FRAG micro edition

FRAGme 2004

INFO401/2004

Mobile Wireless Systems for a Micro-size Peer-to-Peer Environment

Project Guide

INFO401 – SOFTWARE ENGINEERING 2004

Project Documentation

FRAGme2004 Documentation
Version 2.0

The info401 software engineering team 2003 were:

Lars Ehrler, Noel Garside, Benjamin Herrmann, Jonathan Lingard, Rodney Tamblyn

- Original Framework Development -

The info401 software engineering team 2004 were:

Nick Elder, Hamed Hawwari, Lincoln Johnston, Bing Leng, Duncan Meyer, Mengqiu Wang, Heiko Wolf, Hui Zhang

- Framework Extension and Game Development -

Table of Contents

Chapter

1

Project Overview

This document describes the FRAGme project, which implements a framework for distributed peer to peer gaming. It was first developed as a 2nd term group project for the INFO401 Software Engineering course in 2003 at the University of Otago and continued in the INFO401 course 2004.

This document includes four types of documentation:

- Description of 2003's FRAGme project and issues that arose for the 2004 development,
- Description of the Software Engineering concepts used in the course of this project,
- Details of how to install the FRAGme on a Java platform, especially regarding installation for the Sharp Zaurus SL-C700 and how to use JGroups (#1) on the Zaurus,
- Interface documentation for application programmers,
- System documentation for framework developers.

What is FRAGme?

FragME2004 is a framework for wireless application development that takes care of connection and peer management as well as efficient object creation. The name is inherited from the FRAG project of the TU Munich. The original meaning of the abbreviation "FRAG" could not be traced back (there is no remark on that in the 2003 documentation as well), but "me" characterises the "micro edition", which means it is aimed at small, wireless devices. As we grew so used to the name FRAGme, we kept it for FRAGme2004 where it now stands for:

Framework for Robust Autonomous Gaming micro edition

Description of FRAGme 2003 structure

The FRAGME framework was developed directly as a result of the client requirement to deliver an object distribution framework for small PDA-like devices. The original project direction

involved a plan to convert the existing framework FRAG from the TUM Arena project onto the Sharp Zaurus platform.

However, it was found that it was not possible to make the conversion because of a number of structural issues with the way that Arena was coded. One of the issues appear to be with the excessive amount of memory required by the application, built on top of FRAG, called "Sword". It was also agreed that there were also problems with thread management. For example, the Sword program had problems running on PowerMacs, so the inability of the product to run on a PDA is not surprising.

This, together with the necessity for the Munich system to support the independent developments in Germany, meant that it was impossible to continue with the Arena architecture.

The FRAGME logo reflects the heritage from the original ARENA project, as it was a goal of the FRAGME project to complete some of the ARENA targets. The FRAGME logo also shows the Roman Arena, but this time from the outside, as it was a key goal of the project to build a modular architecture separate from any particular game. It should be noted that one of the problems encountered in the ARENA framework is that the separation between the coding for the Framework and the "Sword" application became very blurred.

In second half of 2003 members from the INFO401 group from Otago University began to develop the FRAGME framework. This group consisted of Lars Ehrler, Noel Garside, Benjamin Herrmann, Jonathan Lingard and Rodney Tamblyn. At the conclusion of 2003 the team had produced a framework that could be used to provide a back end to a distributed application, but lacked in adequate functionality and performance. It was the job of the 2004 INFO401 class, consisting of Nick Elder, Hamed Hawwari, Lincoln Johnston, Bing Leng, Duncan Meyer, Mengqiu Wang, Heiko Wolf and Hui Zhang, to take this existing framework and refactor it so that it could be framework could be considered finished.

Problems of the old framework / proposed extensions

The goal of the 2003 group was to develop a framework that could be easily used by an application programmer to develop a distributed application. During the last days of their final semester they achieved this goal by producing a framework that fulfilled those requirements. The rushed delivery meant that while the functionality was delivered, it did suffer from bad performance due to a lack of testing. Other secondary functionality was alluded in interfaces but was never actually implemented in code. The end result was a buggy framework that needed to be patched, and in some cases completely rewritten.

The major problem in the 2003 Framework was the performance of sending objects over the wireless network. Tests revealed that Java's RMI implementation was the problem. This issue would be addressed by evaluating other approach (namely JGroups, see further on in documentation).

The interface to connect the 2003 framework to the application is not coherent enough and this needs to be changed. Ideally the application programmer would connect to the framework with little trouble, e.g. calling on one or two methods and the connection would be set up. The 2003

framework makes hard work of a simple process. During the first attempts to change it also became obvious that the partition between the parts of the systems was not as clear and consequent as described in the documentation. This issue should also be addressed in the redesign process.

The 2003 framework did not handle peers dropping. When a peer left the network all the objects associated with this peer is lost. In the case of a badly written application, it would cause it to crash. Peer dropout is now handled.

In general the 2003 framework is workable, but needs to be overhauled to make it more user friendly and increase its performance. This is the job of the 2004 class.

Chapter

2

Software Engineering methods

Working in groups was a real eye opener. Things that were not as apparent before came to light, things which only surface under real, practical team development conditions. From the outset of forming the groups through to how to break the tasks down and allocate them to the most suitable candidates based on their skill. As the project progressed certain 'truths' emerged:

- *it is pretty hard breaking tasks up (what best for what person according to skill)*
- *people emerge as leaders, some emerge as technical minded workers*
- *deadlines are fleeting*

After analysing last year's framework, the weak points were known and the actual project work could begin. The following chapter discusses the used Software Engineering approaches, the project management tasks and the tools used during development. This project clearly thought us lessons about how to tackle a project, and that project management and control instruments such as project plans and group work directories are absolutely essential to achieve a successful project end.

Project proposal

The requirements were directly derived from the analysis of last year's project.

- Performance of sending objects
  - Check the possibility of state change notification through multicasting instead of RMI -- is it faster?
  - Evaluate communication packages such as JGroups and Jaxter, whether they can be of use for FRAGme
  - Enhance the performance of FRAGme

- Peers dropout
    - Implement peer dropout handling
- Redesign the interface
    - Implement a clear interface for applications to plug into the framework (should be one class that includes all methods which have to be accessed by applications programmers)
- Check concepts proposed by last year's documentation (generic boundary, transfer of object ownership)
    - Are they important for the scope of our framework? If yes, how can they be implemented?
- General debugging tasks
    - Check the memory management and assure that objects are freed correctly
    - Check the object reuse
        - Are objects reused efficiently?
        - If not, activate object reuse functions
    - Reformat code and redefine data declaration scopes, following coding guidelines

The 2003 FRAGme team developed several applications (Button, Asteroid, Pong) as tests for their implementation. We decided to take Pong as a test case for our framework enhancements. On the old framework, it ran very slow, so it was almost unplayable. If we could speed up Pong reasonably, the weak performance of sending objects would be resolved. To test the object reuse and memory management, a second game was needed which would create a lot of objects. A SpaceBattle game would be ideal for this task because of many flying bullets. The development of a new game would also make use of the new interface and further test our network. Finally, formalized testing was also undertaken to ensure the correct function of the framework (see section "Testing"). Use cases were created to plan the future functionality of the framework. They can be found in Appendix B.

Project Management

The team was divided into two groups, of which one group was mainly concerned with the framework development (Hamed, Heiko, Lincoln and Mengqiu, in the following called "FrameworkTeam") and the other (Bing, Duncan, Hui and Nick, in the following called "ApplicationTeam"), with the development of SpaceBattle. In both teams two people were responsible for project management tasks and distribution of work, Heiko and Mengqiu for the FrameworkTeam and Duncan and Nick for the ApplicationTeam. They were also in charge for coordination of the two teams.

The main part of development was done over four weeks at the end of semester one 2004, while the time before was mainly spent on analysis and tests of the old framework. The milestones set were all reached in time.

Week 1 - 10/5/2004 (start date):
Application Team Deliverables: Gamemaker prototype that illustrates what the week 4 prototype might look like. Agree on the type of game that we want to implement in the final prototype.
Framework Team Deliverables: Proposal of the extension that will be made to the framework. This includes all bugs and deficiencies that will be remedied.

Week 2:
Application Team Deliverables: Complete a basic application that uses the framework. This application will be capable of sending message across the network to other peers. It shall be able to be run on the Zaurus. This purpose of this exercise is to make sure that we know how to use the framework from an application programmer's point of view.
Framework Team Deliverables: Begin to fix errors and implementing proposed extensions. The first aspect to take care of is the object management, especially in regards to reusing objects.

Week 3:
Application Team Deliverables: Begin the final prototype development.
Framework Team Deliverables: Continue implementing extension. Test the extensions using existing games such as Pong, and try to make it run smoothly, i.e. playable.

Week 4 - 4/6/2004 (end date):
Application Team Deliverables: Working prototype that uses the extended version of the FRAGme framework.
Framework Team Deliverables: All proposed extensions and bugs finished. Have Pong running smoothly using the new framework.

Project plan (MS project)

A project resource and time plan was set up using MS Project. The time and Gantt view can be seen in Appendix A.

Development process

For the most part the INFO401 development methodology was pretty similar to that of the Extreme Programming (XP) methodology. This involved small groups who (in most instances) practiced peer programming. With four people in each team, the XP methodology could be applied pretty well. There was also a distribution of tasks, which means two main developers, and two people doing the background research, documentation and support activities. In the FrameworkTeam, the support guys researched JGroups and tested the RMI / Multicasting performance on the Zaurus, while in the ApplicationTeam the support people checked out

Gamemaker for interesting game ideas and came up with a game design that was implemented by the main developers afterwards.

In XP the situation is supposed to one of high client contact in which requirements can be elicited and changed on a frequent 'when necessary' basis. For our situation, we had a client (our lecturing/teaching fellow team) which was relatively easy to communicate with this being due to their work residences' being in the same building as where the development was taking place.

A main issue within the FRAGme project has been the communication between the team members and the staff. We used four main means of communication to keep up-to-date:

1. **Wiki:** The major form of communication throughout the INFO401 community has been the use of the WIKI online bulletin board which made transmitting large volumes of information between the group (lecturers included) extremely easy. This was supported by emails and the Otago blackboard.

2. **Weekly meetings** were held with our client to check progress and attain direction for where the project was to lead with their minutes being made available on the WIKI. More formally than the project meetings, the development was explained and advice given from our lecturers.

3. **Project meetings** in lab 3.01 were used to schedule tasks and talk about the coding development in the last days. Programming concepts are best understood when you can show them actually in the development environment, so these meetings were most valuable. This idea of stand-up meetings was borrowed from the SCRUM (#2) methodology. A feature of the individual team meetings (of which formal ones were also held on a weekly basis) was their informal nature and frequent occurrence. Group-wide meetings were also used to coordinate the two teams.

4. To exchange documents and program code that was not available in the CVS, a **GroupWork directory** was made available and strongly used. This was also necessary as the standard student user account is extremely limited in terms of memory.

Development tools

Several development tools were used to plan the development; create the actual code and keep the different versions concurrent between team members.

Borland JBuilder X

After evaluation of Eclipse and other alternatives this IDE seemed well tried and tested and as though it would be a stable workhorse for us to use. Special features we liked were the CVS support, the automatic indent (just press tab…) and the history and diff feature. It also seemed very stable, while Eclipse crashed a lot in the evaluation period.

CVS (JBuilder CVS plug-in)

Concurrent version control was achieved through the use of a JBuilder plug-in that talks to the UNIX version of CVS. We also evaluated WinCVS, but found it much more convenient to use

CVS from inside our IDE. The version control enabled multiple people to work on the same project without having version conflicts. It is highly recommended for team development efforts.

MS Project XP

MS Project XP allowed us to schedule tasks easily and make the plan available on the Wiki in various diagrams (such as the Gantt diagram in Appendix A). We think is it going to even more useful if you manage bigger projects with more time and resource dependencies than FRAGme. But anyway, it was a good lesson how to use it in later projects.

Borland Together 6.1

For the creation of Class Diagrams, use cases and scenarios Borland Together 6.1 was used. It was an invaluable tool to plan our development process.

Chapter

3

Installing and using FRAGme on the Zaurus

Instructions for installing FRAGme and configuring the Zaurus environment

FRAGME is designed to run on a java platform and requires installation of a suitable virtual machine for operation. It also depends on a special, "stripped down" version of JGroups. In addition, peer discovery operates by way of a subnet which requires configuration on each Zaurus.

Zaurus Java VM Configuration

FRAGme is designed for the J2ME Personal Profile edition, which runs on Zaurus. Further information on J2ME can be found under java.sun.com and in the 2003 FRAGme documentation.

JGroups – the new peer manager protocol

In the 2003 FRAGme framework, the peer manager was using RMI calls to distribute objects. The sending of objects was very slow on the Zaurus, and Pong even stopped for some time when changes were distributed. Only parts of seconds, but too long to enable fast gaming. We did some testing to find the reason for the slowdown of the games.

FRAGme testing

In a test case that was given as assignment 2 of semester 1/2004 we compared the performance of RMI calls and multicasting for object distribution. Tests showed that object distribution via RMI calls needed as much as ten times the time as sending an object via multicasting on Zaurus, while on PC both methods were so fast that no differences could be measured. We did not find out whether the RMI implementation for Zaurus was so bad, but we decided that it was worth to try switching to a multicasting based object distribution.

JGroups evaluation

JGroups is an open source project for reliable group communication based on multicasting. It is also very easy to use and offers all the functionality that we wanted for our peer manager. Some of the functions are described in the following paragraph.

- Group creation and deletion. Group members can be spread across LANs or WANs
- Joining and leaving of groups
- Membership detection and notification about joined/left/crashed members
- Detection and removal of crashed members
- Sending and receiving of member-to-group messages (point-to-multipoint)
- Sending and receiving of member-to-member messages (point-to-point)

Flexible protocol stack:

- Select protocols needed for the application - meet different requirements
- Comes with number of protocols
  - Transport protocols: UDP (IP Multicast), TCP, JMS
  - Fragmentation of large messages
  - Reliable unicast and multicast message transmission. Lost messages are retransmitted
  - Failure detection: crashed members are excluded from the membership
  - Ordering protocols: Atomic (all-or-none message delivery), FIFO, Causal, Total Order (sequencer or token based)
  - Membership
  - Encryption

Channel concept:

```
String props="UDP:PING:FD:STABLE:NAKACK:UNICAST:" +

"FRAG:FLUSH:GMS:VIEW_ENFORCER:STATE_TRANSFER:QUEUE";

Message send_msg;
Object  recv_msg;
Channel channel=new JChannel(props);

channel.connect("MyGroup");

send_msg=new Message(null, null, "Hello world");

channel.send(send_msg);

recv_msg=channel.receive(0);
System.out.println("Received " + recv_msg);

channel.disconnect();
```

```
channel.close();
```

The parameters mean:

- UDP - IP Multicast
- FD - failure detection
- STABLE - distributed message garbage collection
- NAKACK - multicast loss-less and FIFO delivery
- UNICAST - unicast loss-less and FIFO delivery
- FRAG - message fragmentation
- GMS, FLUSH, VIEW_ENFORCER, QUEUE - group membership
- STATE TRANSFER - state transfer (STATE_TRANSFER)

For more information on JGroups and the JavaDoc, see http://www.jgroups.org.

JGroups on the Zaurus

JGroups was originally designed to run on the full J2SE, which includes much more functionality than J2ME PP edition, and not surprising it was not running the first time we tried it. We also found a project called JGroupsME, which was aimed at porting JGroups to the Java kvm for very small devices. We did not want to use JGroupsME for the following reasons:

- It does not include all functionality of JGroups (for example the PullPushAdapter which we are using)
- Id does not seem as if someone is still working on it and we could also not find source code on the web site
- JGroupsME is aimed at devices smaller than the Zaurus

So that is what we did: we tried to "strip down" JGroups and cut out all the functionality not needed and get it running on the Zaurus. We needed to get rid of all XML functionality and some other depending functions, but after that it runs perfectly on the Zaurus. Now we can use the comfort and reliability of JGroups and can concentrate on the implementation of the object management and applications.

Testing environment – Zaurus

Copying files

To facilitate distribution of software packages from the development environment to Zaurus computers there are several options available. For download to a single Zaurus, you can connect it to a PC or Mac via USB cable. Then you can use Zaurus File Transfer program to copy files to

the Zaurus or to delete them from there. Sadly, it was not possible to copy from the Zaurus to the PC, but this is not used very often anyway. In lab 3.01, computer L301_6 has the needed drivers installed. This is also the only PC that has the Zaurus File Transfer utility (you can find it under:

```
C:\Program Files\Sharp Zaurus 2\ZaurusDrive\ZDrive.exe
```

Maybe a bit pressure on TSG would help to install them on all lab PCs… The FRAGme 2003 documentation also explains the use of Unison [#3]; a UNIX file synchronization tool.

Tip: we had quite a hard time to find WHERE the files were actually copied when we used Zaurus File Transfer, because it uses a different directory structure than Zaurus' UNIX distribution. The "FlashMemory" folder in the Zaurus File Transfer program equals /home/samba/MainMemory on the Zaurus.

If you have a memory card you may find it convenient to copy files directly from the desktop computer to the memory card on the Zaurus 1, then you can exchange the card from Z1 to Z2 etc. To copy all the files in directory "bob" on memory card to the home directory /home/zaurus (if you are not logged in as root) type: `cp /mnt/card/bob/* ~`

Enabling Multicasting on the Zaurus

FRAGme requires multicasting to be enabled in order for JGroups to work. You **must** undertake the following step **each time** you restart the Zaurus or change its network setup, or the FRAGme applications will not be able to create a JChannel and thus will not start.

```
su root
route add –net 224.0.0.0 netmask 240.0.0.0 dev eth0
exit
```

To view your IP route table to ensure that subnet mask is set (this is one of the first things to check if you are experiencing unexpected errors) type:

```
route –n
```

Writing a script that does the steps above for you speeds up the process.

Peer to Peer Zaurus Wireless setup info

To set up the wireless card (usually not needed, because they should be already installed):

1) In Settings->Network Setup choose Wireless LAN Card from the pop down menu, then click "edit" or "new" to edit/create a wireless setup

2) A tabulated "Wireless LAN Setting" window appears. Click Config tab. Deselect "Any ESS ID" and enter a name in the ESS ID field (eg "1234". Under Network Types select 802.11 Ad-Hoc"

3) Web and Web Auth should be disabled.

4) Under TCP/IP turn off "Obtain TC/IP Info Automatically", enter ip address unique for each Zaurus, eg 192.168.1.1 on Z1 and 192.1681.2 on Z2

5) Click OK to close Wireless LAN Setting setup screen.

To establish a TCP/IP wireless ad hoc network between two Zauri follow these steps:

1) In the "dock" at the bottom right of the screen, click on the wireless card connect on Z1 (do nothing at this point on Z2). Click connect. The Zaurus will attempt to connect to other devices. After a few moments it will give up and present a failure dialog. Leave the dialog (don't click OK) This Zaurus is now listening for devices.

2) On Z2, click connect. It should find the other Zaurus and establish the network.

3) Now go back to Z1 and click "connect" again. Now this Zaurus will join the network, too. You will notice the activity lights on both Zaurus wireless cards regularly "pulsing" indicating network is up.

Tip: we recommend you assign IP addresses of Zaurus according to last digits of their "IS" number, e.g. IS 2309 becomes 192.168.1.09. This way when you are working with 6 Zauri it's easy to see which IP address goes with which Zaurus.

Launching the FRAGme applications

Miscellaneous Zaurus tips

- When working in VI, "escape" key is the Zaurus Cancel key
- Many standard characters are accessed by clicking the purple function key and the appropriate letter combination.
- Putting Zaurus to sleep and waking it up again resets USB networking

Zaurus UNIX Tips

To execute commands from the history the following may be useful:

- To view history: `history | less`
- To execute line 244 from history: `history 244`
- To execute last command beginning "rou" from history: `!rou`
- To execute last command again: `!!`
- For a simple text editor on Zaurus, try "nano"
- Use up and down arrows to review command history

Chapter

4

Application programmers guide to FRAGme

How to design a FRAGme based application

The FRAGme framework has a clear interface to develop FRAGme applications. The methods required to implement an application are all included in the one **ControlCenter** class. Application developers therefore import the **ControlCenter** class and use it to interact with the framework without the need to worry about any underlying code in the framework.

The FRAGme ControlCenter

The ControlCenter has all the methods required to develop a FRAGme application. Before we can use any of the methods we must import the Class:

```
import org.globalse.fragme.ControlCenter;
import org.globalse.fragem.FMeObject;
```

Once this is done we need to setup the connection.

```
ControlCenter.setUpConnections("spacebattle");
```

The `setUpConnections` method contains all the setup code to get a FRAGme application running. It creates the network connection and notifies the others a new peer has joined. It also takes care of setting up the initial objects. The method takes one argument which is the group name of which to join. The example above is from the Space Battle application and therefore the group it joins to is the 'spacebattle'.

After the connection is setup, we need to receive all the FRAGme objects currently in the application. This is done by calling the `getAllObjects`() method.

```
objs = ControlCenter.getAllObjects();
```

This method then returns a vector of all the FRAGme objects in the game.

New FRAGme objects can not just be created by using the new keyword. Instead we must use the ControlCenter method `createNewObject`. This is because the FRAGME framework uses the Factory Pattern in order to have absolute control over instantiating of relevant (FMeObject, FMeSerialized) objects and to encapsulate object creation in one place. This then allows for sophisticated memory management, as the user cannot simply build instances of these objects, but must use the factory class.

The following is an example from the Space Battle application of how to create a new FRAGme object.

```
airlou = (ShipAirlou)
ControlCenter.createNewObject(ShipAirlou.class);
```

Since the factory class does more than simply return an instance of the desired object, a private constructor for FRAGme Objects should be used in order to prevent normal instantiation of these objects. Also the constructor must not have any parameters to be conforming to the implementation of the factory pattern. If an initialization with parameters is needed for this object, use a separate initialization method.

```
private ShipAirlou() {
}
```

To enable factory support we also need our own private FragMeFactory subclass for each FMeObject and FMeSerialized subclass:

```
private static class Factory
      extends FragMeFactory {
    protected FactoryObject create() {
      return new ShipAirlou();
    }
  }
```

This factory enables access to the private constructor. It enables as well, because it is subclass of FragMeFactory, access to all of the functionality of the Factory.

Next our own private factory has to be registered with the single FragMeFactory for this application (FragMeFactory implements the Singleton pattern).

This is done using a static constructor.

The first time, this class is referenced the class-loader of the Java virtual machine will load the class into the virtual machine. While loading the class, the static constructor is triggered:

```
static {
    FragMeFactory.addFactory(new Factory(), ShipAirlou.class);
  }
```

Methods to override in FMeObject subclasses

To create a FRAGme object it must extend the class FMeObject. Apart from the methods required for the Factory design pattern, the subclasses of FMeObject have to implement the following methods.

```
  public Class getSerializedObjectClassName() {
    return ShipAirlou_ser.class;
  }
```

The method getSerializedObjectClassName() is used by the FragMeFactory to build an FMeSerialized object ser from a normal FMeObject object. To do so the static constructor of ser has to be called before the FragMeFactory attempts to build the FMeSerialized object. Calling ShipAirlou_ser.class on the respective serialization class ensures just that.

The next two methods are responsible for serialising – and deserialising your data. You provide an instance of your serialisation-class in both cases as parameter, then you will have to cast it to your own serialisation-class (in this case ShipAirlou_ser) and then move all your data.

```
public FMeSerialised serialize(FMeSerialised shipAirlouSer) {
    try {
      ((ShipAirlou_ser)shipAirlouSer).setXSpeed(
         this.xSpeed);
      ((ShipAirlou_ser) shipAirlouSer).setYSpeed(
         this.ySpeed);
      ((ShipAirlou_ser) shipAirlouSer).setLocation(
         this.location);
      ((ShipAirlou_ser) shipAirlouSer).setDx(this.dx);
      ((ShipAirlou_ser) shipAirlouSer).setDY(this.dy);
      return shipAirlouSer;
    }
    catch (Exception ex) {
      return null;
    }
  };
```

The Framework calls this method, whenever this FMeObject needs to be synchronized (the provided FMeSerialized object.setObject contains the current state)

```
public void deserialize(FMeSerialised serObject) {
   this.setXSpeed( ( (ShipAirlou_ser)
      serObject).getXSpeed());
   this.setYSpeed( ( (ShipAirlou_ser)
      serObject).getYSpeed());
   this.setLocation( ( (ShipAirlou_ser)
      serObject).getLocation());
   this.setDx( ( (ShipAirlou_ser) serObject).getDx());
   this.setDy( ( (ShipAirlou_ser) serObject).getDy());
 }
```

Deserialize() is called whenever a serialized object containing the data of this FMeObject is needed. This represents the other end of the network communication.

Methods to override in FMeSerialized subclasses

Each FRAGme object must also have a serialized version of the object. This object is used to send changes of an object to all the other peers in the game. The class contains all the same fields, along with getters and setters for these fields, as the FRAGme object. The getters and setter can then be used in the `serialize`() and `deserialize`() methods of the corresponding FMeObject. The serialized class also extents FMeSerialized which therefore requires it to implement some methods, which we will cover in this section.

The FRAGME framework requires subclasses of FMeSerialized to be programmed in accordance to the Factory Design Pattern. Therefore the same methods/objects need to be implemented in this respect as for the FMeObject. These are listed below:

- private constructor without parameters
- private FRAGmeFactory subclass
- static constructor
- implement the abstract method `getFMeObjectClassName`()

For more details in implementing these is shown above in the last section.

How to change objects (super.change())

When an object is changed, the other peers need the change sent to them. This is done in the FRAGme framework by calling the super.change() method from within the object which has changed.

```
super.change();
```

When this method is called it serializes the current object using the serialized version and then sends it to the other peers where it is deserialize and the new values of the fields are coped over to the FRAGme object.

Chapter

5

System Documentation

Details of the structure of the FRAGme system

FRAGME in its current version is a complete framework for the communication and object management in distributed network applications. It offers the following main functionalities:

1. Connection of peers
2. Communication between peers
   - Sending of objects
   - Sending of Notifications
3. Object Management
   - Creation and deletion of objects
   - Keeping synchronization and consistency between objects
   - Memory management
4. Clear API for application programmers
   - Access to all FRAGme functions through a "ControlCenter"

This chapter will discuss the implementation of those functionalities in depth as well as explain which design patterns where used.

System overview

The system consists of three main components. These three components are represented in the Java-code in packages. The three components are

- **ObjectManagement:** This component manages all the distributed objects this peers owns and all the distributed objects this peer has a copy of. It is the central component responsible for distributions of changes. Also the "FMeObject", the base object type in this framework, is defined in the ObjectManagement component.

- **PeerManagement:** The communication layer is responsible for automatic discovery of peers and for the communication between these peers. It relies on JGroups for message transport and works as a communication adaptor between JGroups and the ObjectManagement component.

- **FRAGme Factory:** The factory is responsible for creating FMeObjects, which are the base object type in the framework. It also contains a memory management based on the reuse ob freed objects. The Factory design pattern is explained in the chapter *Design Pattern*.

The application programmer's interface is defined in the class **ControlCenter**, which will also be defined as a component as the framework. Finally, there are FRAGme specific exceptions, which are combined in the **Exceptions** package. The parts of the framework and their connections are shown in the following diagram.

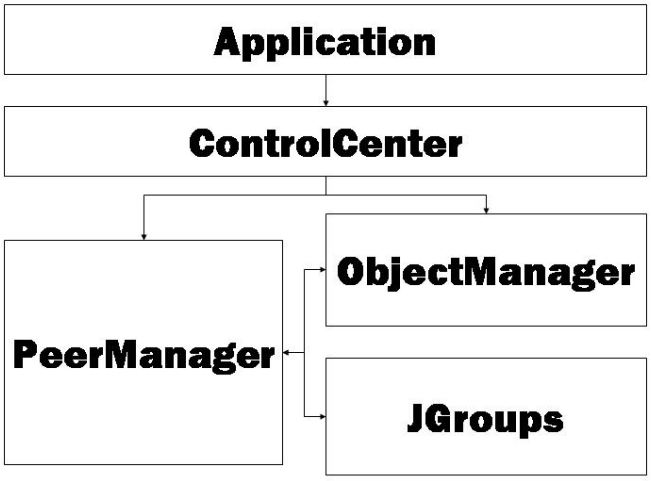


Figure 1 - Main Framework Components

Components

Object management

ObjectManagement is responsible for the management of all distributed objects in the system. Every object has to be registered with the ObjectManager. The ObjectManager uses the Singleton Pattern, which means every peer has only one instance of the ObjectManager. The ObjectManagement also provides the storage for a

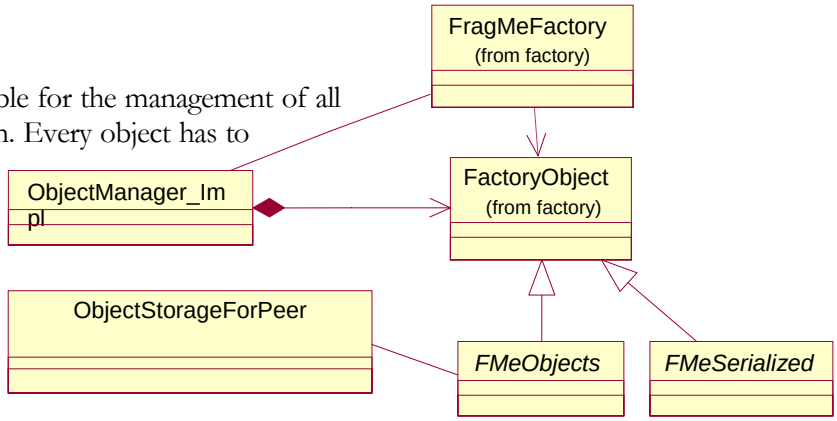


Figure 2 - Object manager class diagram

peer's own objects as well as for objects that were created by other peers (in class "ObjectStorageForPeer"). Important functions provided by the ObjectManager are described below.

```
/* return all stored objects to the application */
Public Vector GetAllObjects()
Public Vector GetAllObjects(Class className)
```

Returns all objects this peer stores in his ObjectStorageForPeer to the application. This method is called via the ControlCenter. If a class is given as input parameter, only objects of that type are returned.

```
public void pushChange(FMeObject object)
```

This method is called by an object's change()-method to register this change with the ObjectManager and to distribute it across the network. First the peer checks if the changed object is one of it's own objects, if so it distributes the change by using the PeerManager's send()-method and passing the object, the "MODIFY" performative and "null" as the receiver address (which signalizes "Broadcasting"). If the object is owned by another peer, it is send to this peer via unicasting (the pm.send()-method with the object owner's address as receiver).

```
public void sendObjectsToNewPeer(Address addr)
```

This method is called by the PeerManager to ask the ObjectManager to send its objects to a newly joined peer. The method iterates through all owned objects and sends them using the PeerManager's send()-method. After sending all objects, the new peer is notified by a message consisting of the "NOTIFY" performative, the "OBJECTS_SEND_TO_NEW_PEER" message and the new peer's address.

```
public void receiveChange(FMeObject object)
```

Whenever the PeerManager receives a new or changed object, it passes the object to the ObjectManager using this method. The ObjectManager checks if the object is its own, if so the object is passed to its own storage. The ObjectStorageForPeer checks whether the object already exists on this peer and adds it if necessary, otherwise the object is updated. It also distributes the change to the other peers (`send("MODIFY", object, null)`). An object from another peer is simply added to storage, using `addObjectsOfOtherPeer()`.

```
/* methods to manage peer storage */
public void allocateSpaceForPeer(Address addr)
public void addCreatedObject(FMeObject object)
```

```
private void addObjectOfOtherPeer(FMeObject object)
public void deleteObject(Address ownerAddr, int id)

public void deletePeer(Address addr)
```

All methods above are used to organize objects using the class ObjectStorageForPeer. The method allocateSpaceForPeer() creates a new object storage for a newly joined peer. Objects are added by addCreatedObject() for own and addObjectsOfOtherPeer for other objects. Single Objects are deleted by deleteObject(), while deletePeer() erases the whole storage for that peer (used in case of peer dropout).

FMeObject and FMeSerialized are abstract classes, which have to be inherited and implemented by the application developer. They represent the distributed data of the application. It is important to override the serialize() and deserialize() methods according to the objects used in the application. Also the methods getSerializedObjectClassName() and getFMeObjectClassName() must me overridden because they specify the type of object sent and stored.

Peer management

PeerManager takes care of all the lower-level transportation and peer management. It uses JGroups to manage the peers as a group, and uses JChannel to send and receive messages. So basically it has JGroups as its core, and wraps around it with an interface to ObjectManager.

At the starting time of each peer in the same application, they register themselves as a member of a particular JGroup, by specifying a group name.

Then each peer creates a JChannel and connects it to the group identified by the group name. This is done via a method call like this:

```
channel = new JChannel(props);
channel.connect(groupName);
```

Than we apply a "PullPushAdapter?" to the channel. That is needed because JGroups only offers pushing of messages and the user has to implement its own receiving-thread. The PullPushAdapter? takes care of this and calls a receive() method whenever a multicasting message arrives.

```
adaptor = new PullPushAdapter(channel, this, this);
```

As a new peer in the group, it will be notified by the JGroups manager of the other peers in the group. In order to receive the notification, PeerManager has to implement the MemberShipListener interface, and provide an implementation of the viewAccepted() method. This method will be called back by the JGroups manager.

In our implementation, we first count the number of currently existing peers, then we identify the peers that are newly joined or have already left by comparing with our previous record of existing peers. From a new peer's perspective, all the other peers are as new peers, because he has not known them before.

Then for each newly joined peer, we make an instance of CheckThread and run it. A CheckThread is mainly used for synchronization purpose. We'll explain in more details later.

When a new peer joins, existing peers must send their objects to this new peer so they share the common game objects. But the problem is that if existing peers send their objects before the new peer has allocated spaces for them, then the objects will not be stored properly by the new peer. So we want the existing peers to send their object only after they make sure that the new peer has allocated spaces for them. That means existing peer has to block and waiting for a notification from the new peer saying that space has been allocated for him. Now back to the new peer's side, a new peer finds one existing peer, send a "space allocated" notification to it, and waiting for him to send the objects. But the problem is that the new peer will never be able to receive those objects from the existing peer in such a simple implementation. The reason is as following:

Recall that we apply a PullPushAdaptor to a JChannel, which runs in a separate thread and takes the object lock of the JChannel when it receives messages. In the preceding scenario a new peer is waiting for objects to arrive from existing peers, this waiting method (whatever it is) takes up the object lock, so PullPushAdaptor cannot receive the objects. This is a dead-lock situation. The solution to tackle this dilemma is to implement the communication between each of the existing peers in a separate thread, as we provided.

Once all the objects have been exchanged, ControlCenter will return the handle to the application, and the setup process is completed.

Once the JChannel has been set up, peer can send and receive objects via PeerManager. ObjectManager calls PeerManager's send method to send all object related messages. This includes modifying, deleting and creating an object. E.g.,

```
PeerManager pm = ControlCenter.getPeerManager();
pm.send(ControlCenter.MODIFY,object,null);
```

When PeerManager receives a message from another peer's PeerManager, it passes the object to ObjectManager by calling ObjectManager's receiveChange() or deleteObject() respectively. E.g.,

```
public void receive(FMeObject object) {
    ControlCenter.getObjectManager().receiveChange(object);
}
```

```
ControlCenter.getObjectManager().deleteObject(senderAddr,((Int
eger)content).intValue());
```

The messages that ObjectManager uses to communicate with PeerManager has a standard format. A message is formed by three parts: Performative, Content, Destination. This implementation is inspired by Speech Act Theory. PeerManager examines the performative part of the message to determine what actions to take. Currently there are three types of Performative for message: Modify, Notify and Delete.

The content of the message should be either a String or an FMeObject, in event of sending notification or updating objects, respectively. If the content of the message is an FMeObject, PeerManager then requests FragMeFactory to serialize the object. If it's a String, PeerManager leaves it intact. After performing the last step, PeerManager constructs a JGroups Message, and sends it to the JChannel, which will in turn multicast it to all members in the group except the sender himself.

When a peer leaves the application, PeerManager calls ObjectManager's deletePeer() method to handle the case. E.g.,

```
ControlCenter.getObjectManager().deletePeer(addr);
```

FRAGme Factory

FRAGme Factory is the implementation of the Factory Design Pattern.

Each FMeObject class has a Factory, which inherits FragMeFactory class. In each FMeObject class there is also a static block which creates an instance of its own Factory, and all it to the collection in FragMeFactory, which contains factories for building different types of objects.

The basic idea is that the behavior of creating an FMeObject is tightly controlled by the Factory. Using a Factory to construct an FMeObject is the only way of making instances of FMeObject in FRAGme System.

ControlCenter

ControlCenter is the place where different components of the system are interconnected. It also behaves as the front-end of the FRAGme system to applications. The methods in ControlCenter that are essential for applications are:

```
public static boolean setUpConnections(String groupName) {}
public static Object createNewObject(Class type) {}
public static Vector getAllObjects() {}
public static Vector getAllObjects(Class className) {}
```

A detailed description of the use of the ControlCenter for application programmers can be found in the chapter "Application programmers guide to FRAGme".

System glossary

- speech act
- used messages (MODIFY, DELETE, NOTIFY)
- synchronization issues
- object creation (object ID)
- JGroups settings
- Peer fostering

Design Pattern

The Polymorphic Factory Pattern

The **static factory( )** method forces all the creation operations to be focused in one spot, so that's the only place you need to change the code.

This is certainly a reasonable solution, as it throws a box around the process of creating objects. However, the *Design Patterns* book emphasizes that the reason for the *Factory Method* pattern is so that different types of factories can be sub-classed from the basic factory (the above design is mentioned as a special case).

However, the book does not provide an example, but instead just repeats the example used for the *Abstract Factory* (you'll see an example of this in the next section). Here is **ShapeFactory1.java** modified so the factory methods are in a separate class as virtual functions. Notice also that the specific **Shape** classes are dynamically loaded on demand:

```
//: c05:shapefact2:ShapeFactory2.java
// Polymorphic factory methods.
package c05.shapefact2;
import java.util.*;
import com.bruceeckel.test.*;

interface Shape {
  void draw();
  void erase();
}

abstract class ShapeFactory {
  protected abstract Shape create();
```

```
  private static Map factories = new HashMap();
  public static void
  addFactory(String id, ShapeFactory f) {
    factories.put(id, f);
  }
  // A Template Method:
  public static final
  Shape createShape(String id) {
    if(!factories.containsKey(id)) {
      try {
        // Load dynamically
        Class.forName("c05.shapefact2." + id);
      } catch(ClassNotFoundException e) {
        throw new RuntimeException(
          "Bad shape creation: " + id);
      }
      // See if it was put in:
      if(!factories.containsKey(id))
        throw new RuntimeException(
          "Bad shape creation: " + id);
    }
    return
      ((ShapeFactory)factories.get(id)).create();
  }
}

class Circle implements Shape {
  private Circle() {}
  public void draw() {
    System.out.println("Circle.draw");
  }
  public void erase() {
    System.out.println("Circle.erase");
  }
  private static class Factory
  extends ShapeFactory {
    protected Shape create() {
      return new Circle();
    }
  }
  static {
    ShapeFactory.addFactory(
      "Circle", new Factory());
  }
}

class Square implements Shape {
  private Square() {}
  public void draw() {
    System.out.println("Square.draw");
  }
  public void erase() {
```

```
    System.out.println("Square.erase");
  }
  private static class Factory
  extends ShapeFactory {
    protected Shape create() {
      return new Square();
    }
  }
  static {
    ShapeFactory.addFactory(
      "Square", new Factory());
  }
}

public class ShapeFactory2 extends UnitTest {
  String shlist[] = { "Circle", "Square",
    "Square", "Circle", "Circle", "Square" };
  List shapes = new ArrayList();
  public void test() {
    // This just makes sure it will complete
    // without throwing an exception.
    for(int i = 0; i < shlist.length; i++)
      shapes.add(
        ShapeFactory.createShape(shlist[i]));
    Iterator i = shapes.iterator();
    while(i.hasNext()) {
      Shape s = (Shape)i.next();
      s.draw();
      s.erase();
    }
  }
  public static void main(String args[]) {
    new ShapeFactory2().test();
  }
}
```

The factory method appears in its own class, **ShapeFactory**, as the **create()** method. This is a **protected** method, which means it cannot be called directly, but it can be overridden. The subclasses of **Shape** must each create their own subclasses of **ShapeFactory** and override the **create()** method to create an object of their own type.

The actual creation of shapes is performed by calling **ShapeFactory.createShape()**, which is a static method that uses the **Map** in **ShapeFactory** to find the appropriate factory object based on an identifier passed to it.

The factory is immediately used to create the shape object, but you could imagine a more complex problem where the appropriate factory object is returned and then used by the caller to create an object in a more sophisticated way. However, it seems that much of the time the intricacies of the polymorphic factory method are not required, and a single static method in the base class (as shown in **ShapeFactory1.java**) will work adequately.

Notice that the **ShapeFactory** must be initialised by loading its **Map** with factory objects, which takes place in the static initialisation clause of each of the **Shape** implementations. So to add a new type to this design you must inherit the type, create a factory, and add the static initialization clause to load the **Map**. This extra complexity again suggests the use of a **static** factory method if it is not necessary to create individual factory objects.

Singleton-Pattern

According to the "Gang of four" (cited by Shalloway), the Singleton's intent is to ensure that a class has one instance, and providing a global point of access to it. This is helpful, if an application is central components.

These Components are in FRAGme

- The FragMeFactory
- ObjectManager
- PeerManager

The pattern is implemented by the following lines of code (adapted to the particular class !)

```
private static GenericBoundary instance;


public static GenericBoundary getInstance(){
  if (instance==null){
    instance=new GenericBoundary();
  }
  return instance;
}
```

The method getInstance (which is static and therefore accessible without instance) checks, if there is already an instance of the GenericBoundary. If, then this instance is returned, otherwise a new one is created.

Observer (A.K.A Dependents and Publish-Subscribe) Pattern.

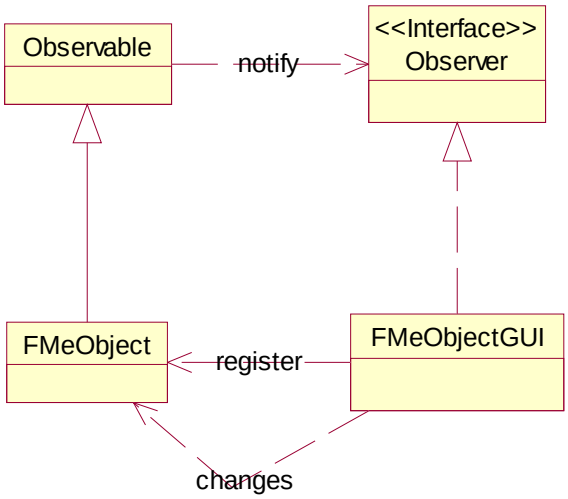
What is the Observer pattern?

The Observer pattern is about decoupling the data objects from the GUI objects. Shalloway suggests that the three categories of patterns: structural, behavioural, and creational, should have the behavioural patterns split into behavioural and decoupling categories. Observer is a pattern that would clearly fit into this new category. The pattern creates a one-to-many relationship between an object and its dependent object so that when an object changes its state all its dependent objects are notified automatically.

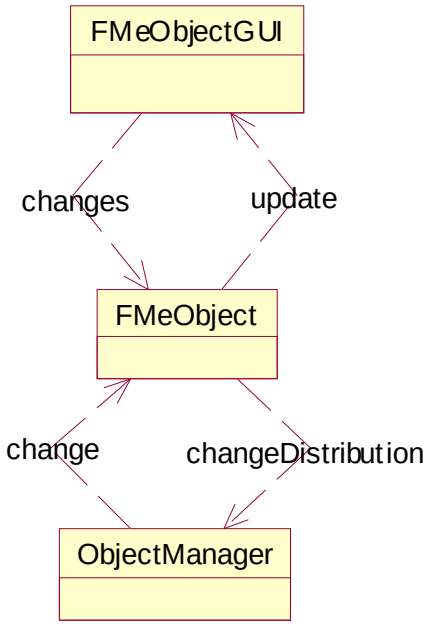
When the dependent objects change e.g. the client application the independent object should not need to change to be able to notify them of its changes of state. This decoupling allows different applications and GUIs to be plugged into the underlying data objects, which precisely suits our

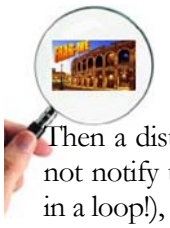
framework. In our system the GUI or application only has to implement the java.util.Observer interface to be automatically notified of the data object changes.

The observer patter is used in the class FMeObject. FMeObject is a subclass of `java.util.Observable`, which means that an application can register implementations of `java.util.Observer` as wished.



Every FMeObject-class gets changed from two points. The Application itself changes the data (for example the GUI), and the ObjectManager changes the Object (for example because another peer changed it).





Then a distinction is necessary – a change from the ObjectManager has to update the GUI, but not notify the ObjectManager (it would then send a distribute a new notification – which results in a loop!), but a change from the GUI has to result in a notification of the ObjectManager.

```
public void setValue(boolean state) {
    this.state=state;
    this.setChanged();
    this.notifyObservers();
}
public boolean getValue() {
    return state;
}
public void changeValue() {

    if(state){
      this.setValue(false);
    } else {
      this.setValue(true);
    }
    this.change();
}
```

The code, which has to be written to make this distinction, can be found in Chapter 3.

The Observer pattern overhead is not necessary for strong one to one dependencies where other dependencies are not likely to be added. This does not apply to our project and more importantly if the notification of events is conditional or the system is run under differing conditions or by different customers each having a different list of required observers. The data object cannot anticipate every object that might need to know about the event if it could then the Observer pattern should not be used. With an open framework this becomes important.

Façade Pattern

The intent of this pattern is to present a unified high-level interface that makes a subsystem easier to use (Shalloway 2002). The interface hides the workings of the underlying subsystem, but makes the functionality available.

In FRAGme, the PeerManager is implemented using the façade pattern. We decided to use this, because this makes it easy to swap and switch. The underlying code is not invisible to the layer above (object manager) as the object manager calls methods on the interface and does not call any methods inside the package peers (one exception is the Button application where the PeerManager_Impl is called directly for test purposes)

The peer manager presents a unified face to the object manager and so does implement the façade pattern.

One of the consequences of using the façade pattern is that by simplifying the use of the subsystem certain functionality may not be available to the client (Shalloway 2002). A file-

swapping program was started but the framework could not provide enough information about the peers without breaking the façade pattern. It would be necessary to call the getIPList() method in the peer manager from the client. This could be remedied by making an object manager interface to implement the façade pattern to allow extra functionality to be passé to the client application.

The advantages of the façade pattern are that it provides a simpler interface for the subsystem and it reduces the amount of objects the client application has to deal with. In the current framework the application has to only deal with the object manager, but it does not have all the required functionality. It is also easier to monitor the use of the data objects as all the method calls go through the object manager. It would be very easy to change the programming of the peer manager interface replacing the current classes that implement the interface. This would be significant in our project if we wanted to change from peer to peer to client server. If the project were continued the next important step would be to implement the façade pattern for the Object Manager.

Reference

Design Patterns Explained Alan Shalloway and James R. Trott, Addison Wesley 2002, The Software Patterns Series.

Areas for future development

…

Test application

…

Pong as an example application
SimpleSpaceBattle

Chapter

6

Testing

Software testing is an important part of Software Engineering. Software testing is a process used to identify the correctness, completeness and quality of developed computer software. Testing involves operation of a system or application under controlled conditions and evaluating the results. The controlled conditions should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should. It is oriented to 'detection'.

Testing

Software testing includes different kinds of software testing activities. Unit testing and integration testing are the most common ones among them. In computer programming, a unit test is a method of testing the correctness of a particular module of source code. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct.

For our project, we use JUnit testing tool which is an unit testing tool for Java programming language, commonly used in Extreme Programming. JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java JUnit is Open Source Software, released under the Common Public License Version 1.0 and hosted on SourceForge.

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing. The goal of integration testing is to verify functional, performance and reliability requirements placed on major design items.

For our FRAGME project, we divided it into **four categories** and for each of them we created unit test cases. Furthermore, the development of the games in the second semester can be seen as a system test for the Framework. Issues that came up there are discussed in this chapter, too..

Test cases for peer start up

First three tests are using the JUnit test environment. Code is available on the group work folder. JUnit tests (1 – 3) are tested on PC using the JBuilder IDE. Test 4 is also tested on the Zaurus.

Test case 1: correct start of peer connection

--- Test passed

This test calls ControlCenter.setUpConnection() and checks whether this function returns true. The test has been done first without another peer running, so that the peer sets up the connection and listens, and with another peer running the same group name, so that the peer actually sets up a connection to the other peer.

Test case 2: starting peer and getting all objects

--- Test passed

In this test, a help function is started first and sets up a connection. Then it creates 50 objects of type "TestObject" and 50 objects of type "TestObject2". After that, the JUnit controlled test function is started and sets up a connection to the same group. It calls getAllObjects() and checks whether 50 objects where transferred.

Test case 3: starting peer and getting all objects of a particular class

--- Test passed after changes

Finally, it is also possible to get only objects of a certain type. Again, the help function creates 100 objects (50 TestObject and 50 TestObject2). The function getAllObjects(TestObject) is then called and checked if it received the 50 objects of that type. In another run, the same test was done with getAllObjects(TestObject2). During the first run, a NullPointerException was thrown while the function tried to read the objects from the peer's storage. The reason for that was that in the FRAGme framework the peer's memory is divided into owned objects and other peer's objects. Because in this test case only the other peer's objects memory contained objects, the program crashed when trying to add objects from its own storage (where null was returned). The affected code has been changed.

Test case 4: parallel start up of several peers and receiving of objects

--- Tests passed, issues need further consideration

The synchronization of peers during start up was one of the major issues during the FRAGme 2004 framework development. To test a correct start up, the following was done:

One peer is started and creates 100 objects. Then two other peers are started at the same time (running a jar at the same time on different computers from the first one). Finally, a fourth peer starts and gets all the objects from the other peers. If everything works out, he should get 400 objects, including his own.

> Because there was always a small delay between the two peers, they never started at the absolute same time, so it can't be said what happens in this case. In case of almost parallel start up, the peers all join the group and transfer their objects correctly.

Three peers are started at the same time and create objects. In the end, every peer should have 300 objects.

> Does not work, every peer stays with 100 objects, and a fourth joining peer just gets 200 objects (connects to one peer). PC and Zaurus showed the same behaviour. Maybe a "search again" for peers would be helpful? Needs further consideration.

Test cases for creating objects.

Test case 1: Check to see if an object it created. (TestControlCenter):

This is checked by using the two methods which check to that an object of the right type is created and with the right id number.

```
 /* Checks to see that the framework creates an object of the
right type.  The
   * object it creates should be of type TestClass.
   */
  public void testCreateNewObject() {
    Class type = TestClass.class;
    Object expectedReturn = TestClass.class;
    Object actualReturn = controlCenter.createNewObject(type);

    assertEquals("Class type", expectedReturn,
actualReturn.getClass());
}
```

Result---- test passed.

Test case 2: Maximum limit test and Time taken.(TestNumberOfObjects Class):

This just uses a while loop to create as many objects as possible and until it runs out of memory. It also gives the time taken to create each 1000 objects.

```
long beforeTime = 0;
    long afterTime = 0;
//    long noOfObjects = 1;
//    long threshHoldTimeValue = 20;
```

```
//    ArrayList list = new ArrayList();
      Vector obj;

      ControlCenter.setUpConnections("CreatingObjectTest");
      obj = ControlCenter.getAllObjects();

      beforeTime = System.currentTimeMillis();
      afterTime = System.currentTimeMillis();

      int counter = 0;
      beforeTime = System.currentTimeMillis();
      while (true) {
        try {

          ControlCenter.createNewObject(TestClass.class);

          counter++;
          if (counter % 1000 == 0) {
            System.out.println("Number of Objects: " + counter);
            afterTime = System.currentTimeMillis();
            System.out.println("Time taken to create last 1000
objects: " + (afterTime - beforeTime));
            beforeTime = System.currentTimeMillis();
          }
        }
        catch (OutOfMemoryError e) {
          System.out.println(e);
        }
      }
```

The results of this test are as shown:

Number of Objects: 1000
Time taken to create last 1000 objects: 791
Number of Objects: 2000
Time taken to create last 1000 objects: 631
Number of Objects: 3000
Time taken to create last 1000 objects: 281
Number of Objects: 4000
Time taken to create last 1000 objects: 430
Number of Objects: 5000
Time taken to create last 1000 objects: 301
Number of Objects: 6000
Time taken to create last 1000 objects: 591
Number of Objects: 7000
Time taken to create last 1000 objects: 380
Number of Objects: 8000
Time taken to create last 1000 objects: 641
Number of Objects: 9000
Time taken to create last 1000 objects: 701

Number of Objects: 10000

The average time to create a single object is 0.0004747 seconds

Test case 3: Simultaneous creation of objects. (TestSimultaneous Class):

This class works by creating two threads. Each thread creates 100 objects. Since they are running at the same time more likely they will be trying to create objects at the same time. We know if it works because there should be 200 objects at the end.

```
 Vector obj;

    ControlCenter.setUpConnections("CreatingObjectTest2");
    obj = ControlCenter.getAllObjects();


    // To test to see if the Framework can create objects at
the same time
    // we have used two Threads.  Each Thread runs at the same
time and creates
    // 100 objects.  If for some reason some of the objects
were not created
    // there would be less than 200.  If they all created fine
then there should
    // be 200 objects.

    TestThread thread1 = new TestThread();
    TestThread thread2 = new TestThread();
    thread1.run();
    thread2.run();
    try {
      sleep(10000);
    }
    catch (Exception e) {
      e.printStackTrace();
    }

    System.out.println("Should create 200 objects");
    System.out.println("Number it created was: " +
obj.size());
  }
```

Test cases for updating objects.

Test case 1: Check to see if an object is updated. (TestControlCenter):

I use JUnit test environment. I created a test class which extends FMeObject with a setVal() method containing **super**.change(). When the new created test class call the method setVal() the

object must be sent to other peer by using the FRAGme framework, then I got the updated one comparing with the original one to see whether it is updated.

```
public void testUpdateObject() {
     Class type = TestClass.class;
     int expectedReturn = 1;
     testClass.setVal(1);
     int actualReturn = testClass.getVal();
     assertEquals("Change object", expectedReturn,
actualReturn);
  }
```

The test result----- this test passed.

Test case 2: testing the time of updating an object.

I create the pieces of code below to show the time of updating 1000 objects. The increment() method contain the method **super**.change() which can be connected with the FragMe framework.

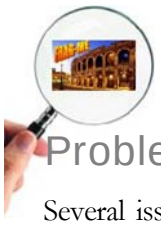
```
public void testUpdate() {
      long prevTime = System.currentTimeMillis();
      System.out.println("starting time: " + prevTime + "
milliseconds");

      for (int i = 0; i < 1000; i++) {
        testClass.increment();
      }
      long postTime = System.currentTimeMillis();
       System.out.println("finishing time: "+postTime+"
milliseconds");

    }
```

The result----

starting time: 1091362080125 milliseconds
finishing time: 1091362080312 milliseconds

The average time to update a single object is 0.187 milliseconds.

Test cases for peer dropout



Problems during game development

Several issues arose only when we started to implement the game "RoboJoust" in the second semester. The game can be seen as a system test which uses all components of FRAGme.

Object deletion

ISSUE:

Deleting objects with FRAGme by a peer that is not the owner.

SCENARIO:

Second peer tries to pick up item, item gets deleted on second peer but never on the first peer.

WHAT WAS WRONG:

When FMeObject.delete() is called, the object managers deleteObjects() method is called. If the peer is the owner of the object, it deletes it and notifies other peers (that works fine). But if it is not the owner, it just deletes it from the PeerStorageForOtherPeers? and tells no one about it, so it is kept by all other peers.

SOLUTION:

When FMeObject?.delete() is called, we now check whether we own that object. If not, we call ObjectManager?.requestDeleteObject?, which sends a message to the owner and requests the delete. Therefore, the new performative REQUEST_DELETE is used. The owner deletes the object and tells the other peers --- works.

CODE:

from FMeObject?:

```
    public final void delete() {
      Address myAddress = ControlCenter?.getMyAddress?();
      if(myAddress == this.ownerAddr) {

ControlCenter?.getObjectManager?().deleteObject(this.ownerAddr
, this.id);
      }
      else {

ControlCenter?.getObjectManager?().requestDeleteObject?(this.o
wnerAddr, this.id);
      }
    }
```

from ObjectManager_Impl?:

```
    public void deleteObject(Address addr, int id){
      if (addr.equals(myAddr)) {
        synchronized(ownObjects){
          ownObjects.deleteObject(id);
        }
```

```
ControlCenter?.getPeerManager?().send(ControlCenter?.DELETE,ne
w Integer(id),null);
    }
    else{
      synchronized (storageForOtherPeers?) {
        ObjectStorageForPeer? peerStorage =
(ObjectStorageForPeer?)
            storageForOtherPeers?.get(addr);
        peerStorage.deleteObject(id);
      }
    }
  }

public void requestDeleteObject?(Address addr, int id){

ControlCenter?.getPeerManager?().send(ControlCenter?.REQUEST_D
ELETE,new Integer(id), addr);
  }
```

from PeerManager_Impl?:

```
      else if (performative.equals(ControlCenter?.DELETE)) {

ControlCenter?.getObjectManager?().deleteObject(senderAddr,
            ( (Integer) content).intValue());
      }
      else if
(performative.equals(ControlCenter?.REQUEST_DELETE)) {

ControlCenter?.getObjectManager?().deleteObject(myAddr,
           ( (Integer) content).intValue());
      }
```

Notification of object deletion

ISSUE:

Notifying another peer that an object has been deleted without pulling objects in an UpdateThread all the time.

SCENARIO:

Peer tries to pick up item. Other peer does not get informed about the deletion unless it asks for the objects in an UpdateThread (what is not efficient). You also can't inform him by having another field (like isDeleted) in the object which would be sent as a change before the deletion, because you can't guarantee that the change will go through before the deletion.

WHAT WAS WRONG:

Not a real error, just the way the framework works. Whenever someone deletes an object, all others delete it too, but there is no notification from the framework to the application layer.

SOLUTION:

The object deletion in the framework has been changed. Now the owner does notify other peers that he deleted an object and leaves the deletion to them. Before a peer actually removes the object from its storage, it calls the method deletedObject() on itself. This method is abstract (similar to changedObject) and has to be implemented by the application programmer. A bit more implementation effort, but no UpdateThread anymore.

Notification of object change

ISSUE:

Deserialize() is called on a changed object and change code is executed (for example in the RoboJoust game, setup() is called on the changed object). In deserialize(), the object has changed, but paint() still paints the object at its old position.

SCENARIO:

Player moves left. Change does not show on the other peer. If the player makes another move, the previous change will be shown, and so forth.

WHAT WAS WRONG:

The problem lies in the FRAGme system of serializing and deserializing objects. Whenever a change is made, this change is broadcasted to all other peers and processed by a peer's receive() method. This method calls deserialize() and then adds the deserialized object to the ObjectStorage?. The problem here was now that the implementation of Robo.deserialize() was used to call repaint, when a change happened. Paint() then tried to get all FMeObjects?. But because deserialize() had not finished, the objects were not added to the PeerStorage? yet and repaint got the "old" object.

SOLUTION:

The second peer has to be notified with another method than deserialize(), if he does not want to use the ugly "update thread" for moveables. The solution I implemented is simple yet effective: FMeObject? got a new abstract method called changedObject(), which is called by the PeerManager? after deserializing and adding a received object. This method has to be implemented by any FMeObject. That means for Robo, that deserialize() only takes care of setting the new x, y values and changedObject() calls setup().

CODE:

from PeerManager_Impl?:

```
    public void receive(Message msg) {
      Address senderAddr = msg.getSrc();
      FragMessage? fragMsg = (FragMessage?) msg.getObject();
      String performative = fragMsg.getPerformative();
      Object content = fragMsg.getContent();
      else if (performative.equals(ControlCenter?.MODIFY)) {
          FMeSerialised? serialised = (FMeSerialised?) content;
          // first we deserialize it
          FMeObject? receivedObject =
 FragMeFactory?.deserialize(serialised);
```

```
            // then we add it to the storage
            receive(receivedObject);
            // now we work with the deserialized object
            receivedObject.changedObject();
        }
    }
```

from Robo:

```
    public void deserialize(FMeSerialised? serObject) {
      map[x][y] = null;
      this.imgOrientation = ( (Robo_ser)
 serObject).getOrientation();
      this.x = ( (Robo_ser) serObject).getX();
      this.y = ( (Robo_ser) serObject).getY();
    }
    public void changedObject() {
      checkOrientation();
      this.setup(x, y);
    }
```

Maintaining object references through serialization

ISSUE:

Peers lose object reference after deserializing objects.

SCENARIO:

Player moves. Then the other peer's player moves. Now the first peer has taken over the controls of peer two's character!

WHAT WAS WRONG:

The FRAGme factory has system of object reuse to save memory. Whenever an FMeObject is sent, it will be deserialized at the receiving peer and put into the object storage. The old representation of this object is just deleted from the object storage. The problem is, that you will lose the reference to an object (because a new one is created) in case of deserialization. But why did we control the first robo now? Well, when we sent the changed robo over, he was deserialized and had a new reference, but his old object was stored for reuse. When the other robo moved, he got this old object. We were still referencing that one, so we took over control over the other robo.

SOLUTION:

Before creating a new object, the FRAGme factory has to check with the object storage already exists. Only if it does not exist, a new object is created. Otherwise the new values will be deserialized in to the old object.

Memory usage

ISSUE:

Memory builds up while playing.

SCENARIO:

The RoboJoust is being played on the Zaurus. After a while, the Zaurus comes up with an Out-of-memory message. On the PC, this does not seem to happen.

WHAT WAS WRONG:

We can only attribute this problem to the garbage collector on the Zaurus which reportedly does not work too well. So every object that is not referenced anymore but not collected by the garbage collector might add to the problem.

SOLUTION:

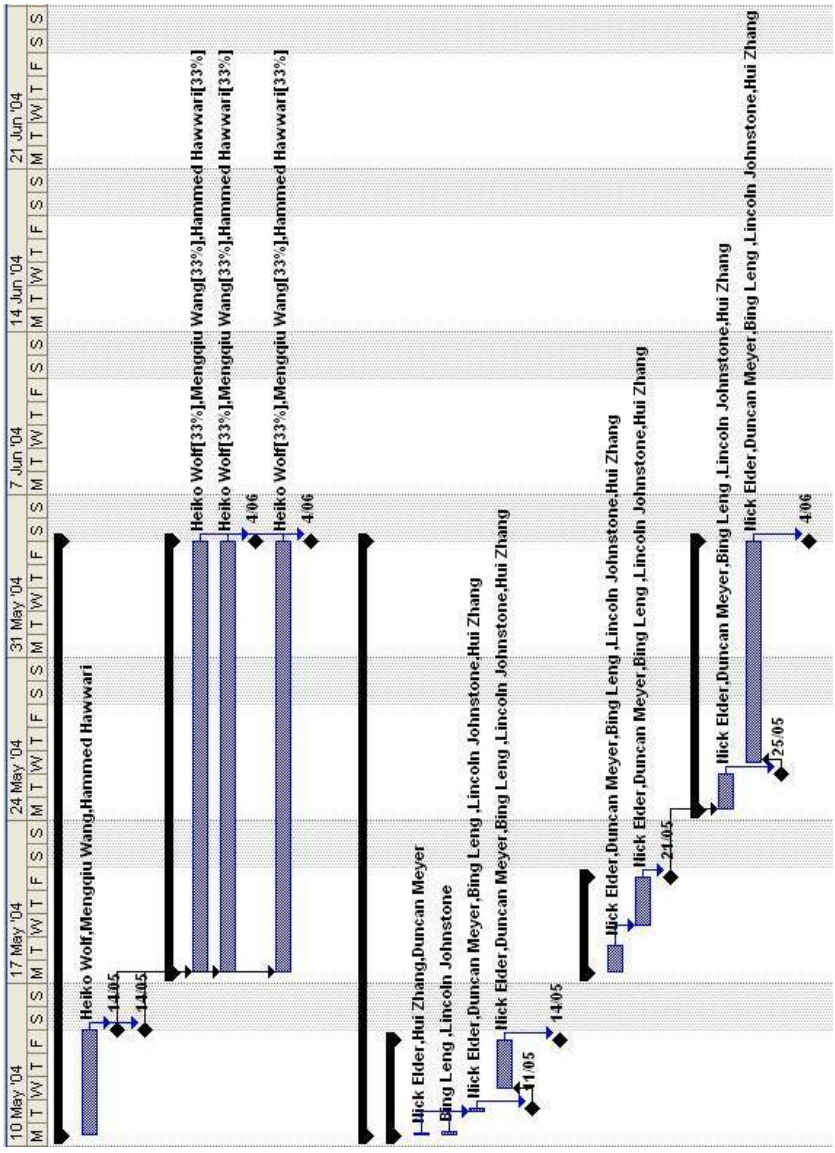
We checked by profiling what data structures and methods were used most and found that especially the FRAGme communication could cause trouble. Every time an object is created, a FRAGmessage is sent. If they are not deleted from the memory, they might cause the memory overflow. Out solution is the implementation of a factory for the FRAGmessages which reuses messages and helps to keep the memory usage down.

The same problem happens with the vector that is created and returned by the getAllObjects() method, which is called quite often. Here we now just use one vector which is reused.

Appendix

A

Appendix A – Project plan diagrams



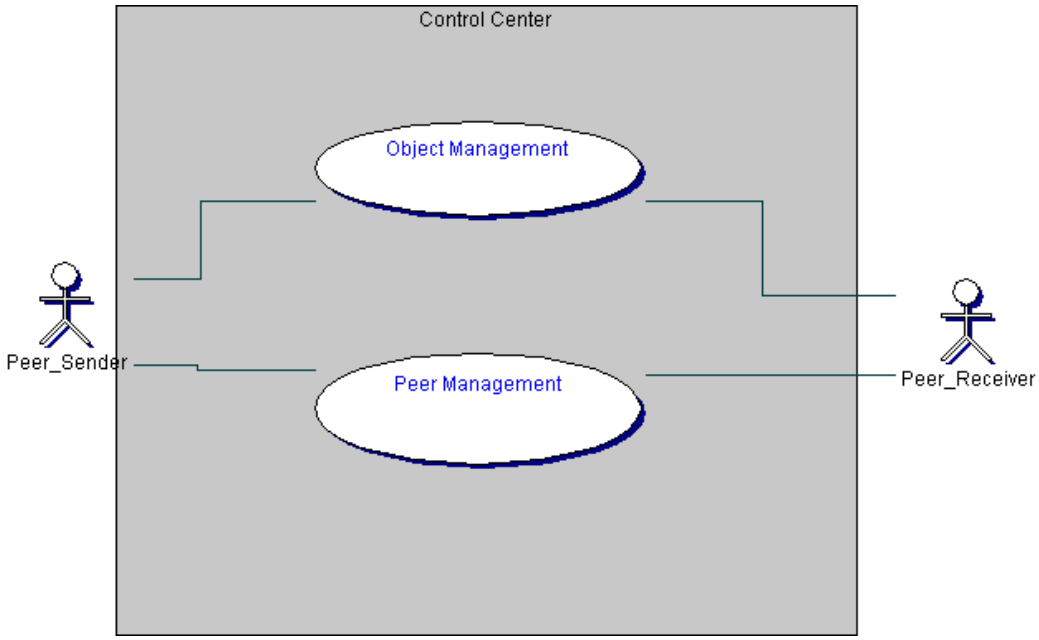
Task Name	Duration	Start	Finish	Predecessors	Resource Names
FragME Framework	**19.2 days?**	**Mon 10/05/04**	**Fri 4/06/04**		
Work proposal	4.8 days?	Mon 10/05/04	Fri 14/05/04		Heiko Wolf,Mengqiu Wang,Hammed Hawwari
Framework extensions proposal	0 days	Fri 14/05/04	Fri 14/05/04	2	
Bug report	0 days	Fri 14/05/04	Fri 14/05/04	2	
Framework Implementation	**14.4 days?**	**Mon 17/05/04**	**Fri 4/06/04**		
Fix bugs	14.4 days?	Mon 17/05/04	Fri 4/06/04	4	Heiko Wolf[33%],Mengqiu Wang[33%],Hammed Hawwari[33%
Implement proposed extensions	14.4 days?	Mon 17/05/04	Fri 4/06/04	3	Heiko Wolf[33%],Mengqiu Wang[33%],Hammed Hawwari[33%
Framework is extended	0 days	Fri 4/06/04	Fri 4/06/04	6,7	
Game improvement	14.4 days?	Mon 17/05/04	Fri 4/06/04	4,3	Heiko Wolf[33%],Mengqiu Wang[33%],Hammed Hawwari[33%
Game runs using the new extensions	0 days	Fri 4/06/04	Fri 4/06/04	9,6,7	
Application	**19.2 days?**	**Mon 10/05/04**	**Fri 4/06/04**		
GameMaker application	**4.8 days?**	**Mon 10/05/04**	**Fri 14/05/04**		
Get familiar with game maker	0.67 days?	Mon 10/05/04	Mon 10/05/04		Nick Elder,Hui Zhang,Duncan Meyer
Get ideas from existing game maker games	1 day?	Mon 10/05/04	Mon 10/05/04		Bing Leng ,Lincoln Johnstone
Agree on game design	1 day?	Tue 11/05/04	Tue 11/05/04	15,14	Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Design the game using game maker	2.8 days?	Wed 12/05/04	Fri 14/05/04	18	Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Game design	0 days	Tue 11/05/04	Tue 11/05/04	16	
Game maker game	0 days	Fri 14/05/04	Fri 14/05/04	17	
Basic application that uses the framework	**4.8 days?**	**Mon 17/05/04**	**Fri 21/05/04**		
Analyze basic functions of existing applications	2 days?	Mon 17/05/04	Tue 18/05/04		Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Implement basic application using those functions	2.8 days?	Wed 19/05/04	Fri 21/05/04	21	Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Basic application running on Zaurus	0 days	Fri 21/05/04	Fri 21/05/04	22	
Prototype development	**9.6 days?**	**Mon 24/05/04**	**Fri 4/06/04**		
Agree on prototype design	2 days?	Mon 24/05/04	Tue 25/05/04	23	Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Implement prototype	7.6 days?	Wed 26/05/04	Fri 4/06/04	27	Nick Elder,Duncan Meyer,Bing Leng ,Lincoln Johnstone,Hui Zh
Prototype design	0 days	Tue 25/05/04	Tue 25/05/04	25	
Prototype running on Zaurus	0 days	Fri 4/06/04	Fri 4/06/04	26	

Appendix

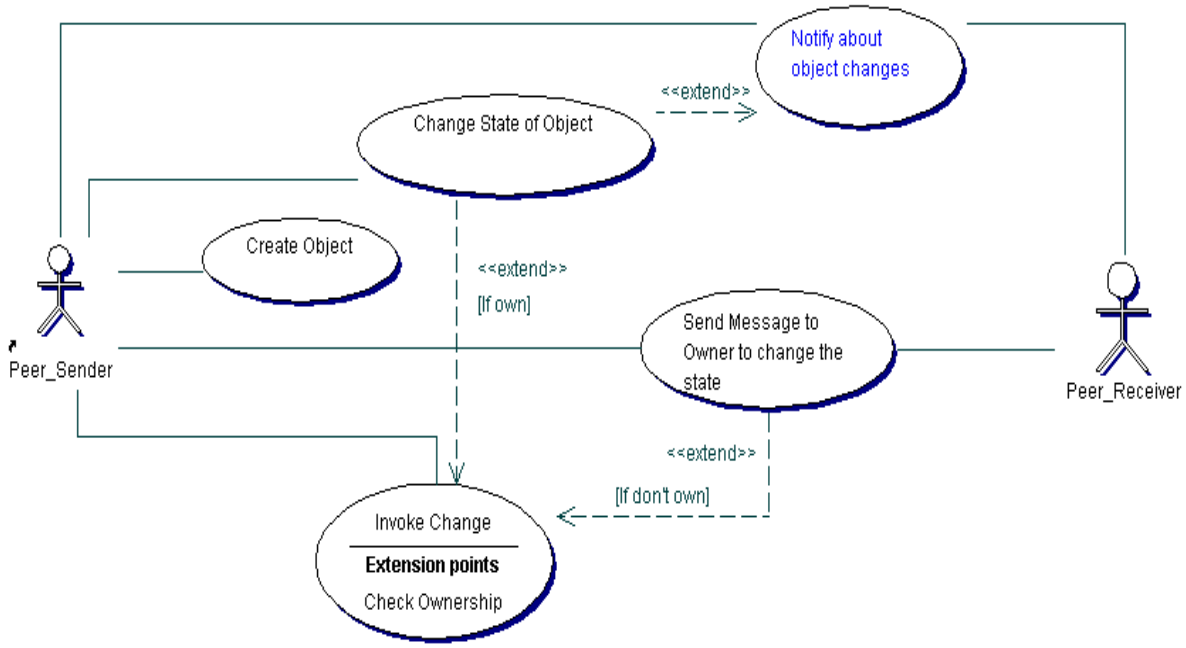
B

Appendix B – Use case diagrams

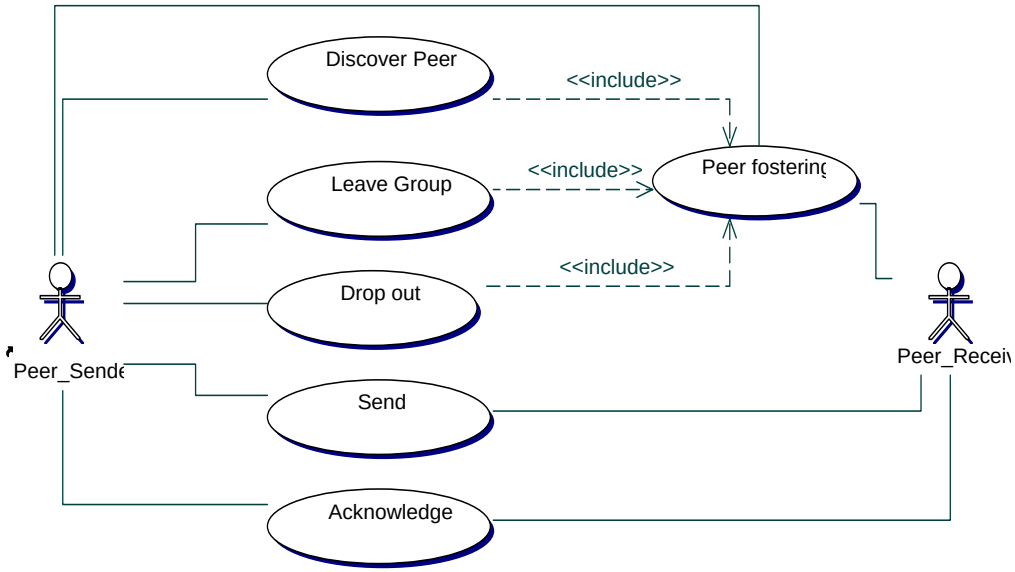
Top level use case:



PeerManager use case:



ObjectManager use case:



1 JGroups see http://www.jgroups.org

2 SCRUM see http://en.wikipedia.org/wiki/Scrum_(in_management)

3 Unison Unix file utility, see: http://www.cis.upenn.edu/~bcpierce/unison/