

國立交通大學

資訊工程系

博士論文

GJMA - 一個泛用的 Java 行動應用程式開發平台

GJMA - A Generic Java Mobile Application Development Framework

研究生：鄭明俊

指導教授：袁賢銘 教授

中華民國九十六年六月

GJMA - 一個泛用的 Java 行動應用程式開發平台

GJMA – A Generic Java Mobile Application Development Framework


研究生：鄭明俊

Student : Ming-Chun Cheng

指導教授：袁賢銘

Advisor : Shyan-Ming Yuan

國立交通大學
資訊工程系
博士論文



A Dissertation
Submitted to Department of Computer Science
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

G J M A - 一個泛用的 J a v a 行動應用程式開發平台

學生：鄭明俊

指導教授：袁賢銘

國立交通大學資訊工程系(研究所)博士班

摘 要

使用行動裝置與無線網路的人愈來愈多，行動應用程式的需求也日益增強，但是行動裝置之間有著很大的差異，且無線網路並不穩定，裝置的差異與網路的不穩定讓開發行動應用程式變得更加困難，開發者必須面對並花費大量的時間來解決這些問題。雖然有許多的研究試圖解決這些問題，像是使用者介面調適，程式語言轉換等等，但是大多數的研究並沒有將行動裝置的計算能力與功能考慮進去，造成這些行動裝置上的資源被忽略或浪費，為了解決這個問題，本篇論文提出一個泛用的 Java 行動應用程式開發平台，稱為 GJMA，它共支援三種運算模式，分別為 BROWSER，STANDALNONE 與 MASTER-SLAVE，GJMA 可以根據行動裝置的使用者介面，計算能力與功能來選擇程式要在哪種模式下運行，使得程式可以被大部分的裝置所存取使用。換句話說，在 GJMA 上開發程式時並不需要考慮該程式要使用何種運算模式，也不需要考慮行動裝置的計算能力與使用者介面為何，所有需要的轉換都是在佈署到行動裝置上時由 GJMA 來自動完成，也就是說，寫一次程式，就可以讓不同的裝置來存取使用。在這篇論文中，有三個調適的機制將被介紹，分別為運算模式的調適機制，使用者介面的調適機制與通訊協定的調適機制。

GJMA – A Generic Java Mobile Application Development Framework

Student: Ming-Chun Cheng

Advisors: Dr. Shyan-Ming Yuan

Department of Computer Science
National Chiao Tung University

ABSTRACT

Although wireless networks and mobile devices have become popular, the diversity of mobile devices and unsteadiness of wireless networks still cause software development much trouble. Mobile application developers are forced to confront these problems, and therefore spend a lot of time developing mobile applications. Although many studies on user interface adaptation and language transformation have attempted to solve the problem, most of them do not consider the computing power and functionalities of end-devices. As a result, these resources are ignored or wasted. To overcome these problems, the author proposes a generic Java mobile application development framework, named GJMA, to help developers build Java mobile applications quickly and easily. The GJMA framework can tailor an application to fit different devices according to user interface formats and the computing power and functionalities of the devices. Every application developed by GJMA can run in one of three computing modes: thin-client computing, distributed computing and fat-client computing. As a result, a mobile application developed on GJMA can enjoy the “write once, run everywhere” benefit. In addition, three adaptation mechanisms are introduced in this dissertation: computing model adaptation, user interface adaptation and network adaptation.

誌 謝

本篇博士論文的完成必須感謝許多人。首先感謝我的指導教授袁賢銘博士，謝謝老師多年來的指導，並感謝老師在我的求學過程中給予充分的發揮空間，讓我可以作不同的嘗試，學習到不同的知識並獲得各種寶貴的經驗。接著要感謝孫春在教授、陳俊穎教授以及曹孝櫟教授，舉凡論文的架構、所採用的技術、投影片的編排都給予我許多的指導。在此要特別感謝所有的校外口試委員，曾黎明教授、楊竹星教授、鄭憲宗教授與張玉山教授，在百忙之中給予我寶貴的建議，讓我可以讓本篇論文更加的完善。

此外我也要感謝實驗室的各位學長，張玉山、梁凱智、何敏煌、焦信達、葉秉哲、許瑞愷、劉旨峰、蕭存喻、林獻堂，給予我許多的指導，在我遇到瓶頸時，指引我一條可行之路。這邊要特別感謝葉秉哲、邱繼弘、吳瑞祥，我們互相打氣，一起參加比賽，一起作研究，一起玩樂，一起寫論文。要不是你們，我的博士班生涯不會多采多姿，也不會充滿著歡笑。除了學長同學外也要感謝實驗室的各位學弟妹給予我的各種幫助，讓我們得以完成各種大小的計畫，藉此機會謝謝所有對實驗室付出心力的人。另外也要感謝系辦的楊秀琴小姐、俞美珠小姐、陳小翠小姐給予我的所有協助。

最後將此論文獻給我的父母與一直鼓勵我支持我的老婆卓宜青，感謝你們提供我如此好的學習機會與環境。求學的過程中，需要感謝的人實在太多，希望在未來可以貢獻所學於社會上。

TABLE OF CONTENTS

CHINESE ABSTRACT.....	i
ENGLISH ABSTRACT.....	ii
ACKNOWLEDGEMENT.....	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES.....	viii
LIST OF TABLES	x
LIST OF LISTINGS.....	xi
Chapter 1 Introduction	- 1 -
1.1. Motivation.....	- 1 -
1.2. Objectives.....	- 2 -
1.3. Organization.....	- 3 -
Chapter 2 Background	- 5 -
2.1. Related Specifications.....	- 5 -
2.1.1. Thin-Client Computing.....	- 5 -
2.1.1.1. Wireless Markup Language.....	- 6 -
2.1.1.2. Compact HyperText Markup Language	- 6 -
2.1.1.3. Extensible HyperText Markup Language.....	- 7 -
2.1.1.4. Markup Language Transform	- 8 -
2.1.2. Fat-Client Computing.....	- 8 -
2.1.2.1. Java ME.....	- 9 -
2.1.2.2. BREW.....	- 11 -
2.1.2.3. Symbian.....	- 11 -
2.1.2.4. .NET Compact Framework.....	- 11 -
2.1.3. Distributed Computing.....	- 12 -

2.2.	Related Works.....	- 12 -
2.2.1.	VNC	- 12 -
2.2.2.	TCPTTE	- 13 -
2.2.3.	J2ME Polish.....	- 13 -
2.2.4.	Nano-X.....	- 14 -
2.2.5.	ART.....	- 14 -
2.3.	Java Class File Structure	- 14 -
Chapter 3	GJMA Development Framework Overview	- 16 -
3.1.	GJMA Concepts.....	- 16 -
3.2.	GJMA System Entities.....	- 18 -
3.2.1.	End-device (GJMClient)	- 18 -
3.2.2.	GJMAApp.....	- 20 -
3.2.3.	GJMServer	- 22 -
3.3.	Three Running Modes in GJMA.....	- 22 -
3.3.1.	BROWSER Mode.....	- 24 -
3.3.2.	STANDALONE Mode.....	- 26 -
3.3.3.	MASTER-SLAVE Mode.....	- 26 -
3.3.4.	Three Running Modes Comparison	- 27 -
Chapter 4	GJMA Design Issues.....	- 28 -
4.1.	GJMServer Architecture	- 28 -
4.1.1.	Adaptive Transport Layer	- 28 -
4.1.2.	Message Routing Layer.....	- 30 -
4.1.3.	Application Runtime Layer.....	- 31 -
4.2.	GJMClient Architecture	- 32 -
4.3.	Initialization Process within GJMAApp.....	- 33 -
4.3.1.	GJMAApp in GJMServer	- 33 -
4.3.2.	GJMAApp in GJMClient	- 34 -

4.4.	Computing Model Adaptation Mechanism	- 35 -
4.4.1.	Adapt to the STANDALONE Mode	- 35 -
4.4.2.	Adapt to the BROWSER Mode	- 36 -
4.4.3.	Adapt to the MASTER-SLAVE Mode	- 36 -
4.4.3.1.	How to Intercept Invocation Actions	- 39 -
4.4.3.2.	How to Reflect Intercepted Actions	- 43 -
4.4.3.3.	How to Create Complementary Objects	- 44 -
4.4.4.	Deployment Process	- 50 -
4.5.	User Interface Adaptation Mechanism	- 53 -
4.6.	Network Adaptation Mechanism	- 55 -
Chapter 5	GJMA Implementation Issues	- 56 -
5.1.	GJMClassLoader	- 56 -
5.2.	GJMAMesg Format	- 59 -
5.2.1.	GJMAMesg for System Use	- 60 -
5.2.2.	GJMAMesg for BROWSER Mode	- 61 -
5.2.3.	GJMAMesg for MASTER-SLAVE Mode	- 61 -
5.3.	Marshalling and Unmarshalling	- 62 -
5.3.1.	Marshalling	- 63 -
5.3.2.	Unmarshalling	- 66 -
5.4.	GJMA Preprocessor	- 67 -
5.4.1.	Generate Wrapper Class	- 68 -
5.4.2.	Convert to Method Invocation Actions	- 74 -
5.4.2.1.	Filed Manipulation Action	- 74 -
5.4.2.2.	Synchronized Action	- 75 -
5.4.3.	Generate Code for Creating Complementary Objects	- 76 -
5.4.4.	Convert Array Type to Class Type	- 77 -
5.4.5.	Insert Code for Intercepting Instance Creation	- 78 -

5.5.	GJMA Analyzer	- 81 -
5.5.1.	Generate Proxy Class	- 81 -
5.5.2.	Generate ObjMngr Class.....	- 82 -
5.5.2.1.	Generate the create Method.....	- 83 -
5.5.2.2.	Generate the invoke Method.....	- 84 -
Chapter 6	Evaluation	- 85 -
6.1.	Programming Framework Comparison.....	- 85 -
6.1.1.	MPI Programming Framework	- 85 -
6.1.2.	Java RMI Programming Framework.....	- 86 -
6.1.3.	GJMA Programming Framework	- 87 -
6.1.3.1.	Hello World	- 87 -
6.1.3.2.	Web Services	- 88 -
6.2.	Performance Evaluation	- 91 -
6.3.	Program Size Evaluation.....	- 93 -
Chapter 7	Conclusion and Future Works.....	- 95 -
References.....		- 97 -

LIST OF FIGURES

Figure 1-1: Computing power requirements for client and server	- 3 -
Figure 2-1: Java platforms for different purposes	- 9 -
Figure 2-2: A VNC2Go screenshot.....	- 13 -
Figure 2-3: J2ME Polish screenshots	- 14 -
Figure 2-4: Java class file structure	- 15 -
Figure 3-1: Three-tier architecture used in GJMA	- 17 -
Figure 3-2: The decision tree for the class org.gjma.application.GJMAApp.....	- 20 -
Figure 3-3: A GJMAApp deployed to different running modes or devices.	- 23 -
Figure 3-4: A diagram for the BROWSER mode.....	- 24 -
Figure 3-5: A screenshot of the GJMA task manager	- 26 -
Figure 3-6: A diagram for the MASTER-SLAVE mode.....	- 27 -
Figure 4-1: The layered architecture of GJMAServer.....	- 28 -
Figure 4-2: The detailed architecture of GJMAServer.....	- 30 -
Figure 4-3. The layered architecture of GJMABrowser.....	- 33 -
Figure 4-4: Logical and physical object views.....	- 39 -
Figure 4-5: Proxy design pattern	- 40 -
Figure 4-6: The relationship between managed and un-managed class	- 42 -
Figure 4-7: Wrapper design pattern.....	- 43 -
Figure 4-8: The sequence diagram for the proxy class.....	- 43 -
Figure 4-9: Insert GJMABObject in the inheritance chaining	- 45 -
Figure 4-10: Separate two associated classes into two different hosts.....	- 47 -
Figure 4-11. The GJMAApp development flow.....	- 51 -
Figure 4-12: A tree structure about user interface	- 54 -
Figure 5-1: Class loader structures	- 58 -

Figure 5-2: Base GJMAMesg format	- 60 -
Figure 5-3: GJMAMesg format for BORWSER mode	- 61 -
Figure 5-4: GJMAMesg format for the MASTER-SLAVE mode	- 62 -
Figure 5-5: Two phases in GJMA preprocessor	- 67 -
Figure 5-6: Replace field manipulation action with SETTER/GETTER	- 75 -
Figure 5-7: Modify codes for creating complementary object	- 77 -
Figure 5-8. How to intercept method invoke action	- 81 -
Figure 5-9: How ObjMngr to process a received GJMAMesg	- 83 -
Figure 6-1: The GJMApp development flow	- 87 -
Figure 6-2: The GJMApp (Hello World) accessed by different GJMClient.....	- 88 -
Figure 6-3: The GJMApp using Web services development flow.....	- 89 -
Figure 6-4: The GJMApp (Web services) accessed by GJMAppStandalone.....	- 91 -
Figure 6-5: The GJMApp (Web services) accessed by WAP browser.....	- 91 -
Figure 6-6: Remote method invocation performance evaluation.	- 93 -



LIST OF TABLES

Table 2-1: Optional packages in Java ME platform	- 10 -
Table 3-1: The GJMA packages	- 21 -
Table 3-2: The comparison table of the three running modes.	- 27 -
Table 4-1: The mapping table among tree element, WML and HTML.....	- 55 -
Table 5-1: Methods to build the first section.....	- 65 -
Table 5-2: Methods to build the sections other than the first section	- 65 -
Table 5-3: Methods to get parameter	- 66 -
Table 5-4: Examples for wrapper class naming.....	- 69 -
Table 5-5: The array class naming convention	- 78 -
Table 6-1: Test environment.	- 92 -



LIST OF LISTINGS

Listing 2-1: WML page example.....	- 6 -
Listing 2-2: C-HTML page example	- 7 -
Listing 2-3: XHTML basic page example	- 8 -
Listing 3-1: Partial device profile.....	- 19 -
Listing 3-2: Partial class profile	- 20 -
Listing 3-3: Java ME MIDP codes vs. GJMAApp codes.....	- 21 -
Listing 5-1: InferenceA source code.....	- 56 -
Listing 5-2: ClassA source code	- 56 -
Listing 5-3: Use the same class loader to load ClassA twice	- 57 -
Listing 5-4: Use two different class loaders to load ClassA twice	- 57 -
Listing 5-5: Use GJMAAppInterface to control GJMAApps	- 59 -
Listing 5-6: The partial source code for ActionBuilder.....	- 63 -
Listing 5-7: A marshalling example	- 66 -
Listing 5-8: An unmarshalling example	- 67 -
Listing 5-9: The source code for test.Foo1.....	- 70 -
Listing 5-10: The source code for test.Foo1's wrapper	- 71 -
Listing 5-11: The source code for test.Foo1's wrapper after replacements.....	- 73 -
Listing 5-12: The source codes for GJMA_ENTER and GJMA_LEAVE.....	- 76 -
Listing 5-13: The partially source code for GJMAObject.....	- 80 -
Listing 6-1: A partial sample code for MPI.	- 86 -
Listing 6-2: A sample interface for Java RMI.	- 86 -
Listing 6-3: A sample RMI server implementation.	- 86 -
Listing 6-4: Hello World sample code.....	- 88 -
Listing 6-5: Web services sample code	- 90 -



Chapter 1 Introduction

In the past decade, the number of mobile devices, such as mobile phones, PDAs, and notebooks, has increased enormously. Wireless networks, such as GRPS, UMTS, WiFi, have also become prevalent. These two factors have changed computing environments tremendously, and many new computing paradigms have been introduced, such as mobile computing [1], pervasive computing [2], and ubiquitous computing [3]. In other words, the requirements for developing mobile applications have increased, and more mobile applications have been developed for these mobile devices.

1.1. Motivation

However, there are many differences between these devices. First, they may have different executing environments. For instance, some of them comply with WAP [4], some with Java ME [5], and some with Microsoft .NET CF [6]. Second, the computing power and functionalities of these devices are diverse and they may have different hardware resources. For example, some have powerful processors, but some do not. Some are equipped with high resolution screens, but some are not. Third, these devices support different kinds of networks. These networks may have different bandwidths, latency, and reliability, and they may disconnect during use. These differences increase the complexity of developing a mobile application capable of supporting them all [7]. Developers have to face these issues, and have spent much time solving them.

Writing an application capable of supporting multiple devices is difficult. Thus, many studies and standards have tried to solve them. For example, Mobile Execution Environment (MExE [8]) defined by a 3GPP working group categorizes these devices into four execution environments, named classmark 1-4, to reduce mobile application development complexity. Different classmarks mean different execution environments. If a mobile application was

developed for classmark 1, it can be run on all devices which conform to classmark 1. Consequently, before developing a mobile application, developers have to decide which classmarks the application will support. This approach makes developers focus on specific execution environments, and implies that the application cannot support devices belonging to other classmarks. To overcome this problem, many studies have been made on adaptations and attribute programming [9], including user interface adaptation [10][11][12][13][14] and programming language transformation [15][16]. They can tailor the application to fit different user interface formats or execution environments. However, most of them do not consider the computing power and functionalities of devices and these resources are ignored or wasted. For instance, some of them focus solely on user interface adaptation. Some of them sacrifice the computing power and functionalities of devices because they can only use functions which all devices support.



1.2. Objectives

The aim of this dissertation is to design and implement a generic Java mobile application (GJMA) development framework. Every application developed from GJMA is capable of tailoring itself to fit different devices or situations according to user interface formats and the computing power and functionalities of the devices. In other words, more powerful devices will do more things in GJMA.

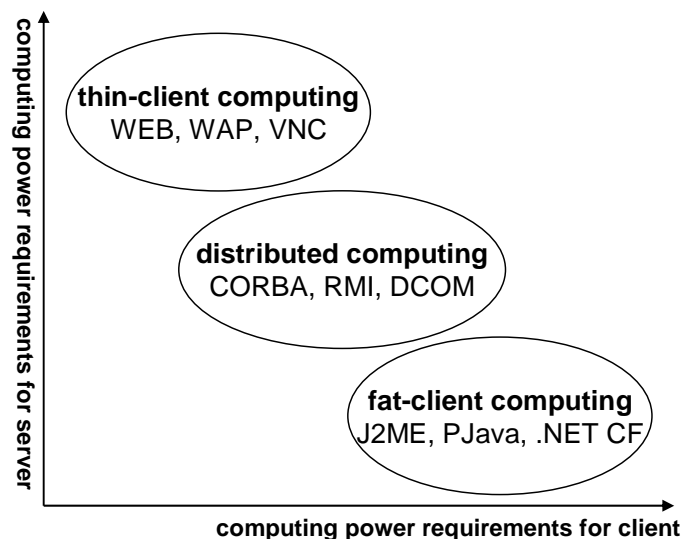


Figure 1-1: Computing power requirements for client and server

A server supports weak devices in this study, helping them do something they cannot do. Thus, every GJMA application can be viewed as client-server computing [17]. Figure 1-1 shows the computing power requirements for three different computing paradigms derived from client-server computing, and every computing paradigm has many different state-of-the-art technologies. In thin-client computing [18], clients are only responsible for user interface, and nearly all application logic is handled by the server. In distributed computing, clients are responsible for some application logic, and other logic is handled by the server. In fat-client computing, all application logic is handled by clients themselves. These three computing paradigms have different computing power requirements for clients. By adapting an application to one of the three computing paradigms, all kinds of devices can be well supported regardless of their computing power and functionalities. In addition, a user interface adaptation mechanism and a network adaptation mechanism are proposed in this dissertation.

1.3. Organization

This rest of this dissertation is organized as follows. Chapter 2 will introduce related background about mobile application development and Java language. Chapter 3 will describe

the concepts of GJMA. Chapter 4 will express the design issues and chapter 5 will discuss the implementation issues. Chapter 6 gives some evaluations and finally chapter 7 gives conclusion as well as future works.

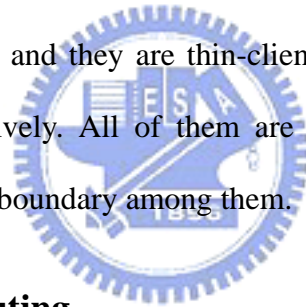


Chapter 2 Background

There are many specifications and works for mobile application developments. Moreover, some researches [19] had surveyed how to develop mobile applications. In this chapter, some important specifications and works are introduced. Also, Java class file structure is described in this chapter.

2.1. Related Specifications

There are many different specifications related to mobile application developments. They can be categorized according to which computing paradigms they used. Different computing paradigms have different way to develop an application. The most three popular computing paradigms are introduced here and they are thin-client computing, fat-client computing and distributed computing respectively. All of them are derived from client-server computing paradigms, so there is no clear boundary among them.



2.1.1. Thin-Client Computing


In thin-client computing, the bulk of business logic is processed on the corresponding server, so it can also be called server-based computing. The responsibility of clients is providing user interface only, thus the computing power requirement of clients is lower.

In this computing model, an application run on server-side can be developed by various platforms, such as Java Servlet[20], PHP[21], ASP[22] and so on. No matter what platform is used, the important is how clients interact with the server. There are many ways for clients to interact with the server and they can be categorized into two categories: standard and proprietary. Basically, clients can use built-in browsers to access the application which will use standard protocols (such as WAP and HTTP) and content formats (such as WML [23],

XHTML [24] and so on). Moreover, clients can use specific programs to access the application which will use proprietary protocols (ex. NTT DoCoMo's i-mode [25]) and content formats (ex. C-HTML [26]). In short, both browsers and specific programs run on client-side are used to render user interface and send user request to the application. The differences among them are content formats and protocols. Some specifications used to create thin-client mobile applications are given below.

2.1.1.1. Wireless Markup Language

Wireless Markup Language (WML) is the content format used in Wireless Application Protocol (WAP) and Listing 2-1 is a WML page example. Currently, almost mobile phones support WAP and they have built-in browsers capable of accessing these WML pages. An application has to output WML pages if it is designed to support WAP-enabled mobile devices. Moreover, the browsers use WAP to communicate with the server.



```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//PHONE.COM//DTD WML 1.1//EN"
"http://www.phone.com/dtd/wml11.dtd">
<wml>
  <card id="main" title="WML Example">
    Hello World
    <p><a href="about.wml">About...</a></p>
  </card>
</wml>
```

Listing 2-1: WML page example

2.1.1.2. Compact HyperText Markup Language

Compact HTML (C-HTML) is the content format used in NTT DoCoMo's i-mode and Listing 2-2 is a C-HTML page example. Basically, C-HTML is a subset of the HTML markup language. In addition, C-HTML adds some features which cannot find in the HTML standard, notably the accesskeys, phone number shortcuts for links and so on. Currently, only i-mode

mobile phones have browsers capable of accessing these C-HTML pages. An application has to output C-HTML pages if it is designed to support i-mode mobile phones. Moreover, the browsers use NTT DoCoMo's proprietary protocols, ALP (HTTP) and TLP (TCP and UDP), to communicate with the server.

```
<html>
  <head><title>C-HTML Example</title></head>
  <body>
    Hello World
    <p><a href="about.chtml" accesskey="1">About...</a><p>
  </body>
</html>
```

Listing 2-2: C-HTML page example

2.1.1.3. Extensible HyperText Markup Language

Extensible HyperText Markup language (XHTML) is a content format similar to HTML but XHTML also conform to XML [27] syntax. In XHTML family, there are two members related to mobile devices: XHTML basic [28] and XHTML mobile profile [29]. The former is designed to support devices which cannot support all XHTML dialects and it is intended to replace WML and C-HTML. Listing 2-3 is a XHTML basic example. The latter is based on XHTML basic and it adds more features for mobile phones. An application has to output XHTML pages if it is designed to support mobile phones equipped with XHTML browser. Moreover, the browsers use HTTP or WAP 2.0 to communicate with the server.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
<html>
  <head><title>XHTML Basic Example</title></head>
  <body>
    Hello World
    <p><a href="about.xhtml">About...</a><p>
  </body>
</html>
```

Listing 2-3: XHTML basic page example

2.1.1.4. Markup Language Transform

There are many different content formats mentioned above and the greater part of them is markup languages. Hence, an application has to output different markup languages to serve different clients equipped with different browsers. XSLT [30] is a technology often used to convert XML data into other markup languages. Thus, an application applied Model-View-Controller pattern [31] can exploit XML document, such as UIML [32], to describe user interface and then use XSTL to convert it into the markup languages supported by the target client on demand. Besides this, there are several similar researches [33][34][35][36].

2.1.2. Fat-Client Computing

In fat-client computing, the bulk of business logic is processed on the client directly. All applications using this computing model can be executed on client without any server assistance. It implies these applications can run offline and the devices must have enough capabilities to execute the applications. This computing model is widely used. Currently, the most three popular development platforms are Java ME, BREW [37], Symbian [38], and .NET Compact Framework.

2.1.2.1. Java ME

Java is an object-oriented programming language developed by Sun Microsystems in the early 1990s. Unlike C programs are compiled to native machine codes, Java applications typically are compiled to Java bytecode which is platform independent and is run on a stack machine named Virtual Machine [39]. Nowadays, there are many different editions for Java, such as Java EE [40], Java SE [41] and Java ME for different purposes as Figure 2-1 shows. Java EE is for enterprise application, Java SE is for general application, and Java ME is for mobile application.

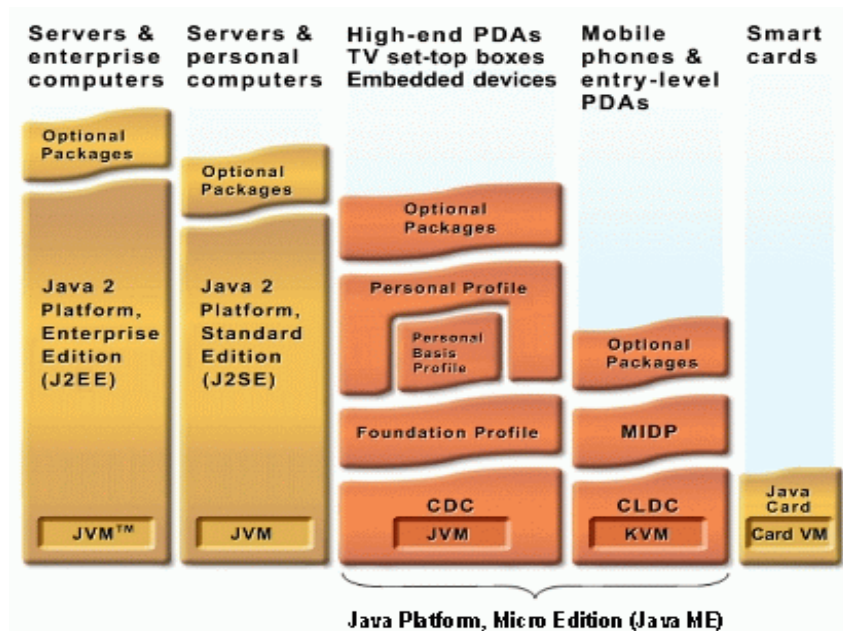


Figure 2-1: Java platforms for different purposes

Because of the mobile device diversity and resource constraints, Java ME architecture is combined of three parts: configurations, profiles and optional packages. It implies that it is impossible to put all codes into a resource-limited device. Thus, each combination of the three parts is optimized for the memory, processing power, and I/O capabilities for target devices. All these things make it clear that Java ME use different combinations to support diverse mobile devices well. For example, some mobile devices are equipped with Bluetooth and some are not.

For devices equipped with Bluetooth, the optional package JSR [42] 82 has been included in the combination to support it. For devices which are not equipped with Bluetooth, the optional package JSR 82 is excluded in the combination to save memory. The functionalities of the three parts are described as follows:

- Each configuration provides base functionalities for particular devices with similar characteristics. Currently, there are two base configurations, CLDC and CDC, in Java ME. The full name of CLDC is “Connected Limited Device Configuration” which is designed for limited mobile devices such as cellular phones. Moreover, the full name of CDC is “Connected Device Configuration” which is designed for more capable devices such as smartphones.
- Each profile is a set of higher level APIs. It defines the application life cycle model, the user interface, and the device specific properties.
- Each optional package offers different functionality as Table 2-1 shows. The capabilities of Java ME can be further extended by combining various optional packages.



Table 2-1: Optional packages in Java ME platform

Optional package name	Version	JSR
Java APIs for Bluetooth Bluetooth, OBEX	1.1	JSR 82
Content Handler API	1.0	JSR 211
Mobile Media API	1.2	JSR 135
Java Binding for the OpenGL® ES	1.0	JSR 239
J2ME Web Services Specification JAXP, JAX-RPC	1.0	JSR 172
Security and Trust Services APIs ADPU, JCRMI, PKI, CRYPTO	1.0	JSR 177
Security JSSE, JCE, JAAS	1.0	JSR 219
Advanced Graphics and User Interface Java 2D™, Swing	1.0	JSR 209
RMI	1.0	JSR 66

JDBC	1.0	JSR 169
JavaTV™ API	1.1	JSR 927

Because Java is platform independent, the greater part of mobile phones is Java ME-enabled currently.

2.1.2.2. BREW

Binary Runtime Environment for Wireless (BREW) is a mobile application development platform created by Qualcomm for mobile phones. It can support GSM/GPRS, UMTS and CDMA. BREW is similar to Java ME but BREW is more powerful because the running level of BREW is lower than Java ME. In other words, BREW is more close to hardware. For example, it can access screen buffer directly.

2.1.2.3. Symbian



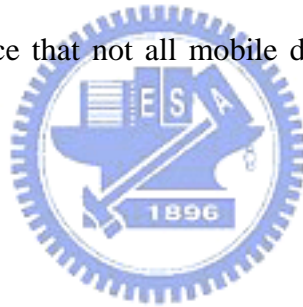
Symbian is an operating system for handheld devices with limited resources. It has different device classes and variants for different mobile devices, such as Series 60, Series 80, UIQ and so on. Basically, they have same basis but may have different look and feel. A Symbian application has to be written in C++ and use classes provided by Symbian libraries. Currently, a Symbian application only can be executed on Symbian OS.

2.1.2.4. .NET Compact Framework

The Microsoft .NET Compact Framework is a version of the .NET Framework that is designed to run on mobile devices. Basically, it is a subset of .NET Framework but it adds some libraries designed specifically for mobile devices. A .NET Compact Framework application can be developed in C# or Visual Basic.NET. Currently, the application only can be executed on mobile devices powered by Microsoft .NET Compact Framework.

2.1.3. Distributed Computing

In distributed computing, the bulk of business logic is spread on different hosts. These hosts provide different services and they cooperate with each other to complete the business logic. This computing model is widely used, such as CORBA [43], Java RMI [44], and Web services [45] are examples. Different distributed technologies have different usages [46]. Some have to write remote interface descriptions [47][48], some have to declare and implement specific interfaces [14], some exploit annotation or attribute to declare remote methods in source codes [49], and some only provide communication library to call [50]. However, for mobile application development, not all technologies is suitable for mobile application and only Web services and Java RMI [51] are two of the most popular technologies to realize distributed computing. It is worth to notice that not all mobile devices support Web services and Java RMI.



2.2. Related Works

2.2.1. VNC

Virtual Network Computing (VNC) [52] is a remote desktop software capable of showing user interface locally (client-side) and controlling the applications remotely (server-side). At the beginning, VNC is not designed for mobile devices but there are some client implementations for mobile devices, such as VNC2Go [53] and so on. In fact, these applications run on server-side are not designed for mobile devices, so the user interface displayed on client-side may be terrible. Figure 2-2 (the figure is captured from <http://www.freeutils.net/vnc2go/index.jsp>) is an example and it only can display small portion of desktop at once. Moreover, VNC uses its proprietary protocol to communicate between server and client.

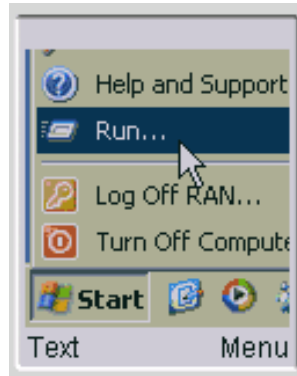


Figure 2-2: A VNC2Go screenshot

2.2.2. TCPTE

The thin-client applications for limited devices (TCPTE) framework [54][55] is a development framework for mobile applications. It can execute Java application on a remote server and display their AWT interface on a local client. It combines the advantages of thin-client computing with the richness user interface and lets programmer develop mobile applications using same process and tools they typical use for desktop applications. Besides this, there are several similar researches and projects [56][57]. They can display user interface remotely also.

2.2.3. J2ME Polish

J2ME Polish [14] is based on Java ME and it is a collection of tools for developing Java ME applications. These tools include device database, preprocessor and utility classes. The device database is used to describe the capabilities of mobile devices and the preprocessor modify source codes to fit target devices according the device database. Furthermore, it provides richer and flexible user interface classes which can use CSS [58] to describe. An example is shown in Figure 2-3 (the figure is captured from <http://www.j2mepolish.org/screenshots.html>). There are three screenshots and they shows the same screen with different CSS. Besides, it also provides some useful features, such as remote method invocation, serialization,

persistence and so on. These mechanisms can help developer build mobile application easier.

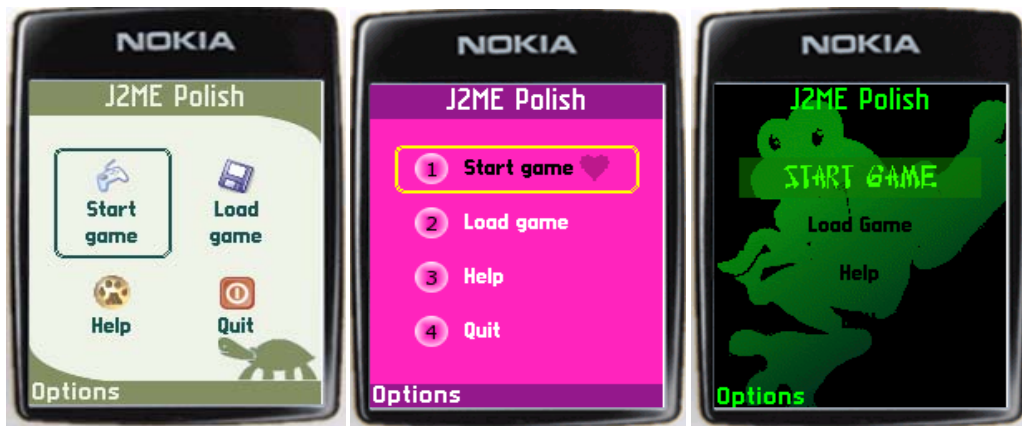


Figure 2-3: J2ME Polish screenshots

2.2.4. Nano-X

The Nano-X Window System (previously Microwindows) [59] is aimed at bringing the features of modern graphical windowing environments to smaller devices and platforms. Basically, it only provides user interface related functionalities. Unlike X Window [61], both server-side and client-side have to be executed on mobile devices in the Nano-X Window System. Because the usage of Nano-X is similar to usage of X Window, developer can use the same process to build mobile application appearance.

2.2.5. ART

Adaptive Remote Terminal (ART) [60] is a mobile development framework capable of executing the application on server-side and displaying user interface on client-side. Besides, it can output different content formats for different browsers. End-users can use built-in browsers or the proprietary browser to access these applications run on server-side.

2.3. Java Class File Structure

Figure 2-4 is the structure of the java class file. There are many parts in a class file, including

header, constant pool, access rights, this class, super class, implemented interfaces, fields, methods and attributes. It is worth to notice that there are many indexes. Every index is used to pointer to an entry in the constant pool. In fact, there is much information stored in the constant pool, such as methods' name, methods' signatures, fields' name, fields' type, class name, and so on. By analyzing the constant pool in a class file, all information can be got and there are many tools capable of modifying or examining the class files. In other words, Java class file is easy to be modified. In GJMA, BCEL [62] is used to do this. It can generate class file or modify the class file on demand, including the bytecode in the methods.

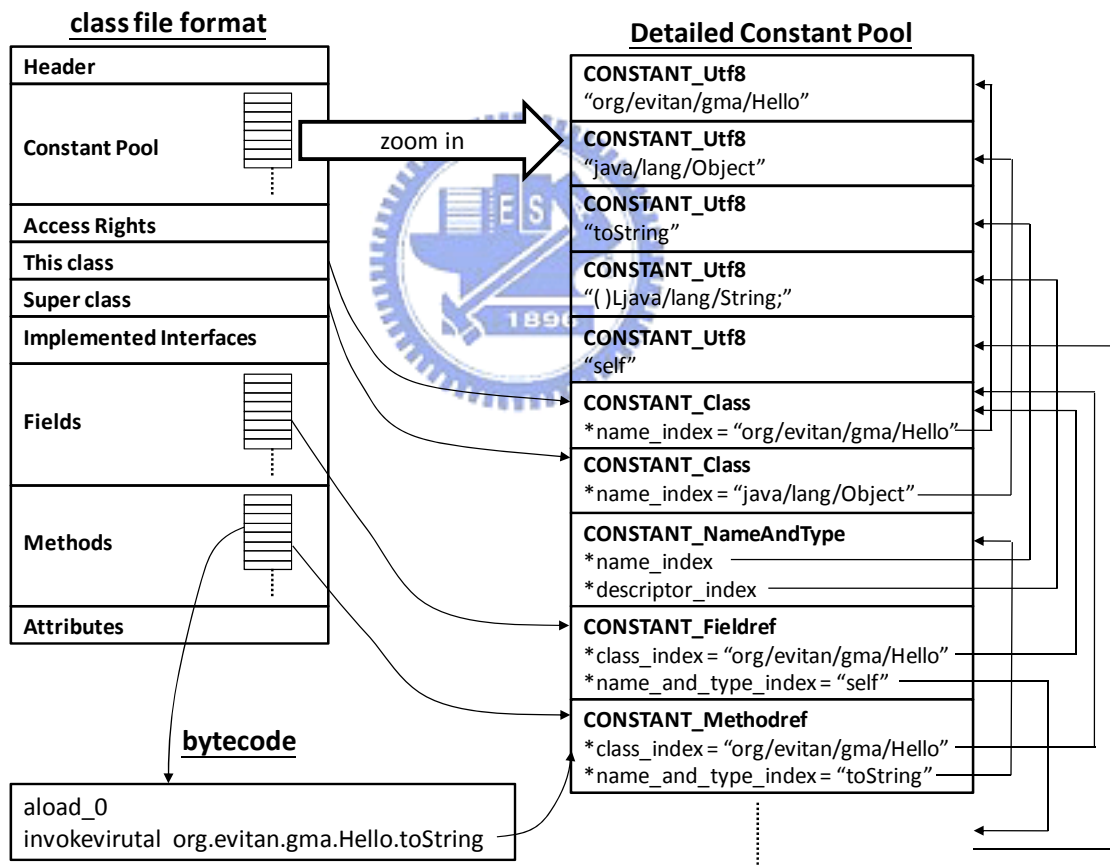


Figure 2-4: Java class file structure

Chapter 3 GJMA Development Framework Overview

This chapter will introduce GJMA concepts, GJMA system entities and three supported running modes.

3.1. GJMA Concepts

It is difficult for a mobile application to support all devices well due to varying computing power and functionalities. A simple scenario follows. Three end-users want to control home appliances via their own mobile devices, named DeviceA, DeviceB, and DeviceC, respectively, using a Java ME MIDP (JSR 118) home appliance control application which uses Web services to control home appliances. These devices have different functionalities. DeviceA cannot run Java ME applications and only has a built-in WAP browser. DeviceB has a WAP browser and is Java ME MIDP compatible, but it does not support Web services (JSR 172). DeviceC is similar to DeviceB, except DeviceC supports Web services. In the scenario, DeviceC can run the home appliance control application directly, but DeviceA and DeviceB cannot. Without an automatically adaptation framework, a developer can only solve the problem in two ways. The first approach is to develop three specific editions for the three devices. The second approach is to develop one general WAP version which can be accessed by WAP browsers on all devices. The former is a tedious task and the latter sacrifices the computing power of DeviceB and DeviceC. One of the primary GJMA framework objectives is to let all devices run at capacity without publishing many editions of an application: write an application once and it can support all kinds of devices well.

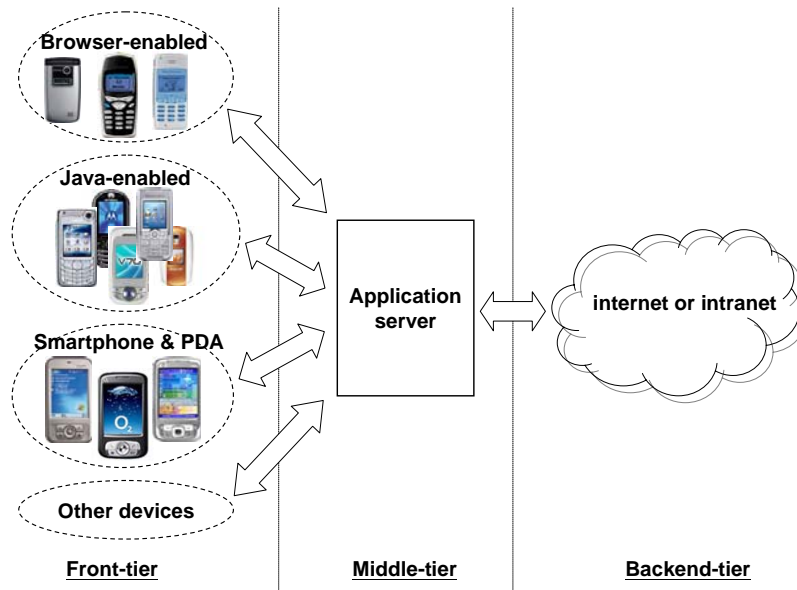


Figure 3-1: Three-tier architecture used in GJMA

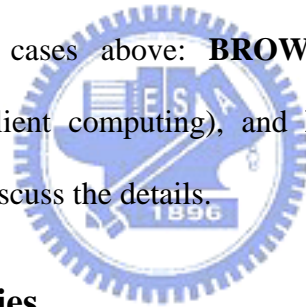
GJMA uses a three-tier architecture, as Figure 3-1 shows, to solve the problems of diverse computing power and functionalities. End-users use their own desktops or mobile devices, called end-devices, in the front-tier to access mobile applications. There is at least one application server in the middle-tier, which provides necessary execution environments and services for running applications and end-devices. An application in the GJMA is designed to be capable of running in the front-tier (fat-client computing), in the middle-tier (thin-client computing), or even in both tiers (distributed computing) simultaneously depending on the computing power and functionalities of end-devices. More front-tier computing power means more codes will be run in the front-tier (implicitly fewer codes will be run in the middle-tier). By analyzing, an application may therefore face three different running cases:

- The computing power of the end-device is not good enough or the device cannot run applications other than built-in applications. The device here is not suitable for the application or cannot run the application. Thus, thin-client computing, such as WEB-based technology, is suitable for this case. Entire application codes must be executed in the

middle-tier, and the front-tier is only responsible for user interface.

- The computing power of the end-device is good enough and the device supports all functionalities which the application requires. In this case, entire application codes can be executed on end-devices. This is a kind of fat-client computing, like running a Java ME MIDP application on a mobile device.
- The computing power of the end-device is good enough but the device does not support all functionalities which the application requires. The device in this situation cannot execute some codes within the application, and these codes have to be handled by the middle-tier application server. This is a kind of distributed computing, such as Java RMI.

To support all kinds of end-devices, GJMA provides three different running modes for an application to fit the three cases above: **BROWSER** mode (thin-client computing), **STANDALONE** mode (fat-client computing), and **MASTER-SLAVE** mode (distributed computing). Section 3.3 will discuss the details.



3.2. GJMA System Entities

The GJMA framework contains three important entities: end-device (GJMClient), GJMApp, and GJMServer. In Figure 3-1, End-device (GJMClient) participates in front-tier and GJMServer involves in middle-tier as the application server. Furthermore, GJMApps are applications capable of running in one of the three running modes.

3.2.1. End-device (GJMClient)

An end-device is any device used by an end-user in the front-tier. These include PDAs, mobile phones, notebooks and so on. End-devices can be divided into two categories according to their programmable characteristics. All end-devices belonging to the programmable GJMClient category can execute applications other than their built-in applications. On the other hand, all

end-devices belonging to the non-programmable GJMAClient category can only execute the built-in applications.

Because there are many differences between end-devices, the GJMA framework contains an end-device database to provide related information. This database helps the GJMA framework adjust applications to fit different end-devices. The end-device database comprises two XML [27] documents: device profile and class profile. Device profile describes related information for end-devices, and class profile describes decision trees used to find the most suitable classes. End-device capabilities are listed in device profile as Listing 3-1 expressed, including execution environments, screen size and other data. Figure 3-2 is an example of a decision tree for the org.gjma.application.GJMAApp class. The class has four implementations for different running modes and Listing 3-2 is the corresponding class profile.

```
<Vendor="SonyEriassion">  
  <Device name="k700i">  
    <Browser type="WAP">  
    </Browser>  
    <ExecutionEnvironment val="J2ME_MIDP">  
    </ExecutionEnvironment>  
  </Device>  
</Vendor>
```

Listing 3-1: Partial device profile

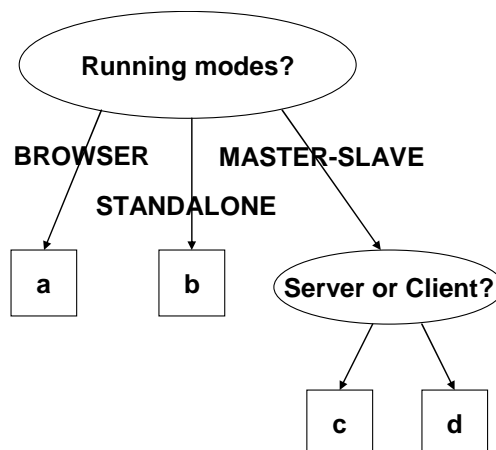
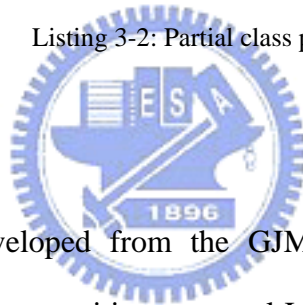


Figure 3-2: The decision tree for the class org.gjma.application.GJMAApp

```
<Class name="org.gjma.application.GJMAApp">
  <Decision var="RunningMode">
    <Equal val="BROWSER">
      <Edition name="a" path="BROWSER\org\gjma\application\GJMAApp.class" />
    </Equal>
    <Equal val="STANDALONE">
      <Edition name="b" path="STANDALONE\org\gjma\application\GJMAApp.class" />
    </Equal>
    <Equal val="MASTERSLAVE">
      <Decision var="MasterOrSlave">
        <Equal val="Master">
          <Edition name="c" path="MS\MASTER\org\gjma\application\GJMAApp.class" />
        </Equal>
        <Equal val="Slave">
          <Edition name="d" path="MS\SLAVE\org\gjma\application\GJMAApp.class" />
        </Equal>
      </Decision>
    </Equal>
  </Decision>
</Class>
```

Listing 3-2: Partial class profile

3.2.2. GJMAApp



Every mobile application developed from the GJMA framework is called a GJMAApp. Developing a GJMAApp is similar to writing a general Java ME MIDP application, but there are something differences between them as Listing 3-3 shows. In Java ME MIDP applications, developers must consider whether or not the classes within the Java ME application are compatible for end-devices. This is because all classes have to be handled by the end-devices. Conversely, GJMAApp developers need not worry about compatibility problems since the server will help end-devices handle all incompatible classes. Classes which need to be executed by the servers are determined in deployment time. A GJMAApp can be deployed in different running modes according to end-device execution environments, computing power, and functionalities. Because different running modes have different requirements for end-devices, a GJMAApp can support the majority of end-devices by adapting to different running modes. As a result, developers do not need to take different devices into account. Instead, they can focus on

business logic only. Section 4.4.4 introduces deployment process details.

<pre> public class TestMIDlet extends javax.microedition.midlet.MIDlet { public TestMIDlet() { //constructor } public void startApp() { //this will be called, when MIDlet is started } public void pauseApp() { //this will be called, when MIDlet is paused } public void destroyApp(boolean unconditional) { //this will be called, when MIDlet is destroyed } } </pre>	<pre> public class TestGJMAApp extends org.gjma.application.GJMAApp { public TestGJMAApp() { //constructor } public void startApp() { //this will be called, when GJMAApp is started } public void pauseApp() { //this will be called, when GJMAApp is paused } public void destroyApp(boolean unconditional) { //this will be called, when GJMAApp is destroyed } } </pre>
---	--

Listing 3-3: Java ME MIDP codes vs. GJMAApp codes

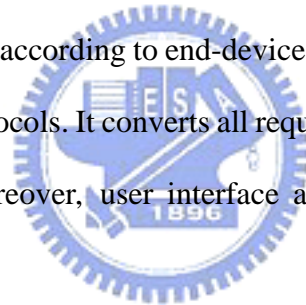
Every GJMAApp has a main class which must inherit from the org.gjma.application.GJMAApp class. This is similar to every Java ME MIDP application which has a main class which must inherit from javax.microedition.midlet.MIDlet class. The GJMA framework, like software development kits, provides classes other than the org.gjma.application.GJMAApp class as Table 3-1 shows. This helps developers build mobile applications efficiently. The GJMA framework prepares many different editions of classes to support all kinds of end-devices in three running modes. Different editions of a class have the same functionalities but have different implementations. When a GJMAApp is deployed, the most suitable classes are chosen according to the class profile in the end-device database. For example, the class org.gjma.ui.LayoutManager is used to arrange widgets, and has two editions. One is for small screens and the other is for large screens. Another example is that the class org.gjma.application.GJMAApp, which initializes all necessary resources in runtime, has four editions. One is for the STANDALONE mode, one is for the BROWSER mode, and two are for the MASTER-SLAVE mode as Figure 3-2 shows.

Table 3-1: The GJMA packages

package name	descriptions
org.gjma.application	core package, including main class, loader class and so on
org.gjma.ui	all user interface related classes are put in this package
org.gjma.io	The classes used to handle I/O are put in this package
org.gjma.service	The classes related to UPnP, Web Service and Jini are put in this package
org.gjma.util	The utility classes are put in this package

3.2.3. GJMAServer

The GJMAServer plays an important role in the BROWSER and the MASTER-SLAVE modes. If a GJMApp is run in the STANDALONE mode, the GJMAServer is unnecessary. Generally speaking, the GJMAServer provides runtime environments and services for GJMApps. The GJMAServer's main functions are application management, communication management, and user interface adaptation. Application management manages the lifecycle of GJMApp. It can load, start, and stop a GJMApp according to end-device requests. Communication management supports different network protocols. It converts all requests into messages, named GJMAMesg described in section 5.2. Moreover, user interface adaptation transforms user interface to different content formats.



3.3. Three Running Modes in GJMA

This subsection discusses the concepts of the three running modes. The mode(s) in which a GJMApp is deployed depends on which end-devices are used, and the decision is made in deployment time. In other words, a GJMApp may be simultaneously deployed in different running modes to support different kinds of end-devices. End-device deployment results may be different even though a GJMApp is deployed in the same running mode because the end-devices may have different functionalities as Figure 3-3 shows.

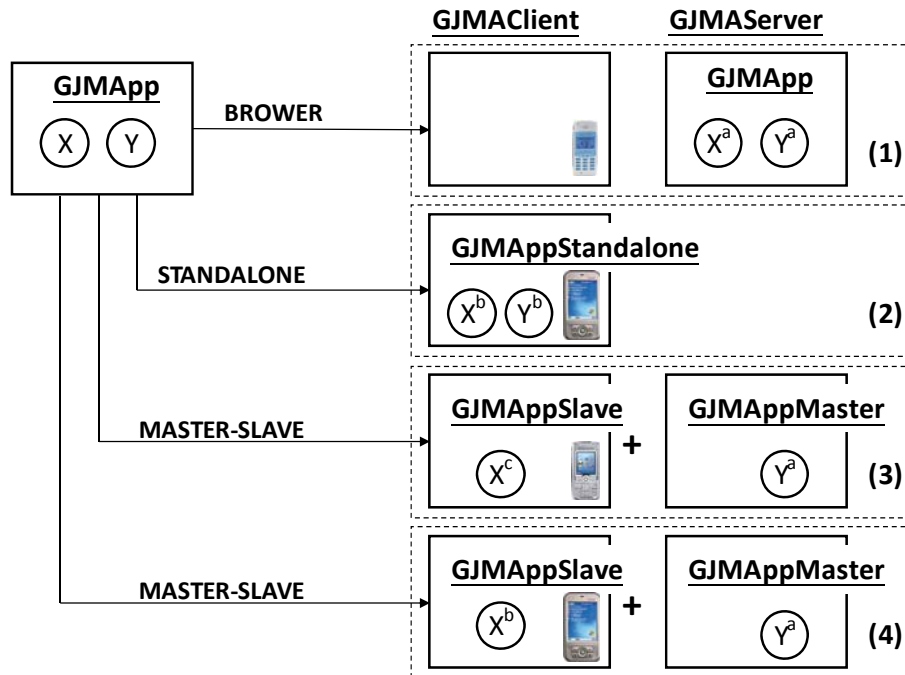


Figure 3-3: A GJMAApp deployed to different running modes or devices.

Figure 3-3 is a sample to demonstrate the result after deployment. Many details, such as proxy class and other necessary classes, are omitted in this figure to keep it simple. For the same class name, different superscript represents different editions/implementations. In Figure 3-3, the original GJMAApp is consisted of two classes, X and Y. Moreover, the GJMAApp is deployed to four different end-devices.

- (1) Deploy the GJMAPP to BROWSR mode. Both X and Y are placed on GJMAServer.
- (2) Deploy the GJMAApp to the STANDALONE mode. Both X and Y are placed on GJMClient, specially named GJMAAppStandalone.
- (3) Deploy the GJMAApp to the MASTER-SLAVE mode. X is placed on GJMClient, specially named GJMAAppSlave and Y is placed on GJMAServer, specially named GJMAAppMaster.
- (4) Deploy the GJMAApp to the MASTER-SLAVE mode also. This case is similar to the case (3) but the target end-device is different. Hence, (3) and (4) have different results. In other words, (3) and (4) used different editions of class X.

3.3.1. BROWSER Mode

In the BROWSER mode, all GJMAApp codes are handled by a GJMAServer. The front-tier in Figure 3-1 is a presentation layer, and end-devices are responsible for user interface only as Figure 3-4 shows. End-users can use many types of devices in the front-tier and the GJMA framework will automatically tailor content formats to fit various end-devices in runtime. When a GJMAApp is deployed in the BROWSER mode, it can be used by a great majority of browser-enabled devices. This mode is also device-independent, and all GJMAApps can be deployed in this mode.

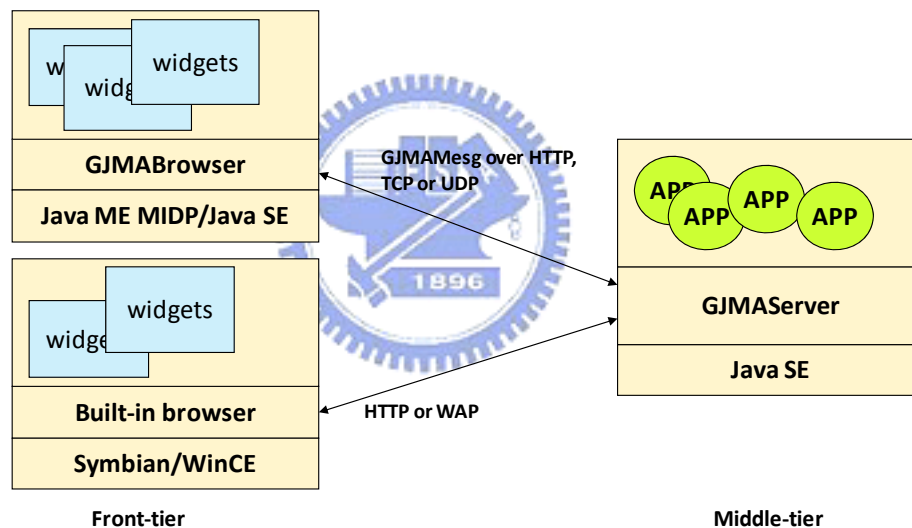


Figure 3-4: A diagram for the BROWSER mode

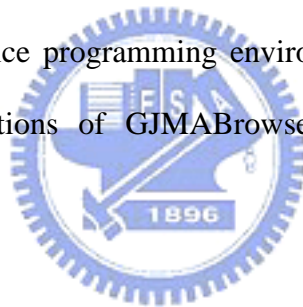
Currently, GJMA systems support two kinds of browsers: built-in browsers and the GJMABrowser. End-users can use either one, but the latter is specifically designed for GJMA use. As a result, it has better display effects and interactive abilities. However, it requires a GJMABrowser installation before use. For built-in browsers, no additional applications need to be installed prior to use.

- Built-in browser

Mobile devices use many different kinds of built-in browsers, such as XHTML browsers, WAP browsers, and others. They may use different network protocols to communicate, including HTTP and WAP. This means that a GJMAServer must support these different protocols. Currently, most mobile devices have a built-in browser. If an end-device is a non-programmable GJMClient, its built-in browser is the only interface to interact with GJMApps.

- GJMABrowser

A GJMABrowser is a mobile application capable of drawing UI widgets and handling end-user actions. A GJMABrowser only can be installed on a programmable GJMClient. It interacts with GJMApps by delivering GJMAMesg between them. The most popular mobile device programming environments are currently Java ME MIDP and .NET CF. Two editions of GJMABrowser are implemented to support both environments.



The front-tier consumes very few resources in this mode because all application codes are handled by the middle-tier. The front-tier is responsible for presentation only. End-users can access several mobile applications simultaneously in this mode, and GJMA provides a menu like the task manager in Microsoft Windows XP (see Figure 3-5). This helps end-users select which GJMApp to start, stop or switch to.

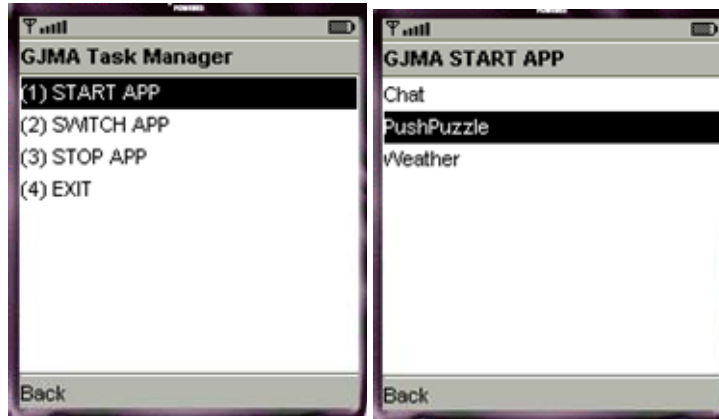


Figure 3-5: A screenshot of the GJMA task manager

3.3.2. STANDALONE Mode

In the STANDALONE mode, all application codes are run entirely on end-devices and do not need any middle-tier assistance, implying that this mode can be used in an environment without a network. When a GJMAApp is deployed in the STANDALONE mode, the entire application, called GJMAAppStandalone, will be executed devices which are powerful enough. This mode is device-dependent, so the application must be re-deployed when changing end-devices. In other words, a GJMAApp has to be deployed in the STANDALONE mode many times for different end-devices.

3.3.3. MASTER-SLAVE Mode

When a GJMAApp is deployed in the MASTER-SLAVE mode, its codes will be divided between end-devices and the GJMAServer as Figure 3-6 shows. The end-device part belongs to GJMAAppSlave, and the GJMAServer part belongs to GJMAAppMaster. Both of them are generated automatically from the original GJMAApp in deployment time. The details will be described in section 4.4.3.

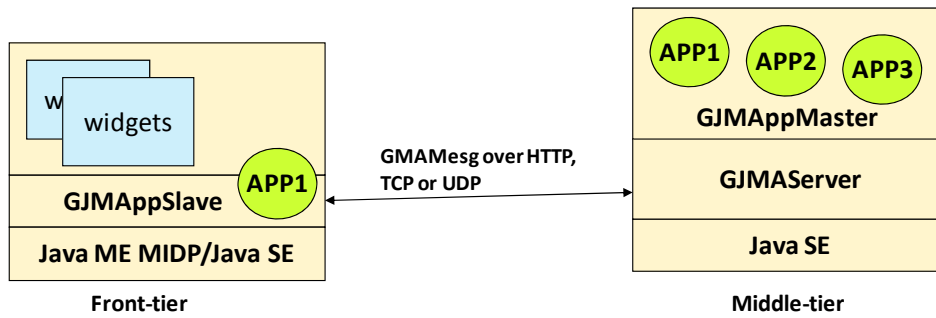


Figure 3-6: A diagram for the MASTER-SLAVE mode

3.3.4. Three Running Modes Comparison

Table 3-2 is a comparison table among the three running modes and there are five criteria in the table. The first criterion is the computing power requirement for end-devices. Today, almost end-devices can access a GJApp in the BROWSER mode. The second criterion is whether or not need network environments when running a GJApp. If a GJApp is deployed in STANDALONE, the GJApp can be executed when off-line. The third criterion is whether or not support to access multiple GJApps concurrently. Because almost end-devices, especially hand-held devices, can only run a KVM at the same time, they can only launch a Java application at one time also. The fourth criterion is whether or not need to install additional program on end-devices. In GJMA, only GJMABrowser has to be installed. The final criterion is whether or not to deploy a GJApp on end-devices before accessing it.

Table 3-2: The comparison table of the three running modes.

	BROWSER		MASTER-SLAVE	STANDALONE
	built-in	GJMABrowser		
1. requirements	Low		Middle	High
2. need network?	Must		Must	No
3. support multi-tasks?	Yes		No	No
4. need installation?	No	Yes	No	No
5. need deployment?	No		Yes	Yes

Chapter 4 GJMA Design Issues

This chapter contains two parts. The first part introduces system architecture and the second part discusses adaptation mechanisms.

4.1. GJMAServer Architecture

GJMAServer plays an important role in the BROWSER and MASTER-SLAVE running modes because some GJMAApp codes are executed by the GJMAServer in these two modes. GJMAServer architecture is shown in Figure 4-1. It can be considered a layered architecture; with an Adaptive Transport Layer, a Message Routing Layer and an Application Runtime Layer from bottom to top. In this way, GJMA can be modularized very well and the layered design makes maintenance and upgrading easier. The following sub-sections will discuss these three layers respectively.

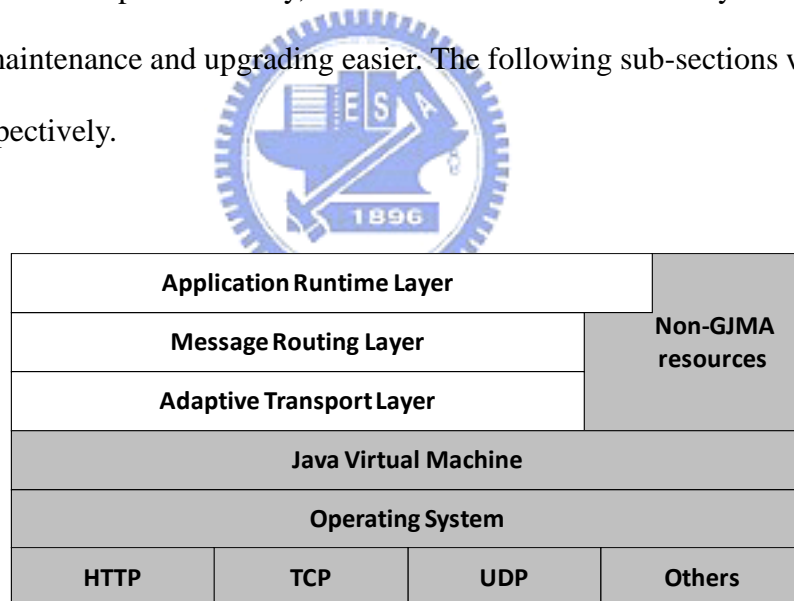
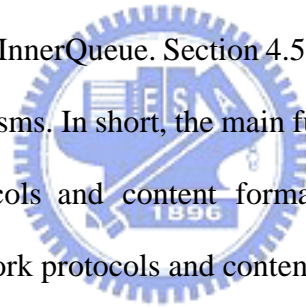


Figure 4-1: The layered architecture of GJMAServer.

4.1.1. Adaptive Transport Layer

The Adaptive Transport Layer enables GJMAServer to communicate with different kinds of GJMAClients which may use different network protocols. Figure 4-2 helps illustrate the detailed GJMAServer architecture.

The primary role in this layer is Communication Manager (CommMngr), which is a super daemon capable of handling many different networks protocols including TCP, UDP, HTTP and so on. It has two missions. First, it establishes the relationship between GJMAServer and GJMClient when a GJMClient sends a login request to GJMAServer. Secondly, after successful login, CommMngr creates a logic process (i.e. a user process, including a UserOutD, a UserInD and a UserOutQ) for the GJMClient. Every logic process might have different components or functionalities depending on which network protocol it uses. The UserOutD is a thread. It is responsible for picking messages called GJMAMesg from the queue named UserOutQ, and sending them to the corresponding GJMClient directly or translating GJMAMesgs to specific content formats. The UserInD is also a thread. It handles GJMAMesgs from its client directly or translates incoming requests from its client to GJMAMesgs, and then put them into the queue named InnerQueue. Section 4.5 and Section 4.6 introduce the details of this layer's adaptation mechanisms. In short, the main functionality of this layer is to transform from different network protocols and content formats to GJMAMesgs and to transform GJMAMesgs to different network protocols and content formats.



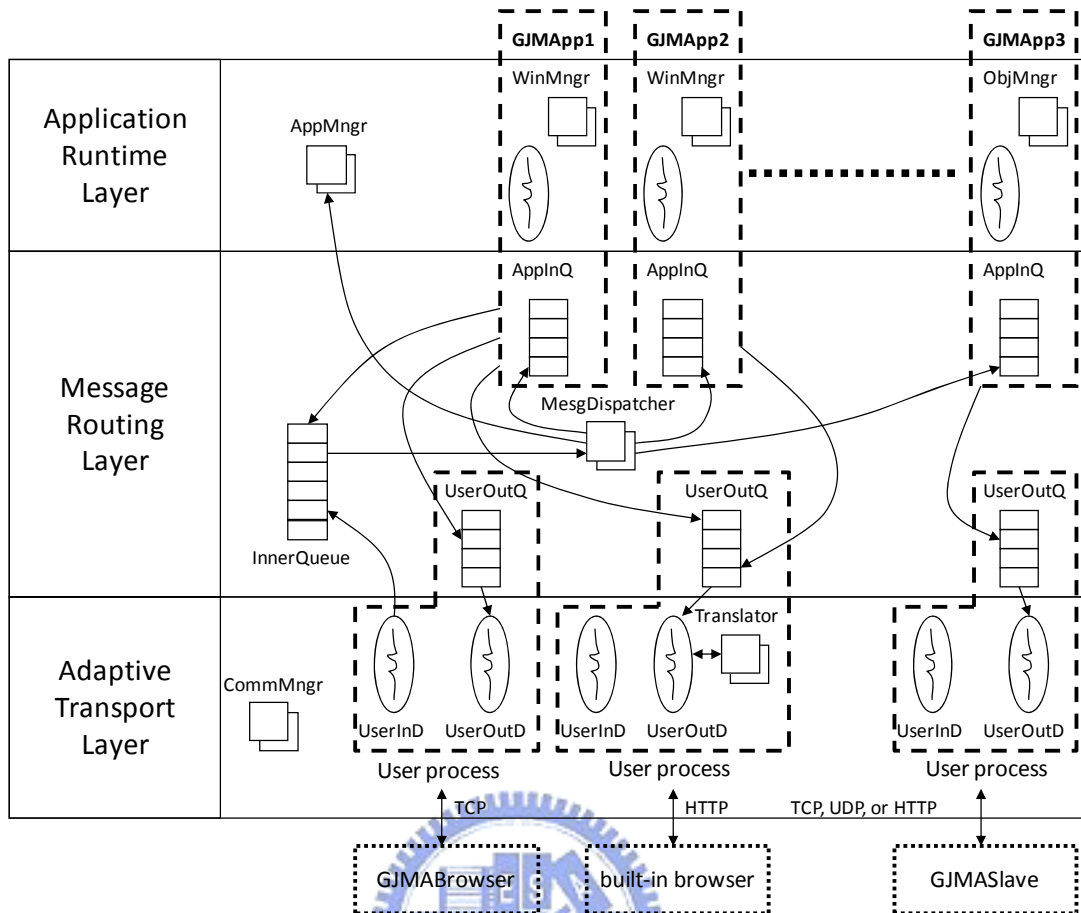


Figure 4-2: The detailed architecture of GJMAServer.

4.1.2. Message Routing Layer

The Message Routing Layer implements an asynchronous message delivery mechanism and the process unit in this layer is GJMAMesg, which's format is introduced in Section 5.2. This layer is responsible for delivering GJMAMesgs to right queue(s) depending on the information encoded in the GJMAMesgs. For example, when a user presses a button or a GJMApp orders the client-side to create a new UI widget, a corresponding GJMAMesg will be generated and routed to the proper destination. The main components in this layer are Queue and Message Dispatcher (MesgDispatcher). The MesgDispatcher is responsible for routing GJMAMesg to the correct queue(s). Moreover, there are three kinds of queues on GJMAServer: InnerQueue, AppInQ and UserOutQ.

Every request received by UserInD is translated to a GJMAMesg and placed it into InnerQueue. Then MesgDispatcher will dispatch them to the some AppInQ in which GJMApp will process these GJMAMesg or call Application Manager (AppMngr) to handle the GJMAMesg.

Once a GJMApp generates a GJMAMesg whose destination is a GJMClient, the message will be placed in UserOutQ within the GJMClient's user process. UserOutD within the user process will later send the message to its client or pass the message to the translator according to the kinds of GJMClient.

If a GJMApp needs to negotiate with other GJMApps on the same GJMAServer, GJMAMesgs will be sent back to InnerQueue and wait for dispatching by MesgDispatcher again.

Since wireless networks are often not stable enough, the GJMA framework uses the asynchronous message delivery mechanism mentioned above for transmissions between GJMAServers and GJMClients. This mechanism decouples the GJMApp and low-level network protocols, helping the GJMAServer handle disconnection situations and preventing GJMApps from accessing the network directly. After a GJMClient reconnected, just rebind the previous used GJMApps. Also, this mechanism enables communication among GJMApps and supports one-to-one, one-to-many and many-to-many modes.

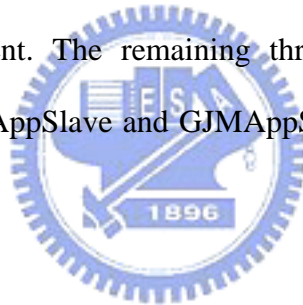
4.1.3. Application Runtime Layer

A GJMAServer can serve many GJMClients at the same time. Also, the GJMClient can access several GJMApps run on the GJMAServer at the same time if the GJMClient is a GJMABrowser or a built-in browser. The Application Manager (AppMngr) is responsible for loading, resuming and stopping GJMApps. Before starting a GJMApp, AppMngr will check if

any instance of the GJMApp already exists in the memory. If it does, AppMngr will then check the startup setting of the GJMApp and decide to create a new instance or bind the GJMClient to the old one. This is useful when a network is temporarily broken. When the GJMClient re-connects to the GJMAServer, previous work can continue. AppMngr uses different class loader instances to load a GJMApp every time to maintain independent space between them. This lets every GJMApp have its own space. How to use different class loader instances to load class is explained in section 5.1.

4.2. GJMClient Architecture

There are four kinds of GJMClient: built-in browser, GJMABrowser, GJMAppSlave, and GJMAppStandalone. It must be noted that the first one can be used by both programmable and non-programmable GJMClient. The remaining three can only be used by programmable GJMClient. Moreover, GJMAppSlave and GJMAppStandalone are generated from original GJMApp when deployment.



GJMABrowser architecture is similar to the GJMAServer as Figure 4-3 illustrates. Because a GJMABrowser can communicate with only one GJMAServer at a time, there are only a couple of InputD and OutputD in the Adaptive Transport Layer. InputD always listens for an arriving GJMAMesgs; if it gets any, it will put the message to the InputQ. At the same time, Message Handler (MesgHandler) retrieves messages from InputQ asynchronously and passes them to the Command Manager (CmdMngr) or Window Manager (WinMngr). The functionality of WinMngr is to manage widgets created on the GJMABrowser. CmdMngr plays almost the same role that the AppMngr on GJMAServer does, but it does not physically load or stop GJMApp instances. It only issues those requests to the GJMAServer and waits for the results.

Every widget has an associated listener. Whenever the status of a widget is changed by its user,

the listener is triggered and generates some corresponding GJMAMesgs. WinMngr then puts these GJMAMesgs into OutputQ, and OutputD will later send them to the GJMAServer.

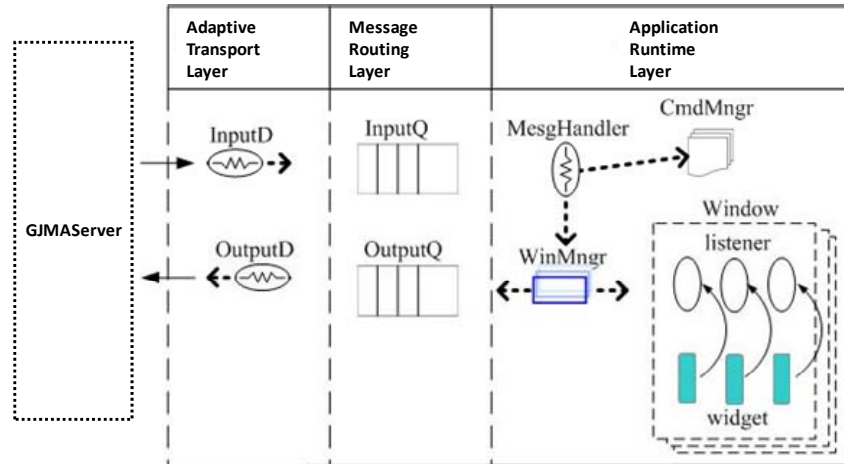


Figure 4-3. The layered architecture of GJMABrowser.

4.3. Initialization Process within GJMApp

Every GJMApp has to be initialized prior to use, and these initialization process is taken care by the constructor of the class `org.gjma.application.GJMApp`. Moreover, different running modes may need different initialization process, and this section will discuss them.

4.3.1. GJMApp in GJMAServer

A GJMApp requires different components when running in different modes on a GJMAServer. A GJMApp deployed in the BROWSER mode needs a window manager (WinMngr) to manage all objects related to user interface. A GJMApp deployed in the MASTER-SLAVE mode needs an object manager (ObjMngr) for both parts to manage remote objects. These necessary components are initialized by the `org.gjma.application.GJMApp` constructor. Hence, when loading a GJMApp, the constructor will be invoked and the necessary components will be initialized automatically. The following content discusses the initialization process of a GJMApp in different running modes.

- **GJMAApp in the BROWSER mode**

The GJMAApp class constructor has to create a window manager (WinMngr) and the queue AppInQ is assigned by AppMngr.

- **GJMAApp in the STANDALONE mode**

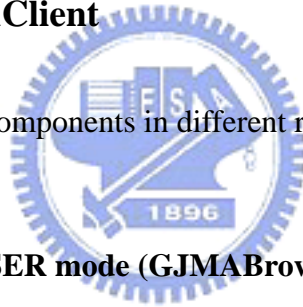
All codes within the GJMAApp are handled by the GJMClient. The GJMServer has nothing to do.

- **GJMAApp in the MASTER-SLAVE mode (GJMAAppMaster)**

The GJMAApp class constructor has to create an object manager (ObjMngr) and the queue AppInQ is assigned by AppMngr.

4.3.2. GJMAApp in GJMClient

A GJMAApp requires different components in different running modes in a GJMClient also.



- **GJMAApp in the BROWSER mode (GJMABrowser and built-in browser)**

All codes within the GJMAApp are handled by the GJMServer. The GJMClient has nothing to do.

- **GJMAApp in the STANDALONE mode (GJMAAppStandalone)**

The GJMAApp has an empty constructor because it does not need to communicate with the GJMServer.

- **GJMAApp in the MASTER-SLAVE mode (GJMAAppSlave)**

The GJMAApp constructor will build a three-layer architecture like the GJMABrowser, as Figure 4-3 shows. The only difference is in the Application Runtime Layer. In this mode, this layer contains an ObjMngr instead of the WinMngr.

4.4. Computing Model Adaptation Mechanism

Three running modes described in section 3.3 are supported in GJMA and every GJMAApps can be adapted to one of the three modes automatically depending on situations in deployment time. In development time, GJMAApp developers do not need to worry about which computing model is used and do not need to write any interface description files such as CORBA IDL [47][48]. In other words, To program a GJMAApps is similar to program a Java ME MIDP application. Moreover, which running mode can be used is determined in deployment time and all necessary transformations are taken care by GJMA. According to the above descriptions, a computing model adaptation mechanisms has to be designed in GJMA. It can tailor GJMAApps to fit one of the three running modes. This subsection introduces how a GJMAApp can be automatically adapted to the three running modes : STANDALONE, BROWSER and MASTER-SLAVE, respectively.



4.4.1. Adapt to the STANDALONE Mode

To tailor a GJMAApp to fit the STANDALONE mode is easy. Just replace some classes with the most suitable classes. In GJMA, there may be many classes having the same class name but they have different implementations for different purposes. For example, there are several implementations for org.gjma.application.GJMAApp class, placed in different directories, and each of them has different initialization process for different running modes as sections 4.3 discusses. In other words, every org.gjma.application.GJMAApp class implementation is suitable for a specific running mode and situation (ex. master or slave part). Hence, if a GJMAApp is determined to deploy as the STANDALONE mode, the computing model adaptation mechanism has to choose an org.gjma.application.GJMAApp class file implemented for the STANDALONE mode from all of them. In addition, in sometimes, the computing model adaptation mechanism has to consider about other criteria, such as execution

environment of end-devices, screen size of end-devices, computing power of end-devices, and so on. For instance, there are two implementations for the `org.gjma.ui.LayoutManager` class. One is implemented for big screen and the other is implemented for small screen. It follows from what has been said that each class may need different criteria and the computing model adaptation exploits different decision trees for different classes when choosing. Furthermore, the decision trees are described in the class profile in the end-device database. In the STANDALONE mode, all codes within a GJMAApp are executed in end-devices. It must be noted that not all end-devices are good enough to execute it, so not all end-devices can run all GJMApps in the STANDALONE mode.

4.4.2. Adapt to the BROWSER Mode

To tailor a GJMAApp to fit the BROWSER mode is same as to tailor a GJMAApp to fit the STANDALONE mode. The only difference is that the computing model adaptation mechanism may choose different class implementations because of different running mode and situation. Take `org.gjma.ui.Canvas` class implementations for example. The implementation for BROWER mode differs from the implementation for the STANDALONE mode. In execution time, the former only generates internal data structures, which are used to transform user interface to specific content formats such as HTML, and the latter will call practical API to draw directly. Consequently, the computing model adaptation mechanism may choose different implementation for different running modes. In BROWER mode, all codes within a GJMAApp are executed in GJMAServer and the majority part of end-devices can access all GJMApps in the BROWSER modes.

4.4.3. Adapt to the MASTER-SLAVE Mode

To tailor a GJMAApp to fit the MASTER-SLAVE mode is a little complicated. Generally, a Java application is consisted of classes and all the classes will be executed by the same host. The

STANDALONE and the BORWSER mode keep this characteristic but the MASTER-SLAVE mode does not. In the MASTER-SLAVE mode, because some codes within a GJMAApp cannot be executed in some end-devices, GJMAServer has to help the end-devices handle these codes. In this situation, the codes within the GJMAApp have to be divided into two parts. One part, called slave part, is handled by the end-device and the other part, named master part, is handled by the GJMAServer. The question then arises about how to automatically divide a GJMAApp into two parts without bothering GJMAApp developers and both parts can cooperate with each other in run-time just like running in the same host. The divided strategy can be fine-grained (ex. method) or coarse-grained (ex. class). To simply the problem, the computing model adaptation mechanism chooses the latter and the minimum dividable unit in GJMA is class file. Hence, the reduced problem is how to segment class files within a GJMAApp into two parts: master and slave.



Before discussion, some terminologies are defined first. They are defined one by one as follows:

- **remote class**

A class is called a remote class if the class is placed in a remote host. The terminology remote represents an opposite relationship. Hence, in the master part's viewpoint, the classes placed in the slave part are called remote class. Moreover, in the slave part's viewpoint, the classes placed in the master part are called remote class too.

- **remote method**

A method is called a remote method if the method belongs to a remote class.

- **managed class**

A class is called a managed class if the class can be replaced by GJMA. It implies an action acted on its instance can be directly intercepted by GJMA.

- **managed object**

An object is called a managed object if the object is the instance of a managed class.

- **complementary object**

In Java, an object is an instance of a class and it is an individual unit of run-time data storage. It implies that an object only contains data and the practical method codes are contained in the class. To be precise, an object is initialized from a class and the class's superclass according to Java inheritance relationship, and the methods associated to the object are contained in the class as well as the class's superclass. Because the class and the class's superclass may be placed in different part in the MASTER-SLAVE mode, an object created from the class is physically divided into two parts. In this case, the run-time data storage of the object and the method codes associated to the object are spread in the two parts' virtual machine as Figure 4-4 shows. An object is called a complementary object if the object logically represents the same object in the other part. In other words, an object and its complementary object must have the same object id. Moreover, a complementary object has two main missions. The first mission is to make the data storage of a logical object can be spread in two parts. The second mission is to make both virtual machines have the same object view. In other words, the both two virtual machines are capable of locating and accessing the same object logically. To achieve the second mission, a corresponding complementary will be created automatically by GJMA in the other part when an managed object was created.

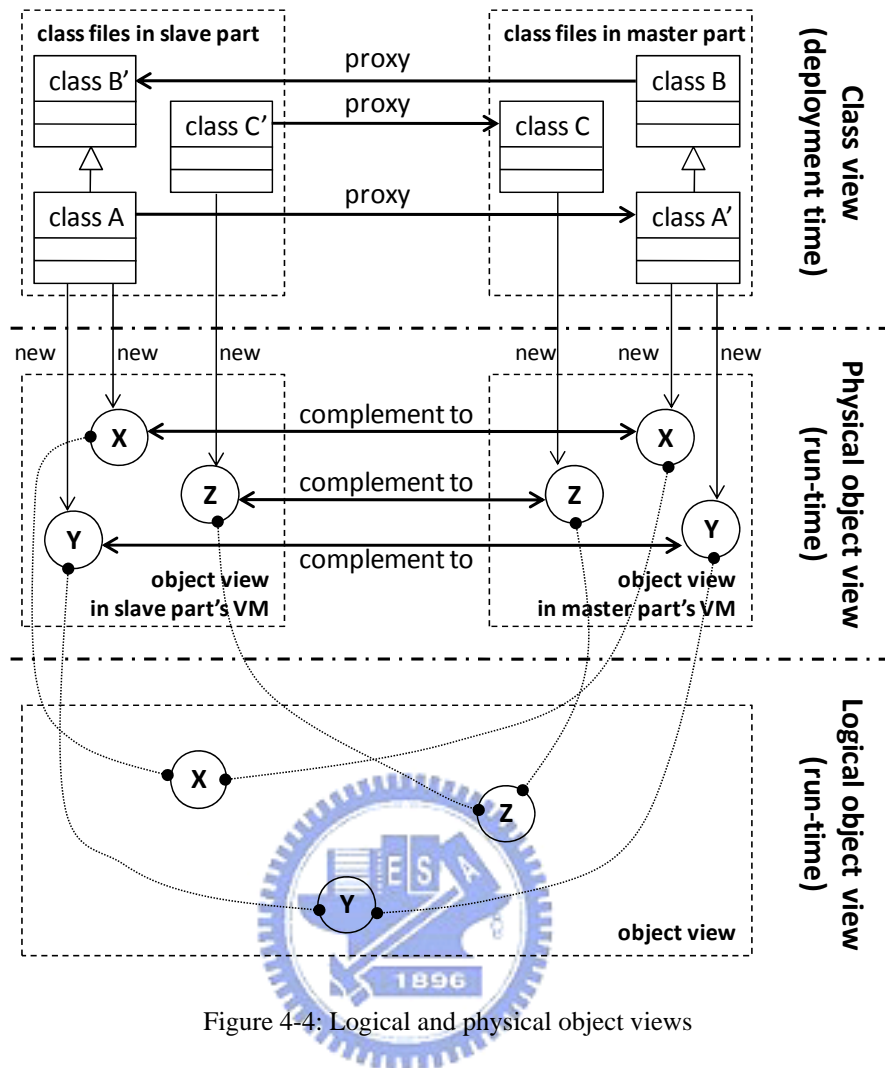


Figure 4-4: Logical and physical object views

In order to make the both parts cooperate with each other, three problems must be solved. The first problem is how to intercept all actions which act on remote classes or remote methods. The second problem is how to reflect these intercepted actions on the corresponding remote classes or remote methods. The third problem is how to create the corresponding complementary object when a managed object was created.

4.4.3.1. How to Intercept Invocation Actions

GJMA exploits proxy design pattern [31] as Figure 4-5 illustrates to solve the first problem. Because many end-devices are JavaME-enabled and Java ME does not support dynamic class loading [63], GJMA generates proxy classes before run-time. Every proxy class has the same

class name, skeleton and inheritance relationship as the original class, but there are no fields in the proxy class as Figure 4-10 shows. Moreover, the codes within a class and its corresponding proxy class are different. The former is practical business logic and the latter is responsible for delegating intercepted actions to the corresponding object managers in the other part. After proxy classes are generated, each original class which is determined to be placed in the other part will be replaced with the corresponding proxy class. Then, actions want to act on remotes classes or remote methods will be intercepted by these proxy classes. When a method within a proxy class is called, the codes within the method are run as the following steps:

1. Encode action type, target object id, method number, and all parameters into an action string. Then, fill the action string into the GJMAMesg. The marshalling details are discussed in section 5.3.1.
2. Deliver the GJMAMesg to the object manager in the remote host and wait until the result returns.
3. Decode results into original return type and return it to the caller. The unmarshalling details are discussed in section 5.3.2.

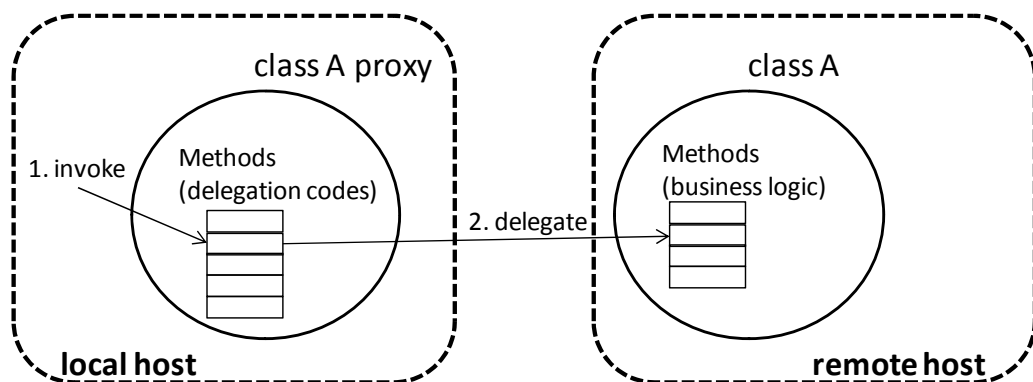
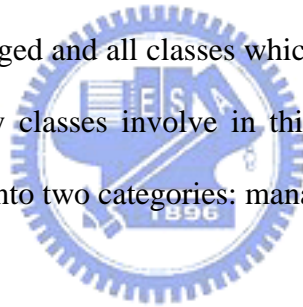


Figure 4-5: Proxy design pattern

On the other hand, because all codes within proxy classes are only executed when the methods were called, the proxy class only can be used to intercept method invocation action. However,

there are other actions having to be intercepted, such as field manipulation action, synchronized action and so on. To intercept all possible actions, the original classes within a GJMAApp have to be modified first. The idea is to convert all actions to method invocation actions and a class modification process is designed to do this. Nevertheless, it must be noted that not all class modifications are useful, because the built-in classes, such as java.lang.String, have the highest class loading sequence. For example, there are two java.lang.String class implementations. One is built-in class placed in rt.jar and the other is modified one placed in somewhere. In the case, both have the same class name but the former is always loaded when JVM requires java.lang.String class. For the reason that the former has higher class loading sequence than the latter has. To sum up, the modified classes for these built-in classes are never loaded and these modifications are useless. To solve the problem, the class names of the modified classes for the built-in classes have to be changed and all classes which references to the modified class have to be changed also. Too many classes involve in this changes and it is hard to complete. Hence, GJMA divides classes into two categories: managed class and un-managed class.



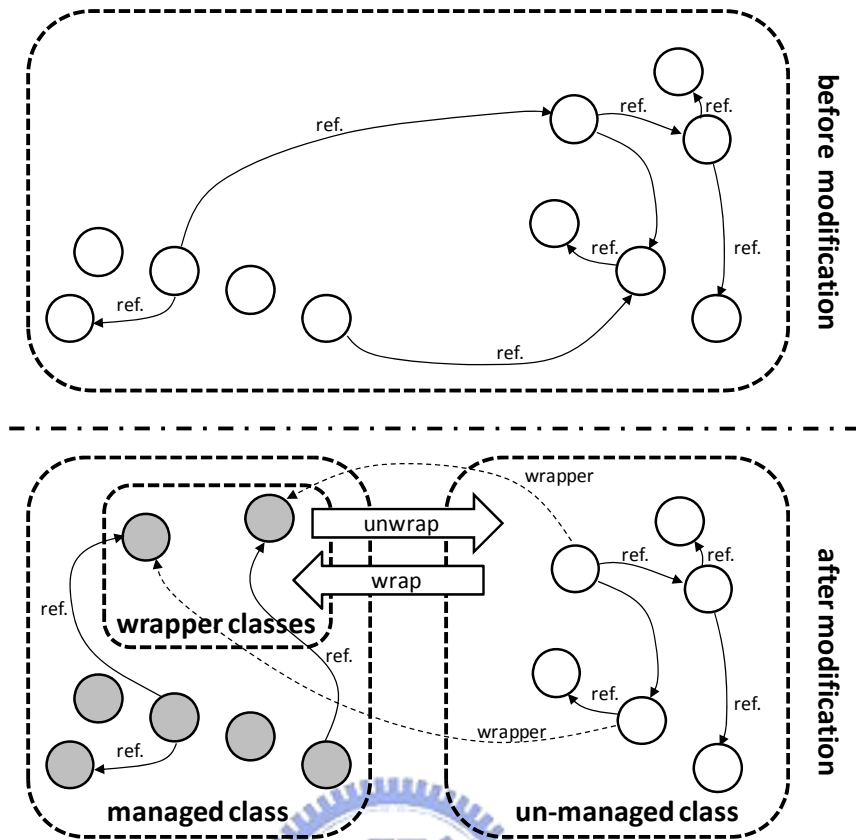


Figure 4-6: The relationship between managed and un-managed class

Moreover, GJMA only modifies classes belonging to managed class and the wrapper design pattern [31] as Figure 4-7 shows is used to bridge between both. Instead of modifying un-managed classes, just generate wrapper classes to wrap them. After modification, all codes referencing to un-managed classes in the managed class are replaced to reference to the corresponding wrapper classes and the codes in the un-managed class are never modified as Figure 4-6 illustrates.

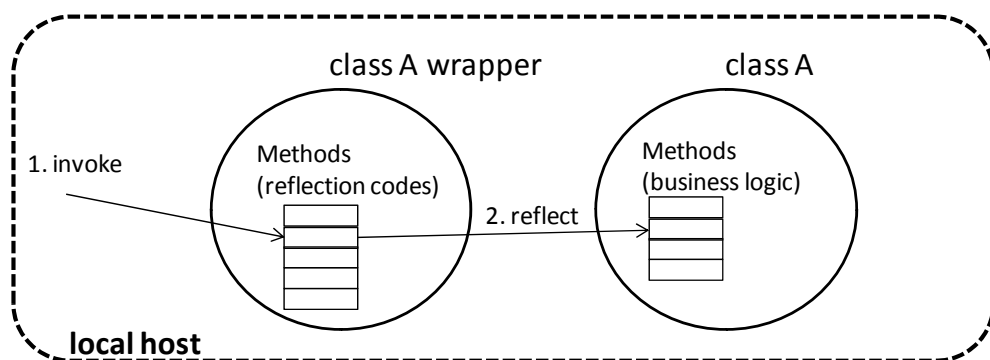


Figure 4-7: Wrapper design pattern

4.4.3.2. How to Reflect Intercepted Actions

Because Java ME does not support Java reflection [64], the GJMA framework generates an object manager class, named `ObjMngr`, for a `GJMAApp` to solve the second problem. All `ObjMngr` classes are responsible for delegating actions to the corresponding classes or methods. Because of lacking Java reflection mechanism, all methods which can be called by an `ObjMngr` have to be determined in deployment time. Thus, a method table is hard-coded in every `ObjMngr` class and the table is generated in deployment time. In run-time, when receiving a command from proxy classes on the other side, the `ObjMngr` will traverse into the method table and invoke the corresponding methods. The steps are as follows:

1. (Unmarshaling) Decode action string in the received `GJMAMesg`.
2. Traverse into the method table according the method number in the action string.
3. Invoke the corresponding methods within the practical object or class.
4. (Marshaling) Encode the results into an action string.

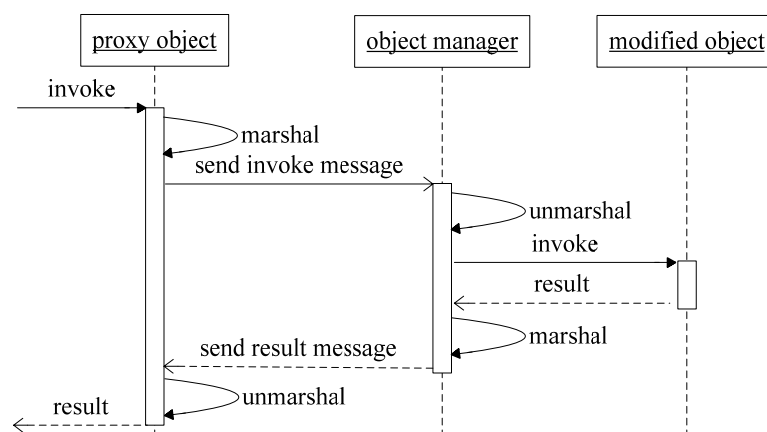


Figure 4-8: The sequence diagram for the proxy class.

4.4.3.3. How to Create Complementary Objects

In order to solve the third problem, instance creation actions have to be intercepted. To do this, GJMA has to find a hook point in an object initialization path and then inject some interception-related codes into it. It is desirable to describe some Java characteristics before moving on to the main topic. Java guarantees that the constructor method of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. In other words, there is a constructor chaining when creating an object. Because the top class in every constructor chaining is always the class `java.lang.Object`, which is the root of the class hierarchy as Figure 4-9 shows, and it has only one constructor `Object()`, the constructor is always called when an object is created. If it is possible to inject interception-related codes into the constructor `Object()`, then instance creation actions can be intercepted. However, there is no way to replace the built-in class `java.lang.Object` as the previous discussion. Hence, GJMA exploits another way to do this. GJMA modifies the original inheritance relationships as Figure 4-9(a) shows by inserting a class `org.gjma.application.GJMAObject` as Figure 4-9(b) shows. The superclass of the class `org.gjma.application.GJMAObject` is the class `java.lang.Object`. After the insertion, the constructor within the class `org.gjma.application.GJMAObject` will be called when every managed object is created and then the interception-related codes can be put in the constructor.

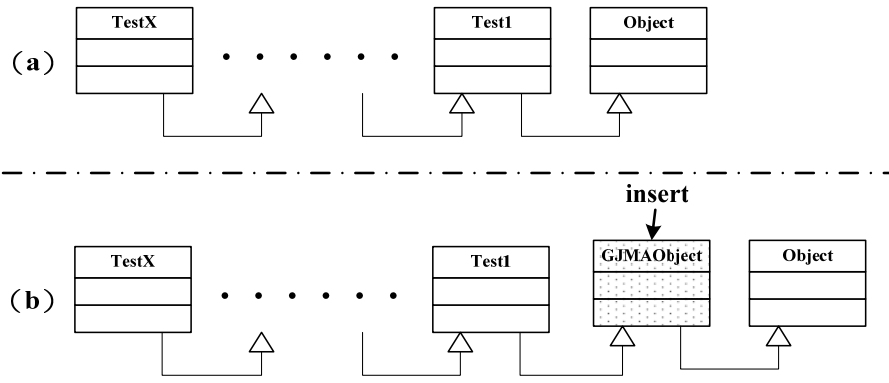


Figure 4-9: Insert GJMAObject in the inheritance chaining

When the interception-related codes are executed, it implies a managed object is being created and the codes respond to send a creation command encoded in a GJMAMesg to the other part's object manager. When the object manager receives the GJMAMesg, the object manager will create the corresponding complementary object. After the complementary object is built, some initialization codes belonging to the other constructors in the constructor chaining will be executed to complete all necessary initializations. The following serves as an example.

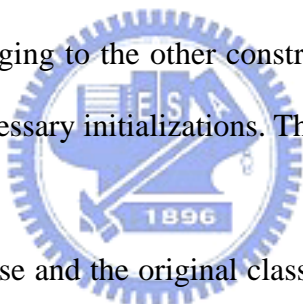


Figure 4-10(a) is the general case and the original classes are placed on the same host. Figure 4-10(b) is the case after a deployment and the original classes are spread into two parts. In order to make the discussion clear, some fields and methods are ignored. In Figure 4-10(a), if ObjectX is created from ClassB, ObjectX will have three fields: field1, field2 and field3. This obeys inheritance associations. If ClassA and ClassB are placed on the same host, there is no trouble. In Figure 4-10(b), ClassA on HostB (GJMAMClient) is replaced by its proxy class and ClassB on HostC (GJMAMServer) is also replaced by its proxy class. In Figure 4-10(b), if ObjectY is created from ClassB, ObjectY will have only two fields: field2 and field3, because of no fields in proxy classes. According to the previous definition, the missed field field1 should be in its complementary object in the other part. In the example, the complementary object of ObjectY will be created from ClassB on HostC, and it will have one field: field1. This means that a logical object may be divided into two parts physically. Traditionally, when an

object is created from a class, its constructor will be called and the constructor will call the constructor of the superclass recursively. For example, in Figure 4-10(a), when an object is created from ClassB, the constructor chaining is executed as following orders:

1. Constructor ClassB() is called and it calls the constructor ClassA().
2. Constructor ClassA() is called, and it calls the constructor Object().
3. Constructor Object() is called and it is the last constructor to be called. It will create the object instance and then return.
4. Return to constructor ClassA(). The field1 is initialized here and the remaining initialization codes in the constructor are executed. After completion, the constructor returns.
5. Return to constructor ClassB(). The field2 as well as field3 are initialized here and the remaining initialization codes in the constructor are executed. After completion, the constructor returns and the object initialization is completed.



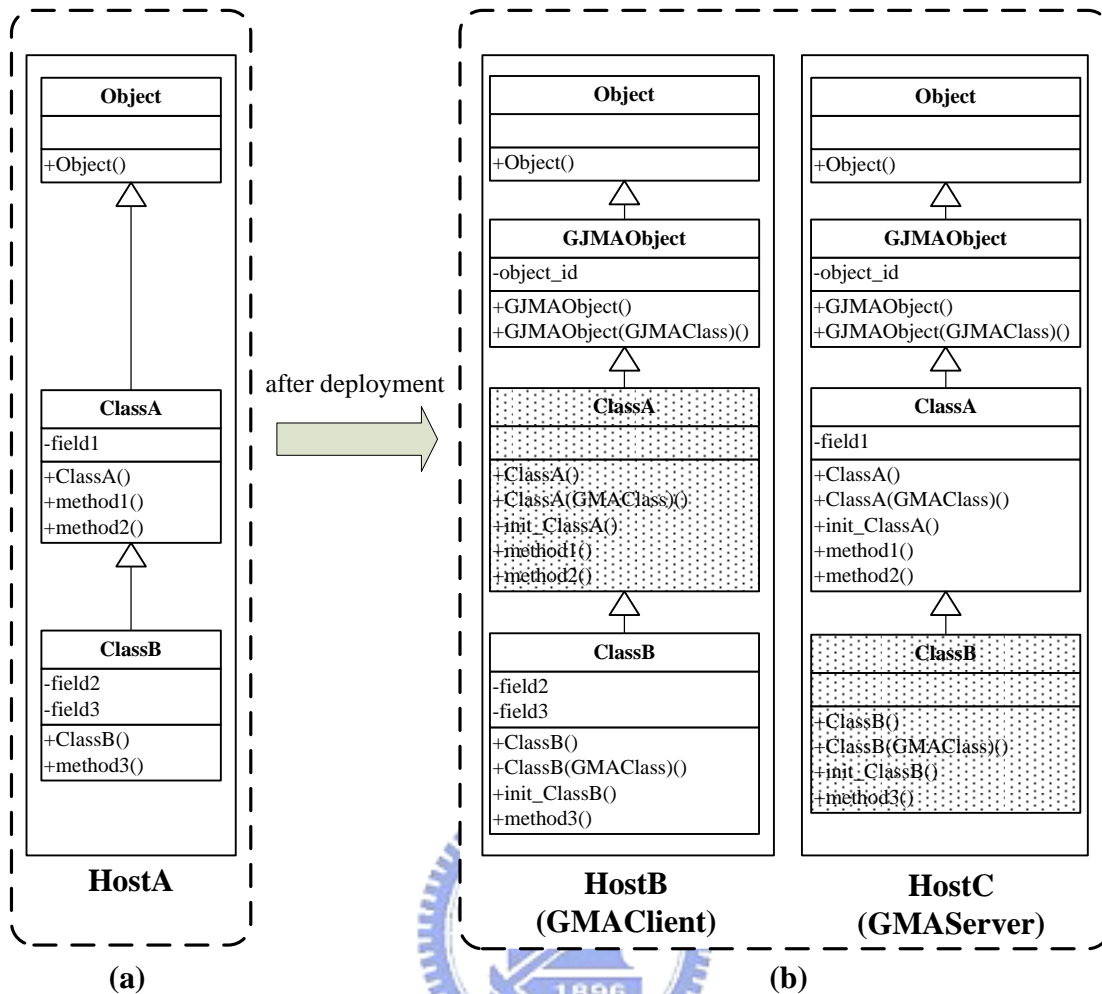


Figure 4-10: Separate two associated classes into two different hosts.

We should notice that the initialization order has to be kept after replacing some classes with proxy classes. Figure 4-10 (b) is an example after a deployment. When an object is created from ClassB on HostB (GMAClient), the initialization order is as follows: (The paragraph starting with **Arabic numerals** is used to describe the actions taken place on HostB. The paragraph starting with **Roman numerals** is used to express the actions taken place on HostC.)

1. Constructor ClassB() is called and it calls the constructor ClassA().
2. Constructor ClassA() is called, and it calls the constructor GJMAObject().
3. Constructor GJMAObject() is called and it calls the constructor Object().
4. Constructor Object() is called and it is the last constructor to be called. It will create

the object instance and then return.

5. Return to constructor GJMABject (). The codes will send a creation command to the object manager on HostC (GJMAServer).

I. The object manager on HostC receives the command. It calls constructor ClassB(GJMApp) to create the complementary object.

II. Constructor ClassB(GJMApp) is called and it calls the constructor ClassA(GJMApp).

III. Constructor ClassA(GJMApp) is called and it calls the constructor GJMABject(GJMApp).

IV. Constructor GJMABject(GJMApp) is called and it calls the constructor Object().

V. Constructor Object() is called and it is the last constructor to be called. It will create the object instance and then return.

VI. Return to constructor GJMABject(GJMApp). The codes will set object id and then return.

VII. Return to constructor ClassA(GJMApp). Do nothing and return directly.

VIII. Return to constructor ClassB(GJMApp). Do nothing and return directly. The empty complementary object is completely created.

IX. Return to the object manager. It will store the relationship between the object id and the complementary object reference and then send result to HostB.

6. Return to constructor GJMABject(). It will store the relationship between the object id and the object reference and then return.

7. Return to constructor ClassA(). Because ClassA is a proxy class, it will invoke a remote method init_ClassA(). The field1 will be initialized in its complementary object on HostC.

8. Return to constructor ClassB(). ClassB is not a proxy class so the field2 and field3 will be initialized here by invoking the method init_classB(). After completion, the constructor

returns and the object initialization is completed.

When an object is created from ClassB on HostC (GJMAServer), the initialization order is as follows: (The paragraph starting with **Arabic numerals** is used to describe the actions taken place on HostC. The paragraph starting with **Roman numerals** is used to express the actions taken place on HostB.)

1. Constructor ClassB() is called and it calls the constructor ClassA().
2. Constructor ClassA() is called, and it calls the constructor GJMAMObject().
3. Constructor GJMAMObject() is called and it calls the constructor Object().
4. Constructor Object() is called and it is the last constructor to be called. It will create the object instance and then return.
5. Return to constructor GJMAMObject(). The codes will send a creation command to the object manager on HostC (GJMAServer).
 - I. The object manager on HostB receives the command. It calls constructor ClassB(GJMApp) to create the complementary object.
 - II. Constructor ClassB(GJMApp) is called and it calls the constructor ClassA(GJMApp).
 - III. Constructor ClassA(GJMApp) is called and it calls the constructor GJMAMObject(GJMApp).
 - IV. Constructor GJMAMObject(GJMApp) is called and it calls the constructor Object().
 - V. Constructor Object() is called and it is the last constructor to be called. It will create the object instance and then return.
 - VI. Return to constructor GJMAMObject(GJMApp). The codes will set object id and then return.
 - VII. Return to constructor ClassA(GJMApp). Do nothing and return directly.

VIII. Return to constructor ClassB(GJMAApp). Do nothing and return directly. The empty complementary object is completely created.

IX. Return to the object manager. It will store the relationship between the object id and the complementary object reference and then send result to HostB.

6. Return to constructor GJMAObject(). It will store the relationship between the object id and the object reference and then return.

7. Return to constructor ClassA(). ClassA is not a proxy class so the field1 will be initialized here by invoking the method init_classA(). After completion, it returns.

8. Return to constructor ClassB(). Because ClassB is a proxy class, it will invoke a remote method init_ClassB(). The field2 and field3 will be initialized in its complementary object on HostB. After completion, the constructor returns and the object initialization is completed.

It will be clear from these examples that wherever the object is created on, the initializations orders are similar to the original initialization orders but the original object is divided into two objects (an object and its complementary object) physically. So far, we have seen that creating an object from ClassB on HostB is equivalent to creating an object from ClassB on HostC.

4.4.4. Deployment Process

According to the above discussion, there are three main functionalities in computing model adaptation mechanism: class replacement, class modification and class generation. Thus, a deployment process is designed to apply the computing model adaptation mechanism. Different running modes require different deployment processes, as Figure 4-11 shows. Three important components participate in the deployment process. They are preprocessor, the analyzer and the deployer.

The preprocessor modifies the bytecodes within original classes and generate some wrapper

classes for the un-managed classes. Every modified class and wrapper class is equivalent to the original class, but bytecodes have small differences. Moreover, the analyzer generates proxy classes and ObjMgr classes by analyzing the class file. Both preprocessor and analyzer exploit a class file manipulation tool, BCEL [62], to handle all class file modifications, including bytecode modifications. The deployer packages necessary classes together. It will lookup end-device database when deployment. The end-device database is consisted of two XML documents : device profile and class profile. The former describes capabilities about end-devices and the latter describes the requirements of classes. The deployer use information in end-device database to choose suitable classes which can be original or generated classes.

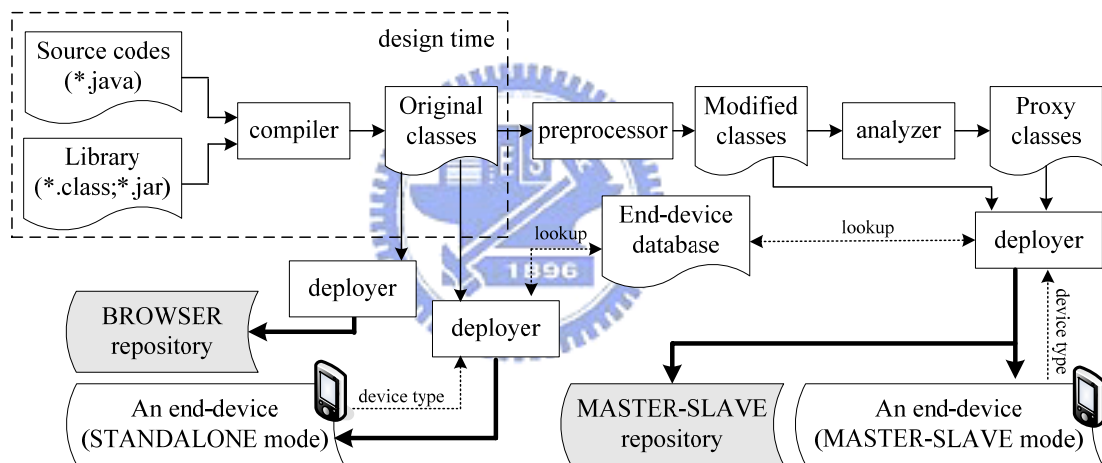


Figure 4-11. The GJMAApp development flow.

The following discussion contains some notations which are defined as follows. If a GJMAApp named APP1 has only three classes, X, Y and GJMAApp, it is represented by the notation $APP1=\{X, Y, GJMAApp\}$. Moreover, notation X' is used to represent the corresponding proxy class of class X. Because some classes provided by GJMA have more than one implementation, the implementation name is denoted by a superscript lowercase letter a, b, c and so on, to distinguish them. For example, X^a and X^b represent two different implementations for class X. In other words, X^a and X^b have the same interface and functionalities, but have different

implementations. Because X^a and X^b have the same interface, they will have the same corresponding proxy class X' .

A GJMAApp can be deployed in three modes. Different modes or runtime environments may require different GJMA class implementations, and deployers are responsible for choosing correct and suitable classes by searching the end-device database. $APP2=\{X, Y, Canvas, GJMAApp\}$ is used as an example in the following. $Canvas^a$ is an implementation for BROWSER mode. $Canvas^b$ is another implementation for the STANDALONE and the MASTER-SLAVE mode. $GJMAApp^a$ is an implementation for the BROWSER mode. $GJMAApp^b$ is an implementation for the STANDALONE mode. $GJMAApp^c$ is an implementation for the slave part and $GJMAApp^d$ is an implementation for the master part in the MASTER-SLAVE mode.

If an application is deployed in the BROWSER mode, it does not need to be modified. Just put the original class file within the application to BROWSER repository, because the default implementation of GJMA classes is just for the BROWSER mode. GJMA classes for the BROWSER mode will intercept all user interface actions and translate them into the corresponding GJMAMesgs. If APP2 is deployed in the BROWSER mode, the result is $APP2=\{X, Y, Canvas^a, GJMAApp^a\}$.

If an application is deployed in the STANDALONE mode, it also does not need to be modified. The deployer will replace some GJMA classes with the correct class implementations. If APP2 is deployed in the STANDALONE mode, the result is $APP2=\{X, Y, Canvas^b, GJMAApp^b, Loader\}$. Loader is a frontend program to load the GJMAApp.

If an application is deployed in the MASTER-SLAVE mode, the application must be modified first, and then generate corresponding proxy classes and ObjMngr classes. In this mode, the

application is divided into two parts, and the deployer packages them. In the slave part, the deployer will insert the ObjMngr class and replace some GJMA classes with the correct class implementations. The deployer will also replace some classes which cannot be executed by end-devices with the corresponding proxy class. In the master part, the deployer will also add the ObjMngr. If the slave part contains class X, the deployer will choose X' to package the master part. If the slave part contains class Y', the deployer will choose Y to package the master part. The master part and slave part are complementary. Furthermore, the class which can be executed on end-devices depends on the end-device database. If APP2 is deployed in the MASTER-SLAVE mode and Y cannot be processed by the end-device, the results are SLAVE={X, Y', Canvas^b, GJMAApp^c, ObjMngr, Loader} and MASTER={X', Y, Canvas', GJMAApp^d, ObjMngr}.

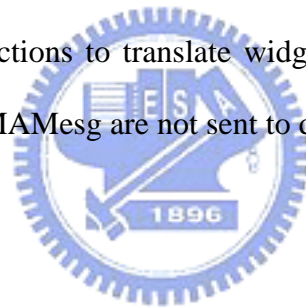
4.5. User Interface Adaptation Mechanism

In the STANDALONE and the MASTER-SLAVE modes, all classes related to user interface are executed on end-devices. In these two modes, user interface adaptation is achieved by replacing user interface related classes within the application with suitable ones according to the class profile in the end-device database. The remaining content in this section will focus on user interface adaptation for the BROWSER mode, in which all classes related to user interface are executed on GJMAServer.

A GJMAApp developer does not need to know what kind of GJMClient the application serves. When a GJMAApp runs in the BROWSER mode, it handles and generates GJMAMesgs regardless of the client type. To serve built-in browsers, the GJMA framework needs a mechanism to translate GJMAMesgs to other content formats, such as WML or HTML, and this mechanism is implemented in adaptive transport layer.

Section 4.1.1 introduces the CommMngr. One of the CommMngr's missions is to create a user process for a client. When a client sends a login request to the GJMAServer, CommMngr will know what kind of end-device it serves. If the client is a built-in browser, a translator component within the user process is combined with many convert functions and a layout manager.

A layout manager is used to intercept GJMAMesgs transferred to a GJMClient. When receiving a GJMAMesgs, the layout manager processes the GJMAMesgs and keeps widgets in tree structures, as Figure 4-12 shows. The layout manager knows how many and what kinds of widgets are created by the GJMAApp based on the trees. In addition, the layout manager can arrange widget positions. Another function of the layout manager is to traverse the tree and call the corresponding convert functions to translate widgets to specific content formats. When serving a built-in browser, GJMAMesg are not sent to directly to the client, but handled by the layout manager.



Every convert function has a different capability to translate a UI widget in the tree structure to a corresponding widget of other content formats. Table 4-1 is a widget mapping among tree element, WML and HTML.

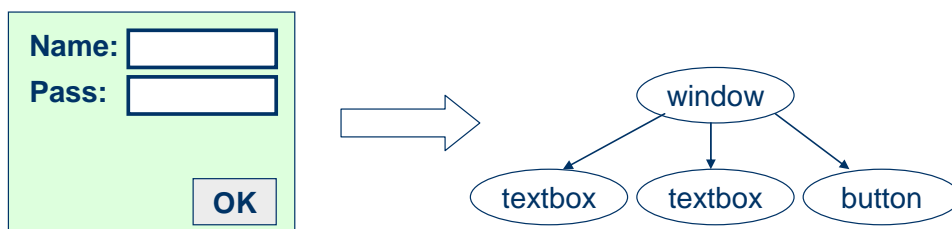


Figure 4-12: A tree structure about user interface

Table 4-1: The mapping table among tree element, WML and HTML

Tree element	WML	HTML
Window	<wml>+<card>	<body>
canvas	<wml> + <card> + 4 direction button+	<body> + 4 direction button +
listbox	<select>	<select>
button	<a href>	<a href>
textbox	<input>	<input>

4.6. Network Adaptation Mechanism

Because of the diversity of mobile devices, network capabilities are not always the same. Some devices support TCP but some devices may support only HTTP. In order to hide these details from mobile application developers, the GJMAServer exploits a loosely coupled design, and the network adaptation is achieved by asynchronous message routing layer as well as adaptive transport layer. In adaptive transport layer, it defines a unified transport interface and all implementation complied with the interface can be plugged into GJMA framework easily. Currently, The GJMA framework supports TCP, UDP, and HTTP.

Chapter 5 GJMA Implementation Issues

5.1. GJMClassLoader

In GJMAServer, all GJMApps are loaded by the same Java Virtual Machine (JVM). It can reduce resource consumption but may cause some problems. These GJMApps can be written by different developers. Hence, these GJMApps may contain some different classes with the same class name. Also, the same GJMApp may be loaded more than twice to serve different GJMAClients concurrently. Without independent running space, a GJMApp may inference with other GJMApps, and may cause incorrect results. An example is given to demonstrate the problems.

```
01 public interface InterfaceA {
02     public int get_next_id();
03 }
```

Listing 5-1: InferenceA source code

```
01 public class ClassA implements InterfaceA{
02
03     private static int id=0; //start with 0
04
05     public synchronized int get_next_id(){ //get next id
06         return(id++);
07     }
08 }
```

Listing 5-2: ClassA source code

```
01 URLClassLoader loader = URLClassLoader.newInstance(urls);
02
03 InterfaceA a1=(InterfaceA)loader.loadClass("ClassA").newInstance();
04 InterfaceA a2=(InterfaceA)loader.loadClass("ClassA").newInstance();
05
06 System.out.println("a1="+a1.get_next_id( )); //a1=0
07 System.out.println("a2="+a2.get_next_id( )); //a2=1
```

Listing 5-3: Use the same class loader to load ClassA twice

```
01 URLClassLoader loader1 = URLClassLoader.newInstance(urls);
02 URLClassLoader loader2 = URLClassLoader.newInstance(urls);
03
04 InterfaceA a1=(InterfaceA)loader1.loadClass("ClassA").newInstance();
05 InterfaceA a2=(InterfaceA)loader2.loadClass("ClassA").newInstance();
06
07 System.out.println("a1="+a1.get_next_id( )); //a1=0
08 System.out.println("a2="+a2.get_next_id( )); //a2=0
```

Listing 5-4: Use two different class loaders to load ClassA twice

If ClassA listed in Listing 5-2 is loaded twice by the same class loader as Listing 5-3 shows, the result is unexpected. In Listing 5-3, ClassA is loaded just one time actually and its running space is the same. Thus, calling the method associated to a1 may inference with the variable value in a2. In other words, if two more GJMApps use ClassA and they use the same class loader, these GJMApps will inference with each other. If ClassA is loaded twice by two different class loaders as Listing 5-4 shows (assume that ClassA is actually loaded by the two class loaders), a1 and a2 will have independent running space and then the code a1.get_next_id() and a2.get_next_id() both return 0. These results lead to the conclusion that GJMA has to use different class loader to create different independent running space.

Therefore, GJMA can provide independent running space for every GJMApps to prevent them from breaking in each other. In GJMA, AppMngr uses different class loader to load every GJMApps and every GJMApp will own an independent space. As a result, the classes with the same name loaded by different class loader will have different space and JVM treats them as different classes.

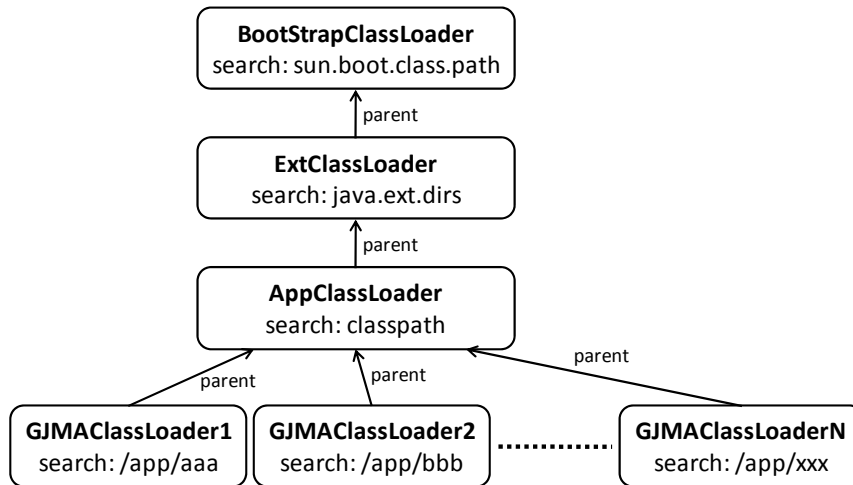


Figure 5-1: Class loader structures

It is helpful to describe how class loaders work in JVM before moving on to the main task. After JVM initialization, there are three default class loaders: Bootstrap loader, ExtClassLoader and AppClassLoader. In JVM, each class loader owns a parent class loader except bootstrap loader as Figure 5-1 shows. When a class loader tries to load a class, the load mission will be taken by its parent class first. If its parent class failed to load, the load mission is then taken by itself. Bootstrap loader is implemented by C/C++ and is used to load classes placed in the path described in parameter **sun.boot.class.path**, such as `rt.jar`. ExtClassLoader is used to load classes placed in the path described in parameter **java.ext.dirs**. AppClassLoader is used to load classes placed in the path described in parameter **classpath**. Thus, in Figure 5-1, when a GJMAClassLoader tries to load a class, the search sequence is expressed as follows.

1. Bootstrap loader will search the path described in parameter `sun.boot.class.path`. If failed, next step continues. If successful, the class is loaded here.
2. ExtClassLoader will search the path described in parameter `java.ext.dirs`. If failed, next step continues. If successful, the class is loaded here.
3. AppClassLoader will search the path described in parameter `classpath`. If failed next step continues. If successful, the class is loaded here.

4. GJMClassLoader will search the path where the GJMApp is placed in. If failed, ClassNotFoundException occurs. If successful, the class is loaded here.

If a class can be found in the first three steps, the class will be in the same running space because the class is loaded by the same class loader. Thus, all classes within GJMApps have been placed in the specific path and the path should not be described in parameters `sun.boot.class.path`, `java.ext.dirs` and `classpath`. It will guarantee that all classes within GJMApps will be loaded by GJMClassLoaders. Moreover, in GJMA, every GJMApp instance is loaded by different GJMClassLoader and each of them will own an independent running space.

According to the discussion in section 4.3, the class GJMApp is responsible for initialization. Thus, every GJMApps has to contain a GJMApp class and the class has to be loaded by GJMClassLoader. It implies the three default class loaders cannot find the class GJMApp. Hence, AppMngr loaded by AppClassLoader cannot access these GJMApp instances directly. To solve the problem, class GJMApp has to implement an interface named GJMAppInterface which can be located and loaded by the AppClassLoader and then AppMngr can control these GJMApp instances by the interface as Listing 5-5 expresses.

```
01 URLClassLoader cl= URLClassLoader.newInstance(urls);
02
03 GJMAppInterface a=(GJMAppInterface)cl.loadClass("APP1").newInstance();
04
05 a.start( ); //start the GJMApp
06 a.stop( ); //stop the GJMApp
```

Listing 5-5: Use GJMAppInterface to control GJMApps

5.2. GJMAMesg Format

There are many GJMAMesg formats in GJMA and the base format of GJMAMesg is illustrated

in Figure 5-2. In the base format, there are only two fields with fixed size and an optional type-specific payload. Whatever the running modes is, the meanings of first two fields for every GJMAMesg is the same. The length field contains the information about the total length of the GJMAMesg and its data type is a short. By reading the length field, the boundary of the GJMAMesg can be recognized. The type field contains the information about which command the GJMAMesg represents, such as create a window (in the BROWSER mode), invoke a method (in the MASTER-SLAVE mode), and so on. Moreover, the type field is also a short and the most two significant bits represent which mode the command belongs to. The remaining of the GJMAMesg is the type-specific payload. Its size and content depends on the type field. For example, the invocationId field only appears in the type-specific payload when the type field value represents a method invocation command. It must be noted that different running modes use different format to communicate. The variable GJMAMesg format makes all information useful in a GJMAMesg and it reduces the size requirement for GJMAMesgs.

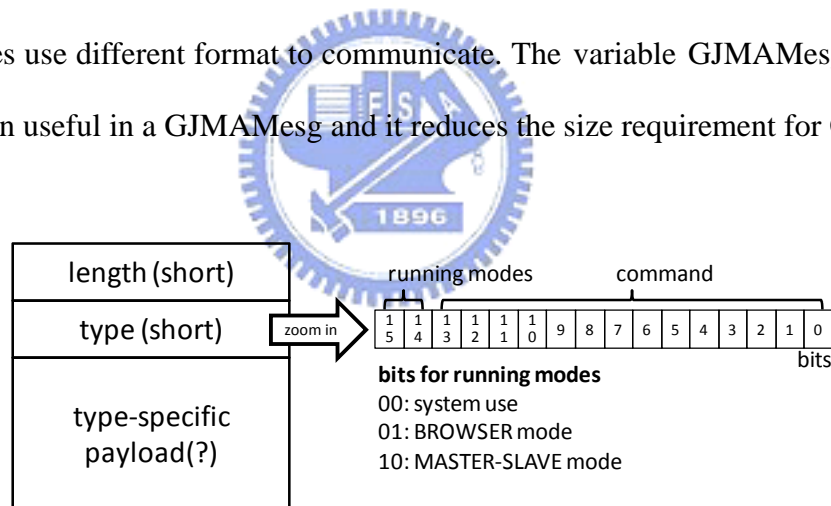


Figure 5-2: Base GJMAMesg format

5.2.1. GJMAMesg for System Use

There are several commands for system use, such as login, logout, start a GJMApp in the BROWSER mode, start a GJMApp in the MASTER-SLAVE mode, stop a GJMApp, and so on. Basically, different commands will need different values in the type-specific payload. For example, the value should be the user id and password if the command is login, the value should be the GJMApp's name if the command is start, and the type-specific should be empty

if the command is logout. In other words, GJMA will treat the type-specific payload value as different meanings according to the type field value.

5.2.2. GJMAMesg for BROWSER Mode

The number of fixed fields in a GJMAMesg for the BROWSER mode is four. The first two fields are the same as the fixed fields in the basic format. The appId (application id) field contains the id of the destination GJMAApp where the GJMAMesg will go. The widgetId field contains the id of the target widget which the command will act on. Moreover, there are also several commands for the BROWSER mode; such as create a form, create a text field, set the window title and so on. In addition, some commands needs extra parameter and the necessary parameter will be filled in the parameter field. For instance, the value in the parameter field should be the window title if the command is setting the window title.

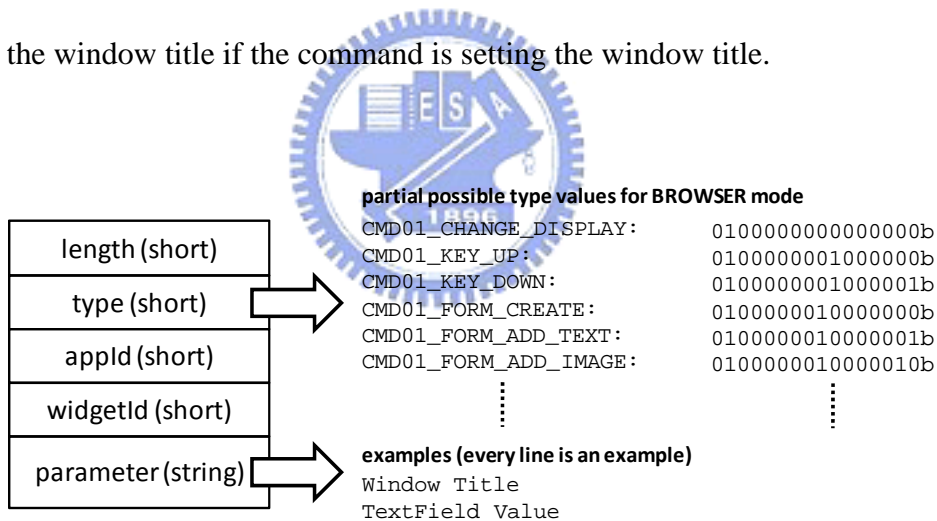


Figure 5-3: GJMAMesg format for BORWSER mode

5.2.3. GJMAMesg for MASTER-SLAVE Mode

The number of fixed fields in a GJMAMesg for the MASTER-SLAVE mode is five. The first two fields are the same as the fixed fields in the basic format. The appId (application id) field contains the id of the destination GJMAApp where the GJMAMesg will go. The invocationId field contains the id of the invocation. It is used to indicate which invocation the result is for. The threadId field contains the id of the caller thread. It indicates which thread is used to

handle the action. Moreover, there are only three commands for the MASTER-SLAVE mode. They are result, invoke and create commands. The action field contains the necessary parameter and it is a string. The details about the action field will be described in section 4.3.

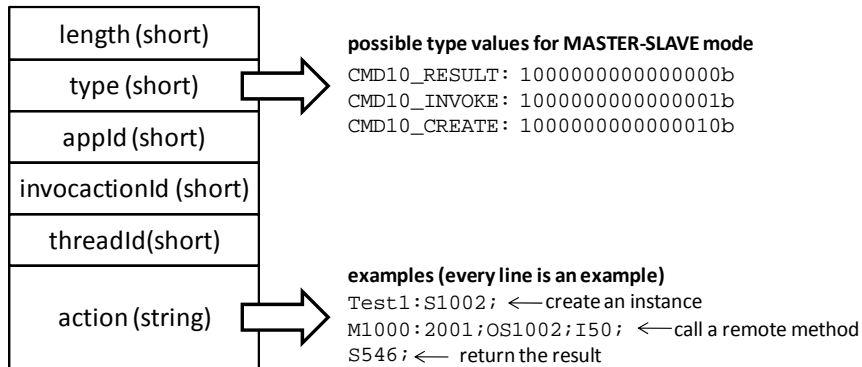


Figure 5-4: GJMAMesg format for the MASTER-SLAVE mode

5.3. Marshalling and Unmarshalling

When a remote method is called or a complementary object is created, the related parameters have to be encoded into an action string. After the remote method was completed or the complementary was created, the result is also encoded into an action string. These encoding actions are called marshalling. On the other hand, when receiving an encoded action string, ObjMngr has to decode it first and the decoding action is called unmarshalling. In the MASTER-SLAVE mode, marshalling and unmarshalling are taken place often, so a utility class named ActionBuilder is implemented to help proxy classes do this. Currently, the encoded format is plain string. In other words, all parameters and results are encoded into strings and then these strings will be filled in the action field in GJMAMesgs.

```

01 public class ActionBuilder{
02
03   public ActionBuilder(String action){//parse the encoded string}
04
05   public String toString(){//return the encoded string}
06
07   public GJMClass getClass(){
08     //return class information encoded in the action string
09     //the class information contains class name and object id
10   }
11
12   public GJMMethod getMethod(){
13     //return method information encoded in the action string
14     //the method information contains object id and method no.
15   }
16
17   public void init_create(String class_name, String object_id){
18     //start to build an action for creation
19   }
20
21   public void init_invoke(String object_id, String method_id){
22     //start to build an action for invocation
23   }
24
25   public void init_result(){
26     //start to build an action for result
27   }
28
29   public void add_int(int a){//add a new parameter with type int}
30
31   public int get_int(int index){
32     //return the parameter encoded in the action string in the order index
33   }
34 }

```

Listing 5-6: The partial source code for ActionBuilder

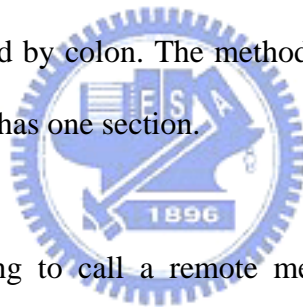
Listing 5-6 is the partial code of the class ActionBuilder. In fact, there are many methods for different parameters types. However, all methods cannot be discussed here for lack of space. Basically, there are one pair methods for every data type in the class ActionBuilder. One is used to encode and the other is used to decode.

5.3.1. Marshalling

GJMA can call methods belonging to the class ActionBuilder to build action strings on

demand. An action string is consisted of at least one section and these sections are separated by semi-colon. The first section is the most important and all action strings own this section because the action type is encoded in it. There are three possible action types: creation, invocation and result. The creation type is used when creating a complementary object. The invocation type is used when calling a remote method. The result type is used when returning a result. Moreover, these three types need different extra information when used.

When building an action string to create a complementary object, GJMA has to know the string represents a creation. Besides, GJMA has to know what class the complementary object belongs to and GJMA has to set the same object id to the complementary object. Thus, the first section in the creation action string should include the action type, class name and object id. The information is separated by colon. The method `init_create` is implemented to do this. The creation action string only has one section.



When building an action string to call a remote method, GJMA has to know the string represents an invocation. Besides, GJMA has to know the invocation is acted on what object and method. Thus, the first section in the invocation action string should include the action type, object id and method id. The information is separated by colon. The method `init_invoke` is implemented to do this. The invocation action string may have other sections if the called remote method needs other parameters. Each section represents a parameter and the sections are appended one by one after the first section. These sections can be appended by calling the methods listed in Table 5-2. Listing 5-7 shows a marshalling example.

When building an action string to return a result, GJMA has to know the string represents a return. Thus, the first section in the return action only includes the action type, so it is a fixed string. The method `init_result` is implemented to do this. The result action string should have

two sections. The second section is used to represent the result and it can be appended by calling the methods listed in Table 5-2.

Table 5-1: Methods to build the first section

string type	method	the first section format
creation	init_create	create:{class name}:{object id};
invocation	init_invoke	invoke:{object id}:{method id};
result	init_result	result;

Table 5-2: Methods to build the sections other than the first section

data type	method	appended string format
byte	add_byte	B{value};
boolean	add_boolean	Z{value};
char	add_char	C{value};
short	add_short	S{value};
int	add_int	I{value};
float	add_float	F{value};
long	add_long	J{value};
double	add_double	D{value};
string	add_string	X{value};
object	add_object	O{value};
void	add_void	V;

```

01 int sum(int a, int b){
02     ActionBuilder invoke=new ActionBuilder( ); //create an instance
03     invoke.init_invoke(object_id, 1001); //initialize invoke action
04     invoke.add_int(a); //add the first parameter into the action
05     invoke_add_int(b); //add the second parameter into the action
06     String action=invoke.toString(); //get the encoded action
07
08     //transfer the above action to the remote host and wait the result
09     //unmarshalling and return result
10 }

```

Listing 5-7: A marshalling example

5.3.2. Unmarshalling

When receiving a GJMAMesg with action filed, GJMA has to parse the action field first. The first step is to use semi-colon as delimiter to break the action string into sections. Then these sections will be processed individually and the processing results are stored in internal structures. After parsing, GJMA can get the information about the action string by calling the corresponding method listed in Table 5-3 according to the parameter types or return type. Moreover, Listing 5-8 shows an unmarshalling example.

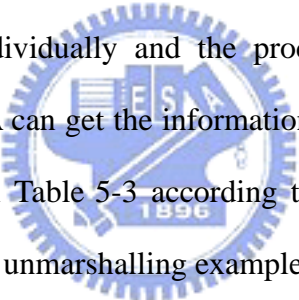


Table 5-3: Methods to get parameter

data type	method
byte	get_byte
boolean	get_boolean
char	get_char
short	get_short
int	get_int
float	get_float
long	get_long
double	get_double

string	get_string
object	get_object

```

01 ActionBuilder ab=new ActionBuilder(m.action); //parse action field
02 ab.get_int(0); //get the first parameters with type int
03 ab.get_int(1); //get the second parameters with type int

```

Listing 5-8: An unmarshalling example

5.4. GJMA Preprocessor

GJMA preprocessor has five main missions. The first mission is to generate necessary wrapper classes. The second mission is to convert all possible actions to method invocation actions. The third mission is to make classes capable of creating complementary object by modifying the original constructors. The fourth mission is to wrap some special type such as array. The fifth mission is to insert GJMAObject into original constructor chaining. All works are made before GJMA analyzer in two phases as Figure 5-5 shows and make GJMA analyzer easier to handle later. In the first phase, the input is original classes and the output is the necessary wrapper classes. In the second phase, the input is original classes as well as the wrapper classes generated in the first phase, and the output is the modified classes. In this section, how to complete the five missions are discussed respectively.

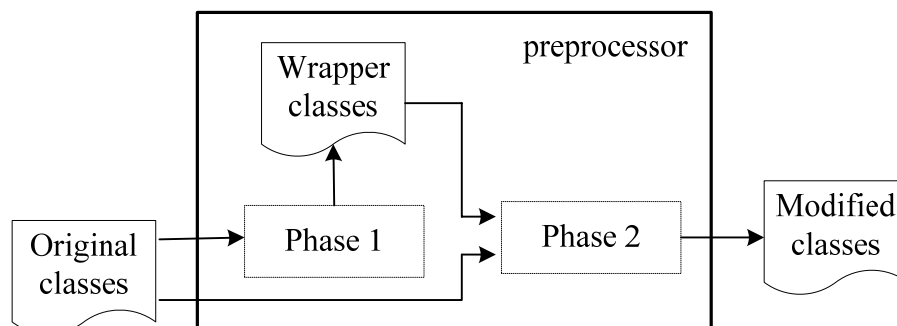


Figure 5-5: Two phases in GJMA preprocessor

5.4.1. Generate Wrapper Class

Because replacement on some classes such as built-in classes is impossible, GJMA needs to generate some wrapper classes to wrap these classes. Moreover, this work is done in the first phase and the workflow is as follows.

1. Find all used classes which cannot be modified.
2. For each class found in step 1, generate its corresponding wrapper class.

The corresponding wrapper class of a class is similar to the original class. Both have the same superclass and all methods in the original class can be found in the corresponding proxy class, but the parameter types and the return type may need to be changed because some classes cannot directly be used in the MASTER-SLAVE mode. In addition, there are two more differences between them. The codes in these methods are different and the field declarations are different also. Because GJMA cannot modify the class loading order, the wrapper class name should be different from the original class name. If both have the same class name otherwise, the wrapper class may never be loaded and all efforts to generate wrapper classes are meaningless. Hence, the different class name guarantees that the proper wrapper class will be loaded when necessary. To maintain the relationships between wrapper class and original class easier, a wrapper class naming convention is designed. The naming rule is simple. Just replace character '.' with character '_' in the original class name and then add a fixed prefix "org.gjma.wrapper.". Table 5-4 is an example.

Table 5-4: Examples for wrapper class naming

Original class name	The corresponding wrapper class name
java.lang.String	org.gjma.wrapper.java_lang_String
java.util.Vector	org.gjma.wrapper.java_util_Vector

An example class test.Foo1 is given to illustrate how to generate its corresponding wrapper class org.gjma.wrapper.test_Foo1. This work is directly made on Java bytecode level in practical but its equivalent Java source code is used to make the explanation clear. The wrapper generation steps are described as follows.

1. The wrapper class name is set to org.gjma.wrapper.test_Foo1. For source code view, it means the package name and class name have to be changed. In the example, lines 1-3 in Listing 5-9 are changed to lines 1-3 in Listing 5-10.
2. Add one field declaration used to store the wrapped object in the wrapper class. The field type is the original class. In the example, line 6 in Listing 5-10 does this.
3. All field declarations in original class do not appear in the wrapper class. Nevertheless, the corresponding SETTERS/GETTERS are generated for these fields. In the example, line 6 in Listing 5-9 does not appear in Listing 5-10 and lines 8-10 in Listing 5-10 are its SETTER/GETTER.
4. Add one static field declaration used to store relationships between all wrapper object and wrapped object. In the example, line 12 in Listing 5-10 does this.
5. Add a new static method named wrap in the wrapper class. The method functionality is to wrap an instance of the original class to the instance of the wrapper class. In the example, lines 14-21 in Listing 5-10 do this.
6. Add a new static method named unwrap in the wrapper class. The method functionality is to unwrap an instance of the wrapper class to the instance of the original class. The transformation direction is contrast to the wrap method's direction. In the example, lines

23-26 in Listing 5-10 do this.

7. Add a new constructor which is only used by the wrap method. This constructor just assigns the field added in the step 3 to an original object. In the example, lines 28-30 in Listing 5-10 do this.
8. Add all constructors in the original class to the wrapper class. For each added constructor, the signature is the same as the original constructor but the codes are changed to delegation codes. In the example, lines 32-34 in Listing 5-10 do this.
9. Add all methods in the original class to the wrapper class. For each added methods, the signature is the same as the original method but the codes are changed to delegation codes. In the example, lines 36-38 in Listing 5-10 do this.

```
01 package test;
02
03 public class Fool                //original class name: test.Fool
04     extends Foo2 {              //Fool's superclass name: test.Foo2
05
06     int a;                       //field declaration
07
08     public Fool(java.lang.String a){ //the constructor
09         //business logic
10     }
11
12     public java.lang.String toString(){
13         //business logic
14     }
15 }
```

Listing 5-9: The source code for test.Fool

```

01 package org.gjma.wrapper;
02
03 class test_Fool          //the wrapper class name
04     extends Foo2 {
05
06     test.Fool v;          //the original class test.Fool
07
08     public int GET_a(){return v.a;}//GETTER for field a within v
09
10     public void SET_a(int value){v.a=value;}//SETTER for field a within v
11
12     private static HashMap ref=new HashMap();    //object references
13
14     public static synchronized test_Fool wrap(test.Fool s){
15         test_Fool t=(test_Fool)ref.get(s); //check existence
16         if(t==null){
17             t=new test_Fool(s);
18             ref.put(s, t);
19         }
20         return(t);
21     }
22
23     public static test.Fool unwrap(test_Fool o){
24         if(o==null) return null;
25         return(o.v);
26     }
27
28     private test_Fool(test.Fool s){
29         v=s;
30     }
31
32     public test_Fool(java.lang.String a){
33         v=new test.Fool(a);
34     }
35
36     public java.lang.String toString(){
37         return(v.toString());
38     }
39 }

```

Listing 5-10: The source code for test.Fool's wrapper

Listing 5-10 is the fully source code of the wrapper class. It is worth to discuss what the method wrap and unwrap do. Because GJMA needs to keep original business logic after using the wrapper class, it is designed to return the same wrapper instance when wrapping the same

object. To achieve this, a field with type HashMap (line 12) is used to store relationships between all wrapper object and wrapped object. When used to wrap an object, the wrap method will check whether the object is ever wrapped before (line 15). If not (line 16), the wrap method will create a new wrapper instance (line 17) and then store the relationship in the field (line 18). Finally, whatever found or not found, the wrap method always return the wrapper instance (line 20). According to the design and implementation, the following two statements' results should be all true.

(1) `obj==obj;` (`obj` is an instance of `test.Foo1` class)

(2) `test_Foo1.wrap(obj) ==test_Foo1.wrap(obj);`

After all necessary wrapper classes are generated, GJMA has to replace the instructions using the wrapped-necessary classes. For example, if an instruction uses the `java.lang.String`, the instruction has to be changed to use `org.gjma.wrapper.java_lang_String`, because `java.lang.String` cannot be replaced and needs to be wrapped. The replacements may occur in three portions in a class file. For every class within target GJMAApp, do following steps:

1. Change the superclass to the corresponding wrapper class if necessary.
2. Change the field types to the corresponding wrapper class if necessary.
3. For each constructor and method, change the signature and instructions to use the corresponding wrapper class if necessary.

Listing 5-11 is an example after replacement. The shaded portion represents the modified-required parts.

```

01 package org.gjma.wrapper;
02
03 class test_Fool                //the wrapper class name
04     extends test_Fool2 {
05
06     test.Fool v;                //the original class test.Fool
07
08     public int GET_a(){return v.a;} //GETTER for field a within v
09
10     public void SET_a(int value){v.a=value;} //SETTER for field a within v
11
12     private static HashMap ref=new HashMap(); //object references
13
14     public static synchronized test_Fool wrap(test.Fool s){
15         test_Fool t=(test_Fool)ref.get(s); //check existence
16         if(t==null){
17             t=new test_Fool(s);
18             ref.put(s, t);
19         }
20         return(t);
21     }
22
23     public static test.Fool unwrap(test_Fool o){
24         if(o==null) return null;
25         return(o.v);
26     }
27
28     private test_Fool(test.Fool s){
29         v=s;
30     }
31
32     public test_Fool(java_lang_String a){
33         java.lang.String p0=java_lang_String.unwrap(a);
34         v=new test.Fool(p0);
35     }
36
37     public java_lang_String toString(){
38         java.lang.String r0=v.toString();
39         java_lang_String r1=java_lang_String.wrap(r0);
40         return(r1);
41     }
42 }

```

Listing 5-11: The source code for test.Fool's wrapper after replacements

5.4.2. Convert to Method Invocation Actions

All actions have to be converted to method invocation actions. This subsection will discuss how to do this and this processing takes place in phase 2.

5.4.2.1. Filed Manipulation Action

Everything occurring after an action took place is injected into respective constructor or method in the previous two actions. However, the solution cannot be applied to intercept a field manipulation action because there is no hook point about this action within the proxy class. Using the proxy class without modifying the original class makes it hard to intercept field manipulation actions. In GJMA systems, all field manipulation actions will first change to SETTER/GETTER, like JavaBean, and all field manipulation actions will be changed to method invocation actions. This change is made by analyzing original Java bytecode, and replacing putfield and getfield instructions with corresponding invokevirtual/invokestatic instructions as Figure 5-6 indicates. This change process is included in the preprocess step in Figure 4-11. For every class within target GJMAApp, do following steps:

1. Find all non-private fields in the class file.
2. Generate the corresponding SETTER and GETTER method for all fields found in the first step. In other words, every non-private field will cause two method generations.
3. Find all putfield/putstatic instructions in the class file, and replace them with the corresponding invokevirtual/invokestatic instructions which will invoke the corresponding SETTER method.
4. Find all getfield/getstatic instructions in the class file, and replace them with the corresponding invokevirtaul/invokestatic instructions which will invoke the corresponding GETTER method.

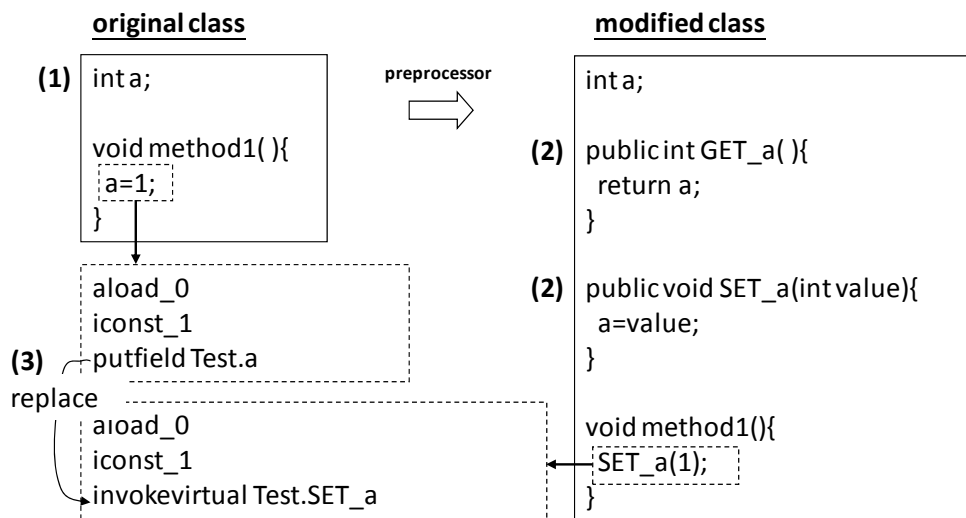


Figure 5-6: Replace field manipulation action with SETTER/GETTER

5.4.2.2. Synchronized Action

In the Java language, a synchronized action acts on an object. Nevertheless, because of the existence of remote objects, the synchronized semantics may become incorrect. In the developer's point of view, a synchronized action occurs on one object only. In GJMA, in fact, the object may have the corresponding remote object, and the synchronized action may occur on both objects at the same time. The problem is that both objects represent the same object logically yet they are different objects physically. Like field manipulation action, there is no hook point related to this action within the proxy class. Thus, in GJMA, all synchronized actions will change to method invocation actions also. This change is made by analyzing the original Java bytecode and replacing the `monitorenter` and `monitorleave` instructions with the corresponding `invokevirtual` instructions. This change process is also included in the preprocess step in Figure 4-11. For every class within target GJMAApp, do following steps:

1. Insert a field named `GJMA_LOCK` into the class file with primitive type `boolean`. The default value is set to `false`.
2. Generate two methods, `GJMA_ENTER` and `GJMA_LEAVE`. The codes within the two

methods are shown in Listing 5-12. GJMA_ENTER is like a barrier, and only one thread can return from it at the same time if and only if GJMA_LOCK is false. The GJMA framework uses these two methods to simulate a monitor, and it is equivalent to the original monitorenter/monitorleave instructions.

3. Find all monitorenter instructions in the class file and replace them with the corresponding invokevirtual instructions which will invoke the corresponding GJMA_ENTER method.
4. Find all monitorleave instructions in the class file and replace them with the corresponding invokevirtaul instructions which will invoke the corresponding GJMA_LEAVE method.

<pre>void GJMA_ENTER(){ synchronized(this){ while(GJMA_LOCK){ wait(); } GJMA_LCOK=true; } }</pre>	<pre>void GJMA_LEAVE(){ synchronized(this){ GJMA_LOCK=false; notifyAll(); } }</pre>
---	---

Listing 5-12: The source codes for GJMA_ENTER and GJMA_LEAVE

5.4.3. Generate Code for Creating Complementary Objects

When an instance is created from a class, the instance's field initialization is done explicitly in the classes' constructors. In other words, the field initialization instructions are placed in the constructors. Besides the field initialization instructions, there are other initialization codes in the constructors. In the MASTER-SLAVE mode, every managed object will have a corresponding complementary object in the remote host. It implies that some codes are placed in the remote host and the initialization process is handled by two threads instead of one thread. It may break the initialization order. Hence, a few modifications have to be made on a class to make it capable of creating an empty complementary object first. Then, all initialization methods are called after the complementary object was created. This work is done in second

phase. For every class within target GJMApp, do following steps:

1. Add a special constructor with one parameter with type GJMClass. This constructor does nothing except to call its superclass's constructor. The constructor is designed to create an empty complementary object.
2. For each original constructor, create a new method and move all codes after the invokespecial instruction within the constructor to the new method.
3. For each original constructor, its instructions are modified to call the new method added in step 2.

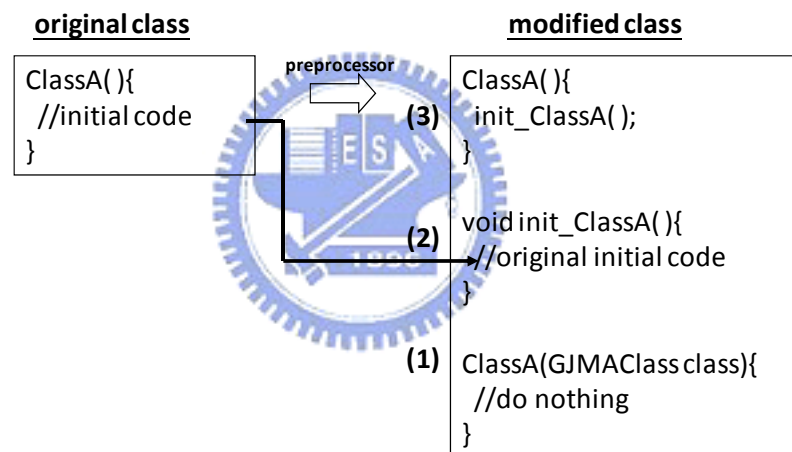


Figure 5-7: Modify codes for creating complementary object

5.4.4. Convert Array Type to Class Type

After the previous discussion, a problem still remains when the return type or some parameter's type is array. GJMA cannot generate the corresponding proxy class because there is no concrete original class for array type. To solve this problem, GJMA transfers all array types to class types generated by GJMA, changes all array manipulation instructions to new instructions or the SETTER/GETTER mentioned above, and changes all method signatures and returns types from array to object. For every class within target GJMApp, do following steps:

Table 5-5: The array class naming convention

Original array type	Generating class name	Static method to create object
byte[]	Byte_Array1	Byte_Array1.create1(int s1)
byte[][]	Byte_Array2	Byte_Array2.create2(int s1, int s2) Byte_Array2.create1(int s1)
int[]	Int_Array1	Int_Array1.create1(int s1)
java.lang.String[][][]	Object_Array3	Object_Array3.create3(int s1, int s2, int s3) Object_Array3.create2(int s1, int s2) Object_Array3.create1(int s1)

1. Find all array types in fields, method parameters and method return type.
2. Generate the corresponding class. The number of constructor parameters is equal to the dimension of the array. Some examples are shown in Table 5-5 to explain this step.
3. Find all newarray, anewarray, and multianewarray instructions in the class file, and replace them with the corresponding invokestatic instructions which will create the corresponding object.
4. Find all caload, castore, iaload, iastore, saload, sastore, laload, lastore, faload, fastore, daload, dastore, baload, bastore, aaload, and aastore instructions in the class file, and replace them with the corresponding invokevirtual instructions which will invoke the corresponding method. In this way the original code semantics will be completely preserved.

5.4.5. Insert Code for Intercepting Instance Creation

According to the previous discussion, GJMA inserts the class `org.gjma.application.GJMAObject` into the original constructor chaining to intercept instance

creation action. When intercepting an instance creation action, GJMA has to create the corresponding complementary object in the remote host. Listing 5-13 is the partially source code of the class `org.gjma.application.GJMAObject`. There are two constructors. The one without any parameters (lines 5-7) is used to intercept method invocation actions because all instance creation actions will always call this constructor. When this constructor is called, it implies an object was created and the constructor does two things. First, the constructor will get a unique object id from the object manager (`ObjMngr`) and the object manager maintains an object table to keep the relationship between the object id and the object reference. Second, the constructor will build a `GJMAMesg` which contains the creation command, including class name and object id. Then, the constructor sends the `GJMAMesg` to the remote host. The other constructor which has one parameter (lines 10-12) is used to create complementary objects. When the object manager received a creation command, the object manager will create the complementary object by using this constructor. This constructor does only one thing. The constructor makes the object manager keeps the relationship between the object id encoded in the received command and the object reference of the complementary object. In brief, an object and its complementary object both have the same object id and they are in different hosts. Besides the constructors, there are other methods overriding the methods in `java.lang.Object` to guarantee that an object and its complementary object are the same logically. For example, the method `hashCode()` acted on an object and on its complementary object have to return the same results.

```

01 package org.gjma.application;
02
03 public class GJMABject {
04
05     public GJMABject() { //used to intercept instance creation action
06         //1. make this object be managed by ObjMgr
07         //2. send create command to the remote host
08     }
09
10     public GJMABject(GJMABject c){//used to create complementary object
11         //make this complementary object be managed by ObjMgr
12     }
13
14     public int hashCode(){ //override other methods in java.lang.Object
15         if(GJMABject.is_slave_part){
16             return(super.hashCode());
17         }else{
18             //call the remote method hashCode()
19         }
20     }
21 }

```

Listing 5-13: The partially source code for GJMABject

If a class's original superclass is `java.lang.Object`, GJMA preprocessor has to replace its superclass with `org.gjma.application.GJMABject`. The steps are as follows.

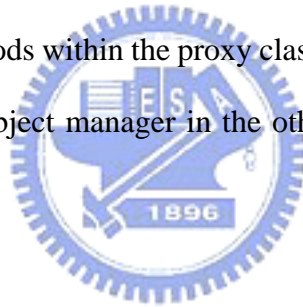
1. Add a new entry with type `CONSTANT_Class` into the constant pool. The entry represents a class named `org.gjma.application.GJMABject`.
2. Modify the superclass field in the class file. After the modification, the value of the superclass field will point to the entry added in STEP 1.
3. Modify the bytecode within the constructors. There is an `invokespecial` instruction in the beginning of every constructor. This instruction is dedicated to call constructors, which is a special method with named "`<init>`". If an `invokespecial` instruction calls the constructor belonging to `java.lang.Object`, the instruction has to be changed. After the modification, the instruction will call the constructor belonging to `org.gjma.application.GJMABject`.

5.5. GJMA Analyzer

The main mission of GJMA analyzer is generating proxy classes and ObjMgr classes. The generated proxy classes are used to intercept method invocation action and the generated ObjMgr classes are used to reflect all intercepted actions to the practical remote classes and remote methods. This sub-section includes two parts. The first part illustrates how to generate proxy class and the second part describes how to generate ObjMgr.

5.5.1. Generate Proxy Class

When a method of the proxy class is invoked, it means the practical business logic is placed in the remote host and the method invocation action has to reflect on the complementary object. Hence, the codes of these methods within the proxy classes are responsible for delegating these actions to the corresponding object manager in the other side. For every class within target GJMAApp, do following steps:



1. Make a copy of the class. It implies the proxy's class name is the same as the class's class name and the proxy's superclass is the same as the class's superclass.
2. For each method excluding constructors, replace the instructions within the method. The codes within the proxy method are responsible to delegate the request to the object manager in other host.

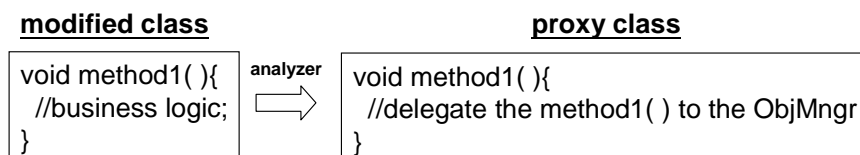
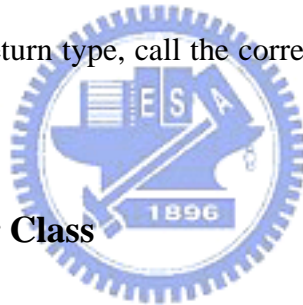


Figure 5-8. How to intercept method invoke action

The delegation codes within proxy methods do following steps:

1. Create an ActionBuilder instance to do further marshalling.
2. Call `init_invoke` method with the instance created on step 1 to create an invocation action string.
3. According to the method parameter types, call the corresponding methods to append necessary parameters to the action string.
4. Call `toString` method with the instance created on step 1 to get the finally action string. Then fill it into the action field in the GJMAMesg and send the GJMAMesg to the object manager in the remote host.
5. Wait the result returns.
6. Parse the action string in the action field in the received GJMAMesg.
7. According to the method return type, call the corresponding methods to get the result and then return it.

5.5.2. Generate ObjMngr Class



The codes in an ObjMngr class can be divided into two parts. One part is fixed and all ObjMngr implementations are the same in this part. This part provides codes to generate unique object id and manage object references. The other part is generated in the deployment time. This part includes two methods. One named `create` is used to create complementary objects and the other named `invoke` is used to call a method. When ObjMngr received a GJMAMesg with the action field, ObjMngr will do unmarshalling on the action field first. After unmarshalling, ObjMngr will know which method has to be called, either `create` or `invoke` as Figure 5-9 demonstrates.

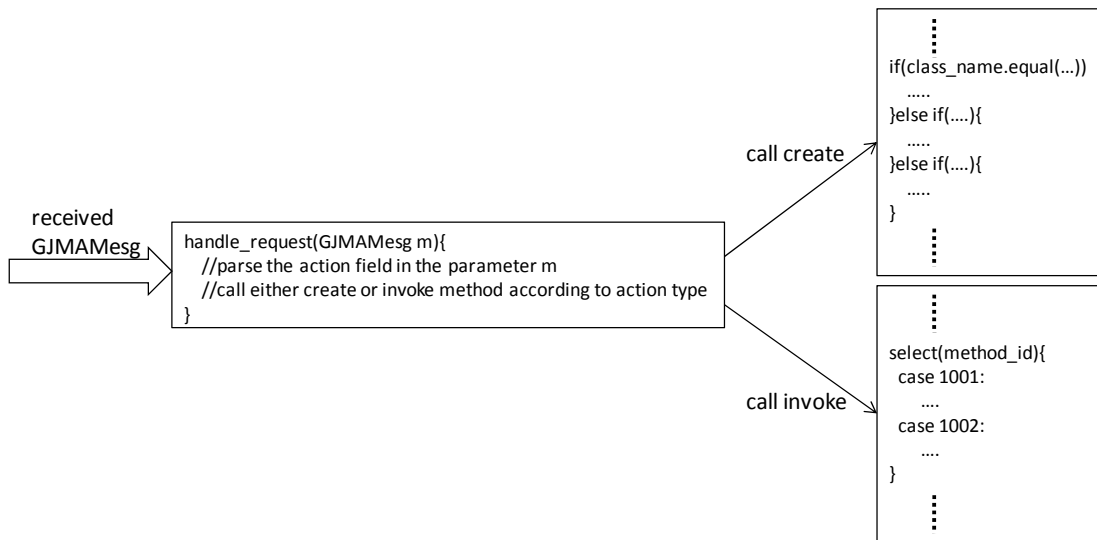


Figure 5-9: How ObjMngr to process a received GJMAMesg

5.5.2.1. Generate the create Method

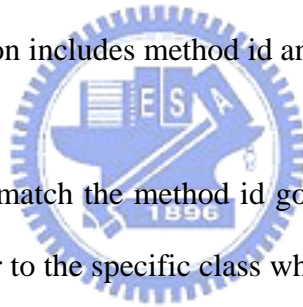
The most important part in a create method is if-else structures. The structure is responsible for comparing the class name encoded in the received GJMAMesg and creating the corresponding complementary object. The create method has a parameter. The only parameter is an ActionBuilder instance created in handle_request method and the return type is a GJMAMesg instance. Moreover, the GJMA analyzer should generate codes to do following steps.

1. Call getClass method with the received parameter to get the information about creation. The information includes class name as well as object id, and it is encapsulated in a GJMAMesg instance.
2. Use if-else statements to compare the class name got in step 1. If a match occurs, create an instance of the class by using the constructor responsible for creating a complementary object. The GJMAMesg instance got in step 1 is used as the constructor's only required parameter. After this step, GJMA will get an empty complementary object.
3. Return the GJMAMesg reference got in step 2.

5.5.2.2. Generate the invoke Method

The most important part in the invoke method is switch structures. The structure is responsible for matching the method number encoded in the received GJMAMesg and calling the corresponding method. Hence, every method within target GJMApp has to be assigned a unique method number. The invoke method has two parameters. The first parameter is an ActionBuilder instance created in handle_request method and the second parameter is a GJMAMesg instance encoded in the received GJMAMesg. Moreover, the return type is String. The GJMA analyzer should generate codes to do following steps.

1. Call getMethod method with the first received parameter to get the information about invocation. The information includes method id and it is encapsulated in a GJMAMethod instance.
2. Use switch statements to match the method id got in step 1. If a match occurs, cast the second received parameter to the specific class which the calling method belongs to. Call the corresponding method with the parameters which can be got from the first received parameter according to the data types. After the method completed, do marshalling on the return result and finally return a String.



Chapter 6 Evaluation

6.1. Programming Framework Comparison

There are many technologies capable of deploying an application to distributed computing, such as MPI (Message Passing Interface) [65] and Java RMI. However, in development time, these technologies are not fully transparent to developers and developers have to handle some extra codes for distributed purposes. In addition, these technologies may use different semantics to handle remote method invocation. For example, a parameter with specific data type may use “call by copy” in a remote method invocation, but the same parameter with the same data type may use “call by reference” in a local method invocation. Hence, developers have to know the differences prior to developments to prevent incorrect semantics. Here, we notice that which codes are remotely are determined in development time in these technologies and they are not suitable for GJMA which makes the decision in deployment time. To be precise, GJMA is fully transparent to developers in development time.

6.1.1. MPI Programming Framework

The Message Passing Interface (MPI) is a language-independent communication protocol. MPI is widely used in parallel computing and it is often used to implement distributed shared memory. Applications in different hosts can communicate with each other by passing messages. Listing 6-1 is a MPI sample code. It is worth to notice that some codes (with bold font) other than business logic are used to communicate. In other words, developers have to handle these communications by themselves.

```

01 int main(int argc, char *argv[]){
02
03 //variable declarations
04
05 MPI_Init(&argc,&argv); //MPI initialization
06 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
07 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
08
09 //business logic
10 MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD); //send
11 //business logic
12 MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat); //recv
13
14 MPI_Finalize(); //MPI finalization
15 return 0;
16 }

```

Listing 6-1: A partial sample code for MPI.

6.1.2. Java RMI Programming Framework

The Java Remote Method Invocation (Java RMI) is a Java application programming interface for performing the object equivalent of remote procedure calls. If developers want to use Java RMI, they have to define interface as Listing 6-2 shows first and then implement its corresponding RMI server as Listing 6-3 demonstrates. The same as MPI mentioned above, Java RMI still need developers to define interface and write some extra codes.

```

01 public interface SampleServer extends Remote{
02     public int echo(int a) throws RemoteException;
03 }

```

Listing 6-2: A sample interface for Java RMI.

```

01 public class SampleServerImpl extends UnicastRemoteObject
02     implements SampleServer{
03
04     public int echo(int a) throws RemoteException{
05         return a;
06     }
07 }

```

Listing 6-3: A sample RMI server implementation.

6.1.3. GJMA Programming Framework

Developing a GJMAApp is similar to developing a Java ME MIDP application as Listing 3-3 illustrates, but developers have not concerned about devices capabilities in GJMA development process. In developers' view, all classes within a GJMAApp are logically run on the same host. In this chapter, two examples are given to demonstrate how to develop a GJMAApp.

6.1.3.1. Hello World

The simplest GJMAApp is only consisted of one class implemented by developers. It implies that developers only need to implement the main class to complete the whole GJMAApp. The general GJMAApp development flow is shown in Figure 6-1. It is worth to notice again that the main class has to be extended from GJMAApp class.

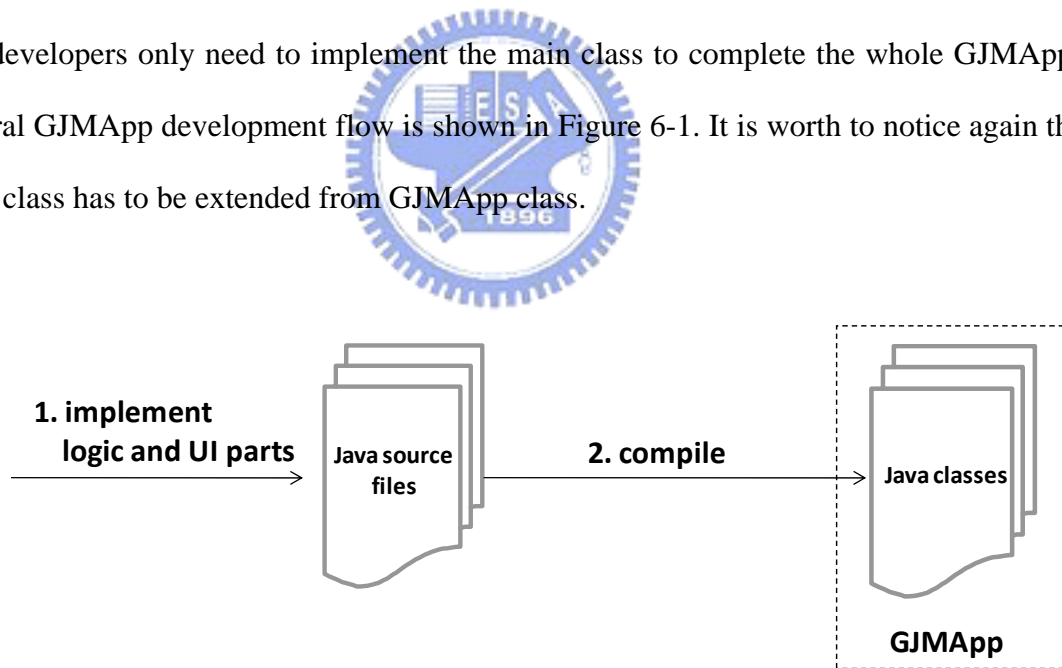


Figure 6-1: The GJMAApp development flow

A GJMAApp which will show “Hello World” on screen is used to demonstrate the simplest GJMAApp. Listing 6-4 is the source code of the main class. The source used two classes provided by GJMA framework. The class GJMAApp is used to initialize necessary resources and the class Window is used to show something in the device screen. The important point to

note is that all programming is based on Java language in GJMA development framework. All developers who are familiar with Java language can develop GJMApps easily without learning any other thing. Figure 6-2 shows snapshots of this example.

```
import org.gjma.application.GJMAApp;
import org.gjma.ui.Window;

public class Main extends GJMAApp{
    public void startApp(){ //be called when the application is started
        //create a window ui widget
        Window w=new Window();
        //set window title
        w.setTitle("Hello World");
    }

    public void stopApp(){ //be called when the application is stopped
        //do nothing in this example
    }
}
```

Listing 6-4: Hello World sample code



Figure 6-2: The GJMAApp (Hello World) accessed by different GJMClient.

6.1.3.2. Web Services

Like Web services client development flow in Java SE, developers have to get WSDL document which describes the Web services definition first and a tool (WSDL2Java) is used to generate Java stub from the WSDL file. Then, developers can directly invoke the methods

of the generated stub to access the Web service described in the WSDL document. The development flow is shown in Figure 6-3.

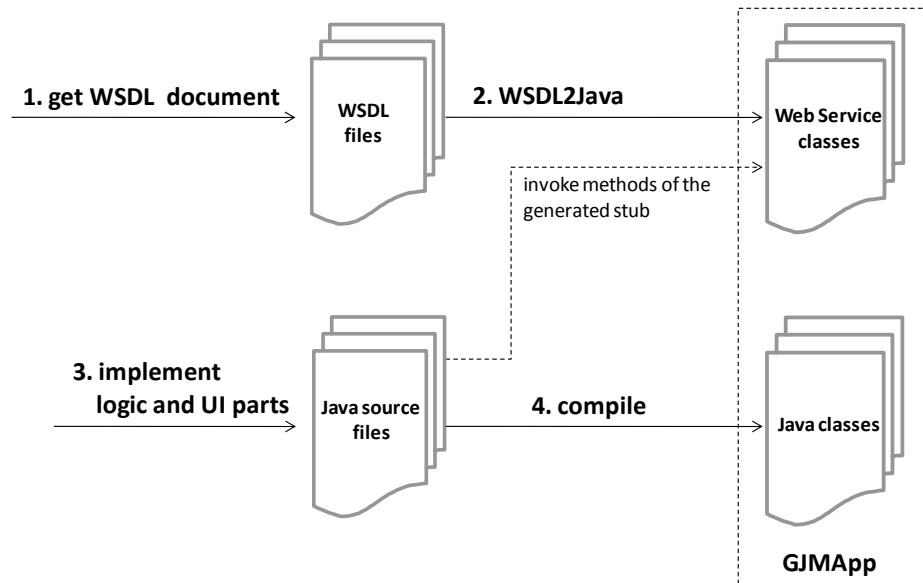


Figure 6-3: The GJMAApp using Web services development flow

In addition, like the previous example, developers still have to implement other business logic, including user interface and main class which extends GJMAApp class, to complete the GJMAApp. Listing 6-5 is the source code for a GJMAApp using Web services. The source used several user interface related classes provided by GJMA framework. Moreover, the class WeatherForecastSoap_Stub and the class WeatherForecasts are automatically generated by the WSDL2Java tools. In addition, we should notice that some event listeners have to be implemented to handle occurring events. In Listing 6-5, a listener is assigned to the button (line 23). In runtime, when the button is pressed, the handler in the listener will be invoked. In this example, the handler will invoke the Web services to get weather information according to the post code got from the text field (lines 14-21). Figure 6-4 and Figure 6-5 show snapshots of this example.

```

01 import org.gjma.application.GJMAApp;
02 import org.gjma.ui.*;
03 import weather.*; //stub for web service
04
05 public class Main extends GJMAApp{
06
07     public void startApp(){
08         Window query=new Window();           //create a window
09         query.setTitle("Query");             //set window title
10         Button ok=new Button("OK");          //create a button
11         TextField f=new TextField("PostCode", "11001"); //create a textfield
12
13         class OkListener implements GJMAButtonListener{//define listener
14             public void actionPerformed(){//invoke web service
15                 WeatherForecastSoap_Stub service = new WeatherForecastSoap_Stub();
16                 try {
17                     WeatherForecasts wf = service.getWeatherByZipCode(f.getText());
18                     text.setText(wf.getPlaceName()); //fill result to text widget
19                 }catch(Exception e){}
20                 result.show(); //show the result window
21             }
22         }
23         ok.addListener(new OkListener()); //set listener for the ok button
24         query.add(field); //add a textfield to the query window
25         query.add(ok); //add a button to the query window
26
27         Window result=new Window();           //create a window
28         result.setTitle("Result");            //set window title
29         Button back=new Button("BACK");       //create a button
30         Text text=new Text();                 //create a text
31
32         class BackListener implements GJMAButtonListener{ //define listener
33             public void actionPerformed(){
34                 query.show(); //show the query window
35             }
36         }
37         back.addListener(new BackListener()); //set listener for the back button
38         result.add(text); //add a text to the result window
39         result.add(back); //add a button to the result window
40         query.show(); //show the query window
41     }
42
43     public void stopApp(){
44         //do nothing in this example
45     }
46 }

```

Listing 6-5: Web services sample code

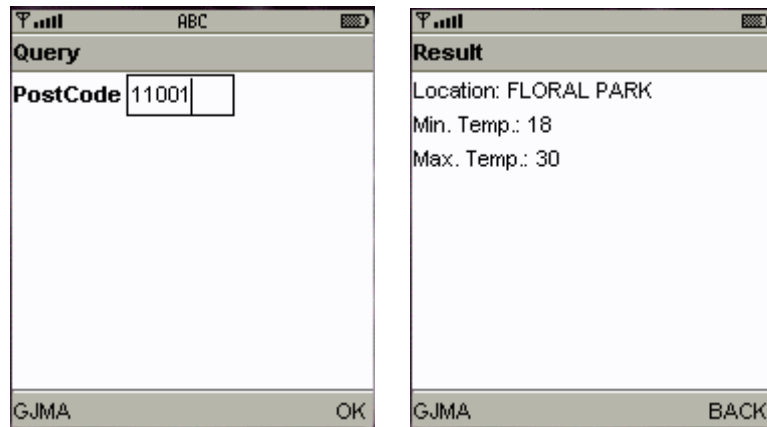


Figure 6-4: The GJMAp (Web services) accessed by GJMApStandalone.

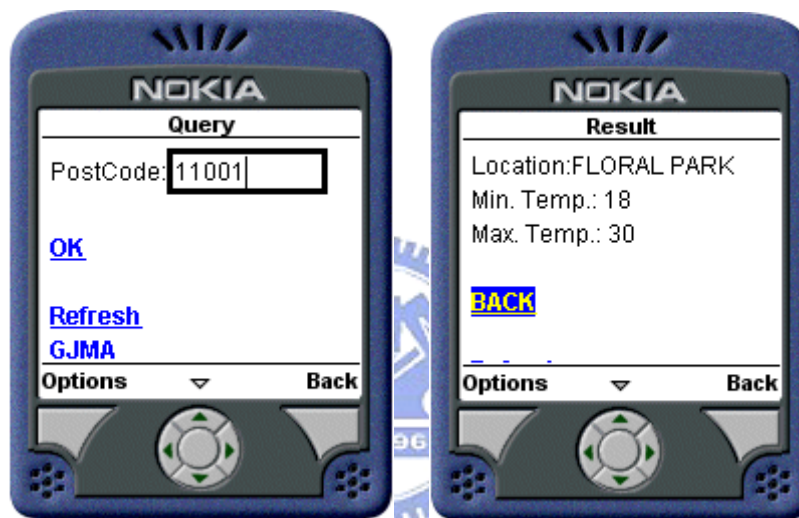


Figure 6-5: The GJMAp (Web services) accessed by WAP browser.

6.2. Performance Evaluation

Because every GJMAp is consisted of original class files (without modifications) in the BROWSER mode and the STANDLAONE mode, the performance does not be influenced by GJMA. Hence, this section only evaluates the performance in the MASTER-SLAVE mode.

```

01 TYPE echo(TYPE t){
02   return(t);
03 }

```

Listing 6-6: The test code template.

In the MASTER-SLAVE mode, some original method invocations are replaced with the corresponding remote method invocations and these remote method invocations are the main factor to influence on the performance. A test code template listed in Listing 6-6 is used to evaluate the remote method invocation performance and the test environments is listed in Table 6-1. In this dissertation, the authors compared the remote method invocation performance between GJMA and Java RMI. Moreover, different parameter types and return types are applied to the test code template and Figure 6-6 shows the result. Every bar in Figure 6-6 represents the elapsed time of every remote method invocation.

Table 6-1: Test environment.

Environment	Description
CPU	Intel Core 2 Duo 1.6 GHz
RAM	2G DDR2 667MHz
Network	Loopback (127.0.0.1)
Java platform	Java Standard Edition 1.6.0_01

For primitive data types (int, short, byte, boolean, char, long, float, double), Java RMI has better performance. The reason is that GJMA exploits the asynchronous message delivery mechanism to handle disconnection situation and Java RMI does not consider about this. In other words, every command in GJMA spends extra time passing from a queue to another queue. For array data type and object data type, GJMA has better performance. The reason is that all parameters are “call by reference” in GJMA to keep the original semantics regardless of the class locations (remote of local). Moreover, Java RMI exploits “call by copy” (serialization) to handle parameters with array data type and some object data types. The serialization action takes long time and it may break the original application semantics.

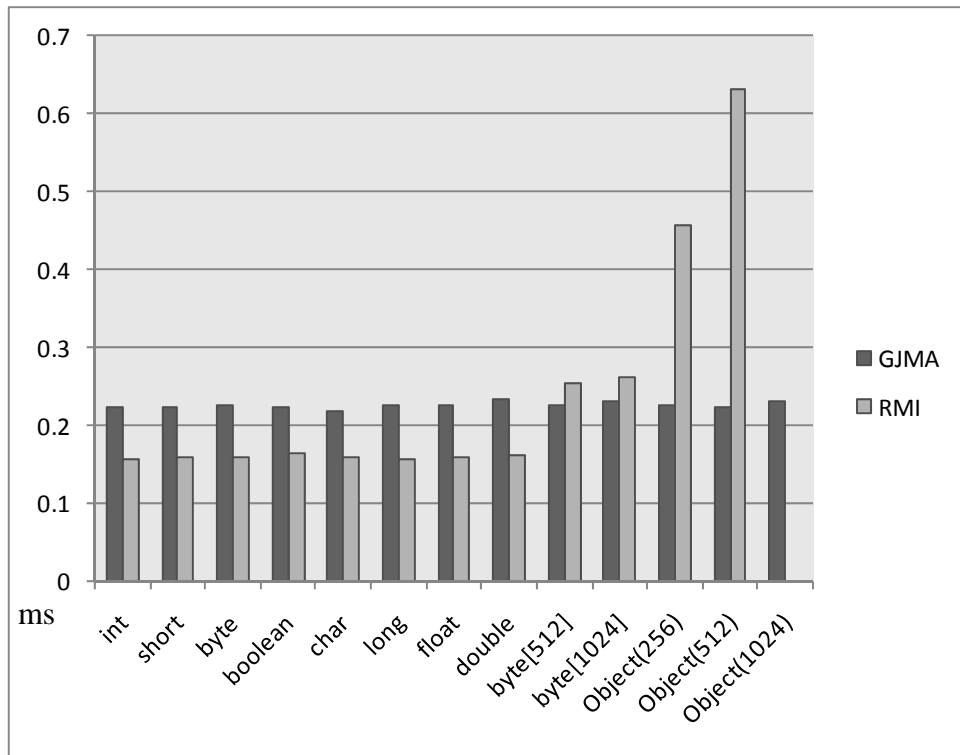


Figure 6-6: Remote method invocation performance evaluation.

6.3. Program Size Evaluation

In the BROWSER mode, end-users can use built-in browsers or GJMABrowser to access GJMApps placed on a GJMAServer. If end-users use built-in browsers, no installation is required and it costs 0 Kbytes. If end-users use GJMABrowser, it had been installed before use and it costs 16 Kbytes (compressed). In the MASTER-SLAVE mode, some classes (modified classes) are modified and some classes (proxy classes) are generated. How many bytes will be increased in a modified class depends on how many constructors and fields the corresponding original class owns. Every constructor in the original class will increase the size about 250 bytes in the modified class and every field in the original class will increase the size about 210 bytes in the modified class. Furthermore, the size of a proxy depends on how many methods the corresponding original class owns and how many parameters these methods have. A method with no parameter and no return costs about 76 bytes in a proxy file

and every parameter needs extra 30 bytes to do marshalling or unmarshalling actions.



Chapter 7 Conclusion and Future Works

In this dissertation, a novel development framework GJMA, which is capable of tailoring mobile applications to fit different end-devices and environments, is proposed and how it works is discussed in the previous chapters. GJMA currently supports WAP and Java MIDP. In other words, GJMA can be used by almost all mobile devices. In addition, three adaptation mechanisms are introduced to solve the problem about the diversity of hardware capabilities and functionalities. To handle disconnection situation, the asynchronous message delivery mechanism is designed and implemented. By using GJMA, when developing a mobile application, developers do not need to concern about the computing power as well as functionalities of the target end-devices and these resources will be effectively used. Moreover, all necessary adaptations are made by GJMA automatically, including computing model adaptation, user interface adaptation and network adaptation. Besides, because XML document is flexible and extensible, anyone can easy to extend the end-device database to support more end-devices. Nevertheless, GJMA has some issues for the moment. First, it is not fully supported Java dynamic class loading. If a GJMApp uses dynamic class loading related codes, the GJMA analyzer cannot recognize these codes and may cause some errors in runtime. Second, remote method invocation action is the performance bottleneck in the MASTER-SLAVE mode. To solve the performance issue, GJMA currently place all codes on the client side unless the codes cannot be handled by the client. In others words, only codes which cannot be executed by the client side will be placed on and handled by the server side. It can reduce the number of necessary remote method invocation actions.

In the future, more user interface widgets will be designed and implemented. For example, Java ME MIDP compatible library will be provided and then existent Java ME MIDP application will be supported by GJMA without modifications. Moreover, the GJMAMesg

formats can be tuned and GJMA may do second phase analysis during the deployment process to make performance better. In addition, a GUI deployment tool will be provided and developers can use the tool to control the deployment process.



References

- [1] G. H. Forman, J. Zahorjan, "The Challenges of Mobile Computing", *Computer*, vol. 27(4), pp. 38-47, 1994
- [2] M. Satyanarayanan, "Pervasive computing: vision and challenges", *IEEE Personal Communications*, vol. 8(4), pp. 10-17, 2001
- [3] M. Weiser, "Some computer science issues in ubiquitous computing", *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 3(3), pp. 10-21, 1999
- [4] WAP, <http://www.wapforum.org/>
- [5] Java Micro Edition (Java ME), <http://java.sun.com/javame/>
- [6] Microsoft .NET Compact Framework,
<http://msdn.microsoft.com/netframework/programming/netcf/default.aspx>
- [7] N. Medvidovic, M. Mikic-Rakic, N. R. Metha, S. Malek, "Software Architectural Support for Handheld Computing", *Computer*, vol. 36(9), pp. 66-73, 2003
- [8] 3GPP TS 22.057 V6.0.0. Mobile Execution Environment (MExE) service description; Stage 1, 2004,
<http://www.3gpp.org/ftp/Specs/html-info/22057.htm>
- [9] Attribute programming, <http://msdn2.microsoft.com/en-us/library/dcy94zz2.aspx>
- [10] M. Butler, F. Giannetti, R. Gimson, T. Wiley, "Device independence and the Web", *IEEE Internet Computing*, vol. 6(5), pp. 81-86, 2002
- [11] W. Mueller, R. Schaefer, S. Bleul, "Interactive multimodal user interfaces for mobile devices", *Proc. of the 37th Annual Hawaii International Conference on System Sciences*, 2004
- [12] J. Plomp, R. Schaefer, W. Mueller, H. Yli-Nikkola, "Comparing Transcoding Tools for Use with Generic User Interface Format", *Extreme Markup Languages*, 2001
- [13] J. Grundy, J. Hosking, "Developing adaptable user interfaces for component-based systems", *Proc. of the 1st Australian User Interface Conference*, pp. 175-194, 2002
- [14] J2ME Polish, <http://www.j2mepolish.org/>
- [15] Tzu-Han Kao, Shyan-Ming Yuan, "Designing an XML-based context-aware transformation framework for mobile execution environments using CC/PP and XSLT", *Computer Standards & Interfaces*, vol. 26(5), pp. 377-399, 2004
- [16] Tzu-Han Kao, Shyan-Ming Yuan, "Automatic adaptation of mobile applications to different user devices using modular mobile agents", *Software Practice and Experience*, vol. 35(14), pp. 1349-1391, 2005
- [17] J. Jing, A.S. Helal and A. Elmagarmid, "Client-server computing in mobile environments", *ACM Computing Surveys*, vol. 31(2), pp. 117-157, 1999
- [18] J. P. Kanter, Understanding Thin-Client/Server Computing, Microsoft Press, 1998
- [19] K. Read, F. Maurer, "Developing Mobile Wireless Applications", *IEEE Internet Computing*, vol. 7(1), pp. 81-86, 2003
- [20] J. Hunter and W. Crawford, Java Servlet Programming, O'Reilly, 2001
- [21] D. Sklar, Learning PHP 5, O'Reilly, 2004

- [22] J. Liberty, D. Hurwitz, Programming ASP.NET, O'Reilly, 2003
- [23] Wireless Markup Language (WML) Specification, <http://xml.coverpages.org/wap-wml.html>
- [24] XHTML™ 1.0 The Extensible HyperText Markup Language, <http://www.w3.org/TR/xhtml1/>
- [25] NTT DoCoMo i-mode, <http://www.nttdocomo.com>
- [26] Compact HTML (C-HTML) for Small Information Appliances, <http://www.w3.org/TR/1998/NOTE-compactHTML-19980209>
- [27] Extensible Markup Language (XML), <http://www.xml.it:23456/XML/REC-xml-19980210-it.html>
- [28] XHTML Basic, <http://www.w3.org/TR/xhtml-basic>
- [29] XHTML Mobile Profile, <http://www.wapforum.org/tech/documents/WAP-277-XHTMLMP-20011029-a.pdf>
- [30] W3C, XSL Transformations (XSLT), <http://www.w3.org/TR/xslt>
- [31] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995
- [32] M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, J. Shuster, "UIML: an appliance-independent XML user interface language", *Computer Networks*, vol. 31(11, 17), pp. 1695-1708, 1999
- [33] J. Grundy and B. Yang, "An Environment for Developing Adaptive, Multidevice User Interfaces", *Proc. of 4th Australasian User Interface Conference*, Australian Computer Society, vol. 18, 2003, pp. 47-56
- [34] M. Bisignano, G. D. Modica, O. Tomarchio, "Dynamic User Interface Adaptation for Mobile Computing Devices", *Proc. of the 2005 Symposium on Applications and the Internet Workshops*, pp. 158-161, 2005
- [35] T. Ziegert, M. Lauff, L. Heuser, "Device Independent Web Applications - The Author Once - Display Everywhere Approach", *Proc. of 4th International Conference on Web Engineering*, pp. 244-255, 2004
- [36] V. Cardellini, M. Colajanni, R. Lancellotti, P. S. Yu, "A Distributed Architecture of Edge Proxy Servers for Cooperative Transcoding", *Proc. of the 3rd IEEE Workshop on Internet Applications*, 2003
- [37] Binary Runtime Environment for Wireless (BREW), <http://brew.qualcomm.com/brew/>
- [38] Symbian, <http://www.symbian.com/>
- [39] T. Lindholm, F. Yellin, Java Virtual Machine Specification, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999
- [40] Java Enterprise Edition (Java EE), <http://java.sun.com/javaee/>
- [41] Java Standard Edition (Java SE), <http://java.sun.com/javase/>
- [42] Java Specification Request (JSR), <http://jcp.org/en/jsr/all>
- [43] CORBA, <http://www.cs.wustl.edu/~schmidt/corba-overview.html>
- [44] Java RMI, <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>
- [45] W3C, Web Services Architecture, <http://www.w3.org/TR/ws-arch>
- [46] N. A.B. Gray, "Comparison of Web Services, Java-RMI, and CORBA service implementation", *Proc. of 5th Australasian Workshop on Software and System Architecture*, 2004
- [47] Yue-Shan Chang, Min-Huang Ho, Shyan-Ming Yuan, "A unified interface for integrating information retrieval", *Computer Standards & Interfaces*, vol. 23(4), pp. 325-340, 2001
- [48] Yue-Shan Chang, Ruey-Shyang Wu, Kai-Chih Liang, Shyan-Ming Yuan, Magic Yang, "CODEX: Content-Oriented Data EXchange Model on CORBA", *Computer Standards & Interfaces*, vol. 25(4), pp. 329-343, 2003

- [49] JavaParty, <http://svn.ipd.uni-karlsruhe.de/trac/javaparty>
- [50] D. Caromel, W. Klauser, and J. Vayssiere, "Towards Seamless Computing and Metacomputing in Java", *Concurrency: Practice and Experience*, vol. 10(11-13), 1998, pp. 1043-1061
- [51] J. Kawash, A. El-Halabi and G. Samara, "Utilizing Object Compression for Better J2ME Remote Method Invocation in 2.5G Networks", *Journal of Computing and Information Technology*, vol. 14, pp. 255-264, 2006
- [52] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hopper, "Virtual network computing", *IEEE Internet Computing*, vol. 2(1), pp. 33-38, 1998
- [53] VNC2Go, <http://www.freeutils.net/vnc2go/index.jsp>
- [54] G. Canfora, G. D.Santo, E. Zimeo, "Developing Java-AWT Thin-Client Applications for Limited Devices", *IEEE Internet Computing*, vol. 9(5), pp. 55-63, 2005
- [55] G. Canfora, G. D. Santo, E. Zimeo, "Toward Seamless Migration of Java AWT-based Applications to Personal Wireless Devices", *Proc. 11th IEEE Working Conference Reverse Engineer*, IEEE CS Press, pp. 38-47, 2004
- [56] Canoo Engineering AG, "Ultra Light Client: Technology White Paper", <http://www.canoo.com/ulc/developerzone/ULCWhitePaper.pdf>
- [57] IBM AlphaWorks, Thin-Client Framework (TCF), <http://www.alphaworks.ibm.com/tech/tcf>
- [58] W3C, Cascading Style Sheet (CSS), <http://www.w3.org/Style/CSS/>
- [59] NanoX, <http://www.microwindows.org/>
- [60] Ming-Chun Cheng, Shyan-Ming Yuan, "An Adaptive Mobile Application Development Framework", LNCS 3824, pp. 765-774, 2005
- [61] N. Mansfield, *The Joy of X: An Overview of the X Window Systems*, Addison-Wesley, 1993
- [62] M. Dahm. "Byte code engineering with the BCEL API", Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, 2001
- [63] S. Liang , G. Bracha, "Dynamic class loading in the Java virtual machine", *Proc. of the 13th ACM SIGPLAN conference on Object-oriented programming*, pp. 36-44, 1998
- [64] G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols", *Communications of the ACM*, vol. 44(10), pp. 95-97, 2001
- [65] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, 1999