THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

Department of Computing

BA (Hons) Degree in Computing
Final Year Project Report
(2001/2002)

A Framework for an Agent-based Development Environment with Jini / JavaSpace –

Internet Integrated Development Environment Framework
(Internet-IDEF)

Supervisor: Dr. Stephen Chan Chi-Fai
Co-Examiner: Dr. Korris Chung Fu-Lai
Student Name: Lam Hoi Kit
Student ID: 98247632d

Date of Submission: 19 April 2002

Abstract

Software development requires the aid of many tools to accomplish a variety of tasks. An integrated development environment may not support particular tools required by specific projects. This project proposes to develop an open, distributed software development platform to solve this problem. Tools are plugged into the platform dynamically and become members of a distributed development environment. A framework, which is set of APIs, was built on top of Jini, making use of JavaSpaces and a number of design patterns. Major components in the framework include a platform infrastructure, utility classes, a file management system, and tool startup, communication, plug-in, tool administration facilities. These components aim at helping tool developers to plug tools into the framework cost effectively. Tool developers do not need a detail understanding of underlying concepts for dynamic plug-in provided by Jini in the development process. Standard communication data structure and protocol make tool interaction possible. A remote Java compiler and a collaborative UML editor, which includes a version engine as another tool, were used as sample applications and testing tools for evaluating the framework. The compiler was plugged into the framework and run successfully. Integration of the collaborative UML editor was designed and partially implemented. The successful integration of these tools into the framework validates the proposed design.

Acknowledgments

I would like to express my sincere thanks to my supervisor, Dr. Stephen Chan, for his guidance throughout the project. He has provided valuable advice and assistance in the whole progress of the project, including meeting, presentation practice, comments and suggestions. He also provides me a sufficient degree of flexibility to manage the project. His contribution to the project is significant.

I also appreciate Dr. Korris Chung, my project co-examiner, to attend my presentation and reading my report.

Finally, I have to specific thanks to my group member, Yim Wai Hin, who give useful support and help on the project and all classmates that give me useful opinions to my project.

Table of Content

List of Figure

List of Table

Chapter 1: Overview

1.1 Aim

The aim of the project was to develop a distributed software development platform for software development tool providers. The platform was based on Jini networking technology and JavaSpaces technology. Tool providers integrate their tools to the platform and a distributed software development environment is constructed.

1.2 Objective

To achieve the aim, certain objectives had to be met. I had to:

1) Identify problems involved in developing software development tools and their interactions so that common features for software development tool can be extracted and implemented into the framework.

2) Investigate new problems involved if tools are put into an Internet/network environment. Problems involve inherent networking problem such as partial failure, distributed resources management and etc.

3) Understand the concepts of Jini network technology and JavaSpaces technology. These two technologies were applied in the project. It is important because these two technologies are fundamental elements for developing API of the framework.

4) Design and implement major components of the framework. Components include the infrastructure for supporting dynamic join and leave of software development tools, a communication facility and etc.

5) Develop and integrate sample tools for the framework. A remote java compiler should be developed. This compiler and a collaborative UML editor, which was my group member's project, are then integrated to the framework. It is used to test the design of framework.

1.3 Proposed Distributed Software Development Environment Architecture

The figure 1, below, is the architecture of proposed distributed software development environment. Distributed software development tools were registered to lookup service, which can be viewed as a tool registry. Client users were able to discover those tools available in the network and to make use of them. The Components, which include development tools and a client application, in environment communicate through the communication facility provided by the framework. The development of communication facility was based on the JavaSpaces technology. A framework for this environment was developed in the project. The outcome was a set of APIs that tool developers can use them to plug their tools into the network. The architecture design will be discussed in Chapter 3 and detail framework design is in Chapter 4.

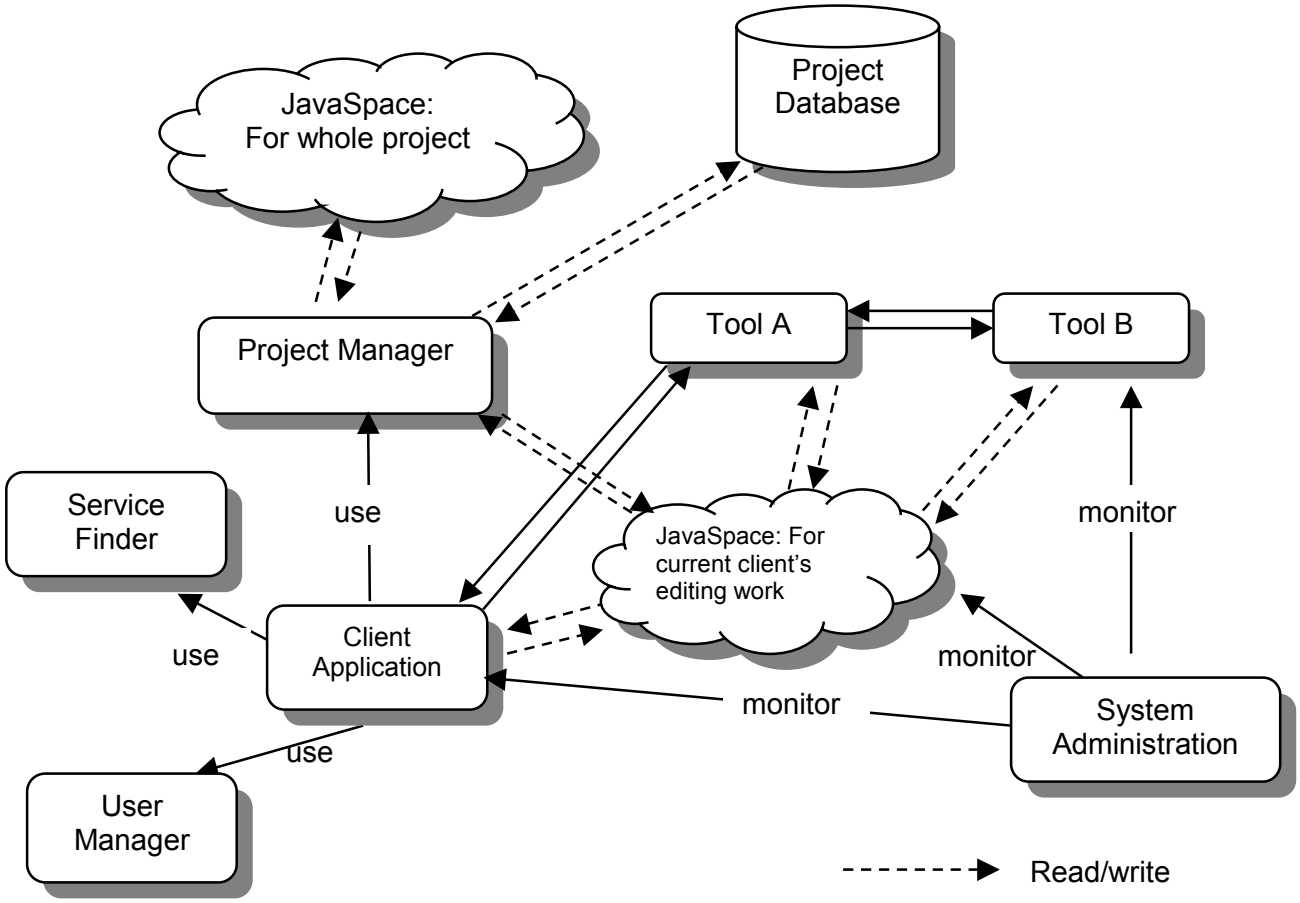

Figure 1: High-level architecture of the distributed integrated development environment.

1.4 Report Outline:

Chapter 2 The background of my project is discussed in this chapter that includes problem statement, related works, technological background and brief description of my work.

Chapter 3 The development process used in the project and the architecture of the framework is discussed in this chapter.

Chapter 4 Detailed design of the framework is illustrated in this chapter. The design for each component will be explained. Design problems and requirements of each component were presented. The solution and the design patterns that were applied in the solution were included.

Chapter 5 Two software development tools were used to validate the framework. The approach used in the integration is explained. Improved design of the existing framework is also provided when the requirement for integration of particular tool cannot be solved by use of APIs in the existing framework.

Chapter 6 This chapter is the result and discussion part of the report. There is a discussion on the component implemented and technical problems involved in the framework.

Chapter 7 The difficulties and the future work in the project will be discussed in this chapter. This includes technical challenges. The work that can be done to evolve and improve the platform was discussed.

Chapter 2: Background

2.1 General Area (Problem Statement):

Software development is a complex process. Many activities are involved in developing a computer system. Many development tools are required to aid the software development. Productivity is reduced when tool required is not readily available.

As time evolves, many integrated development environment (IDE) applications were developed. They attempted to solve the problem described above. An IDE is software that supports most of the tools developers needed. The most common tools are available in an IDE, such as text editor, debugger, complier and etc. However, several problems still exist:

1. It is difficult to provide full-feature development environment for satisfying every developer. This software is usually closed. Extensibility of the software is low. On the other hand, the IDE vendors always provide tools for upgrade. Dependency to the IDE vendors is high and flexibility becomes low.
2. Most of modern IDE tools are either standalone or collaborative work supported at local area network. This leads to geographic limitation. For example, the users need to download and reinstall the IDE when they are required to move somewhere else. It is inconvenience for software developers. On the other hand, developers nowadays require to work on difference platforms and to work on large number of tools. Maintenance of these tools on many platforms requires extra administration.
3. Finally, most of these tools do not support real-time collaborative work. This reduces productivity.

2.2 Related Works

Related projects are going to be discussed in this section. There are two existing projects and proposed project that are similar in nature as the project we proposed.

2.2.1 Existing Project: Netbeans, Eclipse

Recently, several integrated development environments had been developed focusing on extensibility. That is, they had plug-in mechanism or convention for other tool providers.

New modules or extension units can be plugged to the IDEs. Two emerging IDEs, Netbeans[18] and Eclipse[10], are typical example.

2.2.1.1 Background

These two IDEs were capable of supporting dynamic module plug-in for enhancing their functionality of the product. They are open-sourced.
In order to support dynamic module plug-in, two vendors provided a base platform for other tool providers to plug their tools into the systems. In addition, a set of API was developed to help tool provider to develop the plug-in.

2.2.1.2 Plug-in mechanism and technology used

Manifest is a file used to describe the components or modules that are going to be plugged into the IDE platform. It consists of a set of defined tags. All those tags describe all the plug-in related information that is required for making a successful plugging. It also gives instructions to the IDE platform about the way to install the components. The difference of manifest idea between Netbeans and Eclipse is on the format of manifest. Netbeans used its own defined tag in the manifest [12, 13], which is similar to Java JAR manifest, while Eclipse used XML in the manifest [11].

In NetBeans, it developed Open APIs that provided a way to insert pluggable tools. It was based on the Java Extension Mechanism [13]. The APIs provided classes and interfaces for implementing modules so that it is able to plug into the NetBeans platform. On the other hand, NetBeans required that the module should be a JAR-based Module. It defines the format of the JAR files.

In Eclipse, the mechanism was similar but the idea was different. It defined a new term called extension points. Any plug-in can define its new extension point for other new tools to implement on these extension points. The IDE will be like a tree structure of plug-ins. The idea met the requirement of dynamic extension. Some essential extension points were implemented in the IDE base platform as a start point for the tool providers.

2.2.1.3 Analysis

However, both Netbeans and Eclipse are standalone Integrated Development Environment. Although there was a team support in Eclipse, It was for local team only.

The notions of plug-in should be useful for our design of plugging different tools. Since design for local environment and design distributed environment is different, they is not able to completely applied to our project. It was found that Java Extension Mechanism was a good technique for plugging rich GUI existing application. The basic differences of the framework we proposed and the software development environment from existing IDEs are as follow:

1. Software development tools are distributed.
2. The platform lies on the network environment
3. The platform is open. High extensibility was achieved.
4. Collaboration work could be supported.

2.2.2 Proposed system: Jtop [22]

Jtop was a proposed system that aimed as providing a Java integrated development environment on top of Jini while our project aims at providing a software development platform for tool providers. Extensibility of the platform is one of main focuses in out project. Jtop did not have any further development for a significant period. Concepts given in Jtop were primitive and without a rigorous foundation. It provided ideas of the possibility of developing a distributed development environment. It gave a primitive idea by using JavaSpaces for designing the communication channel.

There were no concrete architecture of the system and the architecture was not complete in Jtop. No prototype was implemented. There was only a conceptual idea about developing a Java IDE on top of Jini. In addition, the focus of Jtop is different from our proposed system.

2.2.3 Summarization

A distributed software development framework was developed for other tool providers to plug their tool on it. Manually installation of tools is not required for application software developers when they want to make use of them. In addition, a sample tool was implemented for evaluating the framework.

The final outcome of the project was a set of APIs for tool provider to use. Tool developers utilize the APIs to plug-in their software development tools into the environment, depending on the level of integration they want.

Jini network technology and JavaSpaces technology were applied to develop a distributed software development platform.

2.3 Technological background [Jini, JavaSpaces and patterns]:

2.3.1 JINI network technology

Jini networking technology is an emerging technology. It can be said that the technology is in the adopting phase. Many researches are still going on.

Jini, which was developed by Sun, provides a distributed network infrastructure and programming model. It aims at address existing problems about distributed computing. They claim that distributed computing suffered from network failure, concurrency problems and lack of consistency. It is concluded that distributed application is living in an unreliable environment. As a result, Jini was developed in order to provide a networking architecture and programming model for building robust and reliable distributed application from an unreliable environment.

Infrastructure: Jini promotes a service-based architecture. A Jini-based application is viewed as a set of services interacting with each other. A service has an interface. It determines a set of features that can be used by other clients or other services [1, 2, 20]. Service can be a software component or a hardware device. This provides one more level of abstraction. The following figure shows the mechanism of finding and using a service by a client:

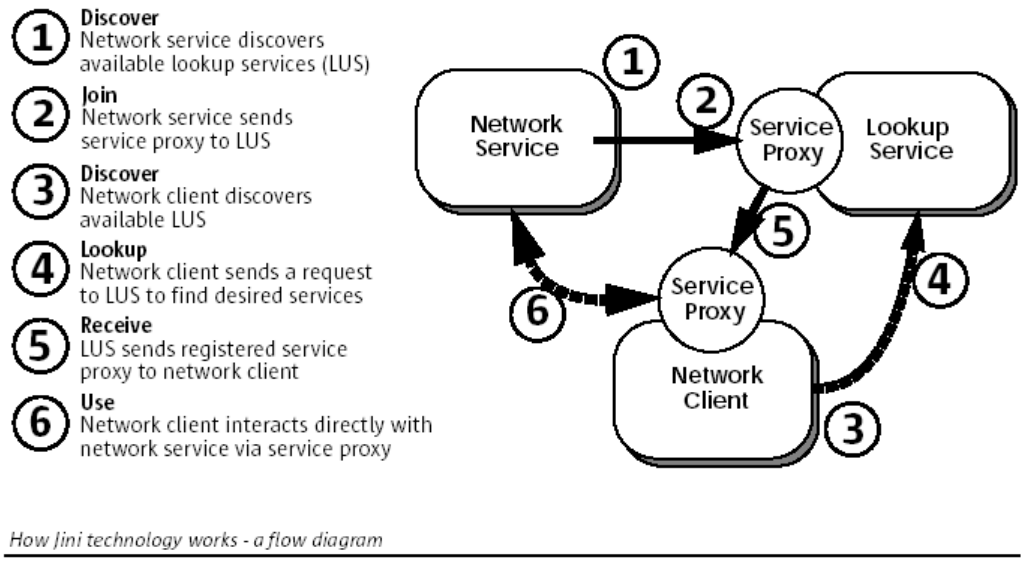

Figure 2: The flow diagram of how Jini technology works. The figure is extracted from [5]

Programming Model: Jini also offers a set of programming model make a network application with ability of dynamic join and leave, promoting self-healing, possibly administration free. Programming model consists of leasing, remote events and distributed transactions.

Leasing in Jini attempts to address self-healing in distributed computing. Self-healing is that distributed system should function for long periods of time without the intervention of a human administrator. Traditional distributed system failed in self-healing because access of resources is always granted indefinitely to the resources holders. If the holder crashes, the resources will be held and other interested parties is not allow making use of it.

Leasing in Jini borrows the real world lease concept. A grantor grants a lease with a period of time to grantee. Before the lease expired, grantee should renew the lease. If the lease is not renewed or is cancelled, the grantee cannot access the resource from the grantor. This ensures that resources are not held by a grantee indefinitely when error occurs. For instance, network failure is a critical problem in developing distributed program since client does not know that the connection to the server is suddenly not available. Clients do not know about that and wait for the reply from the server indefinitely. With leasing, the clients know that the server is not available and take action to make the system function correct by other means.

With leasing, failures are guaranteed to be detected by the grantor because any failed component cannot renew the lease it obtained. In addition, lease grantors are able to clean up failed component automatically. That is, failure detection and resources clean up happen completely within the grantor. These two features promote self-healing because any central authority is not involved in managing resources in a distributed system. Resource grantor can detect failure and then take appropriate action for failure recovery [1].

Remote event model in Jini is like the event model in AWT and Java Bean so that Java developer can easily adapt the remote event model concept without a steep learning curve. However, problems in the network environment should also be taken into account in remote event model. Jini notices about that and define a set of interfaces and conventions for distribute event. Leasing was also heavily used in distributed event.

Distributed transactions addresses the effect of partial failure when an operation of the application requires a number of service working together.

The overall Jini Architecture can be summarized as the table below:

	Infrastructure	Programming Model	Services
Java + Jini	Discovery/Join Lookup service	Leasing Remote Event Distributed Transaction	JavaSpace

Table 1: Summarization of Jini Architecture

The powerful features provided by Jini are the main reason why it was chosen as the fundamental component for developing distributed development environment. It includes dynamic network environment, resources renewal, self-healing. One of features in our framework is any tool that want to join in the application can dynamically integrate to the infrastructure with minimization of modification. The characteristics of the Jini network architecture satisfy the requirement of proposed system.

2.3.2 JavaSpaces Technology

2.3.2.1 What is JavaSpace?

JavaSpaces is an implementation of space-based model for distributed application development. JavaSpaces is greatly influence by similar system called Linda [17], which is the first programming language for space-based programming. The Programming

model of JavaSpace is simply. It only consists of a small set of methods. They are **`read`**, **`take`**, **`write`** and **`notify`**. From [17], it stated that JavaSpace is designed to solve two distributed computing problems: distributed persistence and the design of distributed algorithms.

The programming model for JavaSpaces technology is different from traditional message passing technique and remote method invocation technique. The application that developed with JavaSpaces can be view as a collection of processes cooperating via the flow of objects into and out of one or more spaces [7].

2.3.2.2 Space-based model and Space-based Programming

In JavaSpaces technology, space is a shared, network-accessible repository of objects [7]. Distributed programs coordinate by exchanging objects through spaces. A space is also regarded as persistent for object storage. Programming with JavaSpaces is relatively simple. Only few methods are needed to learn for programming with JavaSpaces. As mention above, they are **`read`**, **`take`**, **`write`** and **`notify`**. Objects can be read or taken from space by value matching. Objects can be written to space. There will be a notification when desired objects were put into the space from other processes. Figure 3, below, shows an overall picture of space-based programming in JavaSpaces.

Design of distributed data structures and distributed protocols that operate on them is the main core for building space-based application.

JavaSpaces was applied as a core technology for communication in this project. It is used to implement a communication channel facility for client and tool communication, and tool and tool communication.

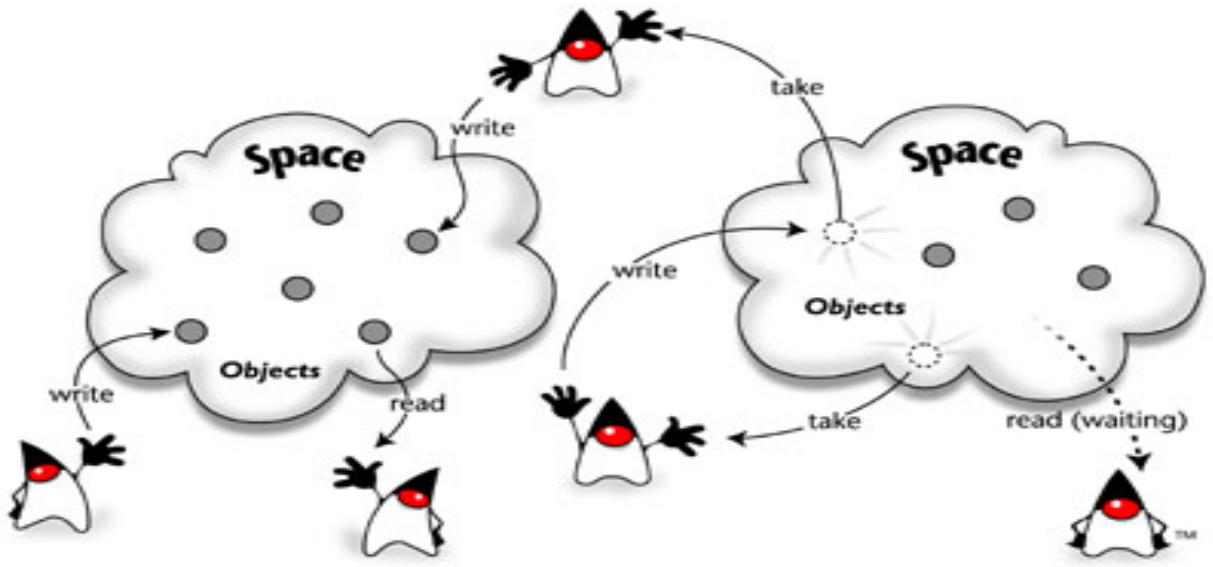

Figure 3: How JavaSpace technology works. Extracted from http://java.sun.com/products/javaspaces/index.html

2.3.2.3 Benefit of using JavaSpaces for building communication facility

The term *Space-based Communication* is used when JavaSpaces is used for communication of different processes. There are three properties of communication: *who* are involved in the communication, *where* are the participants and *when* do participants available [7].

In traditional message passing, senders and receivers need to know each other. They should know the location of each other and both must exist at the same time.

In space-based communication, it is not necessary to know who will receive the message or who did sent the message. It is also not necessary to know where the sender or the receiver comes from. In addition, both sender and receiver do not have to be exist or available at the same time. It is obvious that space-based communication is *anyone*, *anywhere* and *anytime*.

Advantage

From the above, space-based communication is a loosely coupled communication so that communication protocol and facilities can be designed in more flexible ways. It is not necessary to have a communication that obeys *anyone*, *anywhere* and *anytime* in designs. Modeling of traditional message passing in JavaSpaces is allowed.

2.4 Background knowledge of Design Pattern

Pattern Definition

In short, a pattern is a solution to a problem in a context. A pattern is another form of documentation. The power is that the document captures the knowledge from experienced software development expertise in an accepted form that can be shared [8]. Patterns are successful solutions to recurring problems. There are several characteristics for a pattern [6].

- Patterns documents existing, well-proven design experience, not invented or created artificially.
- Recurring – problem occurs many times in a certain contexts. A system of forces exists in those contexts. The solution attempts to resolve and balance the forces.

Common Sections in a pattern form

Following are the common sections that appear in popular pattern form [6, 8, 17].

Name – The name of the pattern. It is one of important elements because this help software designers seeking suitable solution. In addition, pattern name quickly become part of designers' vocabulary. Therefore, a good pattern name is very important.

Problem – This section describes the problem to be solved. It helps designers decide whether to read further when they are selecting appropriate pattern.

Context – This section tell designers when to apply pattern. Pattern only solves the problem if the context of that problem is appropriate. It also determines the impact of the forces.

Forces – Forces is a set of constraint needed to resolve in a pattern. It is a very important factor for understanding a pattern. From [16], it stated that

> *Patterns aren't rules we follow blindly; we should understand them and tailor them to our needs. If we understand the forces in a pattern, then we understand the problem (because we understand the trade-offs) and the solution (because we know how it balances the forces), and we can intuit much of the rationale.*

Solution – This section provides the detail to solve the problem so that designers know what to do. The solution, however, is general enough to address a broad context. Components that involved in the solution are presented. Their relationship and responsibilities is shown. But there is no implementation. This is important because it separate software design from implementation detail so that a pattern can use over and over.

Consequence – Benefit and liability of the solution are discussed in this section. It tells designers which forces are resolved after the pattern is used. Any new problem may arise. Any related patterns are there.

Advantage and disadvantage of using patterns in design

Since the solution in patterns are experienced and proven. Better design of the system can be achieved. In addition, patterns provide common vocabulary for software developer to share ideas in design.

However, since solution in patterns always solve problem by introducing additional of indirection, this may cause the software more complex or decrease the performance of the software. Therefore, design patterns should not be applied blindly. Judgment is required before a pattern is to be used [4].

Charpter 3: Development Process and Architectural Design

3.1 Development Process

The framework was developed in an iterative and incremental manner. A small set of problems and requirements were identified and analyzed one at a time. The solution of the problem was then designed and implemented. After that, next iteration began and similar tasks were preformed for next set of problems and requirements.

On the other hand, several main steps were involved in developing the framework. Firstly, the problem domain analysis was required to figure the problems and tackle them one by one. Understanding the concept of Jini networking technology was the next step. The technology formed the foundation of the project. Architecture of the framework is greatly influenced by the technology. In addition, investigation of the related works was required to get a deeper insight on the software development tool area.

A number of design patterns were used to achieve better design of the framework. As mentioned in the previous chapter, design patterns help developers to obtain a proven and experienced solution in design problems.

Two software tools were used to validate the framework. One is a remote java compiler and another one is a collaborative UML editor, called collaborative ArgoUml. Clients request a standalone java compiler remotely by the remote compiler tool. Collaborative UML editor was the work of my group member and integrated into the framework. A detail discussion will be illustrated in Chapter 5.

3.2 Architectural design of the framework

In this section, architectural design problem and the solution are discussed. At the end of this section, major components of the framework and their relationship will be shown. Detailed framework design will be discussed in the next chapter.

Architectural problem and requirement:

1. Easy plug-n-play
2. Easy to use
3. Easy to learn
4. Wrapper/Adapter should be offered to the tool vendors

5. Different tools vendor/module have different interface. The question was that to investigate the possibility to define a generic interface or type?
6. Vendors do not need to know how to find lookup service and to register the tools itself.
7. Tools have their own mechanism to obtain project resources. It may be different from the mechanism of JavaSpace based.

Figure 4, below, is the high level layered architecture of the whole distributed software development environment. The framework of the environment, named Internet-IDEF, was in the middle of the whole architecture as a platform for tool plug-in. The components for composing the framework will be discussed in the rest of the section.

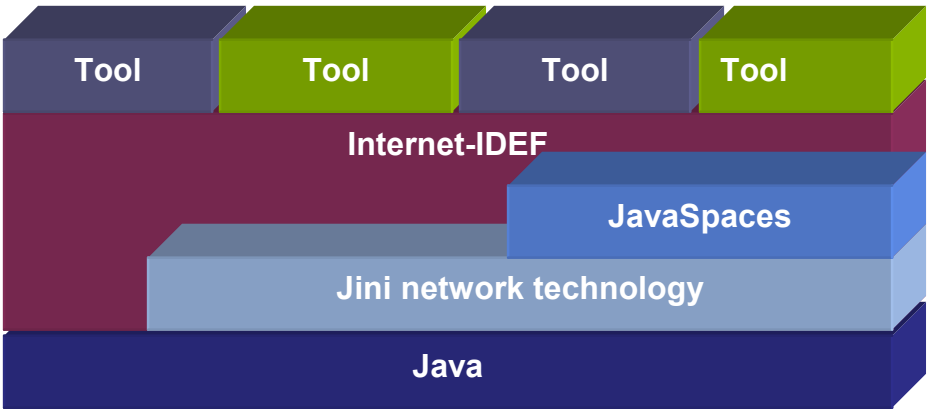

Figure 4: Layered Architecture of the distributed integrated development environment

Major components

- *Dynamic leave and join facility* – Core component of the framework. It supports plug-in for different software development tools, such as remote tools and tool in JAR file format, dynamically in a network environment.
- *Communication facilities* – Another Core component in the framework. Tools work alone reduces the power of Internet-IDEF because the outcomes of different tools are not compatible with each other if no standard communication standard. Extra works are required when users use those tools. A standard of data structure and communication protocol for manipulating those data was defined and implemented.
- *Helper utilities* - Since Jini network technology is the based technology for building Internet-IDEF. We attempted to encapsulate Jini from tool developers in order to reduce the work of tool developers in plugging their tools to the

framework. Tool developers do not need to handle issues about Jini network technology and repetitions of registration.

- *Tool startup mechanism* - Starting up software development tools into a Jini environment is tedious. In order to reduce the cost and effort in integration.
- *Tool administration* – All tools are running in a Jini network environment. The tool administration component provides tool providers to keep track the status of their tools. At this moment, basic administration functions were provided.
- *Network Resource management* – In network environment, network resource management is critical element for provide robust distributed application. Jini network technology offers us basic programming model for resource management, i.e. leasing. Domain specific leasing management was defined.
- *Project resource management* – Project resource management is a typical component in many IDEs. The component was not developed because it is not covered in the project.
- *Client Application* – A client side application is the component that makes the whole system usable. It is involved in supporting dynamic plug-in feature of the system. At this moment, since the focus of project was on the design and implementation core and backend architecture of the framework, tool-browsing function was implemented.
- *User Management* – User registration, role assignment, and access control is managed in the component.

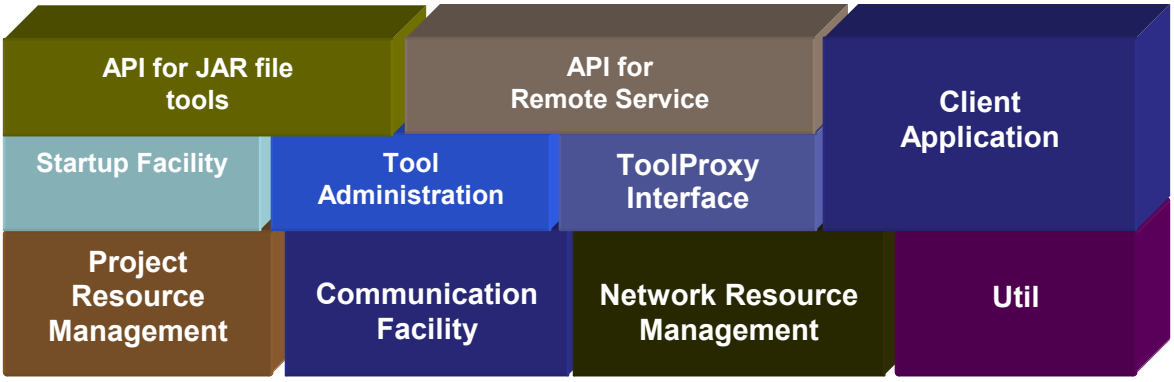

Figure 5: Internal architecture of Internet-IDEF

Chapter 4: Framework Design

In this chapter, design of each component is discussed in detail. The problems and requirements in developing corresponding component are firstly described. Design solution for the problem and requirements is then explained. If there are many constraints/forces to resolve in the problem, design solutions were divided for particular requirement(s). At the end, the overall picture for the component is shown.

4.1 Infrastructure for dynamic join and leave of tools

4.1.1 Problem:

An infrastructure of a system is required to be developed. It is suitable for distributed tools to plug-in dynamically.

4.1.2 General description:

The system is a distributed software development. Our system should be adaptive. New tools can be added and removed dynamically and transparently to our system. Client can use the new system without any modification about the client code. The framework is also an interactive system.

4.1.3 Requirements:

From the Problem statement above there are several requirement to be achieved, which are listed as follow:

Certain forces/constraints are involved in developing the framework:

1. Tools should be added and removed to the framework/system transparently.
2. Functionality of new tools is unforeseeable.
3. The client application may be developed before new tool released. Client should notice about existing of new tools and can use it without modification. On the other hand, user interaction is involved with the new tool. It may be impossible to have universal UI for all different tools.
4. The framework should allow integration of additional service/tools without requiring modifications to its core architecture.

5. Types of new tools are unforeseeable, that is it may be existing application, new developed tool for our system. Our framework should support both of these.
6. The framework should support existing/legacy tool, which may be command line application or GUI application with rich GUI and tightly coupled.
7. Tools may be either local or remote. The framework should be support both of them.
8. Tools can interact with each other so that communication facilities should be support by the framework. How to communicate with client and other tools?
9. Multi-user support should be considered. This involves user management.
10. Client may not know how to use the tool. User Help facilities may be required.

4.1.4 Solution:

Designs of the system structure will be illustrated point by point. Technologies used, design decision to resolve the forces and constraints are included, that is, the patterns that were used and the rationale behind. In the progress, the architecture of the system will be figured out.

Force(s):

- *Tools should be added and removed to the framework/system transparently*

Jini network technology provides an infrastructure for us to resolve the forces and constraints from the above section. The infrastructure offers a good fit solution for the problem of transparent join and leave in the distributed software development environment.

In Jini network technology, three protocols, which are **discovery, join** and **lookup**, forms the heart of the technology. In addition, these protocols, lookup service and distributed security compose the infrastructure of the Jini network technology. The protocols and the lookup service provide the solution for solving the problem that tool should be added and removed to the framework transparently.

Discovery, Join and Lookup protocol:
From [14], it state that discovery and join protocol is a:

Service protocols that allow services (both hardware and software) to discover, become part of, and advertise supplied services to the other members of the federation.

Discovery and Join Protocol

It defines the way for different services to be part of Jini system. Below are the steps for plugging a service to the Jini system.

Discovery

When a service is looking for a lookup service for registration, this is a discovery process.

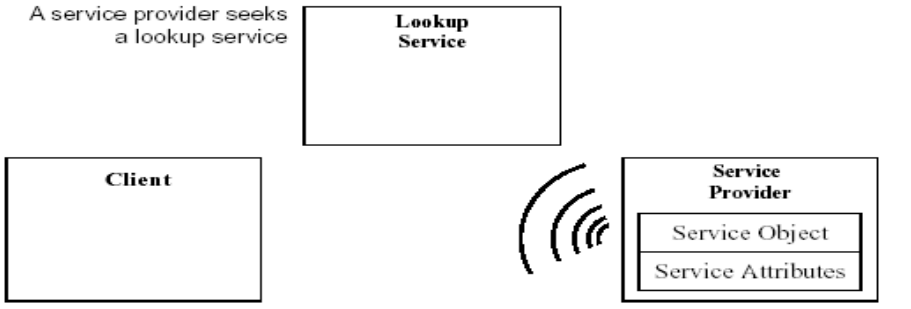

FIGURE AR.2.2: ***Discovery***

Figure 6: Jini's discovery protocol. The figure above is extracted from [14]

Jini 1.2 APIs provided a number of classes and interfaces for lookup service discovery. Jini service developers have a number of ways to discover lookup service. Sample code shown below is a high-level API for discovering a lookup service.

Sample Code:

There is **`LookupDiscoveryManager`**, which is provided by Jini 1.2 API, to mange the lookup service discovery. It is required to know groups, lookup service locator and a **`DiscoveryListener`**.

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

. . .

class Listener implements DiscoveryListener {
    public void discovered(DiscoveryEvent ev) {
        newregs = ev.getRegistrars();
```

```
        System.out.println("Lookup Service is found.");
    }
    public void discarded(DiscoveryEvent ev) {
        ServiceRegistrar[] deadregs = ev.getRegistrars();
        for (int i = 0; i < deadregs.length; i++) {
            //registrations.remove(deadregs[i]);
            System.out.println("Discarded: " + deadregs[i]);
        }
    }
}

. . .

String[] groups = new String[] {"public"};
LookupLocator locator = new LookupLocator(("jini://" + "123.123.23.23" + "/"));
LookupLocator[] locators = new LookupLocator[] { locator };
discoveryManager = new LookupDiscoveryManager(groups, locators, new Listener());
```

Join

When a service has located a lookup service and wishes to join it, this is the join process. The process also includes the discovery process.

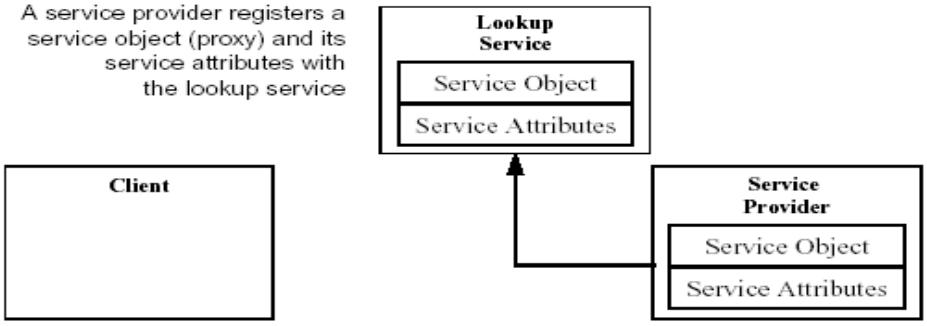

FIGURE AR.2.3: ***Join***

Figure 7: Jini's Join Protocol. The figure above is extract from [14]

Similar to lookup service discovery protocol, there are number of implementation to register a service to a lookup service. Sample Code in here is a high-level API for joining a service.

Sample Code:

JoinManager is responsible for managing the joining process for a service. The service object to be registered to lookup, attributes for that service, a **`ServiceIDListener`** object, a **`DiscoverManagement`** object and a **`LeaseRenewalManager`** object is provided.

```
searcher = new LookupSearcher( groups, locators);
. . .
public void registerToLookups(ToolProxy proxy, long leaseTime, Entry[]
attributes, File idFile) {
    this.idFile = idFile;
    . . .
    LookupDiscoveryManager discoveryManager =
searcher.getLookupDiscoveryManager();

    joiner = new JoinManager(proxy, attributes, this, discoveryManager, new
LeaseRenewalManager());

       System.out.println("register to lookups");
       . . .
    }
```

Lookup Protocol

When a client or user want to use a particular service, then lookup occurs.

In lookup, it involves locate and invoke the service described by its interface type, which is written in the Java programming language, and possibly other attributes. The service-locating step includes the discovery process.

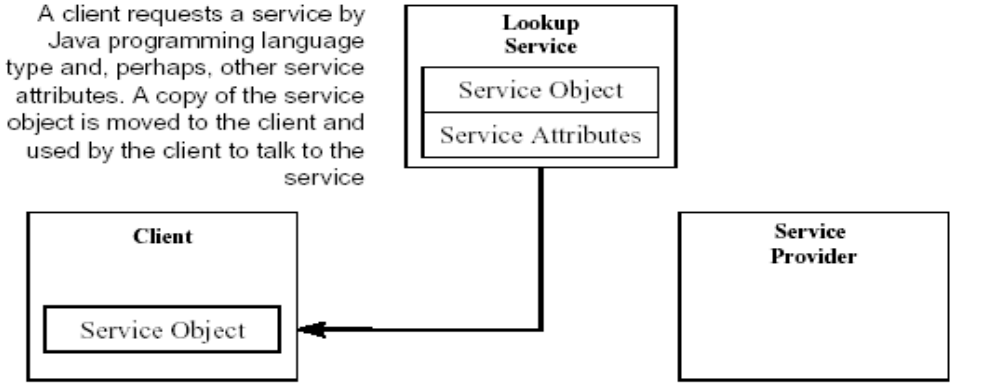

FIGURE AR.2.4: ***Lookup***

Figure 8: Jini's Lookup Protocol. The figure above is extract from [14]

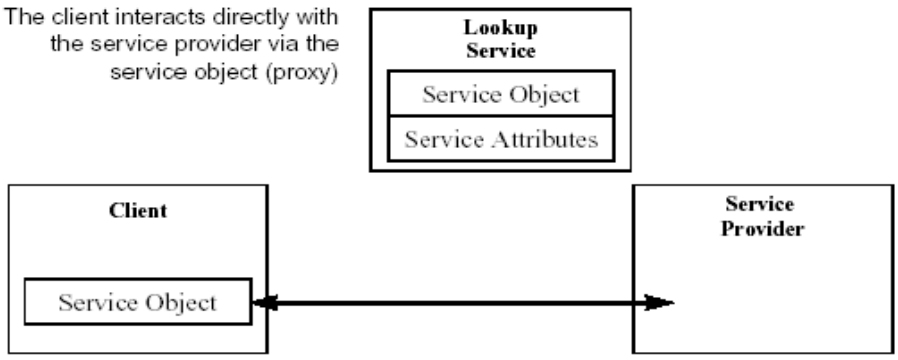

FIGURE AR.2.5: *Client Uses Service*

Figure 9: Client Uses a Jini Service. The figure above is extract from [14]

From the above two figures, client first discover service they want by matching the service they want. When there is such a service, a proxy will be downloaded to client. Client application or object can then associate with and request services to the service provider directly.

In the Jini 2.1 API, there is a **`ServiceDiscoveryManager`** class which encapsulates most of the tasks for discovering a service. Since **`ServiceDiscoveryManager`** is a high-level class, Jini 2.1 API also provided a number of low-level classes so that Jini application developer can implement its own service discovery mechanism for their need. Following are the sample code to illustrate the lookup process when **`ServiceDiscoveryManager`** is used.

Sample Code:

`ServiceDiscoveryManager` is responsible for manage the lookup process for a service.

```
. . .
searcher = new LookupSearcher(LookupDiscovery.ALL_GROUPS, null);
lookupManager = searcher.getLookupDiscoveryManager();
serviceManager = new ServiceDiscoveryManager(lookupManager, renewalManager);
. . .

// find a service
public ServiceItem[] findService(Class serviceType, Entry[] toolInfo,
                                 int       maxMatch,       ServiceItemFilter
serviceFilter) {
        Class[] types = {serviceType};
        template = new ServiceTemplate(null, types, toolInfo);
        return serviceManager.lookup(template, maxMatch, serviceFilter);
    }
```

Lookup Service:

In [14], there is a description of lookup service:

The lookup service serves as a repository of services. Entries in the lookup service are objects written in the Java programming language; these objects can be downloaded as part of a lookup operation and act as local proxies to the service that placed the code into the lookup service.

From the above definition, an entry is registered to the lookup service that can be downloaded and act as local proxy to the service is call the "Jini proxy". As mentioned in [1], this idea of downloadable service proxies is the key idea that gives Jini its ability to use services and devices without doing any explicit driver or software installation. This also is the main reason why Jini is chosen as the base technology of our framework.

From the overview of the Jini network technology infrastructure, it is obvious that the Jini network technology is greatly suitable for distributed application, distributed tools in our case, to be added and removed to a Jini system.

Jini network technology technically provides the solution to resolve the force discussed in this section. This also reduces the effort to design a new infrastructure for addressing the transparency service-locating problem. As a result, Jini was chosen to be the base technology to build our application framework.

As mentioned above, there are many number of ways of implementation for achieve the same goal described above, for example many ways to do a service discovery. In order to simplify the work of tool developers, **Façade** pattern was applied to simplify the interface of those API. Three classes were implemented to do the jobs described above. They were **`LookupSearcher`**, **`ServiceInitiator`** and **`ServiceFinder`**.

Force(s):

- *Functionality of new tools is unforeseeable.*
- *Client application may be developed before new tool released. Client should notice about existing of new tools and can use it without modification.*
- *The framework should allow integration of additional service/tools without requiring modifications to its core architecture.*

The first two forces are competing forces while the last two forces are similar forces. The first two forces are competing forces in the sense that an application can extends it functionalities dynamically while client is undesired to make any modification to support the new functionalities.

Although Jini have helped in dynamic plug-in, any kind of services can be registered to a lookup service. That is, any kind of developing tool can be added to the system as a service with any Java type. For example, a java complier can be implemented with **`Compiler`** Java type while a text editor can be implemented as **`Editor`** Java type. If tools are implemented in this way, modification of client application code is necessary to support all the Java service type. As a result, Jini's service matching mechanism arises another problem during development of our framework. Following is a brief explanation of Jini service matching mechanism.

Jini's service matching mechanism takes Java type (type-base matching) and the attributes of a service (content-base matching) as matching elements. In here, type-based matching is illustrated. In type-based matching, a service, which was registered to the lookup service, is matched when the type of that service is equal to or subtype of the template's type that the client want to match. For example, if a client application want to find a printing service that belongs to the type **`Print`**, any service that have been registered to the lookup with type **`Print`** will be matched and downloaded to the client application. Then client can choose one of them to print the file they want. Here is the sample code for match **`Print`** service.

```
...
Class[] types = new Class[] {Print.class};
ServiceTemplate  template = new ServiceTemplate(null, types, null);
...
//find the Print service by this template ServiceTemplate object
```

From the above illustration, client applications or objects, which decide to use the service provided by **`Print`**, should have the knowledge about the **`Print`** service beforehand. If the client later want to find **`Camera`** service, it should modify their own implementation to support the matching of **`Camera`** service and following code is required to add. This code modification is not desirable in the software development.

```
...
Class[] types = new Class[] {Camera.class, Print.class};
ServiceTemplate template = new ServiceTemplate(null, types, null);
...
```

In [3], it stated the problem described above and provides a solution to solve the problem. The solution is that an Adapter service is provided. The service will wrap the unsupported Java service type into already support Java service type. The solution was not suitable for our case. First, it only solves the problem partially. If there is a tool that does not belongs to any of service type registered, client application code is still needed to modify. In addition, it is necessary to encapsulate concepts of Jini as much as possible to reduce the burden of development of software tool. However, since the Adapter is a service, developer must have a sufficient knowledge of Jini technology. On the other hand, developers are required to know the semantic of all registered service but the documentation for those services may not be available.

A client application can accept all the service types. Users make their decision which tool will be used. However, the users do not know how to request services of the tool they want. This implies that when there is a new tool added to the system with a new interface type and the client wants to make use of the tool, the client application should make changes. This code modification is not desired.

In order to solve the above problems and constraints, a standard service interface was defined for software development tools. The interface is called `ToolProxy`. The idea is like the Java applet that there is an init method that is for overriding by tool developers.

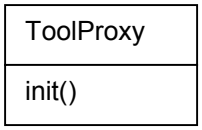

Figure 10: ToolProxy class with init() method only

Force(s):

- *Types of new tools are unforeseeable, that is it may be existing application, new developed tool for our system. Our framework should support both of these.*
- *The framework should support existing/legacy tool, which may be command line application or GUI application with rich GUI and tightly coupled.*
- *Tools may be either local or remote. The framework should be support both of them.*

Tool applications that will be as a part of our system can be divided into three main categories: 1) command line tool, 2) tools with rich Graphic User Interface (GUI) and 3) new development tools for our framework. For new development tools, since they can be tailored to fit with our framework, this will not be discussed.

Command Line Tool

For command line tool, we decided to make use of remote Adapter. That is, existing command line tools resident in a remote host. A set of classes and interface were defined, i.e. a set of mechanisms and conventions, so that tool developers can follow.

It is often the case that a command line tool performs one or few tasks. For example, a compiler does only the compilation job. According to this context, **Command** Pattern was used. This provided a way for client objects to make requests to command line tools. A request is implemented as an object. From the [4], the intent of **Command** pattern state that,

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

There are several benefits in it:
Firstly, **Command** decouples the client that makes the request from the one that knows how to handle the request.
Secondly, preprocessing and post-processing is allowed before and after making a request.
Finally, it is easy to add new command line tools since there is no need to change exist client class.

`Command` object was defined and the **`Command`** object can serve as a sending command for tool-to-tool communication

Adapter pattern is a pattern for converting the interface of a class into another interface client expected. Its variant, which is called pluggable adapter, is use for incorporating existing system with different interface to a new system. It can be a wrapper also. The canonical class structure is shown below:

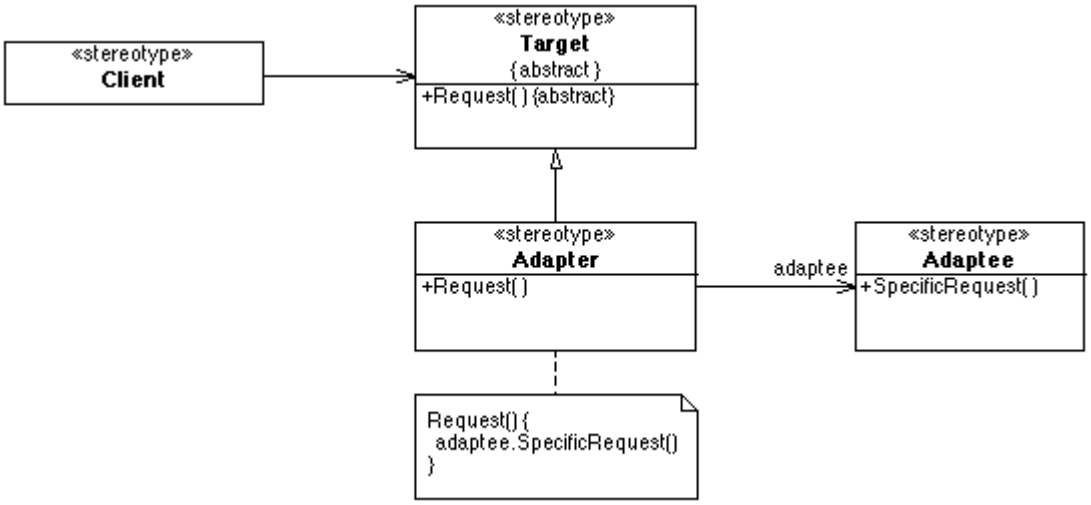

Figure 11: Canonical class structure of Abstract Pattern. Extracted from http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/Abstract-factory.html

From the class diagram above, client objects knows about the interface of class **`Target`**. If the client objects want to request to an object with different interface (**`Adaptee`**) but with similar responsibilities, an **`Adapter`** class is defined by subclassing the Target class. The request in the **`Adapter`** class is overridden to make request to the **`Adaptee`** object.

In the design of the project, the **`Adaptees`** were the tools that resident in remote host, mainly command line tools. **`RemoteService`** class, which implements the **`Remote`** class, is the implementation of **`Target`** in the **Adapter** pattern. The class structure of plugging command line tools shows as below. The dynamic structure would be illustrated in chapter 5.

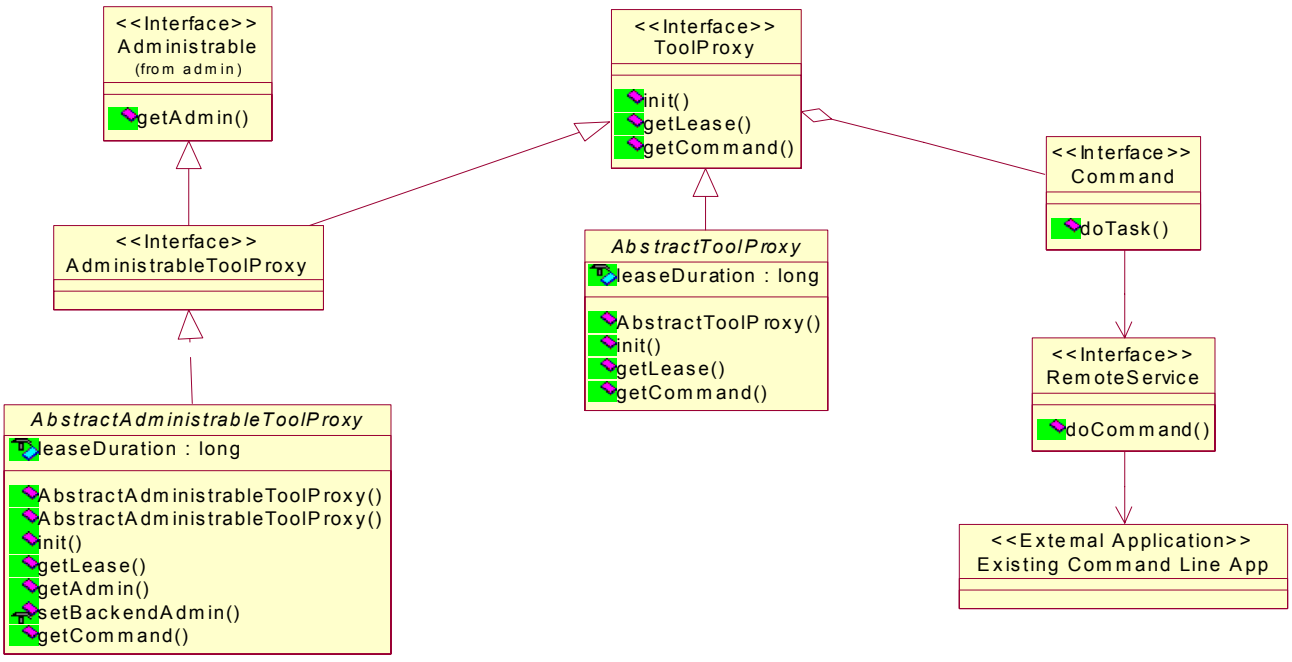

Figure 12: Remote software development tools and newly developed Class diagram.

Rich GUI or interactive tool

For tools with rich GUI, which is always a JAR application, another approach should be taken because Command pattern is not suitable for rich GUI application because many client interactions with the application are involved. The technique of Java Extension Mechanism was applied to solve this problem [9].

Although tool developers are also needed to modify the code for supporting tool communication, the need of modification should be minimized. In the Java Extension Mechanism, we mainly focused on using two classes: **`URLClassLoader`** and **`JarURLConnection`**. Two classes, **`JarRunner`** and **`JarClassLoader`**, which are used for running classes in the JAR file, were defined [9]. Reflective API is also used for invoking methods. A Jini proxy, called **`LocalToolProxy`**, was defined by extending the **`AbstractToolProxy`** and makes requests with **`JarRunner`**. Below shows the class structure for running JAR application.

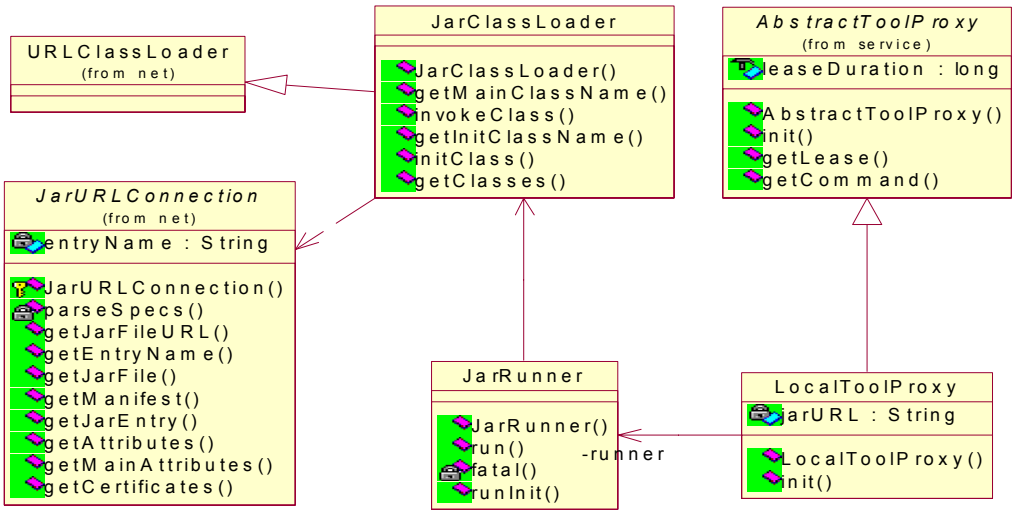

Figure 13: Rich GUI software development tools class diagram.

Convention for the jar manifest

In order to run a jar application after loading by **`JarRunner`**, a certain convention for jar manifest was required. The convention will be shown as follow:

```
Manifest-Version: 1.2
Init-Class: MainFrame
Class-Path: MainFrame2.jar
```

```
Manifest-Version: 1.2
Main-Class: MainFrame
Class-Path: MainFrame2.jar
```

The **`JarRunner`** supports two types of initialization of the application at client side. One is running the application by **`main`** method, same as invoke **`main`** method when running standalone program. It is indicated in a manifest with **`Main-Class`** tag. The tag specifies which class contains the **`main`** method. Another one is running the application by **`init`** method. It is indicated in a manifest with **`Init-Class`** tag. The tag specifies which class contains the **`init`** method. It is important to note that there should be a blank line at the end of the manifest file.

Other Force(s)

- *Tools can interact with each other so that Communication facilities should be support by the framework. How to communicate with client and other tools?*
- *Multi-user support should be considered. This involves user management.*
- *Client may not know how to use the tool. User Help facilities may be required.*

The first force will be discussed in detail in the section of Communication Data Structure and Protocol. The last two forces were not cover in this project so that they will be as the further enhancement of the project.

From the first force, interaction between tools was required. Method **`init`** in **`ToolProxy`** class is not enough when one tool wants to make use of another tools. It is because the **`init`** method is responsible for initialized the tool, typical for starting up a GUI at the client side. **`Command`** object is invoked by GUI component when necessary. As a result, one more method is defined. The method is called **`getCommand`** method. Method **`getCommand`** returns a **`Command`** object to the client object that invokes this method. Both tools can then communicate each other.

.

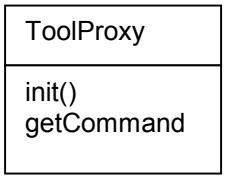

Figure 14: ToolProxy class with both getCommand() method and init() method.

4.2 Communication

4.2.1 Description

The framework provided communication facility for data transfer with client-to-tool communication and tool-to-tool communication.

4.2.2 Features

A standard communication data structure and protocol was provided for tool developers. All tool developers are able to develop their tools that can communicate with each other in a standard way.
In addition, a standard data transfer API was implemented so that tool developers do not need to do duplicate work and coding for data transfer.

4.2.3 Design Strategy

Because of the advantages of space-based communication mentioned in Chapter 2, space-based programming was used as a technique for implementing the communication facility. JavaSpaces was used as a base for communication facility.

Benefit of space-based communication - summarization

JavaSpaces technology is used as a basic tool for communication facility. JavaSpaces technology and space-based programming were used because of several reasons:

- Few methods are used for space programming: **`write`**, **`take`**, **`read`** and **`notify`**. They are simply but powerful.
- It is loose couple. Coupling between sender and receiver is loose.
- It is flexible to construct the communication facility according to specific problem domain.

Was JavaSpace a bottleneck for communication?
When more tools are joined to the framework and more clients use the framework, communication traffic increases. This will be the bottleneck of the system, as shown in the figure 15, below. Clustering should be considered for this scalability problem. The issue was not covered in the scope of this project. It can be considered as a further enhancement of the project. A commercial JavaSpaces product [21] was developed, which clustering was supported. It also provides valuable information on scalability problem in JavaSpaces.

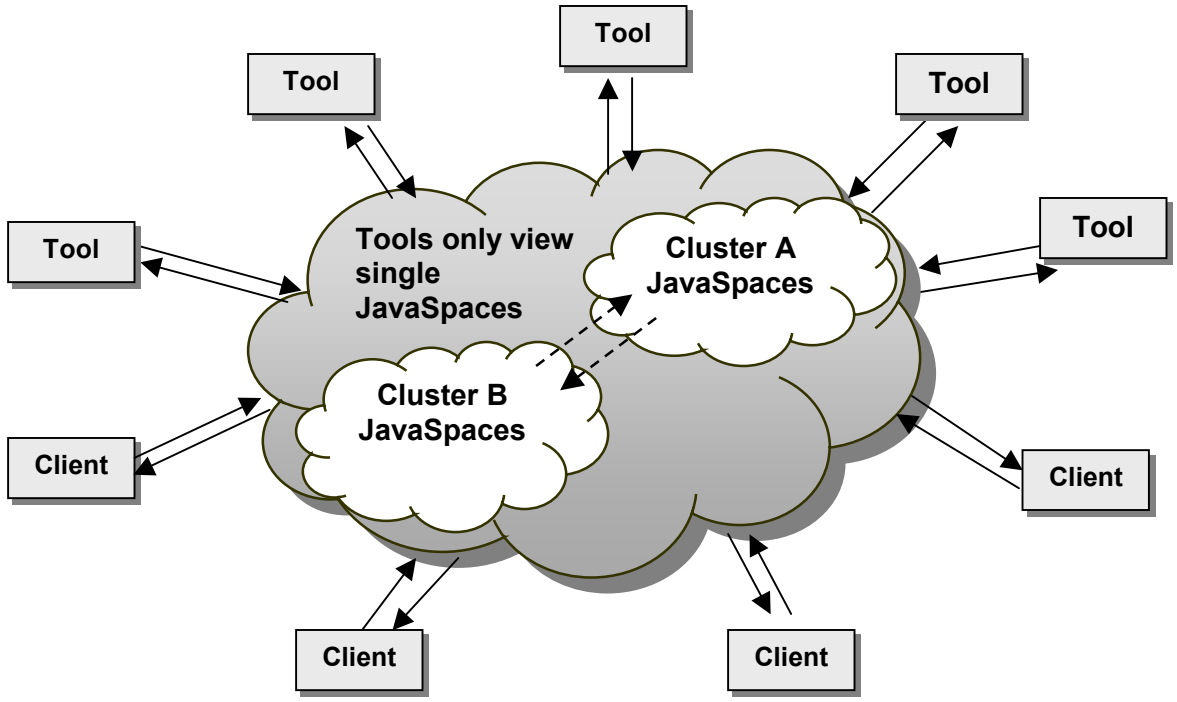

Figure 15: Clustering of JavaSpaces increase scalability.

4.2.4 Next Step: Design Communication Data Structure and Protocol

Communication Data Structure and Protocol were important design problems that should be addressed for achieving tools collaboration when JavaSpaces were applied to build the communication component of the framework. Communication component is one of important elements in the framework. The rest of this section discusses about the detailed design of the communication data structure and design of the communication protocol.

4.2.4.1 Communication Data Structures

The data structure below shows the elements in the **`DataObject`**. **`Receiver`**, **`Sender`** and **`Message Number`** forms the identifier of the data object. **`Receiver`** tells whom the data object is to be received and **`Sender`** tells who sent the data. **`Message Number`** will be assigned when the data object is written to a space.

DataObject

```
Receiver,
Sender,         /*this may store the serviceID*/
Project Name,
Name,           /*Name of the project resource*/
Type,           /*resource type*/
Version Number /*for supporting version control*/
Content/value,
Message Number
```

`MessageNumber` is an object that indicates the largest message number used in this channel, which is identified by **`from`** and **`to`** attribute.

MessageNumber

```
from,     /*from and to attribute identifer a unique channel*/
to,
number    /*the last message number of these channel*/
```

Why receiver and sender are required in the DataObject data structure?

This is because when the data object is written to the JavaSpaces, any other applications can take it away and the task they want although those applications may not know how to manipulate them. On the other hand, in the framework, a data object usually created and written by specific tool's Jini proxy and then the intended tool backend or receiver handle the data object. The **`receiver`** (**`to`**) and **`sender`** (**`from`**) form the identity of the channel. Detail for entry matching mechanism can be found in [7].

According to this scenario, an identity to a data object should be provided so that other (anonymous) application, which follow our protocol, take-and-modify the data object is not allowed.

Why does the framework not support tools with similar functionalities competing to do a task on the same project resource?

It is because it is difficult to manage. Users may not get trust with particular tools. For example, a client write the source Java file to the JavaSpace and let any compiler that was registered the lookup service to compile the source. In this case, it is not guaranteed that one particular tool is not used to compiler the file if the client find does not trust with the compiler.

Design alternatives

From the data structure defined, both send information and the receiver information should be obtained. We had been seeking design alternatives in order to have a better data structure.

Alternative 1: Remove the **`receiver`** attribute from the original design

We only need the sender part (from client side) and remove the receiver part because it is already know which tool will handle the task.

There will be a risk when the Java remote method invocation is not used for a request. It is not practical because a sender may send many data object to different receivers. Specific receiver does not know which data object it should get.

Alternative 2:Use a **`channel`** attribute instead of Sender attribute and Receiver attribute.

The structure is like this:

```
String channel
Integer position
Object content
...
```

It is not practical because the channel is setup at the client side. Receivers of the message do not know the channel before hand. Receivers cannot return the result object back to the sender if there is no **`sender`** attribute in the **`DataObject`**. On the other hand, it can be implemented but the channel set up procedure may take many steps and complex.

Alternative 3: Use Entry ID attribute for each data object instead of **`sender`** attribute and **`receiver`** attribute.

The design is similar to alternative 1 but use an ID instead. Base on knowing that receiver is not need in the data structure, an id can be assigned for each message or entry object. But useful information about the sender and receiver is missed and similar risks have to take. This design also restricts the flexibility of protocol design.

The original design seems the best design amongst all these alternatives because it provides flexibility for protocol design. The information for both sender and receiver is the most. It is not a problem to obtain sender information and receiver information

because the information can be obtained when the Jini proxy of particular tool is initialized.

4.2.4.2 Communication Protocol (Communication Mechanism)

Figure 16, below, is one scenario of high-level protocol design involved when a client object requests a tool to do task. Details about the communication protocol will be discussed later in this section. Typical procedure of the protocol is as follow:

1. Write data to the JavaSpace
2. Request tool to do task
3. Tool takes or reads data to the JavaSpace
4. Do process
5. Write the original data (application dependent) and the result data to the JavaSpace
6. Handle the result

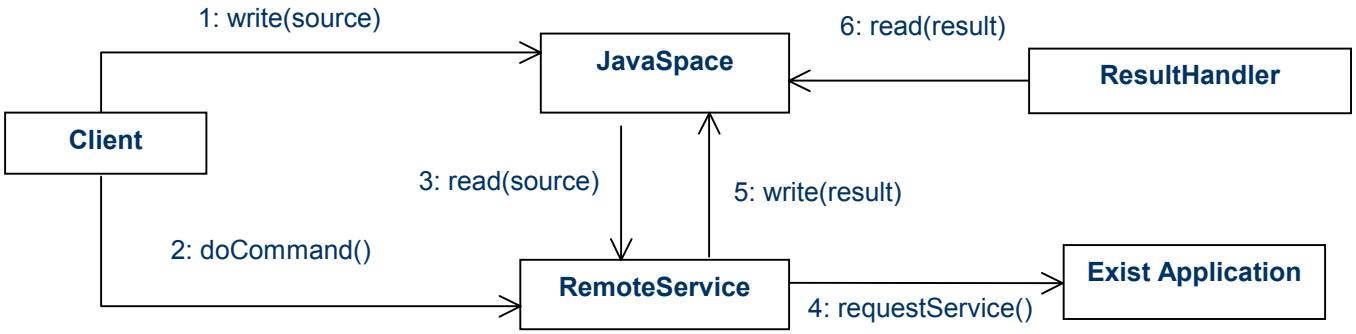

Figure 16: A collaborative diagram of the protocol design

Detail Communication Protocol Design

Since the tool's remote part does not know when will be the communication is needed, so the communication is started up by the client after a Jini proxy for corresponding tool has been downloaded. There are two design choices. One is let the message reader wait until a data object is put into space. Another is sender will call the reader to read the data in the space according to the channel the sender belong and the message of that object. The latter approach was chosen because readers do not have to wait for a request indefinitely. In addition, the proxy of particular tool takes the role to communicate back to its remote backend. It is not a problem to locate the remote backend if remote method call is used. Based on the data structure previously defined, design about the communication protocol is as follow:

Protocol Design:

1. Client creates a **`Channel`** object identified by the **`from`** and **`to`**
2. When the **`Channel`** object is created, a **`MessageNumber`** object will be created for this channel with **`from`** and **`to`**.
3. Client gets the message number from **`MessageNumber`** object
4. Client sends data to the channel object
5. Client pass the **`Channel`** object to the receiver side by RMI method called **`doCommand`**
6. Receiver use the **`Channel`** object to receive the data source
7. Receiver does the task
8. Receiver sends the result back to the sender, the result object have the same message number as the original object.
9. Client receives the result by matching the message number then the client does the further processing to the result.

Based on this design, each client creates a channel for itself when communication with particular tool is needed. Each client has to send the **`Channel`** object back to the receiver in the **`doCommand`** method in **`RemoteService`** class. This is one of drawback of the design because tool developers should think about how to send the **`Channel`** object back to the remote service. It is not a problem in our framework because it is often the case that Jini proxy of that tool make request to the remote side of the tool.

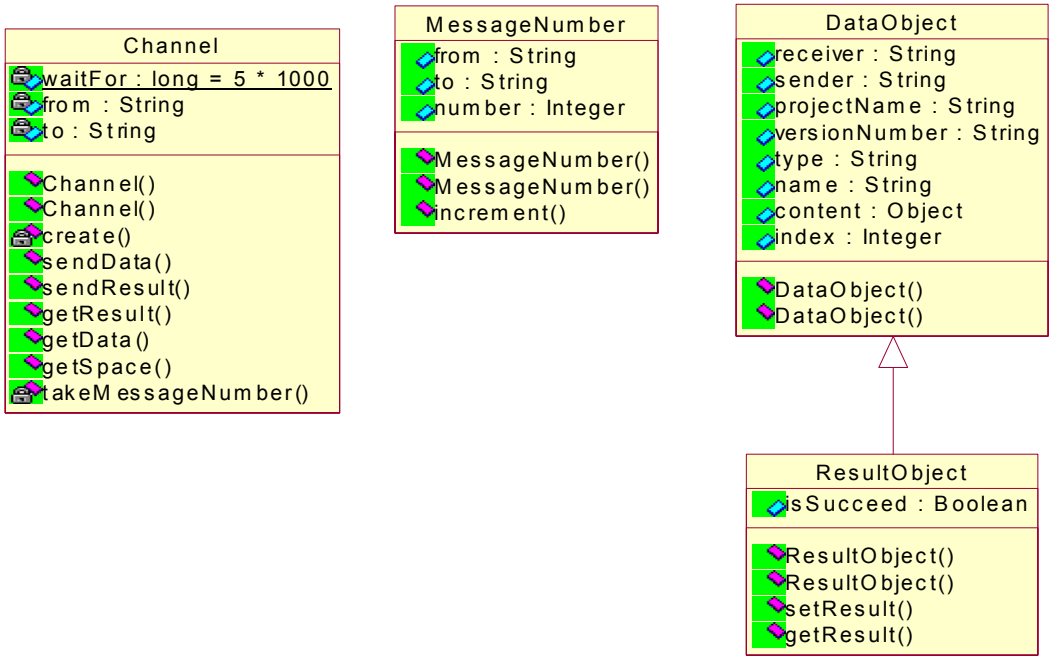

Figure 17: Class Diagram for the communication data structure and protocol

The drawback of the design is that batch process is not allowed. For example, if there is a tool doing a batch job periodically, the tool didn't know which data objects should be read from the space. This scenario does not happen because most software development tools process requests immediately. The solution for handling batch job is to add a type attribute call **`type`** in the **`MessageNumber`** class definition. This indicates the message is a head or a tail so that the tool is able to get all data object from the last data object processed. Modification for methods in the **`Channel`** class is required for supporting head and tail.

4.3 Client application

One of requirement of the client application is to support the dynamic tool plug-in that provided by the infrastructure implemented. Software development tool that can dynamically join and leave to the Jini network is not enough. The client application is last component unit for the whole dynamic tool plug-in infrastructure. It should be able to keep track the status of the Jini network. In order to make the client to notice about tools' join and leave status, classes in Jini's APIs were applied. The APIs provides a set of classes that are responsible for service discovery, that is tool discovery in our framework. Classes that were used in the implementation are **`ServiceDiscoveryManager`**, **`LookupCache`**, and **`ServiceDiscoveryListener`**. **`ServiceDiscoveryListener`** were used to implement the client application so that the client application can listener for the change of join and leave of different tools.

A **`ServiceFinder`** class was defined for simplifying and encapsulating the job of service lookup. Tool providers use instances of the class without having the knowledge of service lookup in Jini. **`ServiceFinder`** has a method for start the **`LookupCache`**, called **`startLookupCache`** (as class diagram shown below). This **`LookupCache`** class existed in the Jini APIs. The responsibilities of this class are to keep track the services lookup and preloading all the services required. When services are added, removed and changed, **`LookupCache`** keep updating its state.

Basically, the main function of the client application is to act as a tool browser. It keeps track the status in the Jini network. When there is any tool plugged or removed, it will update its status as soon as possible, Tool information will be in the client application for the end user to choose their desired tool according to the information provided. Figure 18, below, shows the class diagram of the client tool browser. Demonstration can be referred to appendices.

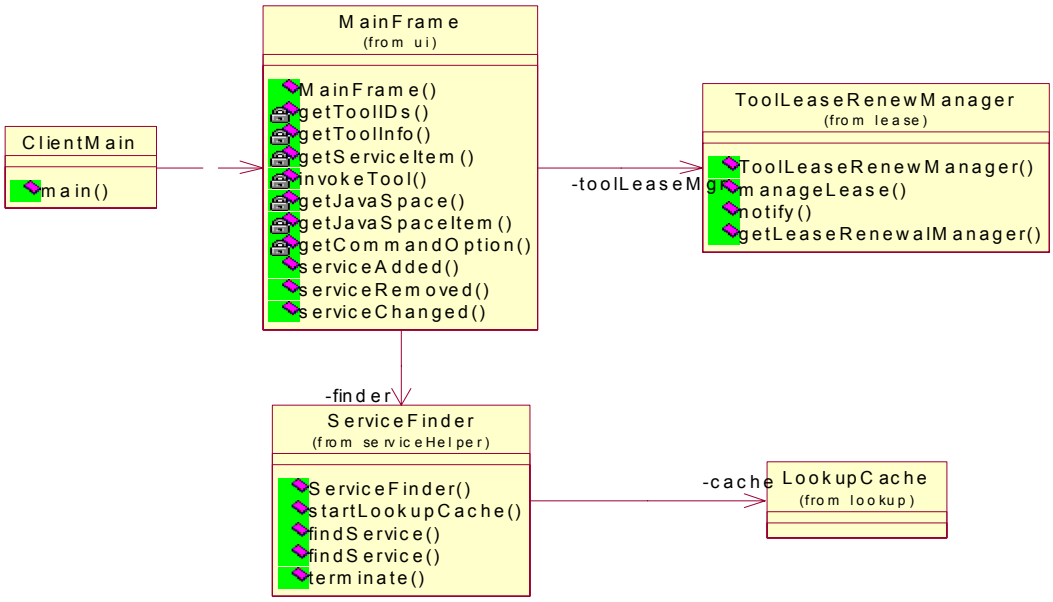

Figure 18: The client tool browser class diagram.

4.4 Startup mechanism

4.4.1 Problem:

It is burden and tedious to write similar code to startup software development tools.

It was found that there was similar procedure to startup process tools and to join to the framework. Tool developers have to implement more or less the same startup coding for their tool to startup.

This involves finding the lookup service, joining process, creating tool information object and command option object. This steps repeat time to time when a tool developer implements several tools and plugs them to Internet-IDEF.

As a result, a start up facilities should be offered to ease the burden of startup process.

4.4.2 Features:

The component helps tool developers to join to Internet-IDEF.
The component also encapsulates the joining process from the developer so that tool developers do not have to write repetition code for joining tool.

4.4.3 Forces:

The startup process should be performed automatically when possible. Single startup processes should adapt for all tools to startup. This implicitly implies that there is no information about the proxy to be created at compilation time.
Since proxy that is needed to create is not known at the compilation time. How can a startup object determine which proxy class is used to create the proxy object at run time is another problem. Extensible property setting was required.

4.4.4 Design Strategy

From the problem described early in this section, the procedure for startup different tool is similar. This context is good fit for applying **Template Method** pattern. **Template Method** pattern was used for providing a template of an algorithm. The pattern was used to implement our startup mechanism.

From the [4], the intent of **Template Method** pattern state that:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Refine the steps in an algorithm without changing the algorithm's structure.

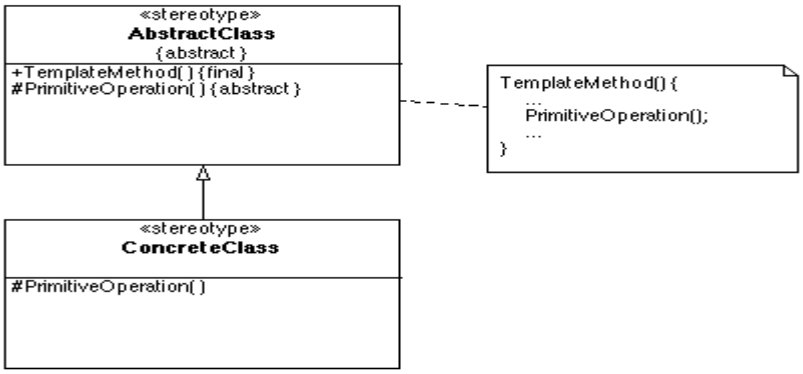

Figure 19: Canonical structure of Template Method Pattern. Extracted from http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/Template-method.html

Diagram above shows the canonical structure of the **Template Method** pattern. Our implementation of template method is vary from the one described at [4]. It is because startup operation is mainly for instantiating the target object and registering it to the lookup service. The template method of the operation was implemented by making use of Reflective API, which is provided by Java, and **Abstract Factory** pattern so that tool provider does not need to subclass the class that consist of template method and the burden for tool developers are lighter. The tool developers are only required to modify specific Property file.

Abstract Factory pattern is applied when families of related or dependent objects are going to be created. It is stated that Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Following is a canonical structure for the **Abstract Factory** pattern.

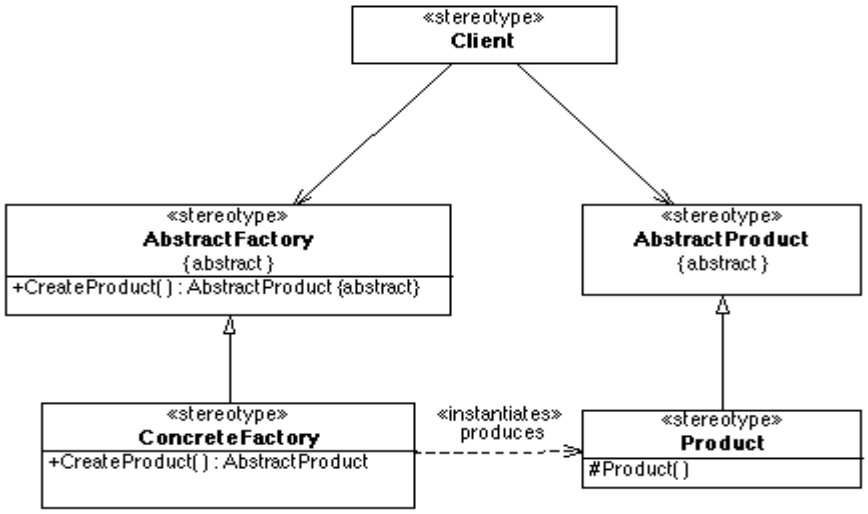

Figure 20: Canonical structure of the Abstract Factory pattern. Extracted from http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/Abstract-factory.html

Our **Abstract Factory**, **`ProxyFactory`**, was used for encapsulating the creation of Jini proxy of different tools. Reflective API is used for dynamic loading of particular proxy factory.

Class definition of **`ProxyFactory`**.

```
public interface ProxyFactory {
    public ToolProxy createProxy(Properties prop);
}
```

A method for dynamic loading of proxy factory and create a **`ToolProxy`** instance by the factory in class **`ToolStartupMain`**.

```
private static ToolProxy getProxyInstance(Properties toolProperties)
                  throws FileNotFoundException, IOException,
                  ClassNotFoundException, NoSuchMethodException,
                  InstantiationException, IllegalAccessException,
                  InvocationTargetException {
    String proxyFactoryName = toolProperties.getProperty("proxyFactory_name");

    //The constructor of concrete ProxyFactory should be a non-arg constructor;
    ProxyFactory factory = (ProxyFactory)
                           (Class.forName(proxyFactoryName)).newInstance();
    ToolProxy tool = factory.createProxy(toolProperties);
    System.out.println("tool is here?" + (tool != null));
    return tool;
}
```

From the sample code above, **`getProxyInstance`** is the implementation of **Template Method** pattern. We can see how **Template Method** pattern is implemented by using java Reflective API. The interface **`ProxyFactory`** is the implementation of **Abstract Factory** pattern. On the other hand, a set of convention was used to startup the tool.

As a result, following mechanism for startup a tool is formed.
Mechanism:

- Startup class first load a property file that contain the name of the proxy factory
- Startup class then instantiates the proxy factory by use of Java Reflective API.
- Startup invokes the factory to create the proxy.
- Startup class instantiates a **`ToolService`** object. **ToolService** constructor will try to join to the lookup service, create backend administration and pass to the proxy. (For remote administration instantiation, please refer to the Tool Administration section.)

A class, called **ToolStartupMain**, was implemented according to the mechanism decribed above. **`ToolStartupMain`** can be view as a builder of **`ToolProxy`** object according to **Builder** design pattern [4]. The diagram below shows the sequence diagram of the startup mechanism.

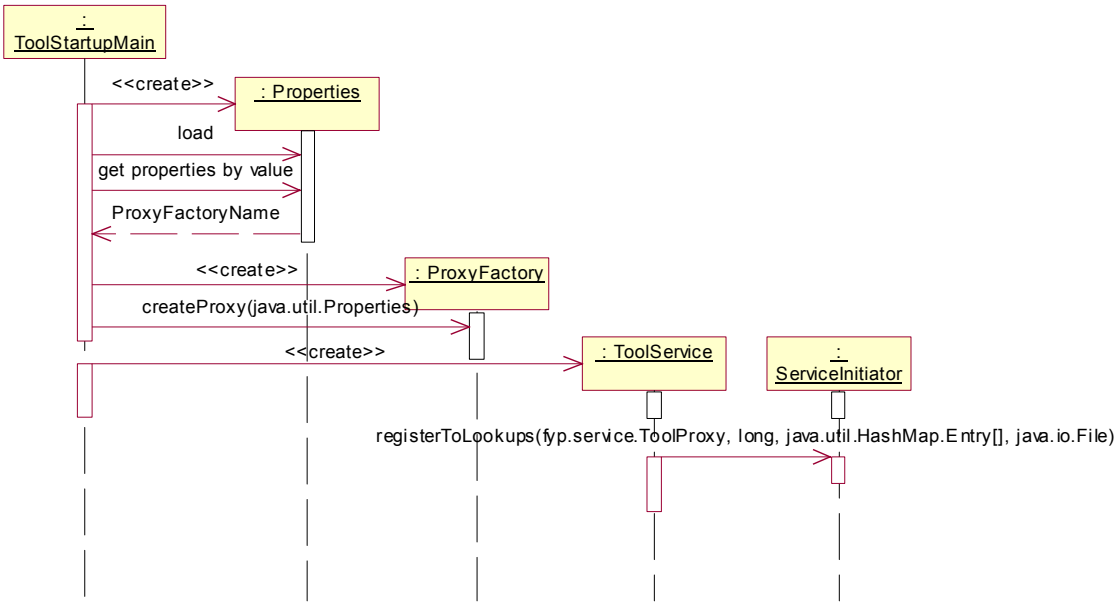

Figure 21: Sequence diagram for the startup mechanism.

Class design:

ProxyFactory – It is the implementation of Abstract Factory pattern and Factory Method pattern. It is used for creating appropriate ToolProxy. Tool developers have to inherit the ProxyFactory to create tool specific tool proxy.

ToolService – It automatically does the following tasks:

1. Founding tool id file location, if any,
2. Registering to be a member of the system, and
3. Obtaining the admin facilities if tool provider has provided this function.

ToolStartUpMain – **Template Method** Pattern and Reflective API was used so that one startup facility for many different tools was allowed. It loads a proxy factory according to the class name provided by the property file setting. Template Method is used to define the proxy creation procedures. That means, it first gets factory class name from the property file, calls a predefined convention of creation of proxy factory and then invokes the factory to create the tool proxy.

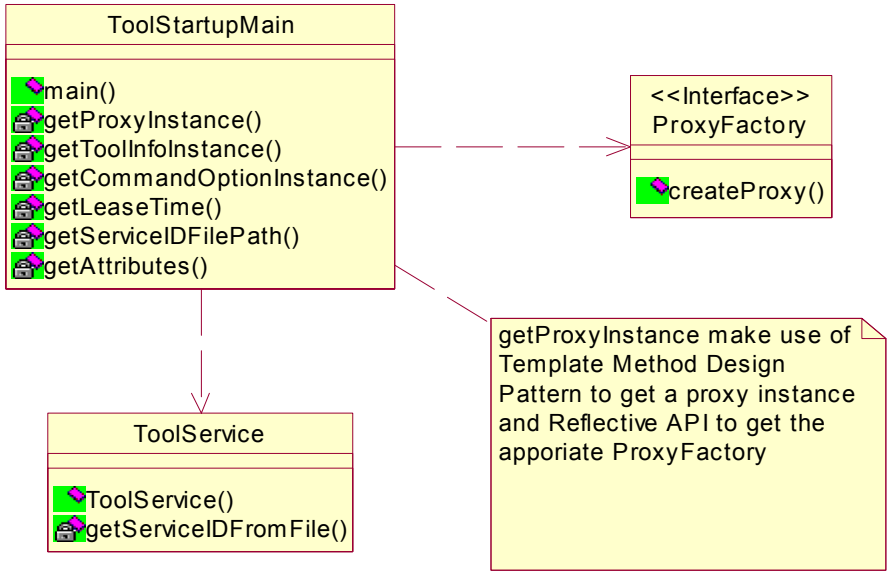

Figure 22: Class Diagram of all related classes in the startup mechanism.

4.4.5 Convention or Property Setting:

There was a set of convention or property setting that tool developers should configure when they are going to use the startup facilities for starting the tool application.

Tool Setting Property File:

Convention

Following is the sample of the tool setting property file –

```
proxyFactory_name=fyp.service.compiler.CompilerProxyFactory
lease-time=600000
toolInfo-filePath=D:\\FYP-InternetIDE\\property\\tool_info.property
ID-filepath=c:\\temp\\compiler.id
commandOption-FilePath= D:\\FYP-InternetIDE\\property\\commandOptions
```

Usage

Tool developers provide the path of tool setting property file. For example, the property file is stored in "c:\tool\ToolSetting.property".

Tool Info Property File:

Convention

Following is a sample setting of the tool info property file –

```
Name=Compiler
manufacturer=Gary
vendor=Gary
version=1.0
model=Null
serialNumber=Null
type=compiler
description="This is a Java Compiler. Thanks for using this tool."
```

Service ID Property File:

It was used to store the service ID of the tool in the framework.

CommandOption Property File:

Each tool has their own argument setting when a command is requested from users. This is especially important for command line application or one-way communication development tools.

Following are the sample of the CommandOption property file –

```
option=source,description=Source File Path,withValue=true
option=destination,description=Destination File Path,withValue=true
```

The convention above is based on the feature of most typical kind of arguments in a command line application. **`option`** is the argument that pass to a command line tool. **`description`** specify the description displayed to the user. **`withValue`** determines specify argument should contain value or not. How those command options should be set and handled depends on a particular tool. On the other hand, this command option property convention can be also used to specify the parameter of any tool when they are initialized.

Remark: There is one requirement for subclasses of **`ToolProxyFactory`** class. All subclasses of **`ToolProxyFactory`** should have a no-arg constructor for **`ToolStartupMain`** class to use the Reflective API to create a **`ToolProxyFactory`** object.

4.5 Tool Administration

4.5.1 Description

It is desirable to have an administration facility to control the tool's configuration in the system. They may want to join to more than one group rather than pre-configure or terminate the tool service from the system.

4.5.2 Features

Basically, a set of functions should be provided to the tool administrator to control the tool.

Jini's administration facilities

Originally, Jini provide a set of interfaces for administrate jini service. It delegates all the administration work to other classes rather build-in in the service.

From the Jini1.2 API documentation, there are following interface:

net.jini.admin.Administrable
net.jini.admin.JoinAdmin
com.sun.jini.admin.DestroyAdmin
com.sun.jini.admin.StorageLocationAdmin

4.5.3 Design Strategy

Proxy pattern was used define the structure of tool administration.

From [4], the intent of the proxy pattern state that,

> *Provide a surrogate or placeholder for another object to control access to it.*

In our case, classes designed was the variant form of the proxy that is known as **Virtual Proxy** pattern or **Remote Proxy** pattern, which is described in [6].

This increases the flexibility for the service developer. Developers can decide their own kind of network protocol for administration. In addition, access control functions can be added to restrict the access right of the tool administration by using Proxy pattern.

There is a backend administration for tool that does all the work of administration job. Then a remote proxy, which is implemented by RMI at this moment, control access the backend administration. In addition, administration job could be done remotely. A remote client application or Java Servlets can be developed to do the administration job.

Static Structure (Class Diagrams):

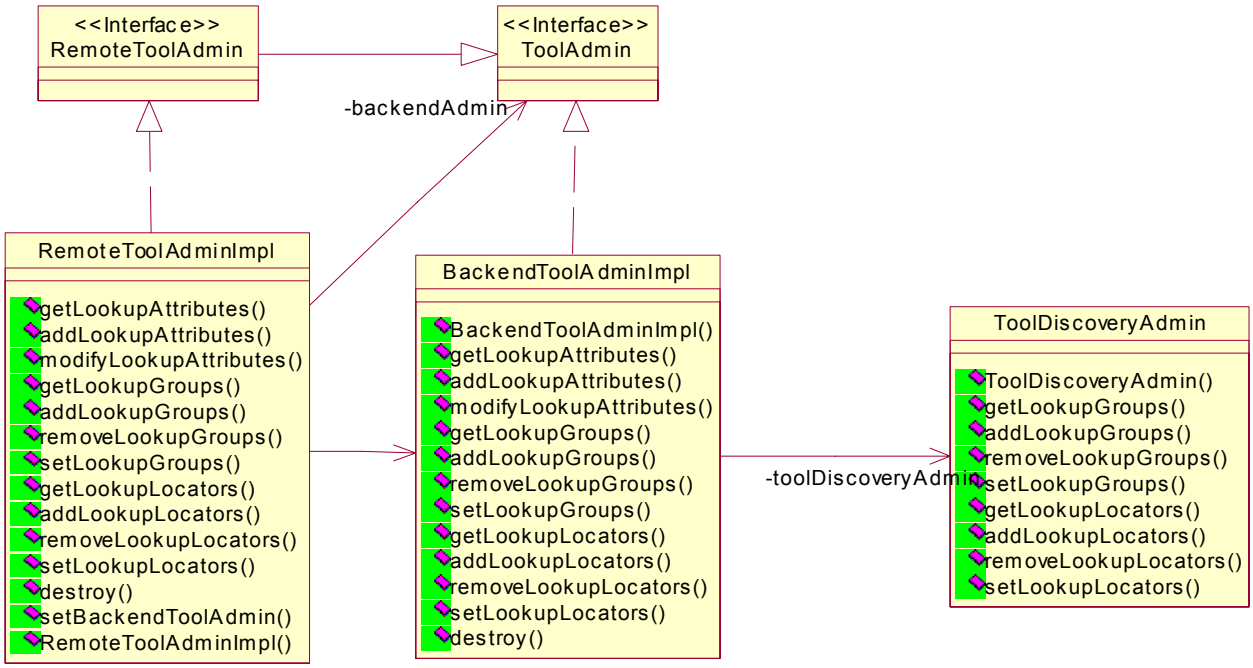

Figure 23: The ToolAdmin class structure

Client of the **`ToolAdmin`** communicate with a representative **`RemoteToolAdminImpl`** rather than to the real administration object **`BackendToolAdminImpl`**.

Dynamic Structure

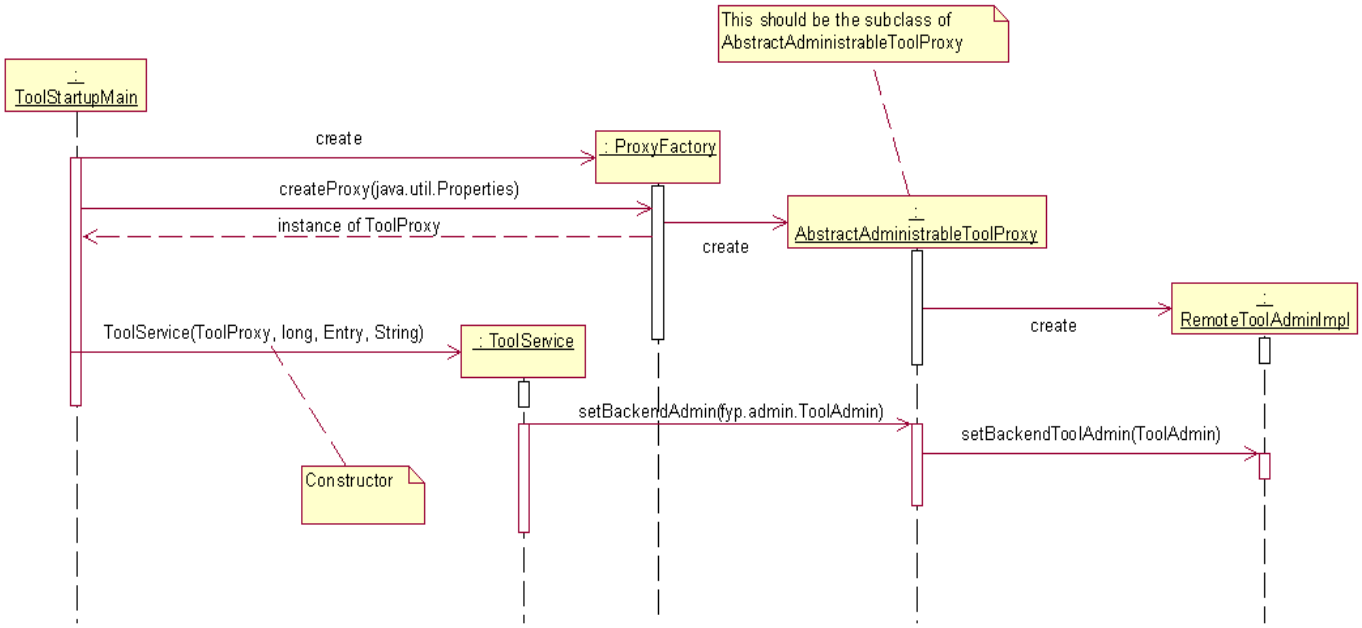

Figure 24: The scenario of instantiating of the remote administration proxy (RemoteToolAdminImpl)

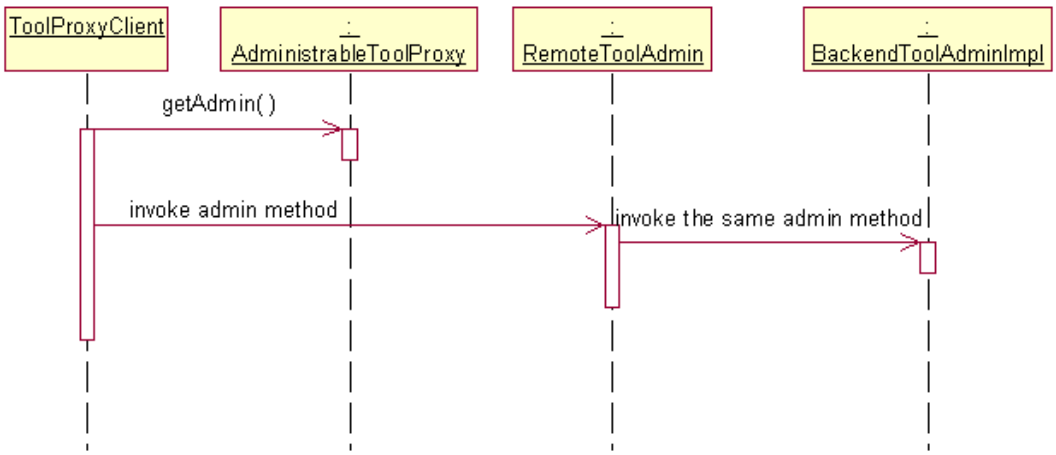

Figure 25: The scenario of getting administration from remote proxy:

4.5.3 Make the ToolProxy class administrable

Jini provides a set of interface and convention for us to do administration job. For those objects who want to obtain an administration object, Jini provides an interface called **`Administrable`**. **`Administrable`** consists of a method called **`getAdmin`**. We should define our proxy with implementing the Administrable object if we want our tool can be managed remotely. On the other hand, according to this convention, administration front-end application could be implemented as similar as the client user application. It means that administration front-end application used the same mechanism to find all administrable tools. Administrator chooses one of the tools and manages the tools. **`ToolProxy`** class should be administrable. Diagram below show the class structure.

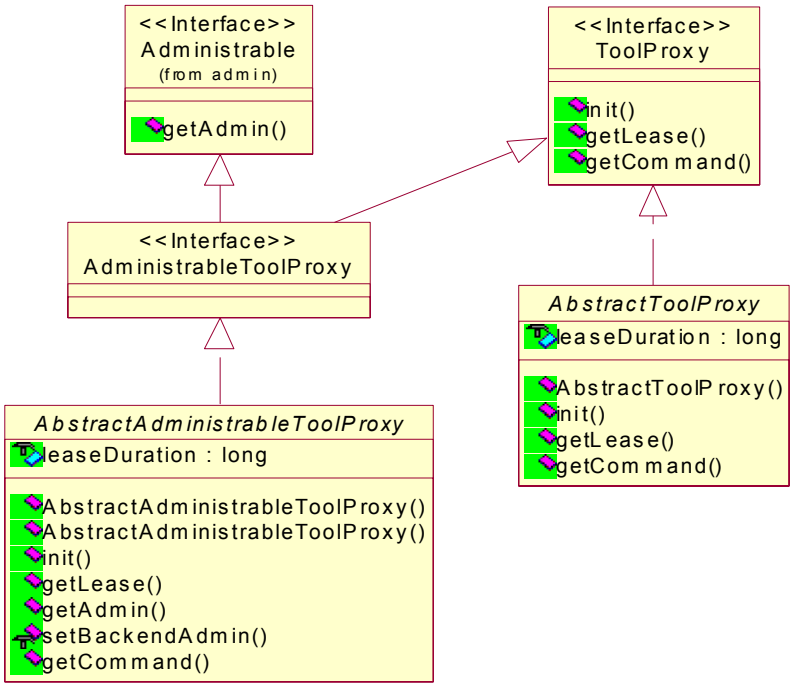

Figure 26: Class diagram to illustrate how ToolProxy becomes administrable.

Chapter 5: Validation of Design

Experience in using the framework is the best way of testing and evaluating the quality of design. A Compiler tool and a collaboration UML editor, which is developed by project member, were used as a testing and validate software development tools of our framework.

We acted ourselves as a software development tool developer to plug a java compiler and a UML editor to the framework. In the rest of section, the characteristic of each tool is described. The approach used to plug them to the framework according to their characteristics is illustrated. This includes the design and the classes implemented.

5.1 Java Compiler

The compiler that was used to implement the remote compiler was obtained from the Java Standard Edition sdk1.3.1 from Sun Microsystem. The compiler is a command line tool call javac.exe. The figure below shows the usage of javac.exe. At the time of writing this report, a simple compilation function was implemented. The remote compile accepts the file to be compiled and the destination of the compiled file, i.e. the ".class" file.

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -O                          Optimize; may hinder debugging or enlarge class
file
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation                 Output source locations where deprecated APIs
are used
  -classpath <path>          Specify where to find user class files
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -d <directory>             Specify where to place generated class files
  -encoding <encoding>       Specify character encoding used by source files
  -target <release>          Generate class files for specific VM version
```

Plug-in approach

The compiler is a command line tool. Based on these characteristics, it was decided that the compiler should be plugged in as a remote complier because of the characteristics of the compiler. Compiler is fitted into this category as we mentioned in the previous section.

Design

In order to plug the compiler as a remote tool, a set of classes from the **`fyp.service`** package was inherited. The super classes are **`ToolProxy`**, **`Command`**, **`ResultHandler`**, **`ToolProxyFactory`** and **`RemoteService`**. The class diagram below shows the relationship of subclass and superclass.

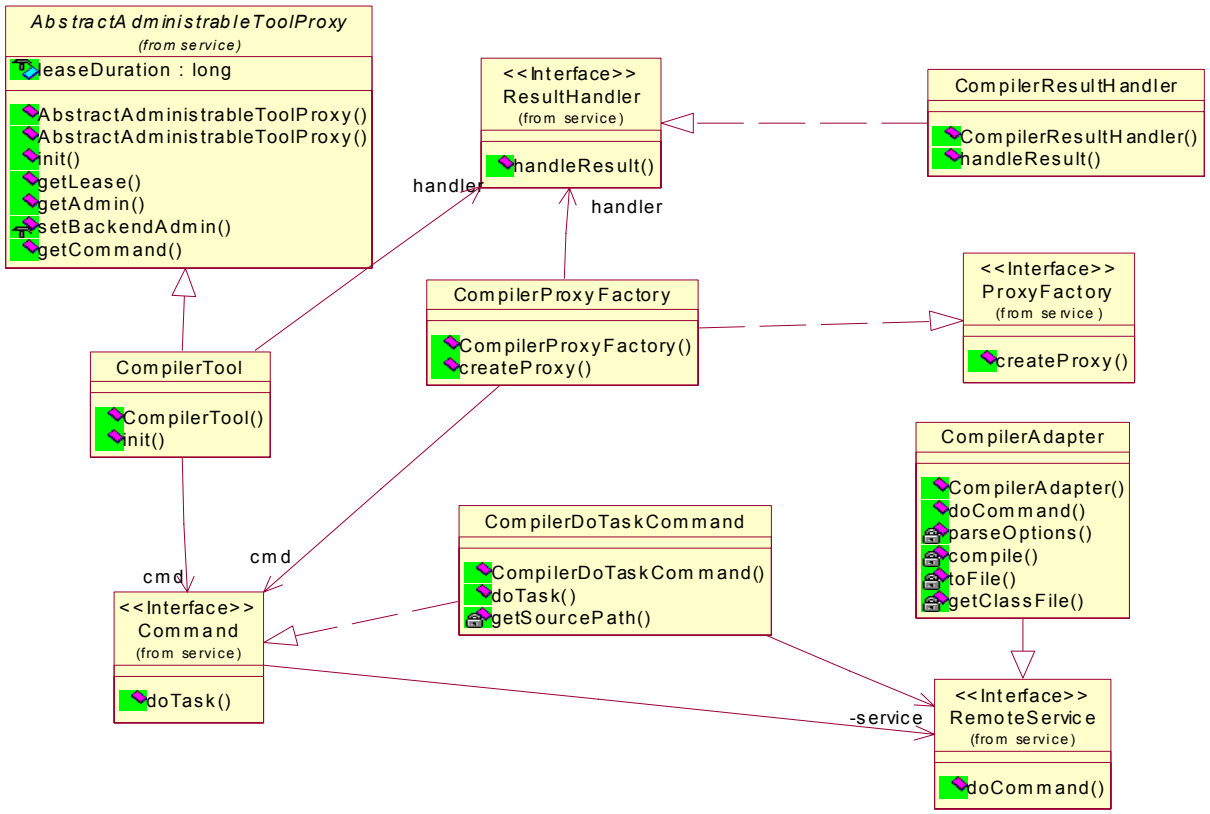

Figure 27: How the remote compiler to be built by use of Internet-IDEF APIs.

From the above diagram, remote compiler developers need to:

1. Extend the any subclass of **`ToolProxy`** that is suitable for the application

2. Extend the **`Command`** class for do all the preprocessing job before invoke an instance of **`RemoteService`**
3. Extend the **`RemoteService`**, which act as an Adapter, for make request to the external application.
4. Extend the **`ResultHandler`** for handling the result after external application finished the job. The instance of **`ResultHandler`** handles the result according to the option setting.
5. Extend the **`ToolProxyFactory`** for create appropriate **`ToolProxy`** instance.

It is easier to understand how the remote compiler works by the explanation of the dynamic behavior and interaction of those classes.

1. Startup

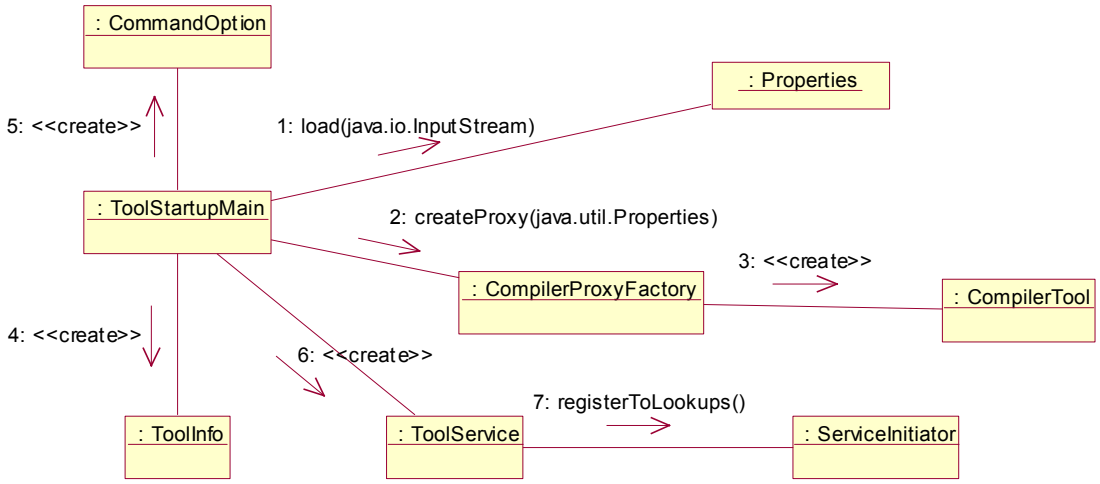

Figure 28: Collaboration diagram of how a remote compiler startup for client to use

The diagram above shows how the remote compiler is setup. Firstly, the **`ToolStartupMain`** object reads the property file path from the command line argument. From the property file, **`ToolStartupMain`** object knows which **`ProxyFactory`** object should be created. In this case, **`CompilerProxyFactory`** object should be created. **`ToolStartupMain`** object will then read the configuration of command options and create a **`CommandOption`** object. In addition, the **`ToolStartupMain`** object will also read the tool information from according to the property file. After that, a **`CompilerTool`** object, which is a subclass of **`ToolProxy,`** was created and registered to the lookup service. After all, whole setup process finished.

2. Handle compilation job

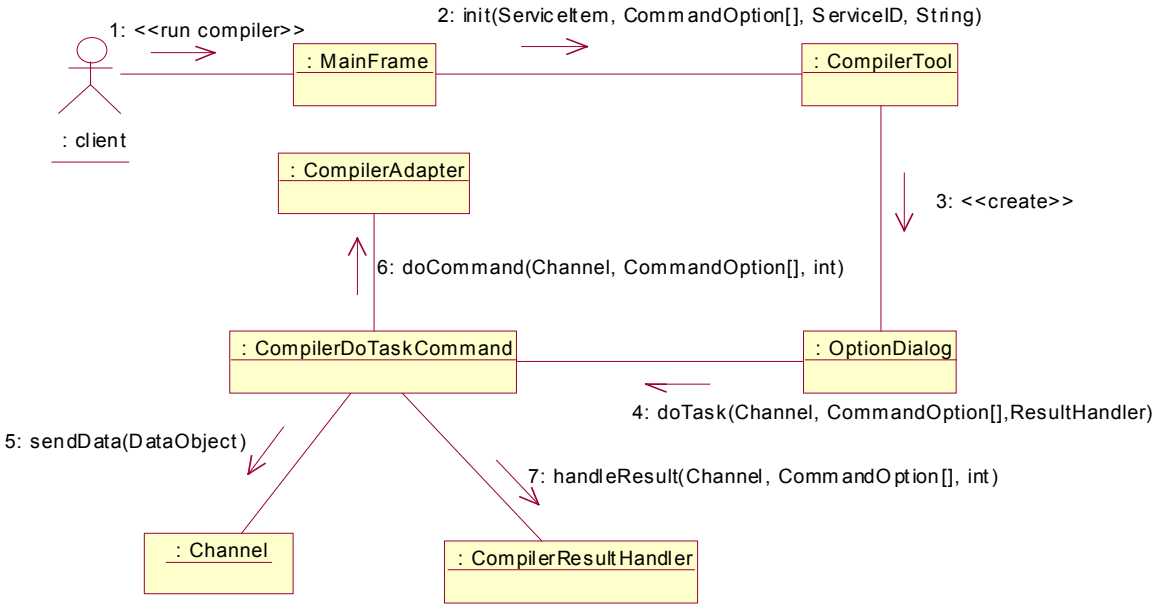

Figure 29: Collaboration diagram of how a client compilation request is handled.

All tools will be downloaded to the client application. After clients choose the compiler and want to use it. They first invoke the downloaded remote compiler proxy by the **`init`** method. After a sequence of configuration, users then run the compiler by invoking the **`doTask`** method, which is defined in the **`Command`** class and implemented in the **`CompilerDoTaskCommand`** subclass. Preprocessing job will be processed before invoking the remote compiler. This includes reading the source java file from the client machine and writing the file to the Javaspace. **`doTask`** method then invokes the **`doCommand`** remote method to compile the file. **`doCommand`** first reads the file back from the Javaspace and then do the compilation job. When the compilation job finished, the result class file is written to the Javaspace and clients will read it back. ResultHandle object then does the post-processing job. After that, whole compilation job finished.

Demonstrations of showing how to compile a Java program by the remote compiler can be referred to Appendix.

5.2 Integration of a collaborative ArgoUML to the framework

Another criterion for passing the test of framework is to integrate a collaborative ArgoUML, which is the project of [19], to the framework. All core APIs in the framework were fully utilized in the integration of the editor, such as API for remote tools, API for communication and API for rich GUI tools. The integration was designed and partially implemented. Following section will discuss about problems was met in the integration and the design approach was used.

Distributed ArgoUML was a collaborated UML editor. It was an interactive application and was a rich GUI application. Communication was another core function for a collaborative application. Versioning engine was implemented for providing collaboration support.

The integration of distributed ArgoUML could be separated into two parts. It was required to integrate 1) versioning engine and 2) ArgoUML. For further information, please refer to [19]. For part 1, the design was similar to the Java remote compiler illustrated in previous section. For part 2, design task was more challenging. The design should be able to ensure that clients can download the whole ArgoUML application and support the communication with other ArgoUML users in the Jini network at the same time. The design shows in the figure below.

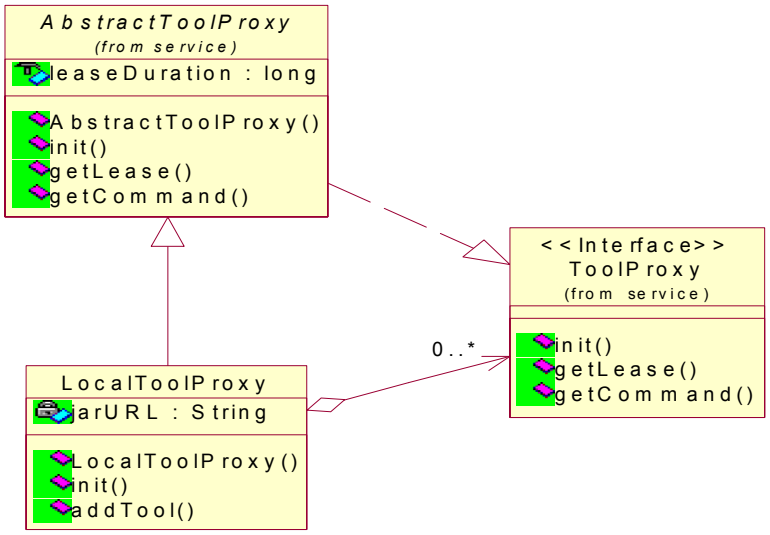

Figure 30: Class structure of how multiple tool proxies composes in one LocalToolProxy.

From the above, it is found that support multiple commands in one tool is required. The design of **`LocalToolProxy`** was modified. There is one method called **`addTool`**.

LocalToolProxy will be responsible for registering all tools that is added to the **LocalToolProxy** object. Method **addTool** should be invoked after a **LocalToolProxy** object has been created. When the **init** method is invoked, all tools added to the **LocalToolProxy** object will be automatically joined to the Lookup Service. On the other hand, when supporting multiple commands for one tool becomes common elements for software tools, the **addTool** method should be moved to **ToolProxy** class, which is the highest level class in the class hierarchy. That is, **Composite** pattern is applied to develop software tool. Sample code below shows how all added tools are registered to Lookup Service.

Sample Code

```
public void init(ServiceItem javaSpaceItem, CommandOption[] options, ServiceID
myServiceID, String clientID) {
   ...
   registerLookup(toolTable);
   ...
}

private void registerLookup(HashMap toolList) {
    Set toolSet = toolList.keySet();
    Iterator toolIterator = toolSet.iterator();
    while ( toolIterator.hasNext() ) {
        try {
            ToolProxy tool = (ToolProxy)toolIterator.next();
            Entry[] attributes = (Entry[])toolList.get(tool);
            ServiceInitiator initiator = new ServiceInitiator();
            File idFile = null;
            initiator.registerToLookups(tool,     Lease.FOREVER,     attributes,
idFile);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Chapter 6: Result and Discussion

6.1 Result:

A set of APIs was implemented. They aim at helping tool developers to plug their development tools into the framework. A compiler and a UML editor were used to validate the design of framework. It was found that the integration of the Java compiler was successful and run properly. The design for the integration of UML editor was finished, as shown in the previous chapter.

Following is a list of components that were already implemented:

- Dynamic Plug-in API for remote service and jar file tools
- **`ToolProxy`** interface
- Service Utilities
- Communication Facility
- Tool Administration
- Client tools browser
- Tool Startup Facility

6.2 Discussion:

From the experience in integration of two tools, it was found compatibility of command line tools into a distributed environment is relatively good. They can be existed without too much collaboration with other tools. It was also found that the **`ToolProxy`** class is relatively thin when the collaborative UML editor, which did several backend collaborative tasks and versioning task other than just provide a standalone application to the client user, was integrated. It was sufficient for standalone tools. But it may become a restriction for tool developers in designing how to integrate tools with high collaboration of tools. Improvement was taken place and **`addTool`** method was added to **`LocalToolProxy`** class so that **`LocalToolProxy`** object can compose a set of tool as a whole.

Alterative designs and design ideas of existing technologies that are appropriate for the problem domain would be necessary. Although the ideas of some existing Java technology, such as applet, Servlet and MIDlet, were borrowed, the core notions or mechanisms behind are not fully understood. These three technologies are similar in nature in providing an interface abstraction for application developers for implementation according to the requirement for client users or client objects. The ideas and design about

how to control the Servlet and MIDlet by internal component such as Servlet container for Servlet and application manager for MIDlet would be useful for improvement of the framework. Those technologies were not investigated because of time limitation.

From all above, enhancing the ability of the client application were needed so that it can support more advancing **`ToolProxy`** interface. If this is the case, refinement of ToolProxy may be necessary.

Chapter 7: Difficulties and Further Enhancements

7.1 Difficulties

Learn and Grasp Jini Concept, infrastructure and programming model

Jini network technology proposes a new view of computer system. In the Jini network environment, everything is either a service or a client. It is called service-based architecture.

Based on this concept, Jini network technology provides the infrastructure and programming model. The concept behind the infrastructure and the programming model are relatively simple to understand. It is difficult to get use to the Jini programming model in a short time period although the basic concept of Jini networking is relatively simply to understand.

On the other hand, another difficulty was the service's to incorporate business logic part with Jini programming model in framework design. It is because Jini often provides interfaces and convention for particular function such as leasing, distributed event and transaction. The semantic behind is depended on the application domain and should be implemented by application developer. Which means that developers have to design the service following the interfaces and convention.

Apply suitable design pattern in the problem in Jini technology context

Since the framework was a middle layer between software development tool and Jini network environment, the framework not only had the responsibility to encapsulate the detail of Jini so that tool developers can focus on implementing functions software development, but also it had the responsibility to provide domain specific functionality that is common for most of software development tools. This makes design and implementation of the framework much harder.

Although applying design patterns provide a way for developers to have a better design of software applications, using them to design Jini service application is a new challenge. It is necessary to fully understand the context and the problem the pattern that is going to solve and then choose suitable pattern. After that, it is necessary to think about how to apply this pattern into Jini technology context. It can be a difficult to judge the pattern applied in this new context is appropriate or not for solving the problem.

Framework always needs evolvement. It may take over years.

Framework development is an evolution process. The design of framework improves when experience gains. Because familiarity in the domain of Integrated Development Environment and software tool development is little, designing the framework is difficult at the beginning.

7.2 Further Enhancements

Refine and evolve the framework

Several components were not yet implemented and designed. They were development project resource management and user management components. In addition, basic functionalities were provided in the tool administration component at this stage. Additional functions, such as property setting management, were required but not yet implemented. Further enhancements also include collaborative software development support.

On the other hand, design of the framework decay during implementation. Refinement and refactoring are required to improve the design. At the time beings, only a set of classical design patterns was applied. Other patterns for distributed computing [15] could be considered to improve the design.

Communication performance should be taken into consideration when the framework evolves. As mentioned in Chapter 4 section 4.2.3, JavaSpaces communication may be the bottleneck of the system. Clustering should be the solution to solve this problem.

During implementation of the framework, it was found that it is worth to develop a shell tool for accessing remotes command line application in the project. The shell tool provides a consistent and familiar user environment for developers to use command line tools.

Try to incorporate Jini with other technologies to the framework

Other technologies such as XML and Java Bean concepts may be useful in improving the functionality and power of the framework. It is required to investigate degrees of enhancement achieved when incorporating those technologies with Jini network technology is applied.

Conclusion

An integrated development environment framework, Internet-IDEF, was developed in this project. The framework aimed at providing a platform for tool providers to plug their tool into the environment. Tool providers are only required focusing on the functionalities of the tool they provided. They do not need to take care about common software development features. The environment is also beneficial to software developers because all tools can be accessed and used when those tools are available in the Jini network. Manually installation is not required. Productivity increases.

With Jini technology, client applications or the client tools do not need to deal with the issue about communication. It is because the whole communication protocol and mechanism are embedded in the Jini proxy that provides the service. Tool developers handle communication issue. The client applications do not need to care about the implementation of service communication. Flexibility and extensibility were increased because client applications do not required to compromise with the service application for communication. That means modification of client code is not required for supporting all communication mechanics.

Internet-IDEF provides an environment for software tools to integrate to the platform dynamically although compatibility for varies kind of tools still needs to investigate. The degree of compatibility or extensibility of the platform can be measured as more and more software tool integrated to the framework.

The most challenging part of this project is to identify appropriate functions or components that should be included in the framework. The design should not be too general for much kind of applications in different domains and should not be too specific to particular kind of tools. The design should meet the requirements in the domain of software development tool. Supporting functions and components should be general enough and useful for software tool development. Design Patterns help to have a better design.

At the time beings, the framework was premature because there were two applications that were used to test the framework. It is believed that a series of refinement and iterative is required when more tools plugged into the framework for validation. As a result, the framework evolves.

Reference

[1] W. Keith Edwards (2001), *Core Jini 2nd Edition*, Sun MicroSystems Press Java(tm) Series.

[2] Scott Oaks, Henry Wong, *Jini in a Nutshell: a Desktop Quick Reference*, O'Reilly.

[3] Vayssiere, J., *Transparent dissemination of adapters in Jini*, Proceedings. 3rd International Symposium on Distributed Objects and Applications, 2001, Page(s): 95 – 104.

[4] Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[5] Sun MircoSystems (2001). *Jini(tm) Technology Datasheet*. Available online: http://wwwwswest.sun.com/jini/whitepapers/jini-datasheet0601.pdf

[6] Frank Buschmann, et al. *Pattern-oriented software architecture: a system of patterns*, Chichester, England: Wiley, 1996.

[7] Eric Freeman, Susanne Hupfer, Ken Arnold, *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley, 1999

[8] Linda Rising, *Design Patterns: elements of Reusable Architectures*, Annual Review of Communications, Vol.49, 1996, pp.907-909. Available online: http://www.agcs.com/supportv2/techpapers/patterns/papers/patterns.htm

[9] Mary Campione, Kathy Walrath, Alison Huml, Tutorial Team, *The Java(TM) Tutorial Continued: The Rest of the JDK(TM)*, Addison-Wesley, 1998. Avalibile online: http://java.sun.com/docs/books/tutorial

[10] An official web site of Eclipse Integrated Development Environment: http://www.eclipse.org

[11] Object Technology International, Inc., *Eclipse Platform Technical Overview*, 2001. Available online: http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[12] The Netbeans IDE. *Netbeans IDE Java Doc - Modules API*. Avaliable online: http://www.netbeans.org/project/www/download/apis/org/openide/modules/doc-files/api.html#how-manifest

[13] Sun Mircosystem, *The NetBeans™ API-A Developer's Guide*. Available online: http://www.sun.com/forte/ffj/whitepapers/netbeansapiwp.pdf

[14] Jim Waldo, Ken Arnold (Editor), The Jini Team, *The Jini(TM) Specifications, 2nd Edition*, Boston, Mass.: Addison-Wesley, c2001

[15] Douglas Schmidt, et al. *Pattern-oriented software architecture:* Patterns for concurrent

and networked objects**,** Chichester; New York, N.Y.: John Wiley & Sons, 2000

[16] James O. Coplien, *Software Patterns,* SIGS Books, New York, New York, 1996. Available at Internet: *http://www.bell-labs.com/user/cope/Patterns/WhitePaper/SoftwarePatterns.pdf*

[17] Sun Mircosystem, *JavaSpaces(TM) Service Specification*, pages 1 - 9. Available online at: http://java.sun.com/products/javaspaces/index.html

[18] An official web site of Netbeans IDE: http://www.netbeans.org/

[19] Yim Wai Hin, *A Framework for an agent-based development environment with Jini/Javaspaces (Real-time collaboration support)*, The Hong Kong Polytechnic University, Final Year Report, 2002.

[20] Jan Newmarch, *Jan Newmarch's Guide to JINI Technologies*, 2001. Available online at: http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml

[21] An official web site of GigaSpaces: http://www.gigaspaces.com

[22] Web site of Jtop Project: http://developer.jini.org/exchange/projects/jtop/

Appendices

Appendix A: Demonstration of compiling a Java program by the remote compiler

1. Client tool browser discovers that there is a compiler in the system. The user wants to run it.

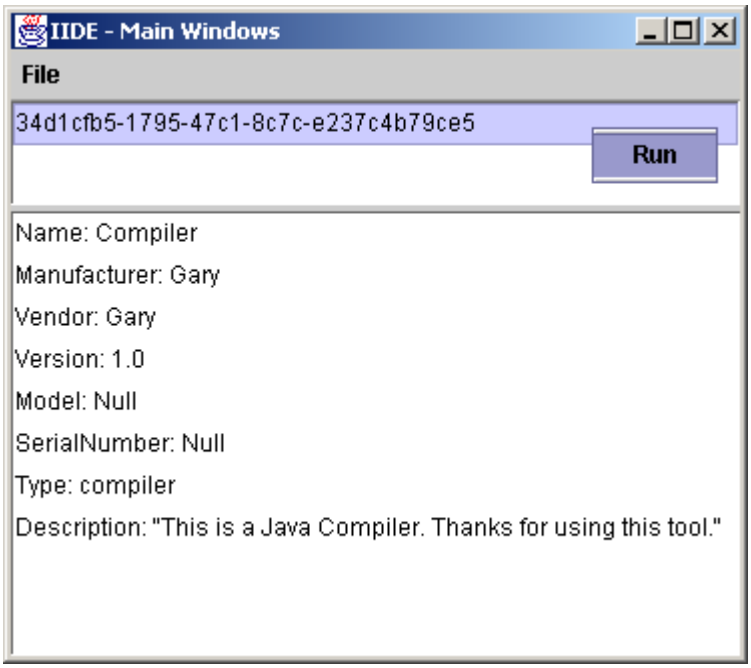

Figure 31: Client Tool Brower

2. A compiler GUI is displayed and user input the parameter needed to compiler the file. User wants to compile a file called FYPTestHelloWorld.java to E drive in the client machine.

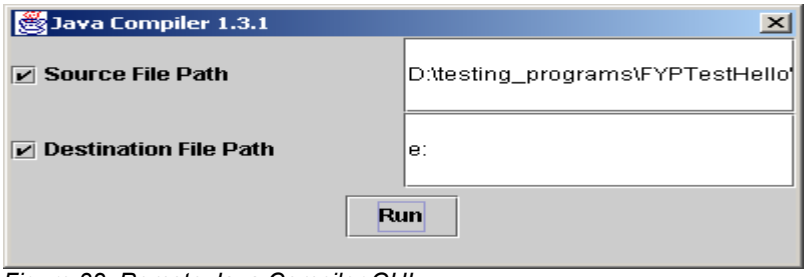

Figure 32: Remote Java Compiler GUI

3. Here is internal process done when a user request to compile the java file by clicking Run command.

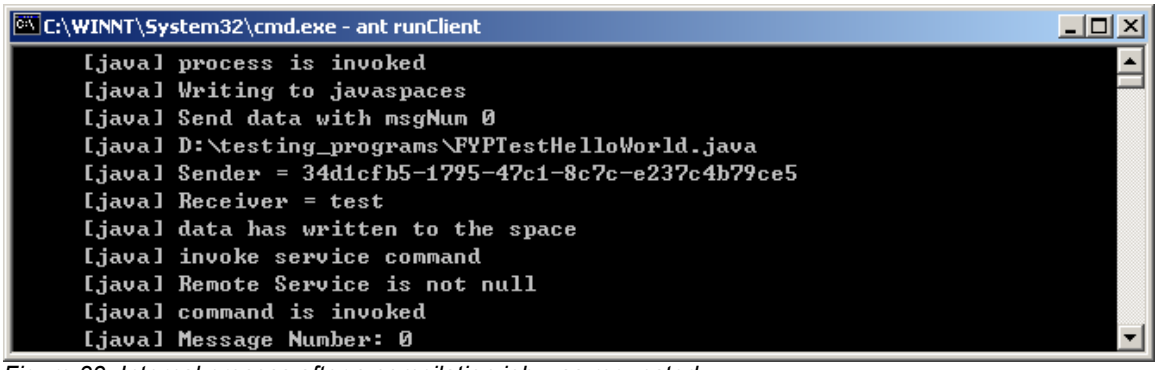

Figure 33: Internal process after a compilation job was requested

A user requested to compile a java file. The Java file was then written into the JavaSpaces and the user invoke the remote compiler to do the compilation job.

4. Here is the backend message in the remote compiler.

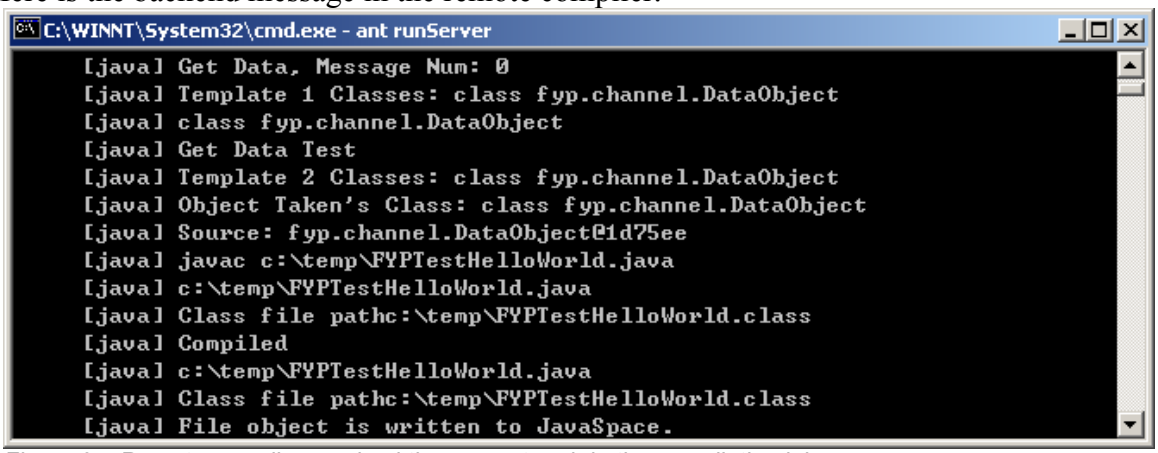

Figure 34: Remote compiler received the request and do the compilation job

Remote Compiler got the java file from the JavaSpaces and put it to a temporary directory. The Java file was compiled and the result class file was written back to the JavaSpaces.

5. Use got back the result class file and stored into E drive.

Figure 35: Client user got back the result class file in the desired directory

Appendix B: Demonstration of running a collaborative ArgoUML from Internet-IDEF

1. Client tool browser discovers that there is an ArgoUML editor in the system. The user wants to run it to draw UML design diagram.

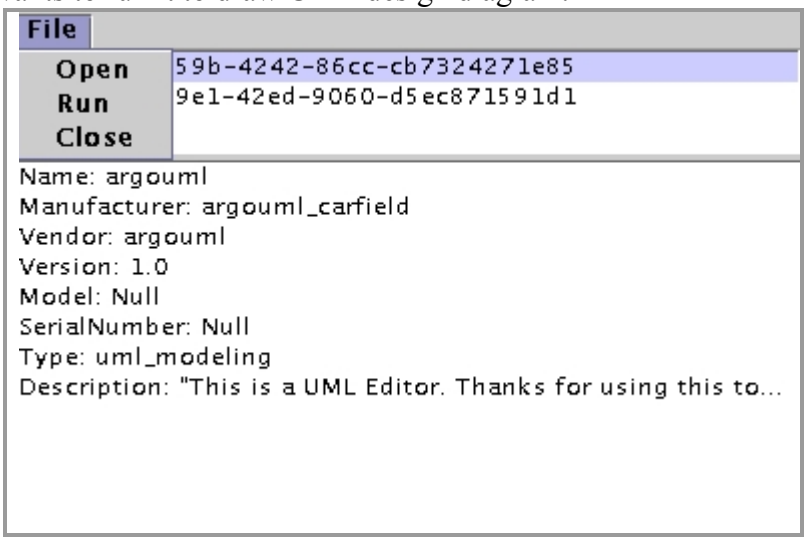

Figure 36: Client Tool Brower (A ArgoUML editor was chosen to run)

2. The ArgoUML editor requested by the user is running.

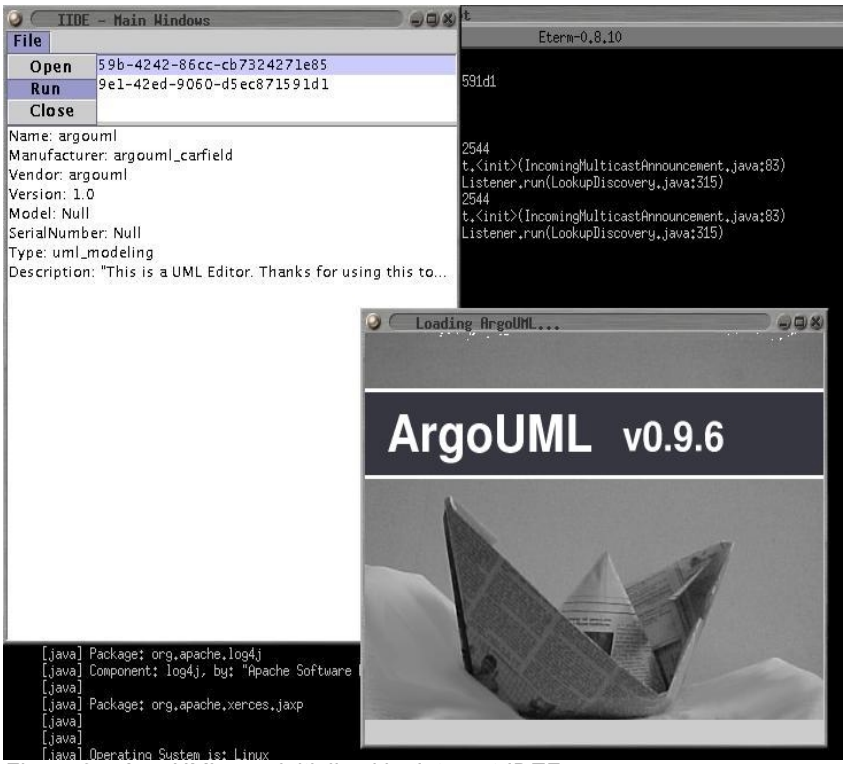

Figure 37: ArgoUML was initialized by Internet-IDEF

3. ArgoUML is initialized and is run successfully from Internet-IDEF.

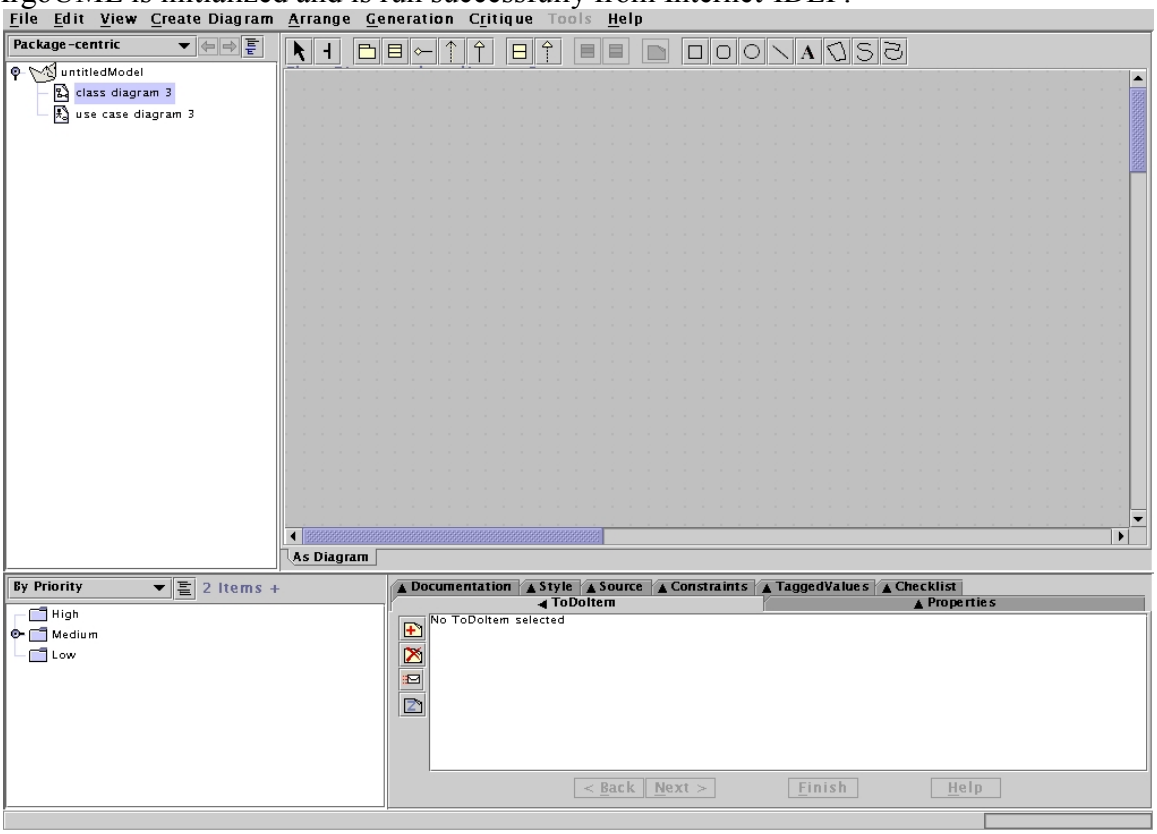

Figure 38: ArgoUml is initialized and run successfully.