
4.1.2 Integration Testing ... 35
4.1.3 Full System Testing ... 36
Chapter 5 Conclusions ... 47

Acknowledgements

Many thanks to Stefan Fornari of Funambol[3] who gave me a much assistance with understanding and writing for the Sync4j Framework, to Aaron Krom and Mike Rhodes for suffering my extensive synchronization arguments and to Stuart Rowan and Julie Campbell for their assistance with testing. I should also like to thank Shuan Leedy for the 'P800 in a Sink' image and my Director of Studies, Graham Titmus.

[3] http://www.funambol.com

Chapter 1 Introduction

The last few years has seen a significant movement towards mobile computing in the form of Laptops, Personal Digital Assistants (PDAs), Mobile Phones and, increasingly, Smart Phones. These all present ideal platforms for providing Personal Information Management (PIM) solutions, enabling users to view their calendars and address books while on the move.

A user will often have a number of these devices, introducing the need for a method of maintaining some level of concurrency between them. While, in an office environment, desktop clients such as Microsoft Outlook can be permanently connected to a central server, such solutions are not viable for embedded devices, due to high connection charges, power usage, slow connection speeds and even unreliable network coverage.

Synchronization offers a method of unifying such data stores at appropriate intervals. It is commonly used within commercial products to allow users to 'share' PIM information between their Desktop Computer and PDA; HotSync allows users to synchronize Palm devices with Windows PCs, while ActiveSync allows users of Microsoft's Pocket PC platform to synchronise with Windows PCs. However, such solutions are often proprietary and only support synchronization with a particular platform. This leads to complications when attempting to perform cross-platform synchronization, with items getting lost or duplicated across synchronizations.

1.1 Synchronization

Synchronization is described as "the process of making two sets of data look identical"[4]. The strategies behind this are well understood, but there exist a number of potentially problematic stages which, if poorly implemented, can cause synchronization software to seem unreliable.

1.1.1 Topologies

In the scenario described above, we potentially find ourselves facing a 'many-to-many' mapping – updating an item on one of n devices requires $n-1$ exchanges to propagate the change. Given the limited network resources and limited processing power of mobile devices, this is not a practical option. The 'client-server' or 'many-to-one' approach is therefore a far more common one when considering the synchronization of mobile devices.

This project approaches Synchronization by adopting the strategy that implementing a server capable of arbitrating between a number of clients will allow for

[4] SyncML White Paper, http://www.syncml.org

significantly lighter client side implementations and allow the topological issues discussed above to be avoided; an intelligent server implementation may be seen as a work-around to the ideal but impractical many-to-many scenario.

1.1.2 Mapping

When comparing the entries within two or more separate data stores, a method must be agreed upon for describing items; a new set of identifiers, Global Unique Identifiers (GUIDs) are used. A mapping to and from these GUIDs must be maintained for each device. In the case of a client-server architecture it is obvious that this mapping should be performed by the server and that items be stored indexed on their GUID. However, in a 'many-to-many' architecture, it is not so, with the worst case requiring a new mapping to be generated each time.

In a 'client-server' architecture, the mapping from Local User Identifier (LUID) to GUID is generated by an initial synchronization stage, often termed 'slow synchronization'. During this stage, the data stores are exhaustively compared in an attempt to find any entries which may already be considered equivalent. Entries deemed such are then mapped to the same identifiers. However, a difference in the depth of data stored by devices and a variety of different representation formats can make this comparison stage highly problematic. Current synchronization systems often perform badly at this initial mapping stage, leaving users with a number of multiple entries; one from each data store. This project aims to focus on some of these problems which face mapping generation.

1.1.3 Change Detection

Given a mapping between two data stores and the time at which they were last declared equivalent, it is possible, with limited effort, to generate a set of the changes made to each data store in the intervening period; a simple modified timestamp proves to be all that is necessary. These changes include entry deletion, creation and alteration and reflect the minimal exchanges required to unify the two data stores.

1.1.4 Conflict Handling

In the case where the sets of entry changes communicated during synchronization are unique, the changes may be made and synchronization can complete. However, if an entry state has been changed on several devices prior to a synchronization, a conflict occurs. Before synchronization may complete, all of these conflicts must be resolved.

There are three popular approaches to conflict resolution; to assume that the server is always right ('server-wins'), that the client is always right ('client-wins') or to query the user. When synchronizing a single PDA with a Desktop Computer, it is quite reasonable to present the user with a dialog box asking how a conflict should be resolved.

However, when considering remote synchronization and attempting to unify a large number of clients this is no longer viable.

1.2 Previous Work

The basic principles of synchronization are well understood and have been implemented in a wide range of applications. However, many of these solutions are proprietary and undocumented. I shall therefore limit this discussion to several open solutions.

1.2.1 vCards and vCalendars

The need for standard formats for communicating calendaring and contact information existed long before synchronization, in the form of the vCard and vCalendar formats, specified initially by the Versit Consortium. Later variations to the standards include iCalendar, which offers a greater selection of fields and xCalendar, an XML equivalent. These are used widely within the mobile phone platforms, where they can be embedded in text messages to allow users to send such information over SMS, or exchanged over the Object Exchange Protocol (OBEX), a Hyper-Text Transfer Protocol (HTTP) like transport layer for Bluetooth and Infra Red. While vCards and vCalendars do not represent a synchronization protocol, their widespread acceptance makes them a viable option for data representation.

1.2.2 SyncML

SyncML is an attempt at an industry standard synchronization protocol. It is defined by the Open Mobile Alliance[5] and is backed by a number of key companies, including Nokia, Sony Ericsson, Starfish and Symbian. This support has allowed SyncML to see good, widespread market acceptance and it is now supported on a range of mobile platforms. Additionally, a number of SyncML library and server side implementations are available.

The SyncML specification defines a number of transport layers, including HTTP and OBEX. Additionally, it relies on the specific implementations for the data representation, allowing data to be sent MIME encoded. Most implementations to date make use of the vCard and vCalendar standards for encoding PIM items.

1.2.3 OpenGroupware

There are a number of PIM Server Applications, possibly the most notable of these being OpenGroupware[6]. This is an Open Source project whose aim is '*to create […] the leading open source groupware server to integrate with the leading open source office suite products and all the leading groupware clients […] through open XML-based*

---

[5] http://www.openmobilealliance.org
[6] http://www.opengroupware.org

interfaces and APIs.' The project hosts both a SyncML project aiming to integrate support for SyncML and an Evolution Connector Project producing a connector for the Ximian Evolution desktop application. While both clearly provide some implementation of synchronization logic, neither focuses on this area and the separation of the two projects suggests that there might be issues with using the two together.

1.2.4 MultiSync

MultiSync is an Open Source synchronization utility for Linux with an extensive plug-in architecture and a wide range of plug-ins, including a SyncML implementation. A brief investigation showed MultiSync to be quite poorly documented and to provide more of a synchronization 'bridge', allowing users to select pairs of devices and stores to be synchronized rather than offering an extensive server implementation. For limited synchronization solutions as might be offered by ActiveSync, or similar, this would seem to be a reasonable solution for a Linux desktop environment. However, it does not appear to venture into the area of particular interest to this project, which is to support a diversity of PIM architectures.

Chapter 2 Preparation

2.1 Requirements Analysis

The implementation may be divided into two specific areas of development; that of Server and Clients. In this case, a certain amount of code will be common to the two 'applications':

- **Transport** – A transport method must be decided upon and implemented. On the server side, a multi-threaded implementation will be required to allow concurrent synchronization of multiple clients.
- **Protocol** – A synchronization protocol is required to allow a meaningful exchange between the Client and Server.
- **Data Representation** – A separate representation standard must be decided upon for this purpose.

Additionally, server side specific blocks of code are required for:

- **Persistent Storage** – The server implementation must provided a globally mapped data store with some transactional properties. In order to provide reasonable conflict handling, some amount of additional information and history may need to be stored.
- **Conflict Handling** – The server must provide some way of handling conflicts. Ideally, this would be in the form of an automated system with a range of heuristics by which the server could select the winner or merge conflicting items.
- **User Interface** – As discussed in the project proposal, a User Interface could be implemented to assist debugging.

The client side, on the other hand, must implement:

- **Persistent Storage** – This is likely to be a reasonably simple implementation, storing only the information pertaining to the device on which the client is running. In many cases, this could be as simple as a number of API calls to interface with the local store.
- **User Interface** – The user must be provided with some method of performing a synchronization. If the Persistent Storage makes use of a local PIM service then no additional user interface is required as the native applications will provide a method of viewing and manipulating the data.

2.2 Work to be Undertaken

2.2.1 Transport and Protocol

Very early in the timescale of the project, SyncML was selected as the most suitable protocol over which a synchronization server could operate. While SyncML is one of a small number of suitable protocols, its widespread uptake across mobile phone manufacturers makes it a natural choice as it already has a number of client implementations. Supported devices include the Sony Ericsson P800 which I already owned, making testing relatively easy. Additionally, this should assist quick fulfilment of my requirement for supporting synchronization with a range of devices and platforms.

Following a consideration of previous SyncML implementations, it was decided that the Sync4j framework offered a significant advantage to the server development as it reduced the amount of work required in what was already clearly a large project.

2.2.2 Sync4j v1.2

The Sync4j project constitutes both a SyncML framework and basic server and client implementations. The server is a Java Enterprise Edition (J2EE) application and makes use of Java Beans to implement the SyncML HTTP transport bindings. Additionally, it provides a modular architecture to allow the integration of custom data stores and content handlers. In order to minimise the learning required to produce a functional solution, it was decided that the project would be implemented within this modular design.

Making use of the Sync4j server implementation brings with it a number of those components highlighted previously: transport, in the form of HTTP and protocol, in the form of SyncML. The existence of a number of client implementations also means that one is no longer necessary within the scope of the project. Therefore, only the following modules require implementation:

- **Conflict Handling**
- **Data Representation**
- **Persistent Storage**
- **User Interface**

Should additional client implementations be desired, the Java Native Interface (JNI) could be used alongside the SyncML framework provided by Sync4j to integrate with desktop applications such as Microsoft Outlook[7] and Ximian Evolution.

[7] The Sync4j 2.0 Framework which was released towards the end of development provides such a client implementation. See 3.3.

2.2.2.1 Conflict Handling

In a 'many-to-one' or 'client-server' model of synchronization, the client device is aware only of its state relating to the server. Within this, however, we wish to simulate a 'many-to-many' model of synchronization. Therefore, we must consider how conflict resolution is handled when multiple devices are involved. SyncML provides no method of querying the user regarding conflict resolutions – this must be handled by the server implementation.

It is possible to imagine a scenario whereby an address book entry (contact item) is altered independently on two different devices. In this case, we find ourselves wanting to perform a conflict resolution explicitly between the two devices but are forced to go via the server. A simple 'server-wins' strategy would ensure that, irrespective of when the item was updated on each device, the first device to synchronize on the server would 'win'. A similar problem occurs with the 'client-wins' strategy, while querying the user would guarantee the last synchronized device to 'win'.

While it is clearly not possible to know the users' intentions, a greater level of detail might allow a more structured approach to these merges. It is quite valid to assume that two users might edit different fields of an item leading to a situation where a merge of the two conflicting items would be the desired outcome.

A number of more involved approaches to conflict resolution may be envisaged, in which the modification times of items are brought into consideration. It is hoped that by maintaining a history of item modifications on the server, finer grained conflict handling could be achieved by considering which fields have changed in an entry. This would resolve the above problem, with conflicts only occurring when the same field is altered in both devices. Storing modification times might also allow the server to make reasoned guesses about which device performed the update last – the more recently an item were updated on the server, the higher the probability that this update was performed after that on the client. While outside the scope of this project, implementations might provide a range of these heuristics, weighting each to achieve a final decision.

Additionally, in the process of conflict handling and establishing a mapping from LUIDs to GUIDs, device limitations must be taken into account. Low-end client devices are only likely to deal with a small subset of the data stored on the server. In such situations, simple equality tests are ineffective as equivalent items would not be declared equal. Further, a straight replacement when resolving a conflict might result in a number of fields being lost. With this in mind, equivalence tests and merge functions would need to be provided.

Sync4j provides a very basic mapping implementation and a 'server-wins' conflict management implementation. The architecture allows for replacement of classes

by means of reflection and it is proposed that a custom implementation be authored to incorporate some of the ideas outlined above.

2.2.2.2 Data Representation

As discussed previously many SyncML implementations make use of the vCalendar and vCard formats. In fact, the SyncML specification requires that these formats be supported in order for a server implementation to be deemed compliant. As such, the project should provide implementations of parsers for both the vCard and vCalendar standards.

The architecture should facilitate the translation of formats to a generic format to allow support for additional representations to be added with minimal effort. An Abstract Factory Pattern for 'representation handlers' would therefore seem to be a suitable solution.

2.2.2.3 Persistent Storage

The issue of conflict handling places a number of requirements on the implementation of persistent storage: that it must provide some form of history of item changes and that conflict resolutions must be transparent and reversible. Additionally, multi-threading within the Sync4j architecture requires that some form of transactional guarantees be provided.

Therefore, an off-the-shelf database and the Java Database Connector architecture would seem the most suitable solution. It is likely that two separate data stores will be required due to the logical separation between calendar and contacts items, but this can be reflected by separate tables within the database design.

2.2.2.4 User Interface

Embracing Sync4j as a SyncML framework has allowed for a broadening of the scope of the project. It was decided that the implementation of some form of user interface, although optional within the proposal document, was important. Since the application will be running within a J2EE web server, it seems most natural to implement the user interface as a web based interface. This has the additional benefit of providing users with yet another 'mobile' way of viewing their calendaring and contacts information.

A user interface provides a medium through with the user may have some additional control over the process of conflict resolution. It is anticipated that, by means of a web-based user interface, the user could alter the outcome of such conflicts, removing some of the frustration which can be caused by unintuitive resolution strategies. Further, this user intervention could be used to assess unsatisfactory conflict resolutions and alter the resolution approach accordingly.

Additionally, the user interface should prove to be a useful debugging tool, allowing the outcomes of synchronizations to be viewed easily.

2.3 Design Model

The proposal document outlined a number of distinct 2-week work packets which it was anticipated would need to be completed in order to achieve a successful implementation of the software specified. It was realised during the initial design and implementation phases that the scope of the project was too large and, consequently, these work packets were revised in time for the progress report. These revised work packets were constructed so that the final acceptance criteria might still be achieved.

This reassessment reflects closely the Spiral Model of development whereby a software project undergoes a number of re-evaluation and specification refinement stages before a product is finalised. This model works well for such a project, where learning is a key aspect and views are likely to be refined throughout. Figure 1 shows the various stages of development of the project, starting from the centre.

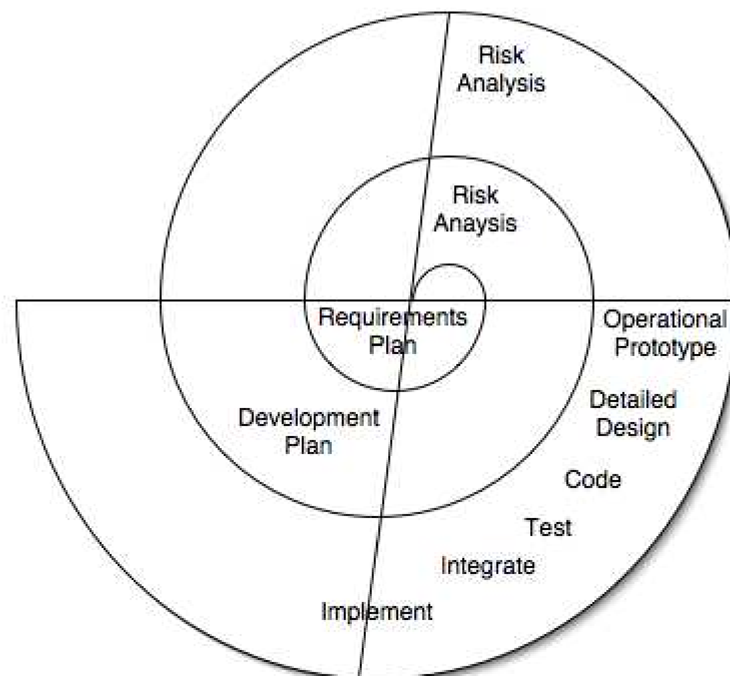

Figure 1 The Spiral Development Model

2.4 Testing

The project breaks down naturally into a number of different components. Where possible, these should be tested by unit testing, authoring test harnesses to ensure the reliability of each module.

Additionally, the Sync4j command line client implementation provides verbose debug output and should prove useful for performing integration testing. As it makes use

of a flat file database, test scenarios can be created and reproduced easily, allowing thorough testing.

2.5 Tools

2.5.1 Language

Hyper Text Markup Language (HTML) and Cascading Style Sheets (CSS) were natural choices for developing the user interface as they see widespread cross-platform acceptance and are already familiar to many users. Additionally, little learning would be required, as I already had significant prior experience of them. I decided to implement the server side processing for this within a Java Servlet. Though this would require some additional learning, it was reasoned that knowledge of Perl based CGI scripting would assist.

MySQL will be used for the database implementation due to its widespread use. A significant amount of SQL had to be learnt and the locking provided by the MySQL implementation had to be considered. See Section 3.2.4.2, page 29.

2.5.2 External Libraries

Extensive use was made of Version 1.2 of the Sync4j framework by authoring a module for the Sync4j server architecture. Additionally, the MySQL Connector/J JDBC Drivers were used. No other libraries were used except those provided within the Java framework.

2.5.3 Backup

As discussed within the proposal document, regular backups were taken to a number of external servers including the University Computing Service's Pelican Backup System. Additionally, regular, incremental backups were taken to CD-RW, with CVS being used during coding of the application.

The cross platform nature of Java made it an ideal choice for implementing both a synchronization server and client. While my experience of Java was limited to that taught earlier in the course, I felt that a good experience of C++ and Object Orientation concepts would prevent this being problematic. Although Eclipse seems to be the widely accepted IDE for Java Development, the need for heavily customized Ant[8] build scripts when generating Sync4j modules led me towards more simple editors, such as Emacs, with most of the final development being undertaken using Apple Xcode. Commenting for JavaDoc required little additional learning since I had already had extensive experience with Doxygen, a documentation utility based on Javadoc.

---

[8] http://ant.apache.org

Chapter 3 Implementation

Before authoring the Sync4j module, a significant amount of time was spent in becoming familiar with the architecture and setting up the development and testing environment. Due to poor documentation and a lack of experience with Java Enterprise Edition, this proved problematic, necessitating much reliance on the Sync4j mailing lists. The JBoss[9] J2EE web-server was used to deploy Sync4j server implementation.

3.1 Sync4j Module Authoring

A Sync4j Module is expected to provide a concrete implementation of sync4j.framework.engine.source.AbstractSyncSource. This declares the following abstract functions and represents the entry point to the module. The functions are used by the Sync4j server implementation to access the persistent storage, allowing it to select the items modified after a given time and for a particular user.

```
SyncItem[] getAllSyncItems(Principal principal)
      throws SyncSourceException;

SyncItemKey[] getDeletedSyncItemKeys(Principal principal,
                                     Timestamp since)
      throws SyncSourceException;

SyncItem[] getDeletedSyncItems(Principal principal,
                               Timestamp since)
      throws SyncSourceException;

SyncItemKey[] getNewSyncItemKeys(Principal principal,
                                 Timestamp since)
      throws SyncSourceException;

SyncItem[] getNewSyncItems(Principal principal,
                           Timestamp since)
      throws SyncSourceException;

SyncItem getSyncItemFromId(Principal principal,
                           SyncItemKey syncItemKey)
      throws SyncSourceException;

SyncItem[] getSyncItemsFromIds(Principal principal,
                               SyncItemKey[] syncItemKeys)
      throws SyncSourceException;

SyncItem[] getUpdatedSyncItems(Principal principal,
                               Timestamp since)
      throws SyncSourceException;
```

[9] http://www.jboss.org

```
void removeSyncItem(Principal principal,
                    SyncItem syncItem)
      throws SyncSourceException;

void removeSyncItems(Principal principal,
                     SyncItem[] syncItems)
      throws SyncSourceException;

SyncItem setSyncItem(Principal principal,
                     SyncItem syncInstance)
      throws SyncSourceException;

SyncItem[] setSyncItems(Principal principal,
                        SyncItem[] syncItems)
      throws SyncSourceException;

SyncItem getSyncItemFromTwin(Principal principal,
                             SyncItem syncItem)
      throws SyncSourceException;

SyncItem[] getSyncItemsFromTwins(Principal principal,
                                 SyncItem[] syncItems)
      throws SyncSourceException;
```

The Sync4j server uses serialised Enterprise JavaBeans to decide which module to load when a particular encoding and resource is requested. An example of the XML used to configure the server to load the project's calendar module is:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
  <object class="uk.co.in7.insync.source.InSyncVCalendarSource">
    <void property="name">
      <string>calendar</string>
    </void>
    <void property="type">
      <string>text/x-vcalendar</string>
    </void>
    <void property="sourceURI">
      <string>calendar</string>
    </void>

      ...

  </object>
</java>
```

Additionally, Sync4j is very explicit about the structure of modules, which are essentially jar packages with a '.s4j' extension. Ant build scripts within these packages are used to

allow customization of the server; the Sync4j install script unpacks the custom modules and executes the Ant files. Additionally, initialization files may be provided to configure the database. These too are executed by the Sync4j installation script. Both of these are used within the Project module to initialize the database and deploy the User Interface.

3.2 Package Structure

Implementation of the Sync4j Module can be divided into two different bodies of code: that relating specifically to the Server implementation and library functions which may find valid reuse within other projects. The package layout reflects this; generic libraries and functions may be found in 'uk.co.in7', while specific module code may be found in 'uk.co.in7.insync'. This follows the Java conventions for using domain names as package names.

Project specific code is further divided into a number of sub-packages which reflect the code structure outlined within the preparation section:

- **User Interface** – uk.co.in7.insync.ui
- **Persistent Storage** – uk.co.in7.insync.database
- **Data Representation** – uk.co.in7.insync

The logical distinction between Calendaring and Contact information leads to some code separation within these packages. Where possible, common interfaces and abstract classes are defined in an attempt to minimise code duplication.

Figure 2 gives a brief overview of these packages. Detailed discussion follows.

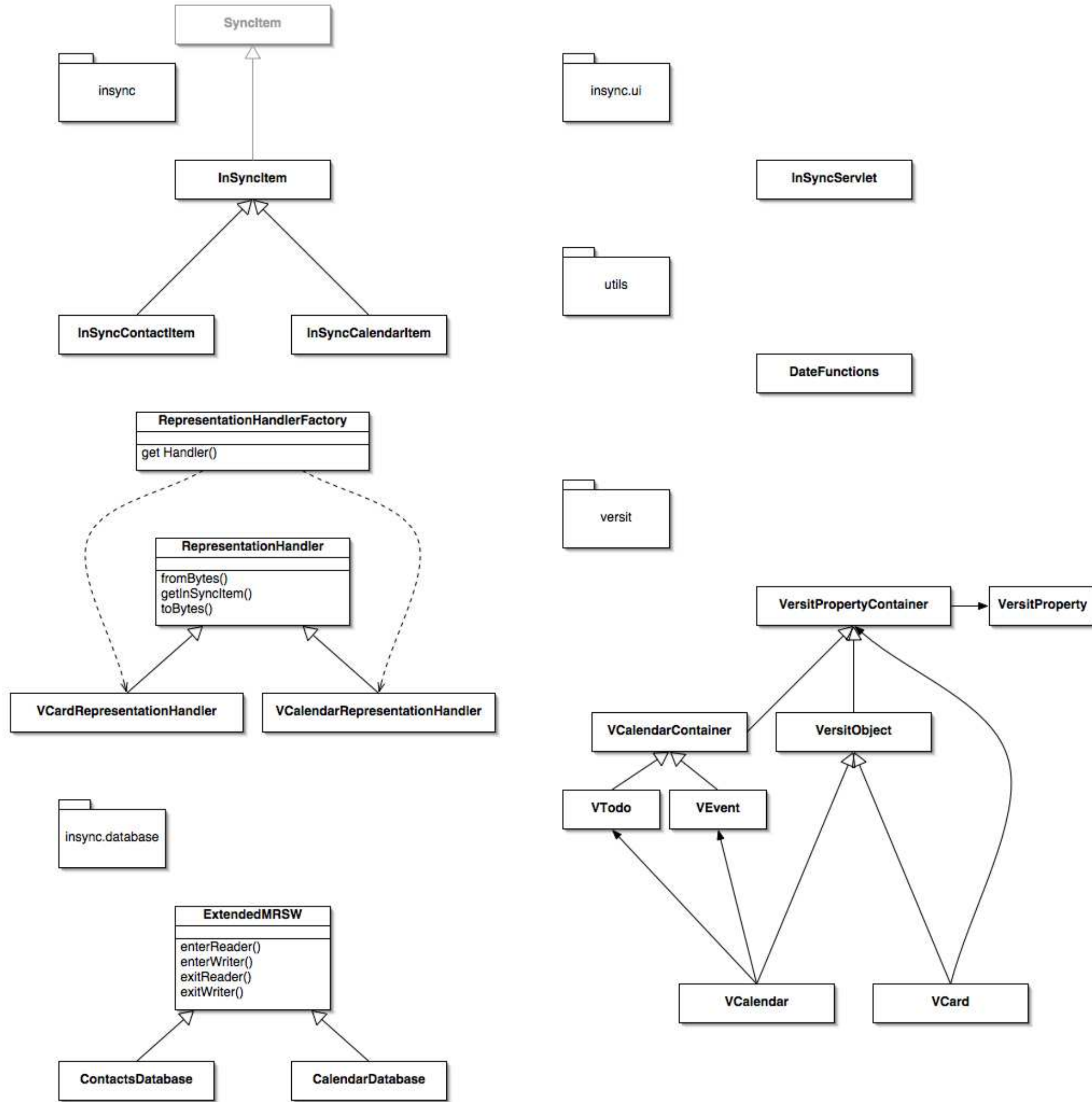

Figure 2 InSync Framework Overview

3.2.1 Versit Utilities and Objects - uk.co.in7.Versit

This package implements a number of classes to deal with the representation and parsing of the Versit vCard and vCalendar formats.

3.2.1.1 Versit Objects

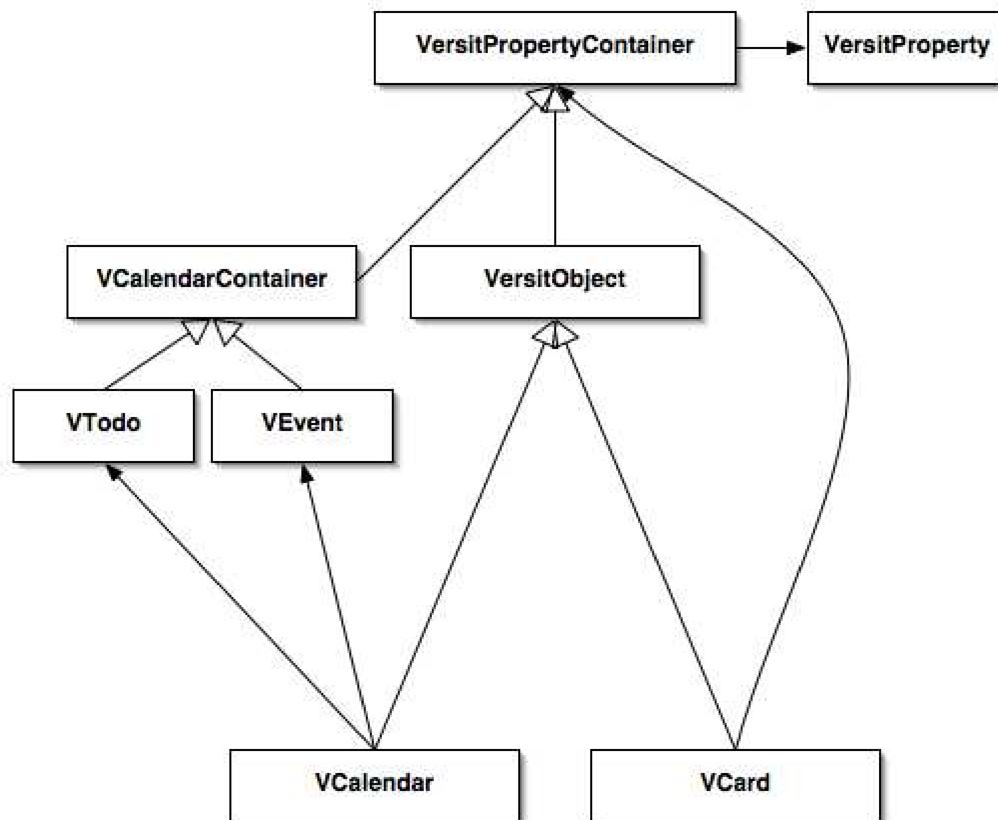

Figure 3 Versit Package Class Structure

The Versit vCard and vCalendar representation formats are essentially a collection of 'properties' – name-parameters-values triplets. A vCard or vCalendar may contain multiple properties of the same name. These can be differentiated by their parameters. For example, two address ('ADR') properties, one containing a home address and the other, a business address would be differentiated by the parameters 'HOME' and 'WORK'. In the case of a vCard, a property may contain a fourth field relating to its grouping – items with the same grouping value are considered to be related.

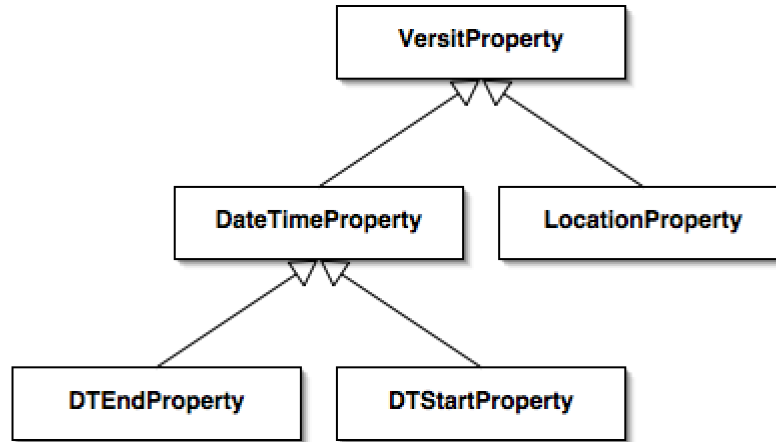

Figure 4 VersitProperty Inheritance Diagram

When designing the object structure for vCards and vCalendars it seemed natural to directly represent these ‘properties’ as objects. The VersitProperty class in Figure 4 reflects a property element, providing appropriate accessors for the various fields. As can be seen, inheritance is used to provide a specific implementation for each property name. This enables custom accessors relating to the specific information stored within the property to be implemented. Figure 4 shows a small portion of this class hierarchy. Note that both DTEndProperty and DTStartProperty inherit from DateTimeProperty. This avoids code duplication, since both contain the same type of information and require the same accessors.

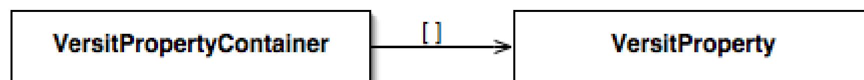

Figure 5 VersitPropertyContainer

To ensure code reuse, VersitPropertyContainer (Figure 5) was defined to provide a number of accessors for maintaining and manipulating arrays of VersitProperty objects.

Properties stored within vCalendars and vCards may be logically divided into two distinct sections: those relating to the Versit object itself, such as its encoding, and the actual information stored. The VersitObject abstract class (Figure 6) is the superclass for vCard and vCalendar implementations, defining those functions common to both. Specifically, abstract functions are declared which convert a VersitObject to and from a byte array. In addition, VersitObject extends VersitPropertyContainer as can be seen in Figure 3; the properties stored in this way are those which relate directly to the vCard or vCalendar, defining the locale, encoding or similar.

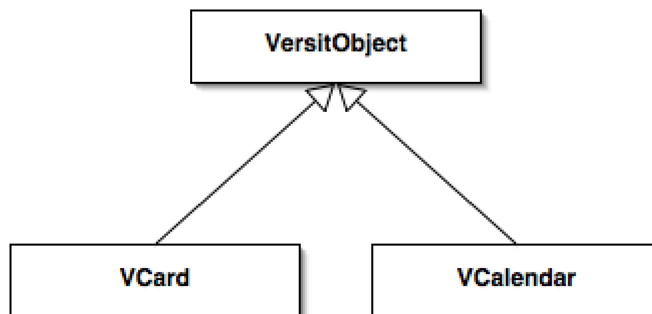

Figure 6 VersitObject Class Structure

3.2.1.1.1 VCard

As mentioned previously, the vCard specification introduces the idea of property grouping, whereby properties may be associated with one another; for example, a comment could be paired with a telephone number. A VCard object must be able to reflect these groups in its internal structure.

The VCard implementation stores groups in a VersitPropertyContainer array, with each VersitPropertyContainer representing a different group. The VCard equals() comparator works by essentially performing unification, comparing this VersitPropertyContainer array. A special VersitPropertyContainer is used to store ungrouped properties – those not associated with a group. In order to prevent confusion, a number of wrapper functions were authored to allow the grouped and ungrouped properties to be accessed directly. When generating the String representation of the VCard object, the groups are printed iteratively, assigning sequential identifiers to them, 'A' for the first group, 'B' to the second and so on. This method conforms to the vCard specification.

3.2.1.1.2 VCalendar

The vCalendar specification introduces the idea of two separate entities which may be contained within a vCalendar object in a similar way to groups within vCards. They differ from vCard groups, however, in that they are delimited by means of start and end tags.

A vTodo object is denoted by:

```
BEGIN:VTODO
END:VTODO
```

While a vEvent, by:

```
BEGIN:VEVENT
END:VEVENT
```

As their names suggest, a vTodo represents 'to-do' entries while a vEvent represents calendar events. While a logical separation is made between these when performing synchronization, it was decided that the VCalendar implementation should represent the vCalendar format as closely as possible. In hindsight, this was possibly unnecessary. However, its implementation did not take long.

As can be seen within Figure 3, two classes were authored, VTodo and VEvent. These extend VersitPropertyContainer and reflect the vTodo and vEvent entities. Additionally, they override the toString() functions to include the start and end tags described above. References to instances of these are then held within VCalendar.

3.2.1.2 Versit Object Parser

A brief search revealed no open source vCard or vCalendar parser and as such it was decided to implement one for the purposes of the project. It was also felt that extensive knowledge of these standards would be of use when implementing the merging and equality tests discussed within Preparation.

The parser described within the following section was implemented as a helper function which, when passed a byte array, returns an array of VersitProperty objects. This is used within the conversion functions implemented in VCard and VCalendar objects. They then iterate over the VersitProperty array, adding each VersitProperty to the appropriate VersitPropertyContainer. The VCalendar implementation is slightly more involved; a reference to the active VersitPropertyContainer is held and altered when vEvent or vTodo start or end tags are encountered.

The string representation of a 'property' is presented within the vCard specification[10] as follows:

PropertyName [';' PropertyParameters] ':' PropertyValue

In addition to this, groups are specified by appending a common character followed by a dot ('.') to each grouped property. Including this, a 'property' may be defined as:

[Group '.'] PropertyName [';' PropertyParameters] ':' PropertyValue

In a number of cases, the 'PropertyValue' element itself constitutes a serialised array of items, similar to the PropertyParameters. Therefore, the final representation of a property is as follows:

---

[10] http://www.imc.org/pdi/

[Group '.'] PropertyName [';' PropertyParameters] ':' PropertyValue [';'PropertyValues]

In keeping with the Spiral Development Model, the functions for parsing Versit objects underwent a number of design iterations.

Initially, extensive use was made of the String.split() function to parse the Versit Objects into arrays of properties, then iterate through, parsing the component parts of the properties, the 'parameters' and 'values' into respective arrays. However, manufacturers seem to be reasonably liberal with their implementations of vCards and vCalendars and using this parsing technique proved problematic, especially when faced with vCard and vCalendar objects produced by Microsoft Outlook 2000.

The use of a parser generating tool such as Antlr[11] was considered but dismissed; a lack of experience of such tools meant it would require an unreasonably long time to configure and use. Furthermore, it was felt that the stricter parser that would be created by such a tool might prove more problematic than the method employed initially.

A state machine was therefore implemented (Figure 7). A number of convoluted sections were necessary to handle Versit Objects output by certain PIM applications. Despite this, the parser has been tested on a range of vCards and vCalendars from different applications and has proved reliable. Special notice should be taken of the transition from state VALUE to state EQUALS and its dependence on **quoted-printable**. This reflects the specification which allows 'Carriage-Return, Line-Feed' (CRLF) occurrences conforming to the MIME specification[12] if the property parameter 'QUOTED-PRINTABLE' is set. This addition was necessary, since, within some implementations, this encoding was not consistent with the usage of CRLF in none quoted-printable values.

---

[11] http://www.antlr.org
[12] RFC 1341

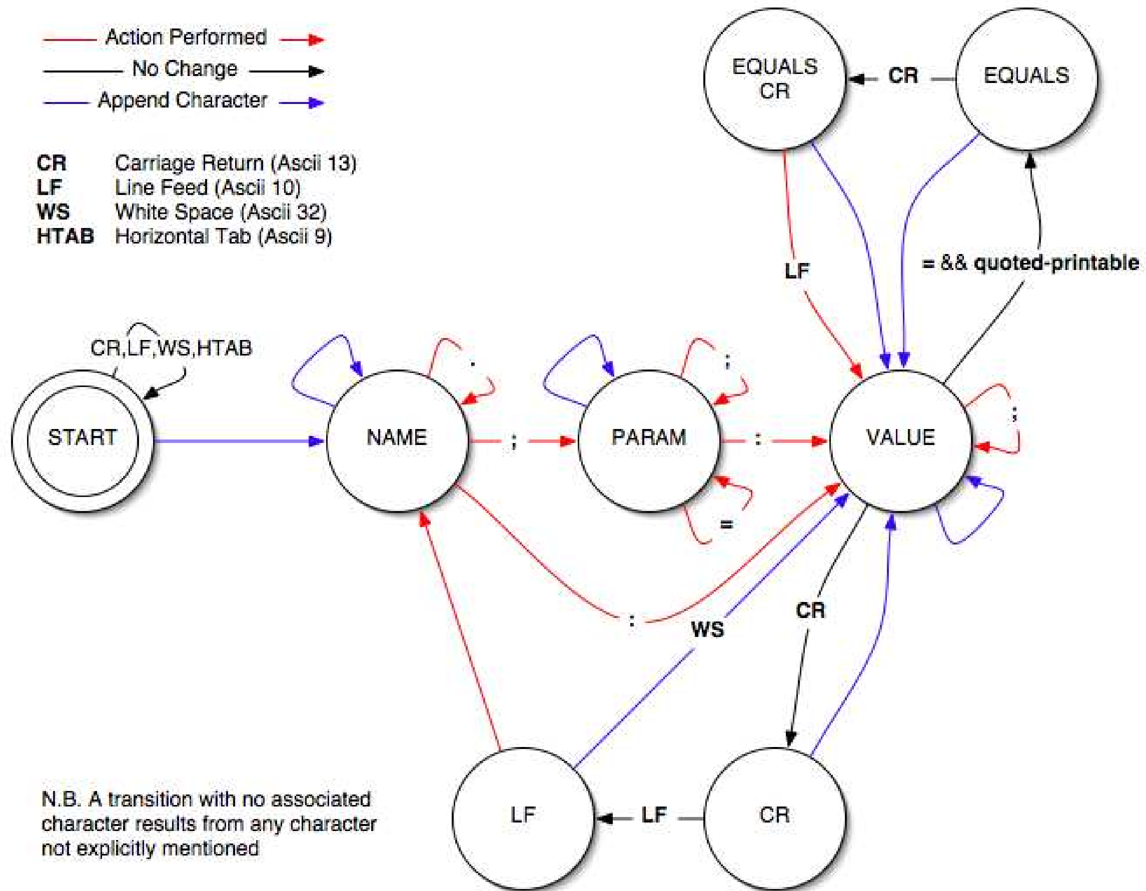

Figure 7 Versit Parser State Machine

This parser was implemented by means of a series of switch statements for each state. The complete source may be found in Appendix A.

```
switch( state ) {
      case STATE_NAME {
            switch( byteArray[ i ] ) {
                  ...
            }
      }
      ...
}
```

As the parser processes each Property line, a new VersitProperty object is instantiated and propagated with the property data. Following this, it is passed to a helper function which inspects the property name value and instantiates the appropriate subclass of VersitProperty.

3.2.2 Helper Functions - uk.co.in7.utils

It was anticipated that a number of small helper functions would need to be implemented during the project. Consequently, it was elected that these would be placed in a more generic package to reflect the fact that they did not necessarily relate directly to the project.

3.2.2.1 DateFunctions

While Java provides extensive date support by means of the GregorianCalendar class, a concrete implementation of Calendar, it was felt that all the functionality offered was unnecessary. DateFunctions provides a small number of calendaring functions, which are mostly used when displaying the calendar items:

```
String getDateString( int day, int month, int year )

String getDayString( int day )

String getLongDayString( int day )

String getMonthString( int month )

int getDayOfWeek( int day, int month, int year )

int getNumberOfDays( int month, int year )

boolean isLeapYear( int year )
```

These calculate the date as a offset from January 1st 1960. It was reasoned that few systems would require support for dates prior to this. Leap years are calculated by the commonly known check which defines a leap year as being divisible by 4 and not by 100, or by being divisible by 400.

3.2.3 Synchronization Items - uk.co.in7.insync

Key to the implementation of the project is the design of the objects which represent the items involved in synchronization.

3.2.3.1 SyncItem

Internally, the Sync4j server implementation uses a concrete implementation of sync4j.framework.engine.SyncItem to represent the items in a synchronization.

In the hope of providing an extensible architecture, java.util.Hashtable is used internally to allow key-value pairs to be stored. Crucially, `PROPERTY_BINARY_REPRESENTATION` and `PROPERTY_TIMESTAMP` are used within the synchronization process to store the binary representation of the item and the

timestamp of the last change to the object respectively. In the case of a vCard or vCalendar item, PROPERTY_BINARY_REPRESENTATION is the byte array representing the vCard or vCalendar.

Additionally, the SyncItem specifies a getKey() method. This key represents the Global Unique Identifier as mapped by the Sync4j server implementation. The server makes use of String objects for the key, appending a '-1' to each new instance of a previously seen Local User Identifier when creating a GUID mapping.

A custom implementation of SyncItem must override these functions.

3.2.3.2 InSyncItem

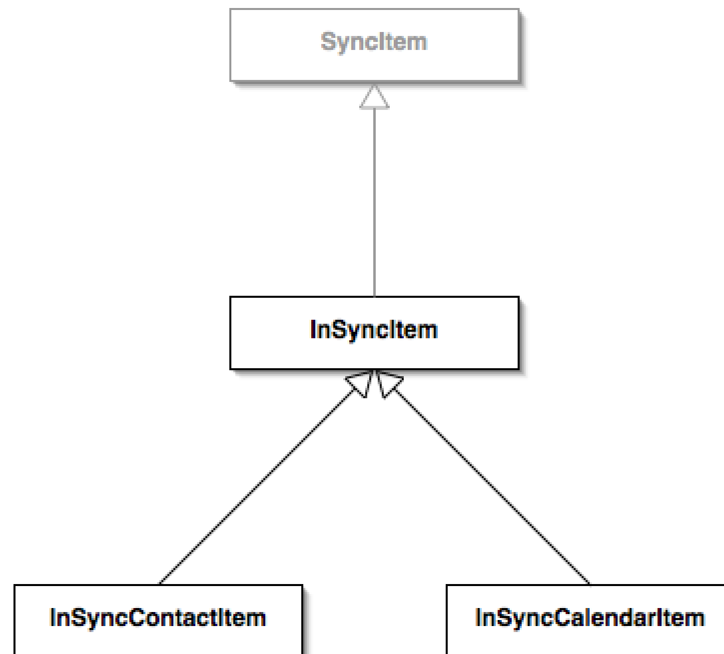

Figure 8 InSyncItem Class Structure

As discussed within Chapter 2, it was initially intended to implement an extensive custom data structure for Synchronization Items, so as to perform conflict handling using a common representation. This architecture offers a means whereby support for additional data representations can be added with ease, each converting to this common data structure. However, the grouping and possible duplication of properties found within the vCard and vCalendar formats made it extremely difficult to find a data structure which effectively represented them. It was decided that simple implementations would be provided for calendar items (InSyncCalendarItem) and contact items (InSyncContactItem) and efforts would be concentrated on producing an extensive representation architecture. Should these implementations prove unsatisfactory at a later date, alternative implementations could be provided within the same framework.

The InSyncCalendarItem and InSyncContactItem implementations hold references to VCalendar and VCard objects respectively to avoid code duplication. They implement accessor functions for Calendar and Contact specific data. Should a different implementation be required, it would only be necessary to provide implementations of these functions to work within the framework established. However, this was not abstracted into an interface due to time limitations. In addition, a number of comparator functions were implemented, as proposed within the Preparation chapter:

```
boolean equals( InSyncItem item )
boolean subset( InSyncItem item )
boolean truncated( InSyncItem item )
```

The implementations provided for these are simple, merely calling the respective methods in the VCard and VCalendar instances. In a more fully featured implementation, these would inspect the internal fields of the InSyncItem objects.

The use of a Hashtable presents difficulties when providing an implementation which does not represent a 'flat' data structure, requiring additional internal fields to store the VersitObject references. To ensure the InSyncItem objects still present the same interface to the Sync4j architecture, the accessors associated with the internal Hashtable were overridden. PROPERTY_BINARY_REPRESENTATION calls were caught and the appropriate VersitObject method called. In the case of the getter, the byte array is written to the Hashtable before calling the respective function in the superclass by means of super.method(). This uses the representation handler interface defined within InSyncItem. It is discussed in more detail within the following section.

3.2.3.3 Representation Handler

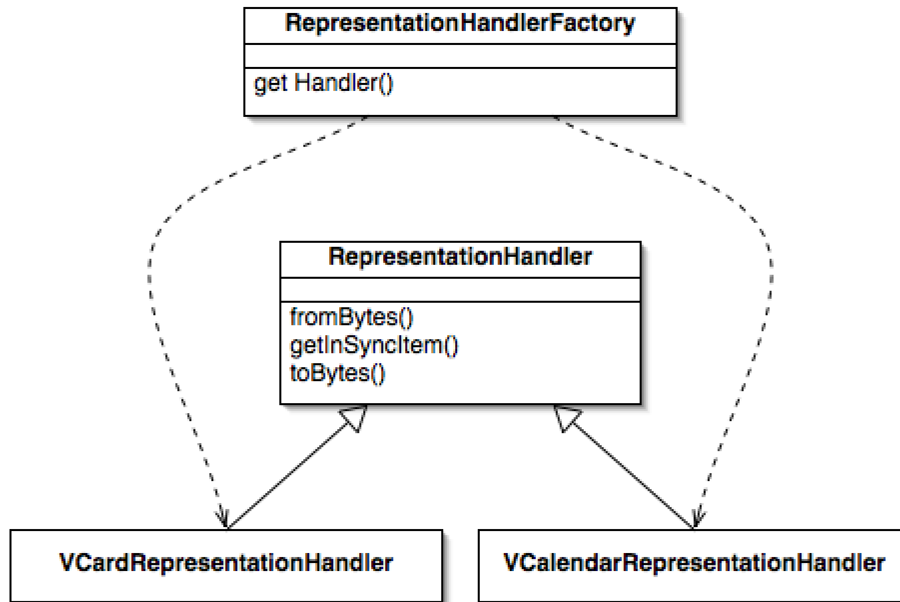

Figure 9 Representation Handler Class Structure

To provide an extensible architecture, the Representation Handler was implemented in accordance with the Abstract Factory Pattern. As shown in Figure 9, the interface, RepresentationHandler, defines those functions required for converting an InSyncItem to and from a byte array, fromBytes() and toBytes(). Additionally, the function getInSyncItem() provides a method for converting a SyncItem to an InSyncItem, so that the SyncItem objects passed by the Sync4j architecture may be easily handled.

VCardRepresentationHandler and VCalendarRepresentationHandler are concrete implementations of this interface, providing support for the vCard and vCalendar formats. As the InSyncCalendarItem and InSyncContactItem implementations contain VersitObject references, these are simple, making use of the VersitObject methods. They are essentially toy implementations with the purpose of demonstrating the architecture. As the RepresentationHandler interface defines the functions in terms of InSyncItem objects, there is no way to enforce that the correct InSyncItem subclass is passed. Whilst not ideal, this is done to maximise code reuse, relying on checks within the RepresentationHandler implementations.

The RepresentationHandlerFactory implements a 'getHandler()' function which may be passed a constant (public static final) to define the RepresentationHandler subclass desired. This implements a case statement which instantiates and returns the appropriate RepresentationHandler implementation. In hindsight, the architecture could have been made more extensible by making use of reflection, so that the implementations need not be hard-coded. However, this would require extensive exception handling, not

currently present within the architecture. The RepresentationHandlerFactory enforces a singleton design pattern on the RepresentationHandler implementations by storing an instance in a Hashtable. If an instance of an appropriate RepresentationHandler already exists, it is returned, otherwise, a new one is instantiated and inserted into Hashtable. This is used to reduce the memory overheads and is possible as the RepresentationHandler is stateless.

In addition to those discussed previously, the InSyncItem class defines a number of methods for handling the representation:

```
byte[] toBytes()
void fromBytes( byte[] bytes )
void fromSyncItem( SyncItem item )
void setRepresentationHandler( RepresentationHandler handler
)
void setRepresentation( int representation )
```

These methods are defined in order to ensure that a generic implementation may be presented to the Sync4j framework. InSyncItems may hold a reference to an appropriate RepresentationHandler, which is then used when the binary representation is requested. In this way, the generic InSyncCalendarItems and InSyncContactItems may provide a specific format to the Sync4j server implementation.

3.2.4 Persistent Storage - uk.co.in7.insync.database

As discussed in the Preparation chapter, MySQL was selected for the database implementation. While the Sync4j architecture provides support for a wide range of databases, it was considered unnecessary to provide this within the Sync4j Module, given the small scale of the project. Additionally, as the Java MySQL Connector, Connector/J, works within the JDBC architecture, adding other JDBC supported databases should prove reasonably easy.

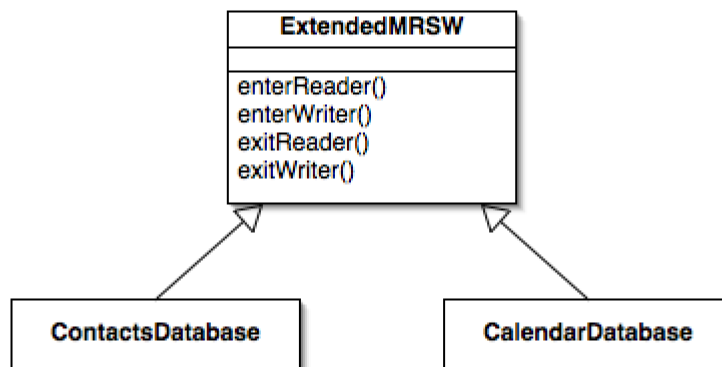

Figure 10 Database Class Structure

Figure 10 shows the structure of the uk.co.in7.insync.database package. The object design reflects the clear distinction between calendar and contacts synchronization tasks. While both of these implement the same methods, a common interface is not used to allow the specific InSyncItem types, InSyncCalendarItem and InSyncContactItem, to be declared. The following methods are implemented by CalendarDatabase:

```
InSyncCalendarItem[] getAllInSyncItems(String user)

InSyncCalendarItem[] getDeletedInSyncItems(String user,
                                           Timestamp since)

InSyncCalendarItem[] getNewInSyncItems(String user,
                                       Timestamp since)

InSyncCalendarItem[] getUpdatedInSyncItems(String user,
                                           Timestamp since)

boolean removeInSyncItem(String user,
                         InSyncCalendarItem item)

int setInSyncItem(String user, InSyncCalendarItem item)

InSyncCalendarItem getInSyncItem(String user,
                                 SyncItemKey syncItemKey)

InSyncCalendarItem[] getInSyncItemVersions(String user,
                                           String luid)
```

In these methods, the Timestamp, since, is passed by the Sync4j framework and represents the last time a synchronization was performed with the device currently synchronizing.

Additionally, CalendarDatabase and ContactDatabase follow the Singleton Design Pattern, providing a getInstance() method. This design pattern is used to allow the implementation of locking on the database and is discussed in greater detail in Section 3.2.4.2.

3.2.4.1 Table Design

A very simple table design was selected to minimise both the work required in writing the java accessors and the number of queries required on the database. The table design discussed here is dependant on the use of VCard and VCalendar objects internal to the InSyncItem implementations.

During the initial design phases, it was intended that VersitProperty objects would be stored as individual rows in one table, alongside an appropriate item identifier or

'itemid'. Rows reflecting the items themselves would then be stored within a second table, indexed on this 'itemid'. Performing a join of the two would result in the items being retrieved from the database. This offers a number of advantages in that SQL queries can be constructed to allow items to be selected based on internal fields. This places any searching load onto the database, which is suitably optimised for such functions.

However, this implementation requires subtly different database designs for the Calendar database and the Contacts database. A simpler design therefore adopted, using a single table with Versit Objects stored as a byte array in a BLOB field. While this places increased load on the Java implementation when selecting items based on fields internal to the Versit objects, this occurs very infrequently, only when displaying items. As the user interface was never intended to be a primary feature of the project, this was considered acceptable.

Items	
Itemid	INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY
Active	BOOL NOT NULL
Deleted	BOOL NOT NULL
Item	BLOB NOT NULL
sourceitem	BLOB
clientitem	BLOB
resolution	INTEGER
modified	TIMESTAMP NOT NULL
User	VARCHAR(244) NOT NULL
Guid	VARCHAR(255) NOT NULL

The database is designed so that three items may be stored: 'item', the active item, 'sourceitem', the item sent by the source in the case of a conflict and 'clientitem', the item sent by the client in the case of a conflict. The use of a third field to store the active item may at first seem unnecessary. However, this allows the active item to be created by merging the two conflicting items. In this case, it is still necessary to store the both conflicting items. In addition, the resolution field stores an integer representing how the conflict was resolved.

The 'user' field is stored so that a single table can store data from multiple users. When any selection is performed, the 'user' field is specified to pull only items relating to the user in question.

This table design allows for a history of item revisions to be stored by making the assumption that the AUTO_INCREMENT 'itemid' field will not use the first available integer, but will use the last assigned integer + 1. This is the case in the MySQL database implementation and ensures that later revisions of an item have higher 'itemid' values and vice versa. Items are stored with their Global Unique Identifier in the 'guid' field so

that an entire history of items may be pulled out by a simple select query. The active item of a set is indicated by the BOOL 'active'. Whenever an item is altered, the new revision is added and 'active' is set to false in any previous versions. In this way, any change is recorded and timestamped using the 'modified' field. Deletion is handled by inserting a new version of the item with the 'deleted' field set to true.

Performing change detection within the database is then a matter of simple SELECT queries. A select for items where the 'modified' field is greater than the timestamp passed by the Sync4j framework gives all those items which have changed after that time. In addition, specifying the value of the 'deleted' field allows deleted items to be separated from modified or created items.

As can be seen from the database functions outlined previously, the Sync4j architecture requires that there is a distinction between newly created items and modified items. The current database design makes it difficult to identify the first item in a history. Therefore a second table is added to store the creation times of each item, indexed by the GUID. Pulling out the creation time of an item is then merely a matter of performing a join of these two tables on 'guid'.

created	
guid	VARCHAR(255) NOT NULL PRIMARY KEY
created	TIMESTAMP NOT NULL

However, maintaining this 'created' table presents some problems. The Sync4j framework defines only one function for inserting a SyncItem into the database, making no distinction between updating an item and adding a new one. This is reflected in the Database implementation with the function setInSyncItem(). This function must therefore be capable of determining whether the item is already present in the database and, therefore, whether a row need be inserted into the created table. In order to reduce the number of SQL queries required when inserting items, the 'IGNORE' keyword is used to perform this check:

```
INSERT IGNORE INTO created (guid, created) VALUES ...
```

The IGNORE keyword means that any row in the insert statement which duplicates an existing PRIMARY KEY in the table is ignored. Therefore, the above statement will only successfully complete the first time an item with a particular guid is inserted. In this way, it is reasonable to execute the statement each time, essentially allowing the MySQL implementation to provide the conditional execution.

3.2.4.2 Locking

The sequential nature of a SyncML exchange means that a device must hold a lock on the data store for the entire duration of a synchronization. As shown within the table design, an item database may be regarded as a number of discrete sets of items, each relating to a different user. As such, two different users may be allowed concurrent access to the database table. The use of table locking through MySQL would therefore be inefficient and seriously hamper the scalability of the server. To avoid this, the locking was implemented by means of an adapted Writer Prioritising Multiple Reader, Single Writer (MRSW) locking architecture as discussed within Part IB Concurrent Systems (Appendix A). The MRSW implementation maintains a counter of the number of readers and the number of writers, only admitting a writer when the numbers of readers reaches zero. An additional counter which stores the number of waiting writers allows readers to be refused when writers are waiting. The MRSW implementation has been extended to provide separate locks for each user, storing multiple instances of the counters, hashed on the username. Since the username is unique, hash conflicts are guaranteed will not occur.

When performing a Synchronization, the SyncSource implementation (see 3.2.5) acquires a write lock in anticipation of writing any changes to the database. Should it require a read lock and attempt to upgrade this to a write lock, the system would either experience deadlock or would fail to hold a lock on the database between releasing the read lock and acquiring the write lock. During this time, a write lock might be acquired by a different 'device'.

3.2.5 Sync4j Interface - uk.co.in7.insync.source

The uk.co.in7.insync.source package reflects the source package within the Sync4j architecture. It contains only two classes, the two AbstractSyncSource implementations, InSyncVCalendarSource and InSyncVCardSource. These handle the vCalendar and vCard synchronizations respectively.

The AbstractSyncSource implementations are responsible for getting a write lock on the database, setting the appropriate representation handlers and calling the suitable functions on the database. This keeps specific code to a minimum, ensuring that support for other representations can be added easily.

3.2.6 User Interface - uk.co.in7.insync.ui

The User Interface was implemented by means of a Java Servlet, using XHTML and CSS for representation. Java Server Pages (JSPs) were not used to minimize additional learning. If time had allowed, this would have been a more ideal solution, as it helps to separate content and layout with templates defined as JSPs. In order to ensure the 'portability' of the User Interface, the design was implemented solely within CSS,

meaning that less-featured web browsers would be able to view a simpler version. The use of CSS and strict XHTML also improves the accessibility of the web page, ensuring a good separation of content from design.

The user interface was divided into two distinct sections, calendar and contacts, in keeping with much of the project. A tab based design was used to provide scope for the addition of further 'views':

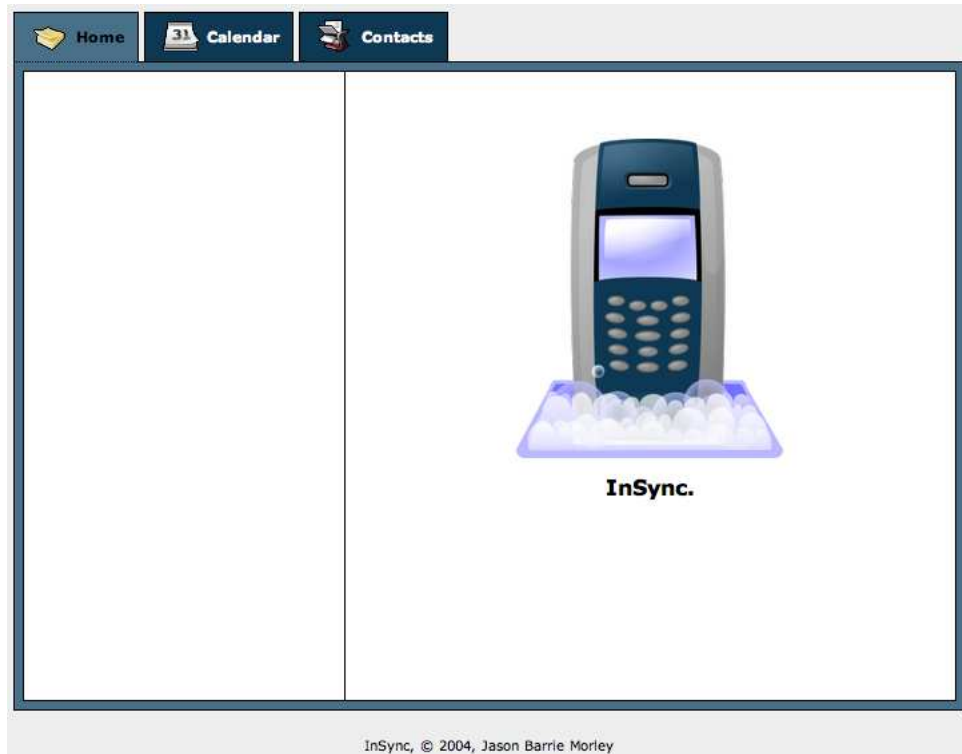

Figure 11 InSync Home Tab

The InSyncServlet defines a number of functions for printing different elements of the HTML, with the tabs and the two panes being separated. A number of methods were defined to inspect and print InSyncCalendarItem objects and InSyncContactItem objects. These essentially print the internal vCard and vCalendar objects, iterating over the VersitProperty objects. Groups, vTodo and vEvents are represented as nested tables:

Figure 12 InSync Contacts Tab

These functions are essentially debug functions to allow the entire vCalendar or vCard to be inspected while on the server. In keeping with the Spiral Development Model, these were initially implemented for the Calendar and Contacts sections, so that a functional user interface could be rapidly developed. The Calendar interface was later extended to provide greater usability.

When instantiated, InSyncServlet inspects the parameters passed and acquires a read-lock on the appropriate database. This is released when the page has been printed.

3.2.6.1 Calendar Interface

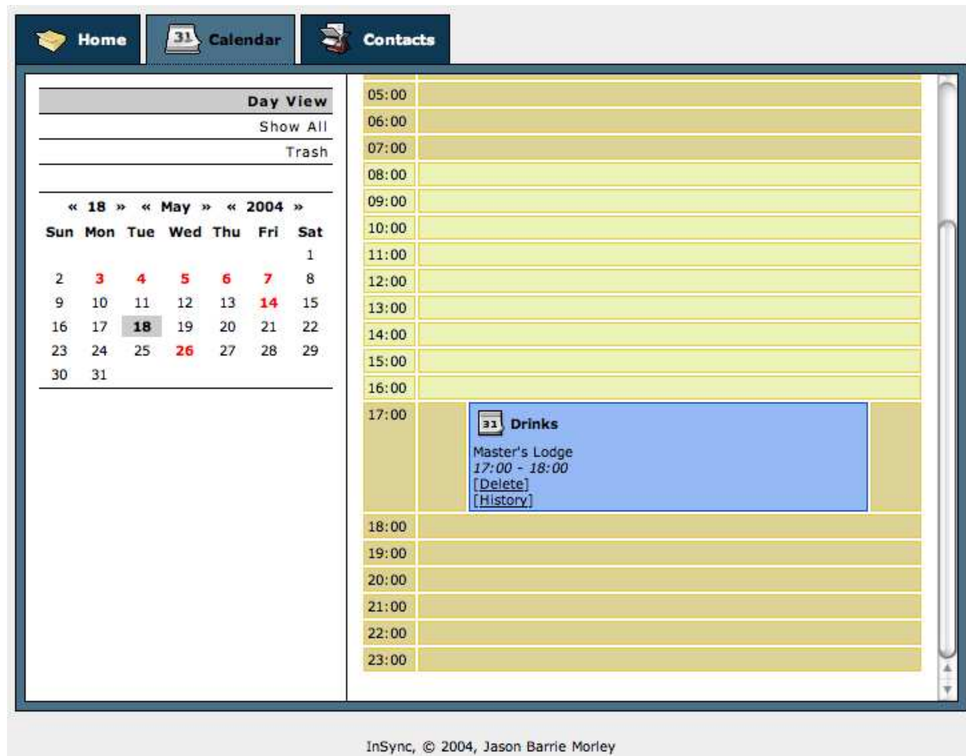

Figure 13 InSync Calendar Tab

The revised calendar user interface provides a number of different views:

- **Day View**
- **Show All**
- **Trash**
- **Item History**

The user interface centres around the 'Day View', which lists the day's appointments in a way similar to Microsoft Outlook or Apple iCal. This day view is encoded as an HTML table to ensure that clients not supporting CSS are able to display appointments. The 'rowspan' and 'colspan' properties are used extensively to allow for both overlapping appointments and appointments spanning multiple hours. Internally, two passes must be made over the InSyncCalendarItem objects to be displayed, the first, to establish the maximum number of overlapping appointments and therefore the table width and the second, to print the rows. When printing items, durations are rounded up to the nearest hour. A number of special cases must be taken into consideration, such as 'reminders' which specify the same start time and end time.

The 'Show All', 'Trash' and 'Item History' views make use of the methods described previously for displaying a list of InSyncCalendarItem objects – they execute

the appropriate queries on the database and print those items returned. In addition, they provide the user with options to revert to a previous version, to delete or to un-delete an item. These cause the InSyncServlet to request a write-lock on the database and make the appropriate changes.

In addition to these views, a Month overview is always visible, providing a quick way for users to navigate to a particular date.

3.2.7 Mapping and Conflict Handling - uk.co.in7.insync.engine

As discussed in the Preparation section, it was initially intended that the project would provide a reimplementation of the synchronization strategy, Sync4jStrategy. While suitable methods were implemented within InSyncItem to allow this, time proved too limited for it to be attempted. The current project implementation therefore relies on the Sync4j framework for mapping and collision handling.

3.3 Sync4j 2.0

Version 2.0 of the Sync4j framework was released towards the end of my development period. While it appeared to offer significant improvement in the API for module developers and a number of additional utilities for server configuration, it was decided not to attempt to port the code to this later framework, since it was not required to demonstrate functionality. This work may be undertaken at a later stage as a contribution to the Sync4j project.

Sync4j 2.0 does however include a far more extensive client implementation, with modules to allow it to synchronize Microsoft Outlook with SyncML. If time allows, this will be used during the evaluation to enable another client to be tested.

Chapter 4 Evaluation

4.1 Testing

Various levels of testing were carried out during the different stages of development. These may be divided up into:

- **Unit Testing**
- **Integration Testing**
- **Full System Testing**

Adherence to the Spiral Development Model meant that code was rapidly integrated, with much of the testing being carried out at this stage. Testing in this manner helped ensure that problems were identified as soon as possible. These various levels of testing will be discussed within the following sections.

4.1.1 Unit Testing

Where possible, classes and methods were tested by means of authoring custom 'test harnesses'. This was used to great effect when implementing the DateFunctions class, where it revealed the calculation of leap years to be incorrect in the initial implementation by the return values against known dates.

Additionally, A toy application was authored when implementing the database functions so as to become accustomed with the Java MySQL Connector. This proved useful in allowing SQL statements to be tested before implementation.

4.1.2 Integration Testing

Integration of the system was carried out in the following order:

- **vCard and vCalendar Parser**
- **InSyncItems and Representation Handler**
- **Database**
- **User Interface**

Different tools were used during each stage of integration to assess each aspect of the project.

4.1.2.1 vCard and vCalendar Parser

During the integration phase, the vCalendar and vCard Parser was tested by making use of the Sync4j logging architecture to print the fields parsed. This proved successful, allowing minor bugs to be identified and the parser to be refined so as to provide support

for a variety of interpretations of the standards. Use was made of the Sync4j command-line client to 'synchronize' a number of test items to the server during this process – vCalendar and vCard files were output from a range PIM applications and added to the Client for synchronization.

4.1.2.2 InSyncItems and Representation Handler

The InSyncItems and Representation Handler architecture were tested in much the same way as the Parser, with feedback being provided by means of the Sync4j logs. Code was authored to allow the comparators to be tested at this stage, so that errors were identified before the functions were relied on for mapping generation.

4.1.2.3 Database

A significant portion of the database implementation was tested during the integration as User Interface specific functions were added when required. phpMyAdmin[13] was used to view the database, allowing for a number of bugs in the supporting methods to be identified. Additionally a number of test scenarios were generated by manually editing the database in this way.

4.1.2.4 User Interface

The user interface is most suited to the Spiral Model of development, with designs being rapidly implemented and tested. Much of this focused around compatibility testing to ensure a consistent interpretation of the CSS based design. Problems were encountered when using Microsoft Internet Explorer, however, it was deemed that this did not unduly affect the usability of the application and these were ignored. Mozilla, Safari, Opera and FireFox all displayed the interface as intended.

4.1.3 Full System Testing

In order to ascertain whether or not the project met the acceptance criterion outlined within this document, it was necessary to perform full system testing with a couple of client implementations. Both 'client-to-server' and 'server-to-client' synchronizations were tested with the three key actions; create, edit and delete. In order to test this efficiently, the following sequence of actions and synchronizations was chosen:

[13] http://www.phpmyadmin.net

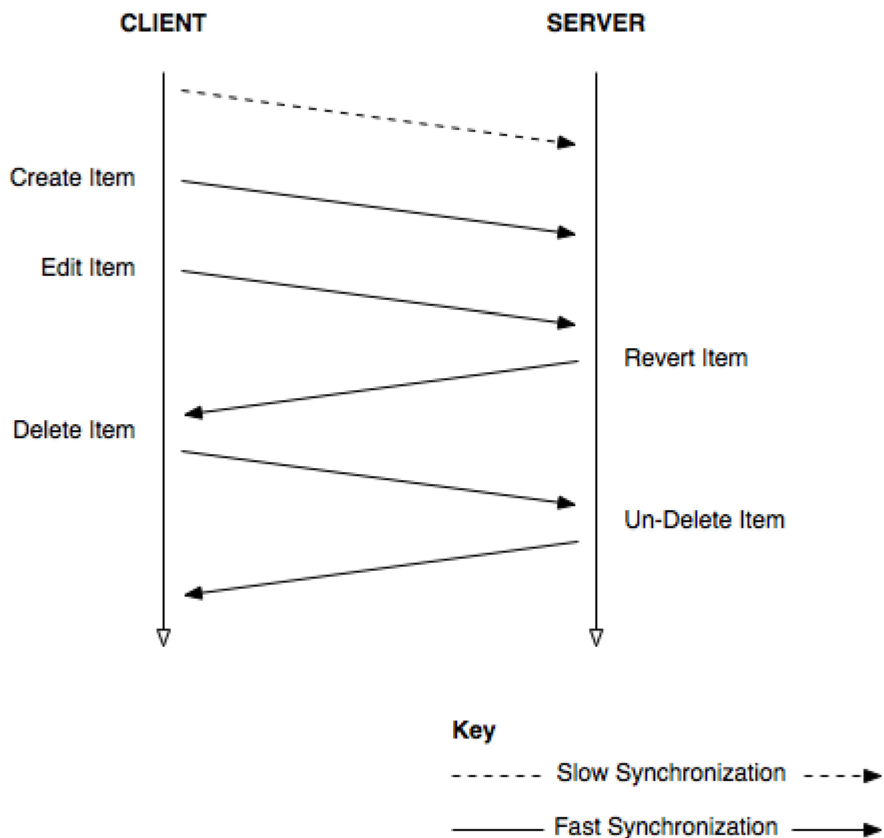

Figure 14 Synchronization Test Sequence

A Sony Ericsson P800 was used for the testing and was configured as appropriate for the Server. This testing focuses on the calendar implementation, as the user interface for the contact implementation was not completed to an appropriate level. As the calendar and contacts implementations are very similar, this serves as an adequate assessment of the contact implementation.

In order to fully test the project, a final test was executed, Synchronizing both the P800 and another client with the database in order to merge their data.

4.1.3.1 Initial Synchronization (Slow Synchronization)

Figure 15 P800 Prior to Synchronization

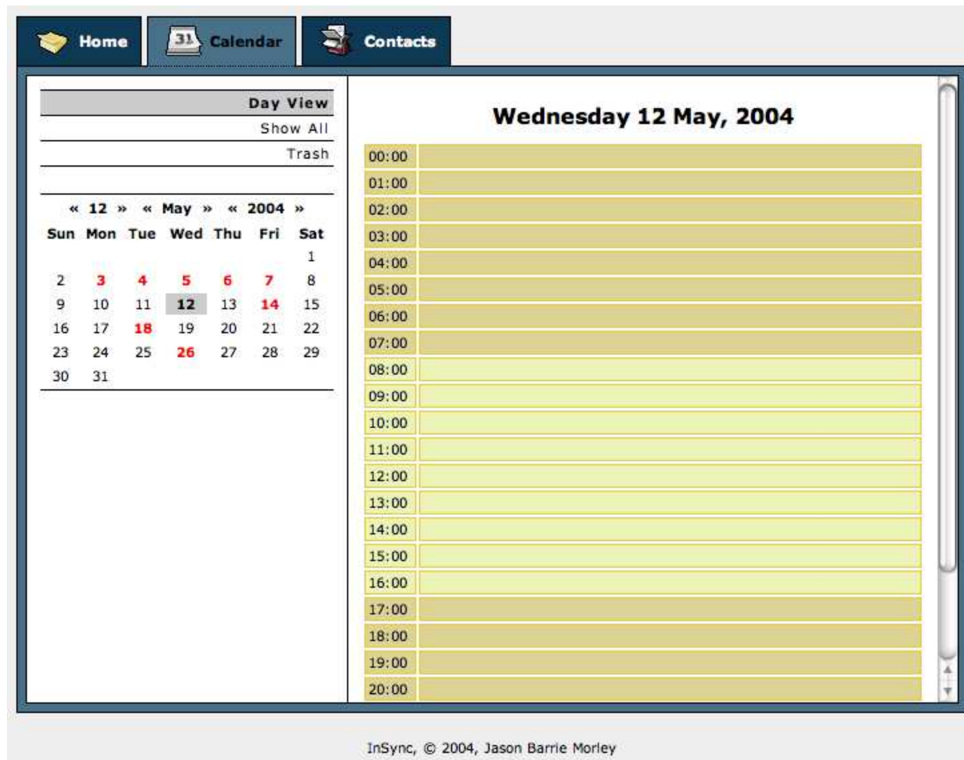

Figure 16 Server Following Slow Synchronization (Calendar)

The initial, or slow, synchronization was performed between the P800 and the Server and proved successful. Comparing the month views of Figure 15 and Figure 16 reveals some apparent inconsistency in the days marked active. However, this is merely due to a different interpretation of appointment types. The P800 differentiates between ‘all-day’ events and ‘normal’ events, leading to the light blue and dark blue entries. The Server does not.

Additionally, Figure 17 shows the contacts tab, demonstrating that contact synchronization also completed successfully during this initial stage.

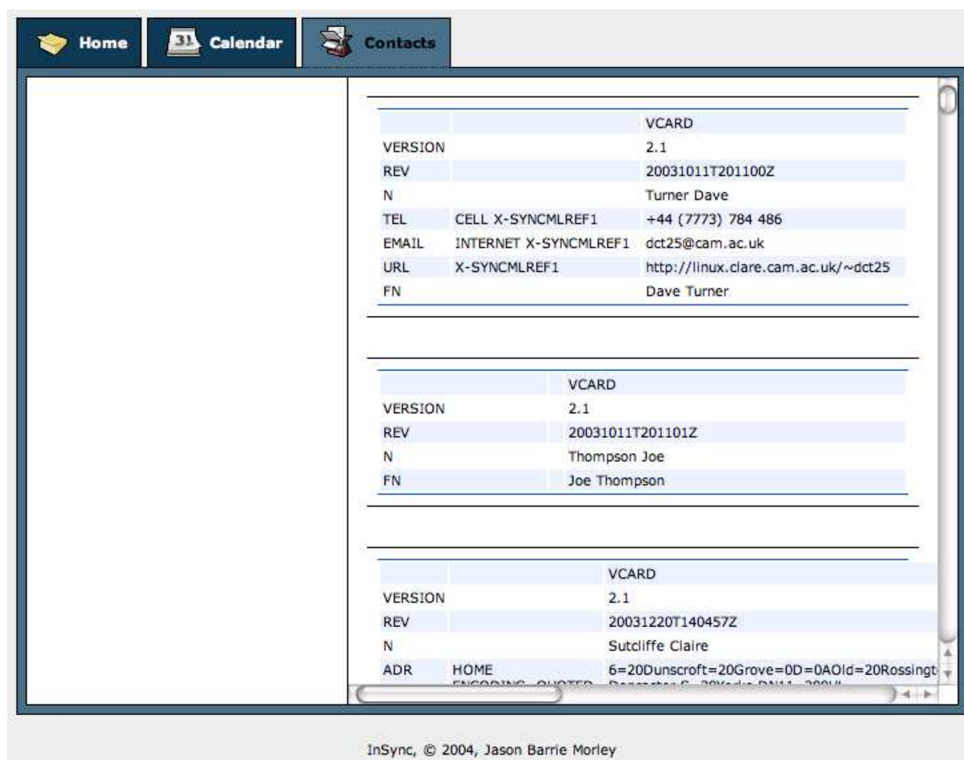

Figure 17 Server Following Slow Synchronization (Contacts)

4.1.3.2 New Appointment

A new appointment was created on the P800 with the name ‘New Appointment’, a start time of 9.00am, end time of 11.00am and the location ‘Gates Building’. Performing a synchronization successfully transferred this to the server as can be seen in Figure 18.

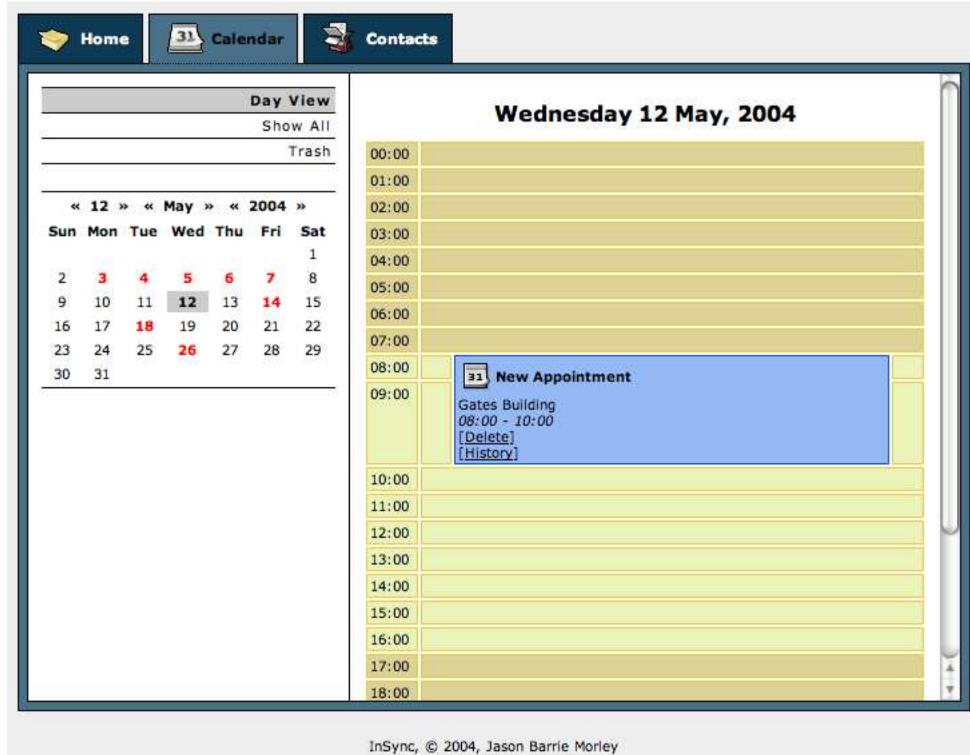

Figure 18 Server Showing New Item From Client

Although the item was synchronized successfully, it displays a start time of 8.00am and an end time of 10.00am. It highlights a bug in the Server implementation whereby the time zone is not considered. The vCalendars are stored and transmitted by the P800 in Greenwich Mean Time (GMT), while they are displayed in British Summer Time (BST). As the Server takes no account of the time zone encoded in the vCalendar format, it displays them in GMT leading to the scenario seen above. While a potential problem, this does not affect the action of synchronizing.

4.1.3.3 Edit Appointment

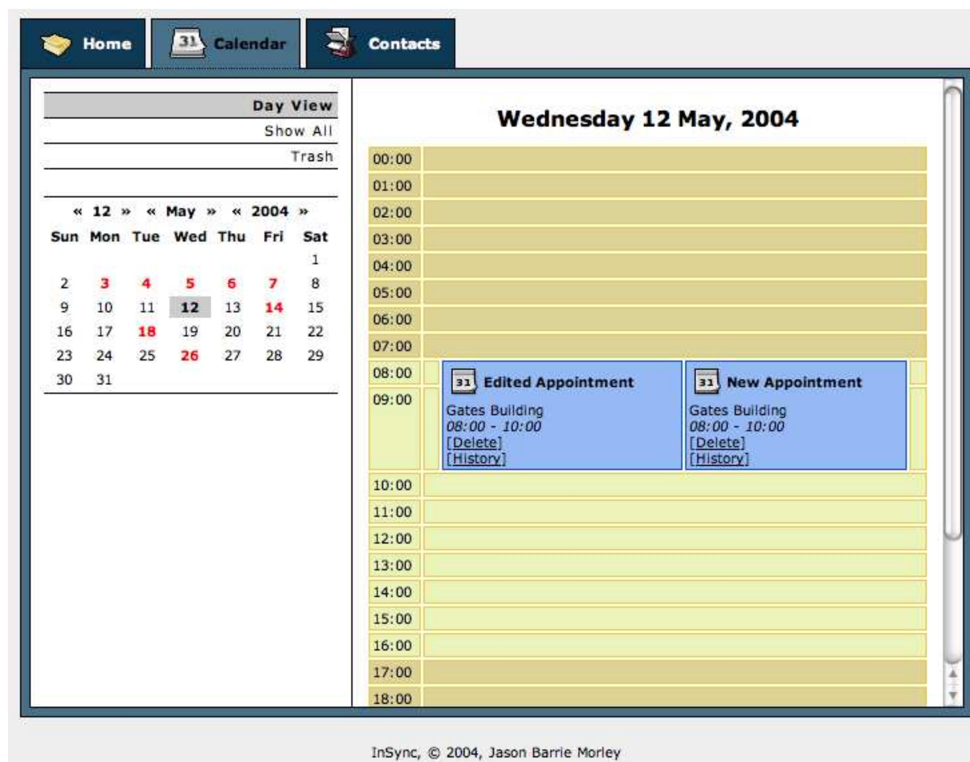

Figure 19 Server Showing Duplicate Items

The newly created item was edited on the P800, with the name being changed from 'New Appointment' to 'Edited Appointment'. Synchronizing this item with the Server proved unsuccessful, with the item being duplicated on the Server as can be seen within Figure 19. An SQL query was performed on the 'calendar_items' database to provide further information regarding this issue:

```
mysql> SELECT itemid, active, deleted, resolution, modified, user, guid FROM
calendar_items;
+--------+--------+---------+------------+----------------+-------+-------+
| itemid | active | deleted | resolution | modified       | user  | guid  |
+--------+--------+---------+------------+----------------+-------+-------+
|      1 |      1 |       0 |       NULL | 20040512084451 | jason | 41    |
|        |        |         |            |                |       |       |
|        |        |         |            |                |       |       |
|    161 |      1 |       0 |       NULL | 20040512090544 | jason | 407   |
|    162 |      1 |       0 |       NULL | 20040512090819 | jason | 407-1 |
+--------+--------+---------+------------+----------------+-------+-------+
162 rows in set (0.00 sec)
```

A sample of the rows returned by the SELECT query can be seen. The last two rows relate to these duplicated items, showing them to have different 'guid' values. This suggests that the Sync4j mapping from LUID to GUID is failing. Further investigation revealed a similar bug filed against the Sync4j framework. Additionally, the server logs

suggested that during the initial mapping phase, the internal 'key' field for the SyncItem was not being set to the GUID, while on other occasions, it was. This would offer an explanation for the presence of these two similar 'GUID' values – the second post fixed by '-1'. The knowledge that Sync4j GUID values are created by appending one or more instances of '-1' to the end of the LUID reinforces this.

In order to test this theory, the item was edited a second time on the P800, changing the end time to 10.00am. Figure 20 shows the items on the Server following the synchronization. Accounting for the time zone issues discussed previously, it can be seen that the synchronization completed successfully, altering the second instance of the item on the server – Figure 21 shows the history of changes for this item. Additionally, while far from authoritative, this goes some way to confirming the suggested explanation. Any subsequent modifications to this item should therefore not encounter this duplication problem.

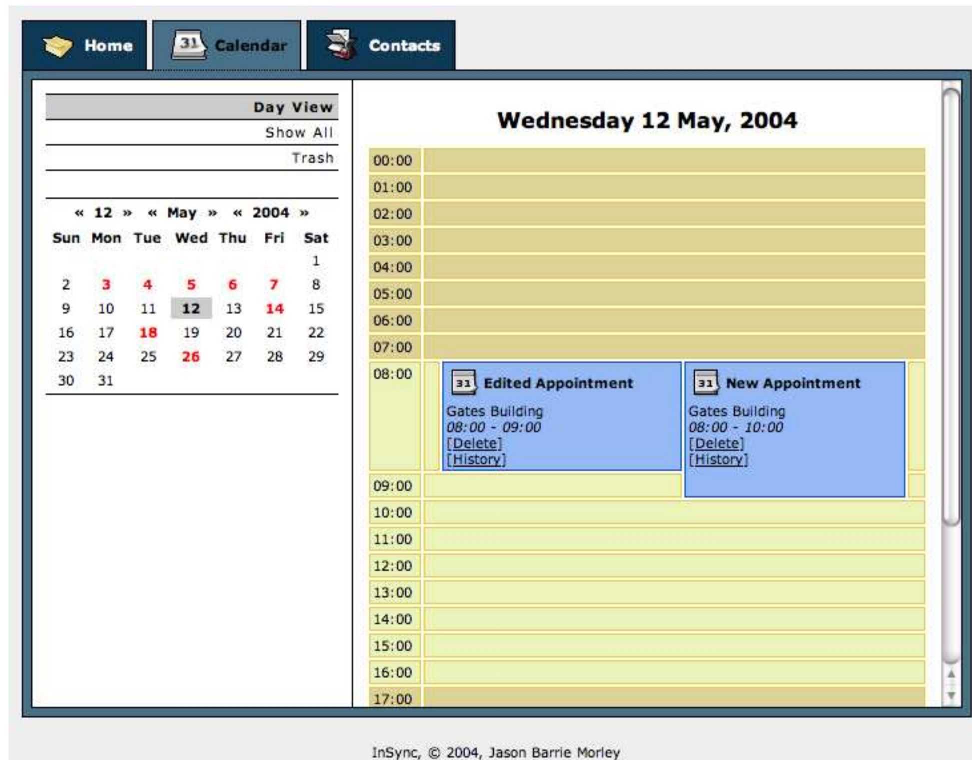

Figure 20 Server Showing Edited Item

Figure 21 Server Showing Item History

4.1.3.4 Revert Item on the Server

By means of the user interface presented in Figure 21, the item was reverted to its previous version. Synchronizing with the P800 successfully transferred this previous version back, showing the modification history implemented on the Server to work.

4.1.3.5 Deleting Item

This item was then deleted from the P800, and again the phone was synchronized with the server. Figure 22 shows this to have been successful, with only one version of the item remaining on the server.

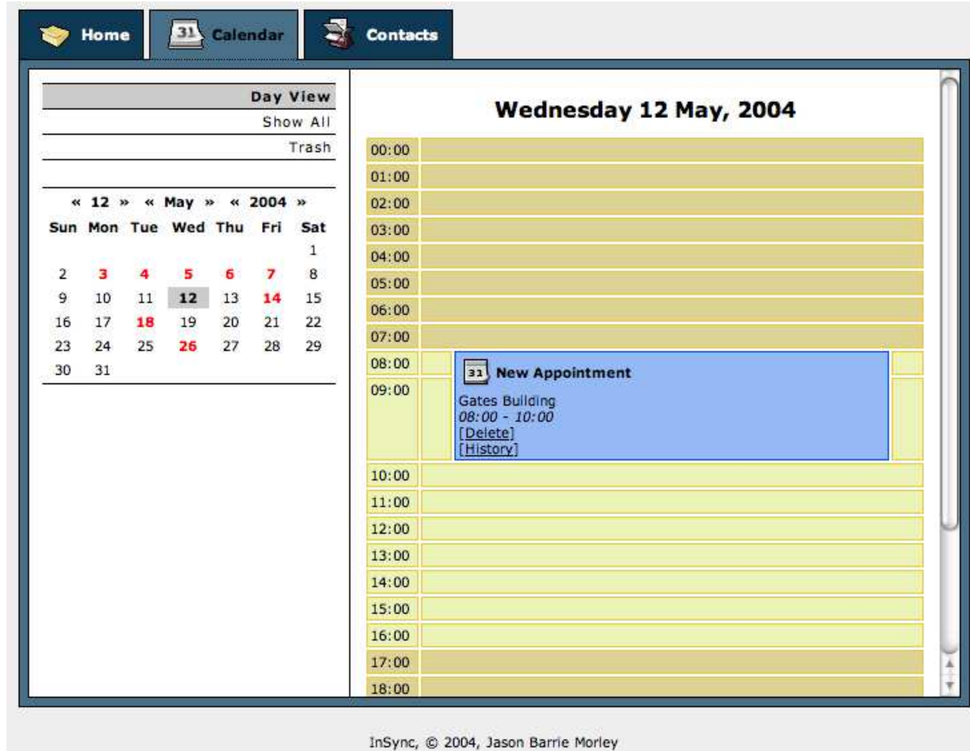

Figure 22 Server Following Item Deletion

4.1.3.6 Un-Deleting Item

Finally, the 'trash' and 'un-delete' elements of the Server were tested. Figure 23 shows the recently deleted item in the 'trash' view, demonstrating this aspect of the user interface to work. By means of the 'Un-Delete' option presented, the item was restored and successfully returned to the 'Day View'. Synchronizing the P800 caused the item to reappear, showing this to be a success.

Figure 23 Server Showing Deleted Item

4.1.3.7 Dual Client Synchronization

In order to fully test the project, the Sync4j command-line client was used to attempt the merging of two sources. Following synchronizing the P800, a number of entries were added to the command-line client and it was synchronized with the Server. This resulted in the entries on the client being mirrored to the Server and the entries from the P800 appearing on the client. A subsequent synchronization of the P800 caused the entries originating on the command-line client to be added to the phone, showing the merge to be successful.

Chapter 5 Conclusions

The project has suffered a number of set backs and ran the risk of not being completed. However, a successful pre-emption of this at the time of the progress report allowed the success criterion defined within the Project Proposal to be acheived – the creation of a merged diary from two different sources.

In addition to this, the project goes some way to implementing those concepts proposed within the Introduction and Preparation chapters:

- **Item Comparators –** The project implements synchronization items with support for a range of comparisons operations suitable for use within conflict handling and mapping generation.

- **Representation Handler** – The project provides an extensible representation handler architecture based on the Abstract Factory Pattern. This anticipates the need to provide support for additional representation formats and currently handles vCalendars and vCards.

- **Revision Control** – The project uses MySQL to implement a persistent store capable of maintaining a history of item modifications. This provides a number of functions, allowing items to be reverted to earlier versions and, implements a framework for altering the outcomes of conflict resolutions.

- **User Interface** – The project provides a web-based user interface through which users may view the data stored on the server. Additionally, this has support for performing item version control and deletion.

Despite this success, a number of additional concepts discussed were not able to be implemented due to time limitations and problems with the underlying framework. These form the basis for a number of possible extensions:

- **Conflict Resolution** – The architecture implemented by the project makes it possible to take a more involved approach to conflict resolution. A wide range of heuristics could be considered, from modification times to Natural Language Processing techniques.

- **Mapping Generation** – While similar to Conflict Resolution this presents a number of unique problems. Experience with the Sync4j framework shows its

implementation to be unreliable, leading to the conclusion that a custom implementation would greatly improve performance.

- **User Interface Extensions** – A natural extension to the user interface would be the addition of support for editing items stored on the server. In addition, a more user-friendly implementation of the Contacts view would be required in any real-world application.

- **Additional Sources** – Many SyncML client applications provide support for synchronizing other items, such as notes or email. The architecture implemented within this project should make supporting these additional formats relatively easy.

- **XML** - Despite good initial intentions, the structure of the Versit Objects became unduly complex and reflected too heavily the initial design of the Parser. On reflection, it may have been wiser to use an XML format internally for InSyncItems. This preempts XML based representation formats such as xCalendar. It also adds scope for the use of XSL Transformations to convert between different representations.

Should time have allowed, these extensions would have progressed the project beyond the basic requirements and provided a synchronization engine with better functionality than those currently in existence.

Appendix A Sample Code

Versit Parser

```
// Constants Representing States.
private static final int STATE_START = 0;
private static final int STATE_NAME = 1;
private static final int STATE_PARAM = 2;
private static final int STATE_VALUE = 3;
private static final int STATE_EQUALS = 4;
private static final int STATE_CR = 5;
private static final int STATE_LF = 6;
private static final int STATE_EQUALSCR = 7;

// ASCII Character Constants.
private static final byte CR = 13;
private static final byte LF = 10;
private static final byte SPACE = 32;
private static final byte HTAB = 9;
private static final byte EQUALS = 61;
private static final byte COLON = 58;
private static final byte SEMICOLON = 59;
private static final byte PERIOD = 46;


/// Parser Method.
public static VersitProperty[] getProperties( byte[] object ) {

  VersitProperty activeProperty = new VersitProperty();
  VersitPropertyParameter activeParameter
    = new VersitPropertyParameter();
  int state = STATE_START;
  String current = "";
  boolean quotedPrintable = false;
  ArrayList arrayProperties = new ArrayList();

  for( int i = 0; i < object.length; i++ ) {

    switch( state ) {

      case STATE_START:
        {

          // Analyze the character.
          switch( object[ i ] ) {

            case CR:
              break;
            case LF:
              break;
            case SPACE:
```

```
          break;
        case HTAB:
          break;
        default:
          {
          current = current + byteToString( object[ i ] );
          state = STATE_NAME;
          break;
          }

        }

        break;
    }
  case STATE_NAME:
    {

      switch( object[ i ] ) {

        case COLON:
          {
          activeProperty.setName( current );
          current = "";
          state = STATE_VALUE;
          break;
          }
        case SEMICOLON:
          {
          activeProperty.setName( current );
          current = "";
          state = STATE_PARAM;
          break;
          }
        case PERIOD:
          {
          activeProperty.setGroup( current );
          current = "";
          state = STATE_NAME;
          break;
          }
        default:
          {
          current = current + byteToString( object[ i ] );
          break;
          }

      }

      break;
   }
  case STATE_PARAM:
   {
```

```
          switch( object[ i ] ) {

            case SEMICOLON:
              {
              activeParameter.setValue( current );
              activeProperty.addParam( activeParameter );
              activeParameter = new VersitPropertyParameter();

              // Check to see if the line is QUOTED-PRINTABLE.
              // If so, set the boolean quotedPrintable to
              // indicate that we are in that state.

              if ( ( quotedPrintable == false ) &&
(current.toUpperCase().equals("QUOTED-PRINTABLE")) )
                quotedPrintable = true;

              current = "";
              break;
              }
            case COLON:
              {
              activeParameter.setValue( current );
              activeProperty.addParam( activeParameter );
              activeParameter = new VersitPropertyParameter();

              // Check to see if the line is QUOTED-PRINTABLE.
              // If so, set the boolean quotedPrintable to
              // indicate that we are in that state.

              if ( ( quotedPrintable == false ) &&
(current.toUpperCase().equals("QUOTED-PRINTABLE")) ) {
                quotedPrintable = true;

              current = "";
              state = STATE_VALUE;

              break;
              }
            case EQUALS:
              {
              activeParameter.setName( current );
              current = "";
              break;
              }
            default:
              {
              current = current + byteToString( object[ i ] );
              break;
              }

          }
```

```
          break;
        }
      case STATE_VALUE:
        {
          switch( object[ i ] ) {

            case EQUALS:
              {
                if ( quotedPrintable == true ) {
                  state = STATE_EQUALS;
                  break;
                } else {
                  current = current + byteToString( object[ i ] );
                }
              break;
              }
            case CR:
              {
              state = STATE_CR;
              break;
              }
            case SEMICOLON:
              {
              activeProperty.addValue( current );

              current = "";
              break;
              }
            default:
              {
              current = current + byteToString( object[ i ] );
              break;
              }

          }
          break;
        }
      case STATE_CR:
        {
          switch( object[ i ] ) {

            case LF:
              {
              state = STATE_LF;
              break;
              }
            default:
              {
              current = current + byteToString( object[ i ] );
              state = STATE_VALUE;
              break;
              }
```

```
        }
        break;
      }
    case STATE_LF:
      {
        switch( object[ i ] ) {

          case SPACE:
            {
            current = current + byteToString( object[ i ] );
            state = STATE_VALUE;
            break;
            }
          default:
            {
            activeProperty.addValue( current );
            arrayProperties.add( activeProperty );
            activeProperty = new VersitProperty();

            current = "";
            current = current + byteToString( object[ i ] );
            quotedPrintable = false;
            state = STATE_NAME;
            break;
            }

        }
        break;
      }
    case STATE_EQUALS:
      {
        switch( object[ i ] ) {

          case CR:
            {
            state = STATE_EQUALSCR;
            break;
            }
          default:
            {
            current = current + "=" + byteToString( object[i] );
            state = STATE_VALUE;
            break;
            }

        }
        break;
      }
    case STATE_EQUALSCR:
      {
        switch( object[ i ] ) {
```

```
            case LF:
              {
              state = STATE_VALUE;
              break;
              }
            default:
              {
              current = current + "=" + byteToString( object[i] );
              state = STATE_VALUE;
              break;
              }

          }
          break;
        }
      default:
        {
        System.out.println( "Parser Error: Unknown State." );
        }

    }

  }

  // Cast the properties to their specific types.
  VersitProperty[] properties
    = new VersitProperty[ arrayProperties.size() ];

  for( int i = 0; i < arrayProperties.size(); i++ ) {
    properties[ i ] = getSpecificProperty( ( VersitProperty
)arrayProperties.get( i ) );
  }

  return properties;

}
```

Extended Multiple Reader Single Writer

```
/*
 * ExtendedMRSW.java
 *
 * Copyright 2004, Jason Barrie Morley
 *
 */

// Package.
package uk.co.in7.insync.database;

// Imports.
```

```
import java.util.Hashtable;

// Class.
class ExtendedMRSW {

  // Variables.
  Hashtable hashReaders = new Hashtable();
  Hashtable hashWriters = new Hashtable();
  Hashtable hashWaitingWriters = new Hashtable();

  // Constructors.

  // Accessors.

  // Locking.

  synchronized public void enterReader( Object keyObject )
    throws InterruptedException {

    System.out.println( keyObject.toString() + ": Requesting Read
Lock" );

    while ( ( getWriters( keyObject ) > 0 )
            || ( getWaitingWriters( keyObject ) > 0 ) )
      wait();

      incReaders( keyObject );

    System.out.println( keyObject.toString() + ": Read Lock
Acquired" );

  }

  synchronized public void enterWriter( Object keyObject )
    throws InterruptedException {

    incWaitingWriters( keyObject );

    System.out.println( keyObject.toString() + ": Requesting
Write Lock" );

    while ( ( getWriters( keyObject ) > 0 )
            || ( getReaders( keyObject ) > 0 ) )
      wait();

    decWaitingWriters( keyObject );

    incWriters( keyObject );

    System.out.println( keyObject.toString() + ": Write Lock
Acquired" );
```

```
  }

  synchronized public void exitReader( Object keyObject ) {

    decReaders( keyObject );
    notifyAll();

  }

  synchronized public void exitWriter( Object keyObject ) {

    decWriters( keyObject );
    notifyAll();

  }

  // Internal Methods.

  // Writers.

  synchronized protected int getWriters( Object keyObject ) {

    Integer integerWriters
      = ( Integer )hashWriters.get( keyObject );
    int numWriters = 0;

    if ( integerWriters != null ) {
      numWriters = integerWriters.intValue();
    }

    return numWriters;

  }

  synchronized protected void incWriters( Object keyObject ) {

    Integer integerWriters
      = ( Integer )hashWriters.get( keyObject );
    int numWriters = 0;

    if ( integerWriters != null ) {
      numWriters = integerWriters.intValue();
    }

    numWriters++;

    hashWriters.put( keyObject, new Integer( numWriters ) );

  }

  synchronized protected void decWriters( Object keyObject ) {
```

```
    Integer integerWriters
      = ( Integer )hashWriters.get( keyObject );
    int numWriters = 0;

    if ( integerWriters != null ) {
      numWriters = integerWriters.intValue();
    }

    numWriters--;

    if ( numWriters == 0 ) {
      hashWriters.remove( keyObject );
    } else {
      hashWriters.put( keyObject, new Integer( numWriters ) );
    }

  }

  // Readers.

  synchronized protected int getReaders( Object keyObject ) {

    Integer integerReaders
      = ( Integer )hashReaders.get( keyObject );
    int numReaders = 0;

    if ( integerReaders != null ) {
      numReaders = integerReaders.intValue();
    }

    return numReaders;

  }

  synchronized protected void incReaders( Object keyObject ) {

    Integer integerReaders
      = ( Integer )hashReaders.get( keyObject );
    int numReaders = 0;

    if ( integerReaders != null ) {
      numReaders = integerReaders.intValue();
    }

    numReaders++;

    hashReaders.put( keyObject, new Integer( numReaders ) );

  }

  synchronized protected void decReaders( Object keyObject ) {
```

```
    Integer integerReaders
      = ( Integer )hashReaders.get( keyObject );
    int numReaders = 0;

    if ( integerReaders != null ) {
      numReaders = integerReaders.intValue();
    }

    numReaders--;

    if ( numReaders == 0 ) {
      hashReaders.remove( keyObject );
    } else {
      hashReaders.put( keyObject, new Integer( numReaders ) );
    }

  }

  // Waiting Writers.

  synchronized protected int getWaitingWriters( Object keyObject
) {

    Integer integerWaitingWriters
      = ( Integer )hashWaitingWriters.get( keyObject );
    int waitingWriters = 0;

    if ( integerWaitingWriters != null ) {
      waitingWriters = integerWaitingWriters.intValue();
    }

    return waitingWriters;

  }

  synchronized protected void incWaitingWriters( Object keyObject
) {

    Integer integerWaitingWriters
      = ( Integer )hashWaitingWriters.get( keyObject );
    int waitingWriters = 0;

    if ( integerWaitingWriters != null ) {
      waitingWriters = integerWaitingWriters.intValue();
    }

    waitingWriters++;

    hashWaitingWriters.put( keyObject,
                            new Integer( waitingWriters ) );

  }
```

```
  synchronized protected void decWaitingWriters( Object keyObject
) {

    Integer integerWaitingWriters
      = ( Integer )hashWaitingWriters.get( keyObject );
    int waitingWriters = 0;

    if ( integerWaitingWriters != null ) {
      waitingWriters = integerWaitingWriters.intValue();
    }

    waitingWriters--;

    if ( waitingWriters == 0 ) {
      hashWaitingWriters.remove( keyObject );
    } else {
      hashWaitingWriters.put( keyObject,
                              new Integer( waitingWriters ) );
    }

  }

}
```

Appendix B User Interface Screenshots

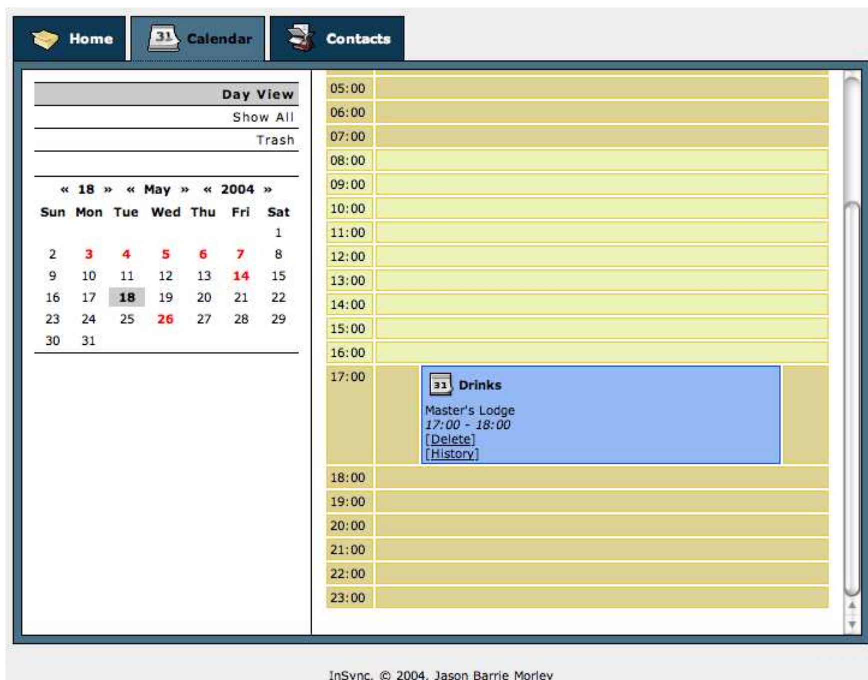

Figure 24 The Calendar Tab showing an appointment

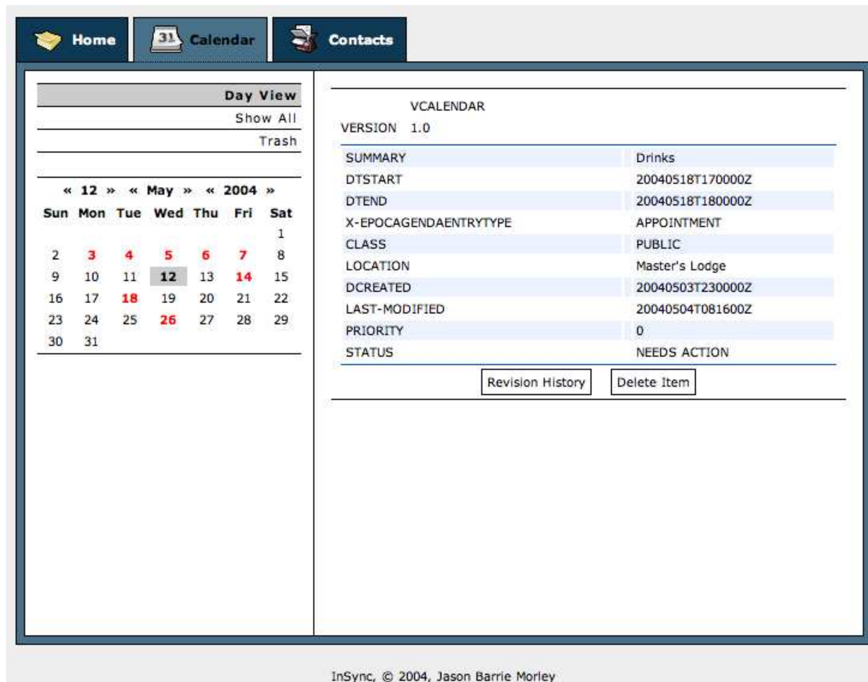

Figure 25 The vCalendar representation of the same appointment

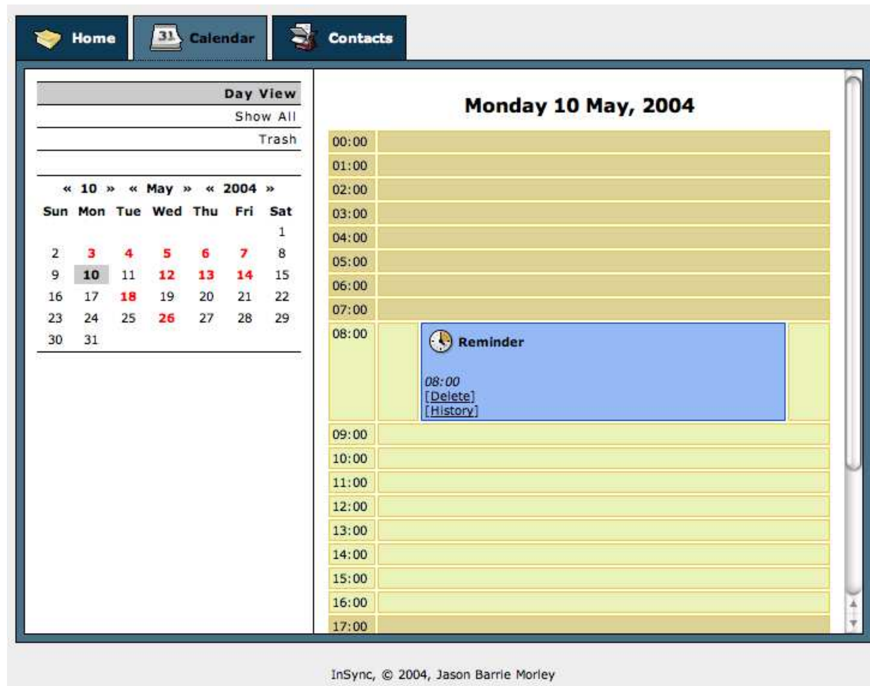

Figure 26 The Calendar Tab showing a reminder

Appendix C Project Proposal

Computer Science Tripos Part II Project Proposal

Calendar and Contacts

Synchronisation Server

J. B. Morley
Clare College
jbm37@cam.ac.uk

24th October 2003

Project Originator: *J. B. Morley*

Signature:

Project Supervisor: *Dr G. Titmus*

Signature:

Director of Studies: *Dr G. Titmus*

Signature:

Project Overseers: Dr *P. Robinson & Dr A. Dawar*

Signatures:

Special Resources Required: *See attached Resource Request Form*

Contents

Contents 64
Introduction 65
Work to be undertaken 65
Assessment Criterion 65
Resources Required 66
Backup Arrangements 66
Background Knowledge 66
Starting Point 66
Work Schedule 67
Michaelmas Term 67
24th October – 7th November 67
7th November – 21st November 67
21st November – 5th December 67
Christmas Break 67
5th December – 16th January 67
Lent Term 67
16th January – 30th January 67
30th January – 13th February 68
13th February – 27th February 68
27th February – 12th March 68
Easter Break 68
12th March – 23rd April 68
Easter Term 68
23rd April – 14th May 68

Introduction

The movement of the consumer market towards mobile computing solutions such as Laptops, PDAs and, increasingly, Smart Phones has brought with it new problems. Many of these devices focus on providing PIM (Personal Information Management) solutions. This introduces a need for a method of maintaining some level of concurrency between devices. Most are sold with proprietary software that allows limited two-way 'synchronisation' with a small number of PIM applications for Microsoft Windows. However, should a user wish to use more than one such device, they require multiple pieces of software to be installed and often encounter problems where data merges are performed incorrectly.

This project aims to implement an extensible synchronisation calendar and contacts server. This should provide support for merging multiple entries from a range of different devices and platforms.

Work to be undertaken

The project divides naturally into a number of different sections:

1. Investigation of the current standards used for transferring contacts between devices and systems. This includes standards like vCal, vCard, OBEX and SyncML. It will also be necessary to refresh my knowledge of Databases and XML.
2. A greater knowledge of these should allow a decision to be made on the internal architecture of the server and how to perform mappings of these various formats to an internal representation. This architecture should provide good support for communicating with networked clients.
3. Implementation of the central server as decided upon in Section 2. The server should provide support for multi-way merges of entries where not all fields are shared.
4. Assuming success of the server implementation, a number of these network transparent client applications could be implemented to allow direct synchronisation of both mobile devices and desktop applications.
5. *A possible extension would be to author a web-based interface, allowing users to view the contents of the server directly. Further, support could be added for group calendars and shared entries.*

Assessment Criterion

It is necessary to have some criterion that I may use to assess the success of the final project. This should be when it is possible to create a merged diary from two different sources; at this point, the initial concept has been tested and proved.

Resources Required

Development will be performed on one of my two systems (an Apple PowerBook G4 and an AMD XP2000+ PC).

The project requires that a test be run with one or more mobile device. The devices listed within this section represent those that I own. Only a couple of these should actually be required. Therefore, should one cease to work, there will be no problem of lack of resources:

- Psion Series 3c
- Psion Series 5
- Sony Ericsson P800
- Compaq iPaq H3660
- Sony Ericsson T68

Additionally, for interfacing with Microsoft Outlook, the Office Development Kit will be required. This should be available through the Microsoft Academic Alliance.

Backup Arrangements

As described above, work will be carried out on my personal computers. Regular backups will be performed to the PWF, Pelican and CD-RW. Should my personal development machine fail, the backups to the PWF should allow me to continue development instantly using the PWF machines. Additionally, CVS will be used with a policy of daily check-ins, to ensure all development is tracked and retrievable.

Background Knowledge

I have no real practical knowledge of databases apart from that gained in the Part IB Databases course. Clearly time needs to be dedicated to this, though hopefully much of the knowledge required can be picked up during development.

Starting Point

It would seem logical to make use of both XML and MySQL libraries and bindings should these be required. Additionally, for connection with PDAs, it would be seem sensible to use off the shelf solutions, since these do not really demonstrate the project area I am interested in. Support for Bluetooth and IRDA standards for devices would be provided by 3rd party software. Any such software would clearly be declared.

To my knowledge, no other material exists that I can make use of.

Work Schedule

Michaelmas Term

24th October – 7th November

- Investigating details of Calendaring Software.
- Preparation of development environment.
- Instigating backup system.

Deliverables: Working Development Environment and Backup System.

7th November – 21st November

- Designing server architecture.
- Experimenting with, and configuring the database for use with the server.

Deliverable: Document outlining the final architecture of the Server.

21st November – 5th December

- Beginning development of the central server – this two week period should be long enough to handle issues arising from actual development.

Deliverable: Basic prototype of the Server.

Christmas Break

5th December – 16th January

- Completion of the coding of the server.

Deliverable: Completed Server.

Lent Term

16th January – 30th January

- Coding a simple client to access the server and the handling of any problems this might reveal. This simple client should allow users to merge a standard file format, such as vCard into the server.
- Authoring the Progress Report.

Deliverable: Project Progress Report

30th January – 13th February

- Completion of the above, or implementing a second synchronisation source. This second source should run natively on a mobile device, possibly being authored in Java.
- Testing.

Deliverable: Second Synchronisation Source.

13th February – 27th February

- Continuation of the previous work packet.
- Testing and bug fixing.

27th February – 12th March

- Evaluating system.
- Draft write-up of project.

Deliverable: Draft Dissertation.

Easter Break

12th March – 23rd April

While most of the Easter break should be used for revision, this period may be used tie up any loose ends should it be necessary.

Easter Term

23rd April – 14th May

- Completing and handing in Dissertation

Deliverable: Completed Part II Project.